!-Graphs with Trivial Overlap are Context-Free

Aleks Kissinger
Department of Computer Science
University of Oxford
Oxford, United Kingdom
aleks.kissinger@cs.ox.ac.uk

Vladimir Zamdzhiev
Department of Computer Science
University of Oxford
Oxford, United Kingdom
vladimir.zamdzhiev@cs.ox.ac.uk

String diagrams are a powerful tool for reasoning about composite structures in symmetric monoidal categories. By representing string diagrams as graphs, equational reasoning can be done automatically by double-pushout rewriting. !-graphs give us the means of expressing and proving properties about whole families of these graphs simultaneously. While !-graphs provide elegant proofs of surprisingly powerful theorems, little is known about the formal properties of the graph languages they define. This paper takes the first step in characterising these languages by showing that an important subclass of !-graphs—those whose repeated structures only overlap trivially—can be encoded using a (context-free) vertex replacement grammar.

1 Introduction

String diagrams are essentially directed graphs, but instead of edges they have a more flexible notion of *wires*. Wires can be left open at one or both ends to form inputs and outputs to the diagram, or can be connected to themselves to form a circle. They have a formal semantics given by monoidal category theory [9], and have found applications in many fields. For example: in models of concurrency, they give an elegant presentation of Petri nets with boundary [15], in computational linguistics, they are used to compute compositional semantics for sentences [4], in control theory, they represent signal-flow diagrams [1, 2], and in theoretical physics, they provide the formal language of categorical quantum mechanics [3]. A *string graph* encodes a string diagram as a typed, directed graph, by replacing the wires with chains of edges containing special dummy vertices called *wire-vertices* (see Fig. 1). By contrast, the "real" vertices, labelled here f, g and h are called *node-vertices*. String graphs—originally introduced under the name "open graphs" in [7]—have the advantage that they are purely combinatoric (as opposed to geometric) objects. Equational reasoning on string diagrams is done by replacing sub-diagrams. The

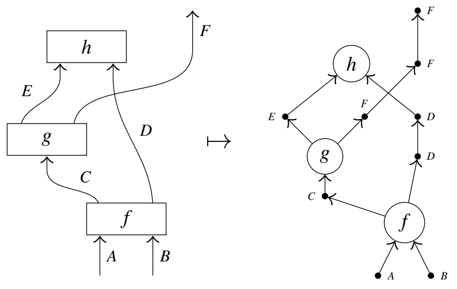
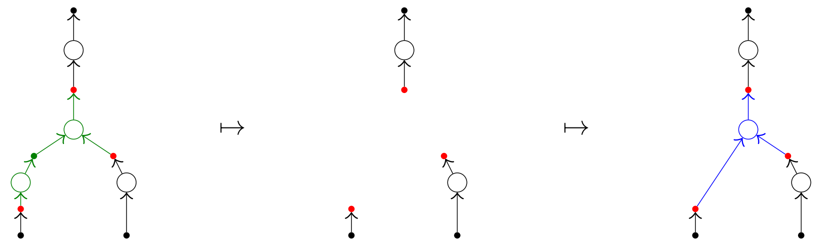


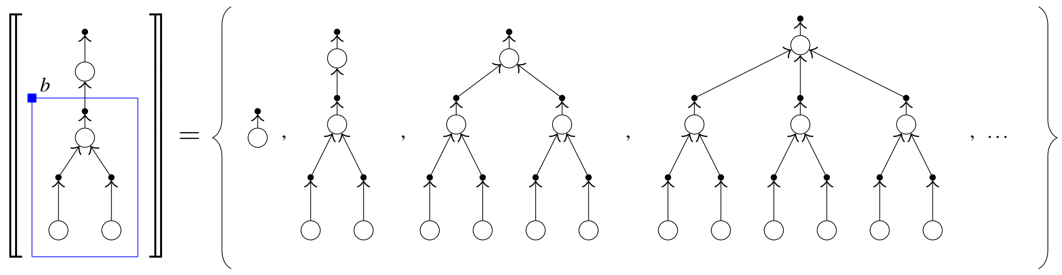
Figure 1: A string diagram, and its encoding as a string graph

presence of dummy vertices in string graphs allows this to be done with double-pushout (DPO) graph rewriting. For example, in:

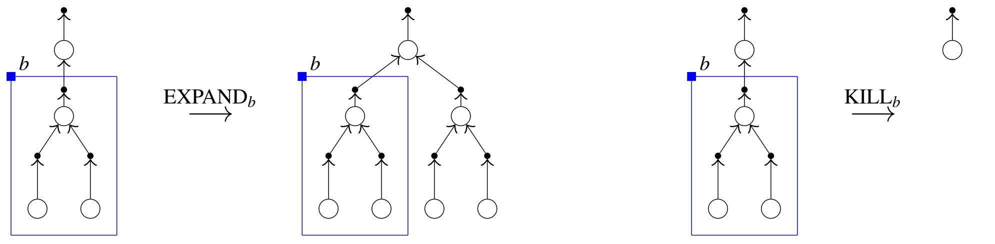
 (1.1)

the LHS (shown in green above) is replaced by the RHS (blue), using the common boundary (red). This type of rewriting for string graphs is implemented in the graphical proof assistant Quantomatic [12].

Often, one wants to reason not just about single string graphs, but entire families of them. However, informal notions of graphs with repetition are ill-suited for automated tools. To address this problem, *!-graphs* (pronounced "bang-graphs") were introduced in [6] and formalised for string graphs in [10]. The idea behind !-graphs is that certain marked subgraphs (along with their adjacent edges) can be repeated any number of times, in a manner somewhat analogous to the Kleene star. These marked subgraphs are called *!-boxes*. For example:



The !-graph in the semantic brackets represents the depicted set of string graphs. More precisely, an instance of a !-graph is obtained by repeatedly applying of the two operations EXPAND and KILL on the !-graph:



(1.2)

until all !-boxes have been eliminated. The set of concrete graphs (i.e. those not containing any !-boxes) obtainable from a !-graph via these operations is called the *language* of the !-graph.

We call the set of languages expressible by !-graphs **BG**. In this paper, we will compare the expressiveness of this language to context-free vertex replacement grammars. We will focus on confluent neighbourhood-controlled embedding grammars, with directions and edge-labels, i.e. **C-edNCE** grammars [14].

In arbitrary !-graphs, the !-boxes are allowed to nest inside of each other, but also overlap. The latter can generate patently non-context-free behaviour. Thus, it is natural to consider a restricted set of languages,

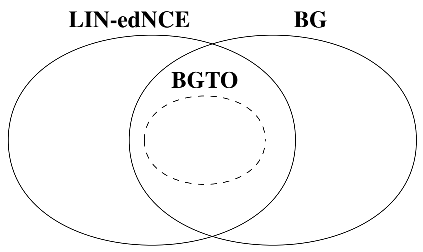


Figure 2: Relationship between **LIN-edNCE**, **BG** and **BGTO**

BGTO of *!-graphs with trivial overlap*. In this paper, we will give an algorithm for producing a C-edNCE grammar that is furthermore *linear* (LIN-edNCE) which reproduces the language of any !-graph with trivial overlap. We therefore show the relationship between graph languages illustrated in Fig. 2.

After introducing some preliminaries on string graphs, !-graphs, and C-edNCE grammars in Section 2, we will give an encoding of !-graphs into LIN-edNCE grammars in Section 3. In that section, we also demonstrate that **LIN-edNCE** contains languages not in **BG** (and hence is strictly larger than **BGTO**). Finally, we give one conjecture and discuss the prospects for reasoning with context-free string graph grammars in a manner analogous to the !-graph case in Section 4.

2 Preliminaries

Definition 2.1 (Graph [14])**.** A *graph* over an alphabet of node labels Σ and an alphabet of edge labels Γ is a tuple $H = (V, E, \lambda)$, where V is a finite set of nodes, $E \subseteq \{(v, \gamma, w) | v, w \in V, v \neq w, \gamma \in \Gamma\}$ is the set of edges and $\lambda : V \to \Sigma$ is the node labelling function. The set of all graphs with labels Σ, Γ is denoted $GR_{\Sigma,\Gamma}$.

Remark 2.2. Note, that this definition of graphs does not allow for self-loops. This is a standard (and convenient) assumption in the C-edNCE literature. By using this notion of graphs, we do not lose any expressivity for string graphs, because any string graph with a self-loop is wire-homeomorphic (see Definition 2.6) to a string graph with no self-loops.

Definition 2.3 (String Graph)**.** A *string graph* is a graph labelled by the set $\{N, W\}$, where vertices labelled N are called *node-vertices* and vertices labelled W are called *wire-vertices*, and the following conditions hold: (1) there are no edges directly connecting two node-vertices, (2) the in-degree of every wire-vertex is at most one and (3) the out-degree of every wire-vertex is at most one.

We will depict wire vertices as small black dots and node-vertices as larger white circles, as we have already done in Section 1. For simplicity, we assume there is only one type of node-vertex, but we could easily capture string graphs like 1 by taking the labels to be $\{N_f, N_g, N_h, W\}$, for example.

We also identify those wire-vertices which form the boundary of a string graph:

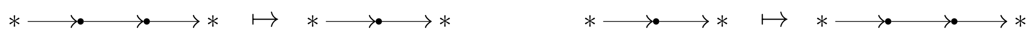
Definition 2.4 (Inputs, Outputs and Boundary Vertices)**.** A wire-vertex of a string graph G is called an *input* if it has no incoming edges. A wire-vertex with no outging edges is called an *output*. We denote with $In(G)$ and $Out(G)$ the string graphs with no edges whose vertices are respectively the inputs and outputs of G. The boundary of G is $Bound(G) := In(G) \cup Out(G)$.

Just as node-vertices represent the boxes in a string diagram, chains of wire-vertices represent the wires.

Definition 2.5 (Wires)**.** A connected chain of vertices where each endpoint is either a boundary or a node-vertex, and all other vertices are wire-vertices is called a *closed wire*. A closed wire minus its endpoints is called the *interior* of a closed wire. A cycle consisting only of wire-vertices is called a *circle*.

Often, it is more natural to consider an equivalence class of string graphs, up to *wire homeomorphism*.

Definition 2.6 (Wire-homeomorphic string graphs)**.** Two string graphs G and G' are called *wire-homeomorphic*, written $G \sim G'$ if G' can be obtained from G by either merging two adjacent wire-vertices (left) or by splitting a wire-vertex into two adjacent wire-vertices (right) any number of times:



That is, two graphs are wire-homeomorphic if the only difference between them is the number of wire-vertices used to represent a wire in the diagram (see Fig.3).

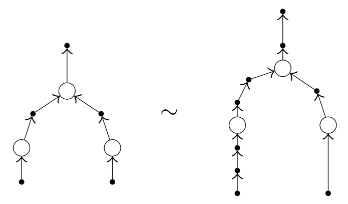


Figure 3: Wire homeomorphism

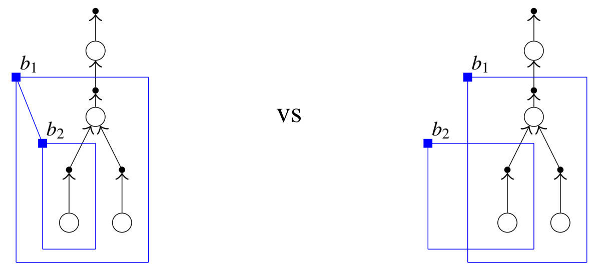
Next, we provide definitions for !-graphs and related notions. First, we need to identify which subgraphs are legal to put in !-boxes.

Definition 2.7 (Open Subgraph)**.** A subgraph O of a string graph H is said to be *open* if it is a full subgraph and furthermore $In(H \backslash O) \subseteq In(H)$ and $Out(H \backslash O) \subseteq Out(H)$.

Definition 2.8 (!-graph)**.** A *!-graph* H is a string graph, a partially ordered set $!(H)$ of *!-boxes*, and an open subgraph $B(b) \subseteq H$ for every $b \in !(H)$ such that $b \leq b' \implies B(b) \subseteq B(b')$.

The notion of openness ensures that we never put only part of a wire inside a !-box, which would cause wires to "split" in the middle as we expand the !-box, which violates the arity conditions for wire-vertices from Definition 2.3.

If $b_2 \leq b_1$, this means that b_2 is *nested* inside of b_1, which we indicate by drawing a line connecting their corners. This is to distinguish from the case where b_1 and b_2 merely *overlap*.



Definition 2.9 (Concrete Graph)**.** A !-graph with no !-vertices is called a *concrete graph* or simply a string graph.

A !-graph represents an infinite set of concrete string graphs, each of which is obtained after a finite number of applications of the EXPAND and KILL operations presented below.

Definition 2.10 (!-box Operations)**.** The two primitive !-box operations are as follows:

- $\text{KILL}_b(G) = G' \backslash B(b)$
- $\text{EXPAND}_b(G) = G \sqcup_{KILL_b(G)} G'$

where G' has the same underlying string graph as G, but with b and all of its children removed from $!(G)$. The expression for EXPAND is the disjoint union of G and G', with the vertices, edges, and !-boxes in the common subgraph $\text{KILL}_b(G)$ identified.

As seen in (1.2), KILL_b removes a !-box b and its contents, whereas EXPAND_b inserts a new copy of the contents of b. See [10] for a more detailed description of these operations.

Definition 2.11 (!-graph Language)**.** The *language* of a !-graph G is the set of all concrete string graphs obtained by applying a sequence of !-box operations on G.

Note that overlapping !-boxes are not explicitly ruled out. However, they can create some odd non-local behaviour. So, naïvely, one might want to rule out overlap entirely, but it turns out that certain instances of overlap are more innocent than others.

Definition 2.12 (Overlap and Trivial Overlap)**.** Given a pair of non-nested !-boxes b_1 and b_2, we say that b_1 and b_2 are *overlapping* if $B(b_1) \cap B(b_2) \neq \emptyset$. b_1 and b_2 *overlap trivially* if $B(b_1) \cap B(b_2)$ consists of only the interior of zero or more closed wires, where one endpoint is a node-vertex only in $B(b_1)$ and the other is a node-vertex only in $B(b_2)$.

In particular, $B(b_1) \cap B(b_2)$ can be empty, so trivial overlap generalises no-overlap. In Fig. 4 b_1 and b_2 overlap trivially, since they only overlap on the interior of a wire connecting node-vertices in the two boxes. b_3 and b_4 overlap non-trivially, since they overlap on a node-vertex. The most subtle case is b_5 and b_6, which overlap non-trivially because the shared wire-vertices are not part of a wire whose endpoints are in distinct !-boxes. Of course, all other pairs of !-boxes overlap trivially, as their intersection is just the empty set.

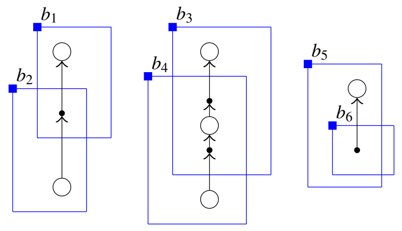


Figure 4: Trivial vs non-trivial overlap

Definition 2.13 (**BGTO**)**.** A !-graph where any two non-nested !-boxes overlap trivially is called a *!-graph with trivial overlap*. The set of all languages induced by these !-graphs is called **BGTO**.

Next, we introduce C-edNCE graph grammars. Graph grammars are a generalization of context-free grammars for strings. A graph grammar consists of a finite collection of productions which specify instructions on how to generate a family of graphs. Like their context-free string counterparts, context-free graph grammars have better structural, decidability and complexity properties compared to other more expressive mechanisms for graph transformation. We use the same definitions and conventions for graph grammars as presented in [14], which also describes C-edNCE grammars and their properties in much greater detail.

C-edNCE grammars are built using graphs with embedding. These provide the information needed to replace a non-terminal node with a new graph, and update connections accordingly.

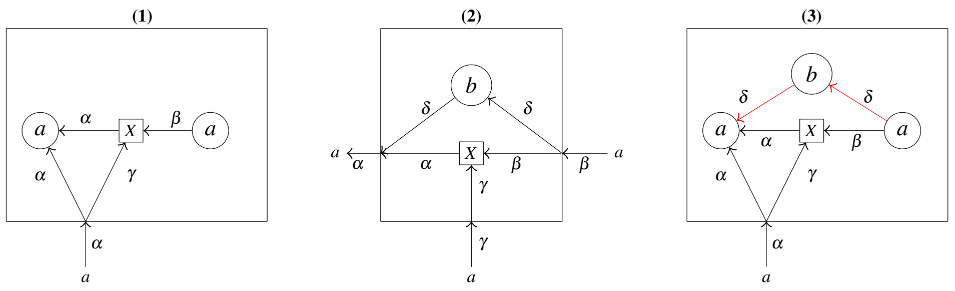


Figure 5: **(1)** Graph with embedding (H, C_H) **(2)** Graph with embedding (D, C_D) **(3)** Result of substitution $(H, C_H)[v/(D, C_D)]$ where v is the vertex with non-terminal label in H.

Definition 2.14 (Graph with embedding [14])**.** A *Graph with embedding* over labels Σ, Γ is a pair (H, C), where H is a graph over Σ, Γ and $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_H \times \{in, out\}$. C is called a *connection relation* and its elements are called *connection instructions*. The set of all graphs with embedding over Σ, Γ is denoted by $GRE_{\Sigma,\Gamma}$.

Graph grammars operate by substituting a graph (with embedding) for a non-terminal node of another graph. Connection instructions are used to introduce edges connected to the new graph based on edges connected to the non-terminal. A connection instruction as $(\sigma, \beta/\gamma, x, d)$ says to add an edge labelled γ connected to the vertex x in the new graph, for every β-labelled edge connecting a σ-labelled vertex to the non-terminal. d then indicates whether this rule applies to in-edges or out-edges of the non-terminal. More formally:

Definition 2.15 (Graph Substitution [14])**.** Let $(H, C_H), (D, C_D) \in GRE_{\Sigma,\Gamma}$ be two graphs with embedding, where H and D are disjoint. Let $v \in V_H$ be a node of H. The *substitution* of (D, C_D) for v in (H, C_H) is denoted by $(H, C_H)[v/(D, C_D)]$ and is given by the graph with embedding whose components are:

$$\begin{aligned}
V &= (V_H - \{v\}) \cup V_D \\
E &= \{(x, \gamma, y) \in E_H | x \neq v, y \neq v\} \cup E_D \\
&\quad \cup \{(w, \gamma, x) \mid \exists \beta \in \Gamma : (w, \beta, v) \in E_H, (\lambda_H(w), \beta/\gamma, x, in) \in C_D\} \\
&\quad \cup \{(x, \gamma, w) \mid \exists \beta \in \Gamma : (v, \beta, w) \in E_H, (\lambda_H(w), \beta/\gamma, x, out) \in C_D\} \\
\lambda(x) &= \begin{cases} \lambda_H(x) & \text{if } x \in (V_H - \{v\}) \\ \lambda_D(x) & \text{if } x \in V_D \end{cases} \\
C &= \{(\sigma, \beta/\gamma, x, d) \in C_H \mid x \neq v\} \\
&\quad \cup \{(\sigma, \beta/\delta, x, d) \mid \exists \gamma \in \Gamma : (\sigma, \beta/\gamma, v, d) \in C_H, (\sigma, \gamma/\delta, x, d) \in C_D\}
\end{aligned}$$

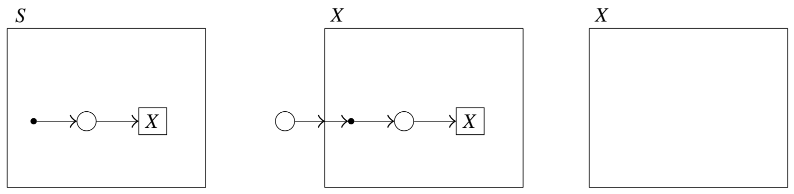
Thankfully, graphs with embedding allow for a simple graphical presentation (see Fig. 5). We draw graphs as in the previous sections, but with the additional convention that nodes with non-terminal labels are drawn as boxes instead of circles. A single connection instruction is drawn as a pair of edges crossing the outer frame. The edge and node types outside of the frame specify neighbouring node(s) which are connected to the non-terminal in the host graph and the edge(s) inside the frame indicate what new edges will be constructed to and from those nodes.

Next, we define the concept of an edNCE Graph Grammar. edNCE is an abbreviation for **N**eighbourhood **C**ontrolled **E**mbedding for **d**irected graphs with dynamic **e**dge relabeling.

Definition 2.16 (edNCE Graph Grammar [14])**.** An *edNCE Graph Grammar* is given by a tuple $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$, where Σ is the alphabet of node labels, $\Delta \subseteq \Sigma$ is the alphabet of terminal node labels, Γ is the alphabet of edge labels, $\Omega \subseteq \Gamma$ is the alphabet of final edge labels, P is a finite set of productions and $S \in \Sigma - \Delta$ is the initial nonterminal. Productions are of the form $X \to (D, C)$, where $X \in \Sigma - \Delta$ is a non-terminal node and $(D, C) \in GRE_{\Sigma,\Gamma}$ is a graph with embedding.

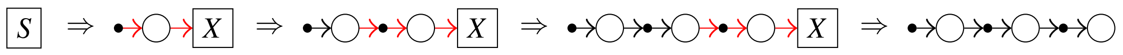
We can graphically depict an edNCE grammar by depicting all of its productions. A production is drawn exactly like a graph with embedding with the addition that we depict its associated non-terminal in the top-left part of the frame.

Example 2.17. The following grammar generates the set of all chains of node vertices with an input and no outputs:



Definition 2.18 (Derivation Step [14])**.** For a graph grammar $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ and graphs (with embedding) $H, H' \in GRE_{\Sigma,\Gamma}$ consider a vertex $v \in V_H$ and an isomorphic copy $p : X \to (D, C)$ of some production $p' \in P$. We say $H \Rightarrow_{v,p} H'$ is a *derivation step* if $\lambda_H(v) = X$ and $H' = H[v/(D, C)]$. If v and p are clear from the context, then we write $H \Rightarrow H'$. A sequence of derivation steps $H_0 \Rightarrow_{v_1,p_1} H_2 \Rightarrow_{v_2,p_2} \cdots \Rightarrow_{v_n,p_n} H_n$ is called a *derivation*. A derivation is called *creative* if H_0 and $rhs(p_i)$ are all mutually disjoint. We write $H \Rightarrow_* H'$ if there exists a creative derivation from H to H'.

Example 2.19. A derivation in the above grammar of the string graph with three node vertices:



where again we color the newly established edges in red.

Definition 2.20 (Starting Graph [14])**.** We say that $sn(S, z) \in GR_{\Sigma,\Gamma}$ is a *starting graph* if it has only one node given by z, its label is S, the graph has no edges and no connection instructions.

Definition 2.21 (Graph Grammar Language [14])**.** The graph language induced by a graph grammar $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ is given by $L(G) = \{[H] \mid H \in GR_{\Delta,\Omega}$ and $sn(S, z) \Rightarrow_* H$ for some $z\}$, where $[H]$ denotes the equivalence class of all graphs which are isomorphic to H.

Definition 2.22 (Confluence [14])**.** We say that a graph grammar $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ is *confluent* if the following holds for every graph $H \in GRE_{\Sigma,\Gamma}$: if $H \Rightarrow_{u_1,p_1} H_1 \Rightarrow_{u_2,p_2} H_{12}$ and $H \Rightarrow_{u_2,p_2} H_2 \Rightarrow_{u_1,p_1} H_{21}$ are creative derivations of G with $u_1 \neq u_2$, then $H_{12} = H_{21}$.

There are simple and easily decidable conditions for determining if an edNCE grammar is confluent. The class of confluent edNCE grammars is denoted C-edNCE.

Definition 2.23 (LIN-edNCE grammar [14])**.** An edNCE grammar G is *linear*, or a LIN-edNCE grammar, if for every production $X \to (D, C)$, D has at most one nonterminal node.

At any given point, only one non-terminal can be replaced, so LIN-edNCE grammars are automatically confluent. Hence, they are a subclass of C-edNCE grammars.

3 !-graphs vs C-edNCE grammars

In this section we outline the relationship between !-graph languages and context-free languages described by C-edNCE grammars and some of their respective subclasses. In subsection 3.1, we show that the language induced by any !-graph H with no overlapping !-boxes can be described by a LIN-edNCE grammar, which can moreover be constructed effectively from H. In subsection 3.2, we first show that there exists a !-graph with trivial overlap of !-boxes whose language cannot be directly represented using any C-edNCE grammar. Next, we show that the language of any !-graph H with trivial overlap between !-boxes can be represented by a LIN-edNCE grammar up to wire-encoding and the grammar can be effectively constructed from H as well. In subsection 3.3 we show that the classes of languages induced by (unrestricted) !-graphs and C-edNCE grammars respectively are incomparable.

3.1 !-graphs without overlap

The main result of this subsection is a theorem stating that the language induced by any !-graph with no overlapping of !-boxes can be directly represented by a LIN-edNCE grammar, which can moreover be generated effectively.

Before we present the main results in this subsection and the next one, we prove a series of lemmas which are used in the proof of the two main theorems. Each lemma describes how to build bigger LIN-edNCE grammars out of smaller ones which in turn describe the language induced by certain subgraphs of a given !-graph. All of our constructions are effective in the sense that they can be performed by a computer.

We introduce some conventions that are used throughout our proofs. Given a !-graph H, its vertices will be $\{v_1, v_2, ..., v_n\}$. To these vertices, we will associate non-final edge labels $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ which will be used in the productions of our grammars. Informally, an edge labeled with α_i will have as its source or target either the original vertex v_i or one of its copies. This is used in some productions of our grammars, so that we can easily refer to all copies of such a vertex at once and connect them to other vertices.

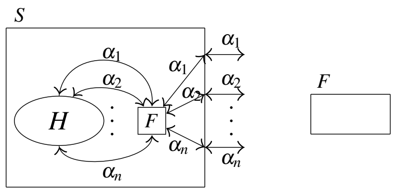
Throughout the proof, we will use grammars satisfying the following conditions:

1. The grammar is in LIN-edNCE, thus every production has at most one non-terminal node in it
2. There is a single final production which is the empty graph
3. Every sentential form with a non-terminal node is such that all terminal vertices are connected with an edge to the non-terminal node (in both directions) and every such edge has as label some α_i
4. Every production except the final one has connection instructions which specify that both incoming and outgoing edges of type α_i are connected to the non-terminal. Graphically, we will depict that using bidirectional edges as a shorthand notation for an edge in each direction.
5. For every i, there is at most one terminal vertex in the productions of a grammar which is incident to an edge with label α_i

For convenience, we will refer to a grammar satisfying 1-5 as being in *!-linear form*.

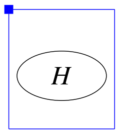
Lemma 3.1. *Given a concrete string graph H, there exists a LIN-edNCE grammar $\mathcal{G}$ which generates the language $\{H\}$. Moreover, this grammar can be effectively generated and is in !-linear form.*

Proof. This is done by the following grammar:



where the set of vertices of H is $V_H = \{v_1, v_2, ..., v_n\}$ and each edge with label α_i has as source or target v_i and the non-terminal F. □

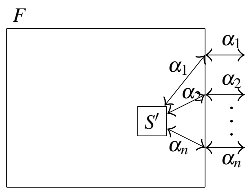
Lemma 3.2. *Given a !-graph H and a !-linear form grammar $\mathscr{G}$ which generates the same language as H, there exists a grammar $\mathscr{G}'$ which generates the same language as the following !-graph:*



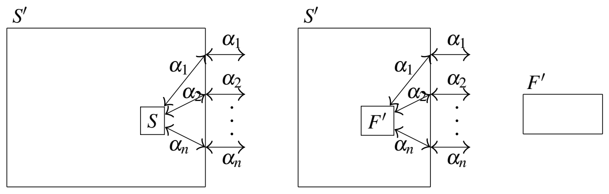
Moreover, $\mathscr{G}'$ can be effectively generated and is in !-linear form.

Proof. Let's assume that S is the starting production of $\mathscr{G}$, F is the final production of $\mathscr{G}$ and that S' and F' are not productions of $\mathscr{G}$.

First, we modify the production F to be the following:

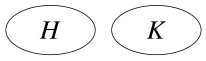


Finally, to the productions of $\mathscr{G}$ we add the following productions, where S' is the starting one:



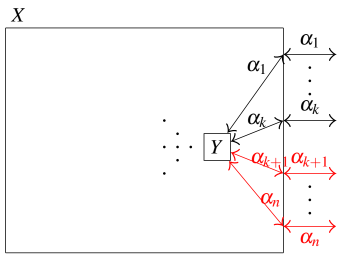
A derivation $S' \Rightarrow S \Rightarrow \cdots \Rightarrow F$ creates a concrete string graph from the language of H, so it is simulating a single EXPAND operation applied to the top-level !-box, together with a concrete instantiation of the !-boxes in H. By construction, we can iterate this, thus allowing us to generate multiple disjoint concrete graphs, all of which are in the language of H. A derivation $S' \Rightarrow F'$ simulates the final KILL operation applied to the top-level !-box. □

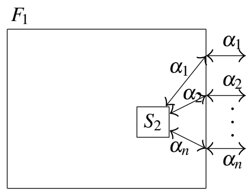
Lemma 3.3. *Given disjoint !-graphs H, K and grammars $\mathcal{G}_1, \mathcal{G}_2$ which generate the same languages as H and K respectively, then there exists a grammar $\mathcal{G}'$ which generates the same language as the following !-graph:*



Moreover, $\mathcal{G}'$ can be effectively generated and is in !-linear form.

Proof. Let the vertices of H be $\{v_1, v_2, \ldots, v_k\}$ and let the vertices of K be $\{v_{k+1}, v_{k+2}, \ldots, v_n\}$. Also, let S_i and F_i be the starting and final productions respectively of $\mathcal{G}_i$. First, we modify each production X of $\mathcal{G}_1$, except F_1, by adding connection instructions for edge labels $\alpha_{k+1}, \ldots, \alpha_n$ in the following way (left):

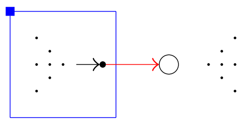




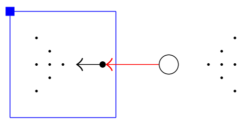
where the new additions are colored in red. This doesn't change the language of $\mathcal{G}_1$ and is done so that we can put the grammar in the required form. Similiarly, modify all productions of $\mathcal{G}_2$, except F_2, by adding to their connection instructions the missing edge labels $\alpha_1, \alpha_2, \ldots, \alpha_k$. Finally, modify F_1 to be the production depicted above (right), so that we can chain together the two grammars.

The required grammar $\mathcal{G}'$ has as its productions the modified productions of $\mathcal{G}_1$ and $\mathcal{G}_2$ with starting production S_1. A derivation $S_1 \Rightarrow \cdots \Rightarrow F_1$ creates a concrete graph from the language of H and a derivation $S_2 \Rightarrow \cdots \Rightarrow F_2$ creates a graph from the language of K. By chaining the two grammars, we simply generate two disjoint concrete graphs, one from the language of H and one from the language of K, as required. □

Lemma 3.4. *Given !-graph H, where H contains a !-box b and given a !-linear form grammar $\mathcal{G}$ which generates the same languages as H, there exist grammars $\mathcal{G}'$ and $\mathcal{G}''$ which generates the same languages as:*

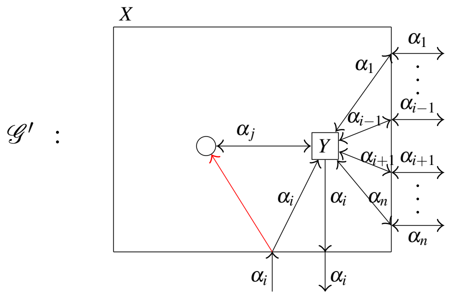


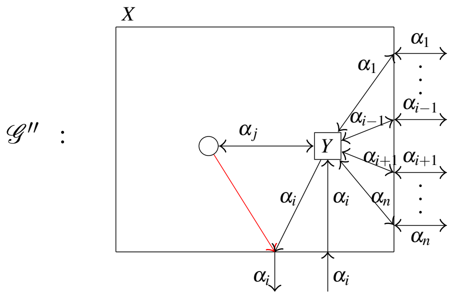
and



respectively, where in both cases, the newly depicted edge (colored in red) is incident to the !-box b in H and the edge is also incident to a node-vertex in H which is not in any !-boxes. Moreover, these grammars can be effectively generated and are in !-linear form.

Proof. In both cases, for the newly depicted edge, identify the wire-vertex as v_i and the node-vertex as v_j. To get $\mathscr{G}'$, identify the unique production X of $\mathscr{G}$, such that X contains a node-vertex incident to an edge with non-final label α_j. Then add to its connection instructions a new edge in the following way (left):





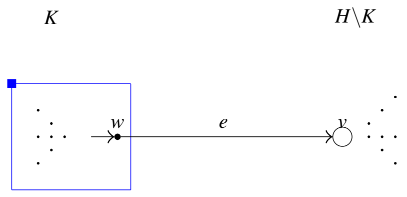
where the red-colored edge is the new addition. To get the grammar $\mathscr{G}''$, follow a similar same procedure (shown on the right above). This modification has the effect that we connect all copies of the wire vertex to the single node vertex (in the appropriate direction), which is the only change required compared to the concrete graphs of H. $\square$

Theorem 3.5. *Given a !-graph H such that it doesn't have any overlapping !-boxes, there exists a LIN-edNCE grammar G which generates the same language as H. Moreover, this grammar can be effectively generated and is in !-linear form.*

Proof. We present a proof by induction on the number of !-boxes of H.

For the base case, if H has no !-boxes, then lemma 3.1 completes the proof.

For the step case, pick any top-level !-box and let's consider the full subgraph of H it induces. Call this subgraph K. Any vertex v of $H \backslash K$ which is adjacent to K must be a node-vertex, because otherwise this would violate the openness condition of !-boxes. Let $w \in K$ be a wire-vertex that v is adjacent to and let e be the edge connecting v to w.



If v is in some !-box b, then the openness condition of !-boxes implies that w must also be in b. However, we have assumed that H does not contain overlapping !-boxes, so this is not possible and thus v is not in any !-boxes. Therefore, we can use lemma 3.4 to reduce the problem to showing that we can effectively generate a grammar for $H \backslash e$. Similarly, by applying the same lemma multiple times, we can reduce the problem to showing that we can effectively generate a grammar for the !-graph consisting of the disjoint !-graphs K and $H \backslash K$. Applying lemma 3.3 and the induction hypothesis then reduces the problem to showing the theorem for K. Finally, we can apply lemma 3.2 to K and then the induction hypothesis to complete the proof. $\square$

3.2 !-graphs with trivial overlap

We begin by providing an example of a !-graph with trivial overlap of !-boxes which cannot be directly represented using any C-edNCE grammar. Then, we prove the second main theorem of this work which shows that the language of any !-graph with trivial overlap of !-boxes can be represented by a LIN-edNCE grammar up to wire-encoding. The construction of the grammar is also effective.

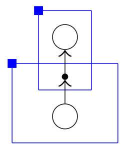


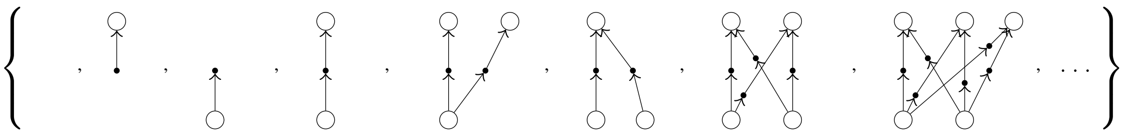
Figure 6: !-graph with no direct encoding in **C-edNCE**

Proposition 3.6. *The generative power of C-edNCE grammars and Hyperedge Replacement grammars on string graphs is the same.*

Proof. The graph $K_{3,3}$ is not a subgraph of any string graph. Then, the proposition follows immediately from the main result in [5]. □

Proposition 3.7. *The language induced by the !-graph with trivial overlap of !-boxes shown in Fig. 6 cannot be directly described using any C-edNCE grammar.*

Proof. The language of the above !-graph is given by:

$L :=$ 

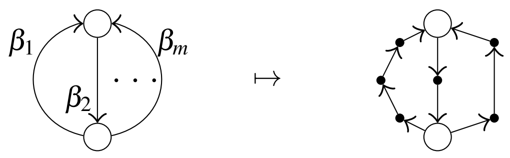
In other words, it's the set of complete bipartite graphs $K_{m,n}$ where each node-vertex is connected to other node vertices through a single wire vertex.

Let us assume for contradiction that there exists a C-edNCE grammar which generates L. From proposition 3.6, it follows that L can be described using a hyperedge replacement grammar. Then, a simple application of the pumping lemma for hyperedge-replacement languages [8] yields a contradiction. □

Note, however, that the graph language of complete bipartite graphs $K_{m,n}$ (without wire vertices in-between) can be generated by a LIN-edNCE grammar. Moreover, this language is wire-homeomorphic to the language in the above proposition. However, removing the wire vertices from our C-edNCE language brings another complication – we need to account for parallel edges between node vertices by introducing additional labels on the parallel edges, because in most of the literature, C-edNCE grammars do not allow for parallel edges of the same type.

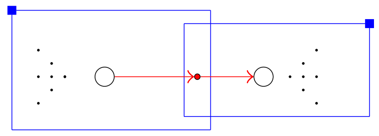
Using these two ideas, we define the notion of wire-encoding and then show the main result of this subsection.

Definition 3.8 (Wire-encoding)**.** We say that two graphs H and H' are equal up to *wire-encoding*, if we can get one from the other by replacing every edge with special label β_k by a closed wire with endpoints the source and target of the original edge.



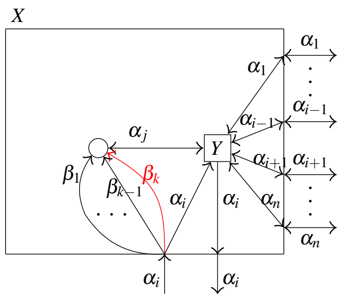
We also say that two graph languages L and L' are equal up to wire-encoding, if there exists a bijection $f : L \to L'$, s.t. for every $H \in L$, H and $f(H)$ are equal up to wire-encoding.

Lemma 3.9. *Given !-graph H which contains non-nested !-boxes b_1 and b_2 and given a !-linear form grammar $\mathscr{G}$ which generates the same languages as H, there exists a grammar $\mathscr{G}'$ which generates the same language, up to wire-encoding, as the following !-graph:*



where the new additions are coloured in red and the newly depicted wire-vertex is in both b_1 and b_2. Moreover, this grammar can be effectively generated and is in !-linear form.

Proof. For the newly depicted edges, identify the source node-vertex as v_i and the target node-vertex as v_j. To get the desired grammar $\mathscr{G}'$, identify the unique production X of $\mathscr{G}$, such that X contains a node incident to an edge with non-final label α_j. Then, add to its connection instructions a new edge with (final) label β_k:

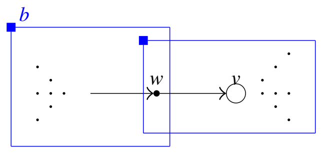


where the new addition is coloured in red. Note, that with this construction, we are not creating the depicted wire-vertex, nor any of its copies. We are connecting all copies of the node-vertex v_i to all copies of the node-vertex v_j directly with edges labelled with β_p.

□

Theorem 3.10. *Given a !-graph H such that the only overlap between !-boxes in H is trivial, then there exists a LIN-edNCE grammar G which generates the same language as H, up to wire-encoding. Moreover, this grammar can be effectively generated and is in !-linear form.*

Proof. The proof is the same as for the previous theorem, except that we have to consider an additional case, namely, when the node-vertex v is in a !-box.



In this case, the wire-vertex w is in both !-boxes and we can apply lemma 3.9 to reduce the problem to showing that $H \backslash w$ can be handled. If we set W to be the set of all wire vertices which overlap with b and another !-box, then applying the same lemma multiple times reduces the problem to showing that $H \backslash W$ can be handled. However, we have assumed that only trivial overlap between !-boxes exists in H and therefore in $H \backslash W$ there is no overlap between b and any other !-boxes. Then, the proof can be finished using the same arguments as in the previous theorem. □

3.3 The power of context-free languages

In this subsection, we will show that there exists a LIN-edNCE language which cannot be induced by any (unrestricted) !-graph. We start by providing some definitions.

Definition 3.11 (Maximum distance)**.** For a graph G and vertices $v, u \in G$, the *distance* between u and v is the length of the shortest path connecting u and v. If there is no path between u and v then we say that the distance is -1. The distance between a vertex and itself is 0. The *maximum distance* for a graph G is the largest distance among all pairs of vertices.

Definition 3.12 (Bounded maximum distance)**.** For a set of graphs $\mathscr{G} = \{G_1, G_2, G_3, \ldots\}$, we say that $\mathscr{G}$ is of *bounded maximum distance* if there exists an integer $n \in \mathbb{N}$, such that the maximum distance for every graph in $\mathscr{G}$ is smaller than n. If such an n does not exist, then we say that $\mathscr{G}$ is of *unbounded maximum distance*.

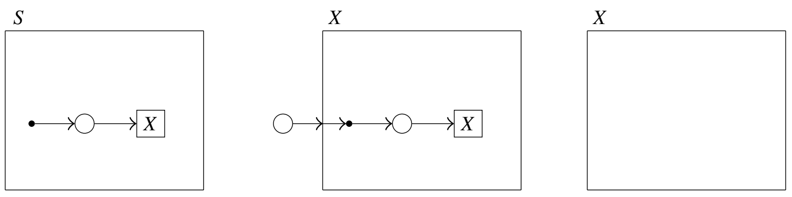
Proposition 3.13. *The language induced by any !-graph is of bounded maximum distance.*

Proof. Consider a !-box $B \subset G$. Applying a KILL operation to B cannot increase the maximum distance. Applying an EXPAND operation once could potentially increase it, however, applying an EXPAND more than once will not increase it any further. Thus, regardless of how many times an EXPAND operation is applied to B the maximum distance can only increase by a fixed amount. Also, because of the symmetric properties of the EXPAND map, any nested !-boxes within B or any overlapping !-boxes can increase the maximum distance with a fixed amount as well regardless of how many EXPAND operations are applied to any of them.

By combining the above observation with the fact that G has finitely many !-boxes we can conclude that the set of graphs induced by G is of bounded maximum distance. □

In the next proposition, we show that even severely restricted graph grammars can generate languages which are not induced by any !-graph, thereby establishing that **C-edNCE** $\not\subseteq$ **BG**.

Proposition 3.14. *The language of the LIN-edNCE grammar of Example 2.17:*



is not induced by any !-graph, even up to wire-encoding.

Proof. The language generated by this grammar is of unbounded maximum distance and thus it cannot be generated by any !-graph. □

4 Conclusion and future work

We have shown that the language of a !-graph with trivial overlap can always be encoded using a context-free grammar. **BGTO** languages are in fact a natural restriction to **BG**, and have already arisen in alternative !-box formalisations, such as the (non-commutative) !-tensors described in [11]. Thus, gaining a better understanding of **BGTO** languages is valuable in its own right. The presence of non-trivial overlap tends to cause highly non-local effects when expanding !-boxes, so it could be the case that the property of being context-free actually *characterises* trivial overlap. In other words:

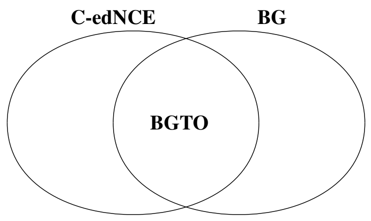
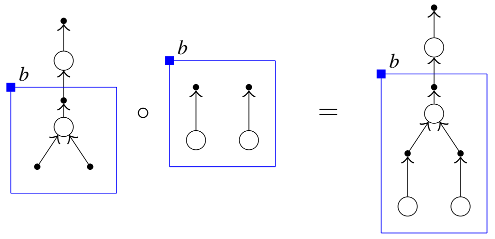


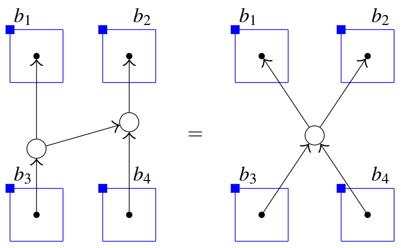
Figure 7: Conjectured relationship between **C-edNCE** and **BG**

Conjecture 4.1. *The language induced by any !-graph which contains !-boxes whose overlap is non-trivial cannot be described by a C-edNCE grammar, even up to wire-encoding.*

If the conjecture holds, then the Venn Diagram in Section 1 would simplify to the one in fig. 7, which lends credence to the notion that BGTO languages are the string graph analogue to regular languages.

Quite aside from classification issues, the next step for context-free string graphs grammars is to develop the tools for working with them. For example, since they are built on top of string graphs, !-graphs can be plugged together along inputs and outputs to get new !-graphs, and more importantly, they can be used to define *!-graph rewrite rules*:





Just as !-graphs represent families of string graphs, !-graph rewrite rules represent families of string graph rewrite rules. Furthermore, new !-graph rules can be introduced from concrete ones using a technique called !-box induction [13]. The natural next step in this program then is to see if composition, rewriting, and a notion of graphical induction carry through to the context-free case.

Acknowledgements. We would like to thank the anonymous referees for their feedback and we wish to gratefully acknowledge financial support from EPSRC, the Scatcherd European Scholarship, and the John Templeton Foundation.

References

[1] John C. Baez & Jason Erbele (2014): *Categories in Control*. Available at `http://arxiv.org/abs/1405.6881`. ArXiv:1405.6881.

[2] Filippo Bonchi, Paweł Sobociński & Fabio Zanasi (2015): *Full Abstraction for Signal Flow Graphs*. In: *Principles of Programming Languages, POPL'15.*, doi:10.1145/2676726.2676993.

[3] B. Coecke (2010): *Quantum picturalism*. *Contemporary Physics* 51(1), pp. 59–83, doi:10.1080/00107510903257624.

[4] B. Coecke, E. Grefenstette & M. Sadrzadeh (2013): *Lambek vs. Lambek: Functorial vector space semantics and string diagrams for Lambek calculus*. *Annals of Pure and Applied Logic* 164(11), pp. 1079 – 1100, doi:10.1016/j.apal.2013.05.009.

[5] B. Courcelle (1995): *Structural Properties of Context-Free Sets of Graphs Generated by Vertex Replacement*. *Information and Computation* 116(2), pp. 275 – 293, doi:10.1006/inco.1995.1020.

[6] L. Dixon & R. Duncan (2009): *Graphical reasoning in compact closed categories for quantum computation*. *Annals of Mathematics and Artificial Intelligence* 56(1), pp. 23–42, doi:10.1007/s10472-009-9141-x.

[7] L. Dixon & A. Kissinger (2013): *Open-graphs and monoidal theories*. *Mathematical Structures in Computer Science* 23, pp. 308–359, doi:10.1017/S0960129512000138.

[8] A. Habel & H-J. Kreowski (1987): *Some structural aspects of hypergraph languages generated by hyperedge replacement*. In: *STACS 87*, *Lecture Notes in Computer Science* 247, Springer Berlin Heidelberg, pp. 207–219, doi:10.1007/BFb0039608.

[9] A. Joyal & R. Street (1991): *The geometry of tensor calculus, I*. *Advances in Mathematics* 88(1), pp. 55 – 112, doi:10.1016/0001-8708(91)90003-P.

[10] A. Kissinger, A. Merry & M. Soloviev (2012): *Pattern graph rewrite systems*. In: Proceedings 8th International Workshop on *Developments in Computational Models*, 143, pp. 54–66, doi:10.4204/EPTCS.143.5.

[11] A. Kissinger & D. Quick (2014): *Tensors, !-graphs, and non-commutative quantum structures*. In: *Proceedings of QPL 2014*, 172, pp. 56–67, doi:10.4204/EPTCS.172.5.

[12] Aleks Kissinger & Vladimir Zamdzhiev (2015): *Quantomatic: A proof assistant for diagrammatic reasoning*. Available at `http://arxiv.org/abs/1503.01034`. ArXiv:1503.01034.

[13] A. Merry (2014): *Reasoning with !-Graphs*. DPhil Thesis, University of Oxford, `http://arxiv.org/abs/1403.7828`.

[14] G. Rozenberg, editor: *Handbook of graph grammars and computing by graph transformation, vol 1: Foundations*. World Scientific, 1997.

[15] P. Sobocinski (2010): *Representations of Petri net interactions*. In: *CONCUR 2010 - Concurrency Theory, LNCS* 6269, Springer, pp. pp 554–568, doi:10.1007/978-3-642-15375-4_38.