Graph generation using a graph grammar

Hiroshi Kajino

IBM Research - Tokyo
I will talk about an application of formal language to a graph generation problem

- **Formal language**
  - Context-free grammar
  - Hyperedge replacement grammar (HRG)
  - HRG inference algorithm

- **Application to molecular graph generation**
  - Molecular hypergraph grammar (a special case of HRG)
  - MHG inference algorithm
  - Combination with VAE
A molecular graph should satisfy some constraints to be valid

- **Learning a generative model of a molecular graph**
  - **Input:** set of molecular graphs \( G = \{g_1, g_2, \ldots, g_N\} \)
  - **Output:** probability distribution \( p(g) \) such that \( g_n \sim p \)

- **Hard vs. soft constraints on \( p(g) \)'s support**
  - **Hard constraint:** valence condition
    - \( \exists \) rule-based classifier that judges this constraint
  - **Soft constraint:** stability
    - \( \nexists \) rule-based classifier, in general

**Formal language can help**
Why should we care about a formal language?

A formal language defines a set of strings with certain properties; an associated grammar tells us how to generate them.

- **Formal language**

  Typically defined as a set of strings

  - **Language** point of view
    
    $\mathcal{L} = \{a^n b^n : n \geq 1\} \subset \{a, b\}^* = \Sigma^*$
    
    (= a subset of all possible strings)
  
  - **Generative** point of view
    
    A grammar is often associated with a language
    
    $\mathcal{G} = (\{S\}, \{a, b\}, S, \{S \rightarrow ab, S \rightarrow aSb\})$
Why should we care about a formal language?

A formal language defines a set of graphs satisfying hard constraints; an associated grammar tells us how to generate them.

- **Formal language**
  
  Typically defined as a set of graphs:
  
  - **Language** point of view
    
    $\mathcal{L} = \{ \text{Molecules satisfying the valence conditions} \} \subset \{ \text{All possible graphs} \}$
  
  - **Generative** point of view
    
    A grammar is often associated with a language
    
    $\Rightarrow$ how to generate graphs in $\mathcal{L}$
I will talk about an application of formal language to a graph generation problem

- **Formal language**
  - Context-free grammar
  - Hyperedge replacement grammar (HRG)
  - HRG inference algorithm

- **Application to molecular graph generation**
  - Molecular hypergraph grammar (a special case of HRG)
  - MHG inference algorithm
  - Combination with VAE
Context-free grammar

CFG generates a string by repeatedly applying a production rule to a non-terminal, until there exists no non-terminal

- Context-free grammar $G = (V, \Sigma, R, S)$
  - $V$: set of non-terminals
  - $\Sigma$: set of terminals
  - $R$: set of production rules
  - $S \in V$: the start symbol

- Example
  - $V = \{S\}$
  - $\Sigma = \{a, b\}$
  - $R = \{S \rightarrow ab, S \rightarrow aSb\}$
  - $S$
Context-free grammar

CFG generates a string by repeatedly applying a production rule to a non-terminal, until there exists no non-terminal

- **Context-free grammar** \( G = (V, \Sigma, R, S) \)
  - \( V \): set of non-terminals
  - \( \Sigma \): set of terminals
  - \( R \): set of production rules
  - \( S \in V \): the start symbol

- **Example**
  - \( V = \{ S \} \)
  - \( \Sigma = \{ a, b \} \)
  - \( R = \{ S \rightarrow ab, S \rightarrow aSb \} \)
  - \( S \)
Context-free grammar

CFG generates a string by repeatedly applying a production rule to a non-terminal, until there exists no non-terminal

- **Context-free grammar** $G = (V, \Sigma, R, S)$
  - $V$: set of non-terminals
  - $\Sigma$: set of terminals
  - $R$: set of production rules
  - $S \in V$: the start symbol

- **Example**
  - $V = \{S\}$
  - $\Sigma = \{a, b\}$
  - $R = \{S \rightarrow ab, S \rightarrow aSb\}$
  - $S$
Context-free grammar

CFG generates a string by repeatedly applying a production rule to a non-terminal, until there exists no non-terminal

- Context-free grammar $G = (V, \Sigma, R, S)$
  - $V$: set of non-terminals
  - $\Sigma$: set of terminals
  - $R$: set of production rules
  - $S \in V$: the start symbol

- Example
  - $V = \{S\}$
  - $\Sigma = \{a, b\}$
  - $R = \{S \rightarrow ab, S \rightarrow aSb\}$
  - $S$
CFG generates a string by repeatedly applying a production rule to a non-terminal, until there exists no non-terminal

**Context-free grammar** $\mathcal{G} = (V, \Sigma, R, S)$

- $V$: set of non-terminals
- $\Sigma$: set of terminals
- $R$: set of production rules
- $S \in V$: the start symbol

**Example**

- $V = \{S\}$
- $\Sigma = \{a, b\}$
- $R = \{S \rightarrow ab, S \rightarrow aSb\}$
- $S$

$aaSbb$
Context-free grammar

CFG generates a string by repeatedly applying a production rule to a non-terminal, until there exists no non-terminal

- **Context-free grammar** \( \mathcal{G} = (V, \Sigma, R, S) \)
  - \( V \): set of non-terminals
  - \( \Sigma \): set of terminals
  - \( R \): set of production rules
  - \( S \in V \): the start symbol

- **Example**
  - \( V = \{ S \} \)
  - \( \Sigma = \{ a, b \} \)
  - \( R = \{ S \rightarrow ab, S \rightarrow aSb \} \)
  - \( S \)

\[ aaSbb \]
Context-free grammar

CFG generates a string by repeatedly applying a production rule to a non-terminal, until there exists no non-terminal

- Context-free grammar $G = (V, \Sigma, R, S)$
  - $V$: set of non-terminals
  - $\Sigma$: set of terminals
  - $R$: set of production rules
  - $S \in V$: the start symbol

- Example
  - $V = \{S\}$
  - $\Sigma = \{a, b\}$
  - $R = \{S \rightarrow ab, S \rightarrow aSb\}$
  - $S$

$aaabbb$
I will talk about an application of formal language to a graph generation problem

- **Formal language**
  - Context-free grammar
  - Hyperedge replacement grammar (HRG)
  - HRG inference algorithm

- **Application to molecular graph generation**
  - Molecular hypergraph grammar (a special case of HRG)
  - MHG inference algorithm
  - Combination with VAE
Hyperedge replacement grammar

Hypergraph is a generalization of a graph

- Hypergraph $H = (V, E)$ consists of...
  - Node $v \in V$
  - Hyperedge $e \in E \subseteq 2^{|V|}$: Connect an arbitrary number of nodes
    
    cf, An edge in a graph connects exactly two nodes
Hyperedge replacement grammar

HRG generates a hypergraph by repeatedly replacing non-terminal hyperedges with hypergraphs

- **Hyperedge replacement grammar (HRG)** \( \mathcal{G} = (V, \Sigma, R, S) \)
  - \( V \): set of non-terminals
  - \( \Sigma \): set of terminals
  - \( S \): start symbol
  - \( R \): set of production rules

A rule replaces a non-terminal hyperedge with a hypergraph

[References: Feder, 71, Pavlidis+, 72]
Hyperedge replacement grammar

Start from start symbol $S$

Production rules $P$
Hyperedge replacement grammar

The left rule is applicable

Production rules $P$
Hyperedge replacement grammar

We obtain a hypergraph with three non-terminals

Production rules $P$
Apply the right rule to one of the non-terminals

Production rules $P$

Hyperedge replacement grammar
Two non-terminals remain:

Production rules $P$:
Hyperedge replacement grammar

Repeat the procedure until there is no non-terminal

Production rules $P$
Hyperedge replacement grammar

Repeat the procedure until there is no non-terminal

Production rules $P$
Repeat the procedure until there is no non-terminal

Production rules $P$
Hyperedge replacement grammar

Graph generation halts when there is no non-terminal
I will talk about an application of formal language to a graph generation problem

- **Formal language**
  - Context-free grammar
  - Hyperedge replacement grammar (HRG)
  - HRG inference algorithm

- **Application to molecular graph generation**
  - Molecular hypergraph grammar (a special case of HRG)
  - MHG inference algorithm
  - Combination with VAE
HRG inference algorithm outputs HRG that can reconstruct the input

**HRG inference algorithm** [Aguiñaga+, 16]

- **Input:** Set of hypergraphs $\mathcal{H}$

- **Output:** HRG such that $\mathcal{H} \subseteq \mathcal{L}(HRG)$
  
  Minimum requirement of the grammar’s expressiveness

- **Idea:** Infer production rules necessary to obtain each hypergraph
  
  Decompose each hypergraph into a set of production rules
Tree decomposition discovers a tree-like structure in a graph

- **Tree decomposition**
  - All the nodes and edges must be included in the tree
  - For each node, the tree nodes that contain it must be connected

* Digits represent the node correspondence
Tree decomposition and (a syntax tree of) HRG are equivalent

- **Relationship between tree decomposition and HRG**
  1. Connecting hypergraphs in tree recovers the original hypergraph
  2. Connection $\Leftrightarrow$ Hyperedge replacement
Tree decomposition and (a syntax tree of) HRG are equivalent

- **Relationship between tree decomposition and HRG**
  1. Connecting hypergraphs in tree recovers the original hypergraph
  2. Connection ⇔ Hyperedge replacement
Relationship between tree decomposition and HRG

1. Connecting hypergraphs in tree recovers the original hypergraph
2. Connection $\Leftrightarrow$ Hyperedge replacement
**Tree decomposition and (a syntax tree of) HRG are equivalent**

- **Relationship between tree decomposition and HRG**
  1. Connecting hypergraphs in tree recovers the original hypergraph
  2. Connection ⇔ Hyperedge replacement

```
Production rule = attach
```

![Diagram showing the relationship between tree decomposition and HRG](image)
HRG inference algorithm

HRG can be inferred from tree decompositions of input hypergraphs.

HRG inference algorithm [Aguiñaga+, 16]

- Algorithm:

  1. Compute tree decompositions of input hypergraphs
  2. Extract production rules
  3. Compose HRG by taking their union

- Expressiveness: $\mathcal{H} \subseteq \mathcal{L}(\text{HRG})$

  The resultant HRG can generate all input hypergraphs.
  (clear from its algorithm)
I will talk about an application of formal language to a graph generation problem

- **Formal language**
  - Context-free grammar
  - Hyperedge replacement grammar (HRG)
  - HRG inference algorithm

- **Application to molecular graph generation**
  - Molecular hypergraph grammar (a special case of HRG)
  - MHG inference algorithm
  - Combination with VAE
We want a graph grammar that guarantees hard constraints

- **Objective**
  
  Construct a graph grammar that *never* violates the valence condition

- **Application**: Generative model of a molecule
  
  - Grammar-based generation guarantees the valence condition
  
  - Probabilistic model could learn soft constraints
A simple application to molecular graphs doesn’t work

- **Input:** Molecular graphs
- **Issue:** Valence conditions can be violated

---

This rule increases the degree of carbon
Our idea is to use a hypergraph representation of a molecule

- **Conserved quantity**
  - **HRG:** # of nodes in a hyperedge
  - **Our grammar:** # of bonds connected to each atom (valence)
    - Atom should be modeled as a hyperedge

- **Molecular hypergraph**
  - Atom = hyperedge
  - Bond = node
A language for molecular hypergraphs consists of two properties

- **Molecular hypergraph as a language**

  A set of hypergraphs with the following properties:

  1. Each node has degree 2 (=2-regular)
  2. Label on a hyperedge determines # of nodes it has (= valence)
MHG, a grammar for the language, is defined as a subclass of HRG

- **Molecular Hypergraph Grammar (MHG)**
  - **Definition:** HRG that generates molecular hypergraphs only
  - **Counterexamples:**
    - **Valence** 😞
    - **2-regularity** 😞

This can be avoided by learning HRG from data [Aguiñaga+, 16]

Use an irredundant tree decomposition (our contribution)
I will talk about an application of formal language to a graph generation problem

- **Formal language**
  - Context-free grammar
  - Hyperedge replacement grammar (HRG)
  - HRG inference algorithm

- **Application to molecular graph generation**
  - Molecular hypergraph grammar (a special case of HRG)
  - MHG inference algorithm
  - Combination with VAE
A naive application of the existing algorithm doesn’t work

- **Naive application of the HRG inference algorithm** [Aguiñaga+, 16]

  - **Input**: Set of hypergraphs
  - **Output**: HRG w/ the following properties:
    - All the input hypergraphs are in the language 🙄
    - Guarantee the valence conditions 😞
    - No guarantee on 2-regularity 😢

This cannot be transformed into a molecular graph
Irredundant tree decomposition is a key to guarantee 2-regularity

- Irredundant tree decomposition
  - The connected subgraph induced by a node must be a path
  - Any tree decomposition can be made irredundant in poly-time
MHG inference algorithm

MHG inference algorithm is different from the existing one by two steps

- **MHG Inference algorithm** [Kajino, 19]
  - **Input:** Set of molecular graphs
  - **Output:** MHG w/ the following properties:
    - All the input hypergraphs are in the language 😊
    - Guarantee the valence conditions 😞
    - Guarantee 2-regularity 😕

Thanks to HRG

Our contribution

1. Convert molecular graphs into molecular hypergraphs
2. Compute tree decompositions of molecular hypergraphs
3. Convert each tree decomposition to be irredundant
4. Extract production rules
5. Compose MHG by taking their union
I will talk about an application of formal language to a graph generation problem

- **Formal language**
  - Context-free grammar
  - Hyperedge replacement grammar (HRG)
  - HRG inference algorithm

- **Application to molecular graph generation**
  - Molecular hypergraph grammar (a special case of HRG)
  - MHG inference algorithm
  - Combination with VAE
Combination with VAE

We obtain (Enc, Dec) between molecule and latent vector by combining MHG and RNN-VAE

- MHG-VAE: (Enc, Dec) between molecule & latent vector

![Diagram showing the combination of MHG and RNN-VAE encoders](attachment:image.png)
First, we learn (Enc, Dec) between a molecule and its vector representation using MHG-VAE

**Global molecular optimization** [Gómez-Bombarelli+, 16]

- **Find:** Molecule that maximizes the target
- **Method:** VAE+BO

1. Obtain MHG from the input molecules
2. Train RNN-VAE on syntax trees
3. Obtain vector representations \( \{z_n \in \mathbb{R}^D\}_{n=1}^N \)
   Some of which have target values \( \{y_n \in \mathbb{R}\} \)
4. BO gives us candidates \( \{z_m \in \mathbb{R}^D\}_{m=1}^M \) that may maximize
5. Decode them to obtain molecules \( \{G_m\}_{m=1}^M \)
Given vector representations and their target values, we use BO to obtain a vector that optimizes the target.

- **Global molecular optimization** [Gómez-Bombarelli+, 16]
  
  **Find:** Molecule that maximizes the target
  
  **Method:** VAE+BO
  
  1. Obtain MHG from the input molecules
  2. Train RNN-VAE on syntax trees
  3. Obtain vector representations \( \{z_n \in \mathbb{R}^D\}_{n=1}^N \). Some of which have target values \( \{y_n \in \mathbb{R}\} \)
  4. BO gives us candidates \( \{z_m \in \mathbb{R}^D\}_{m=1}^M \) that may maximize the target
  5. Decode them to obtain molecules \( \{G_m\}_{m=1}^M \)
We evaluate the benefit of our grammar-based representation, compared with existing ones

- **Empirical study**
  - **Purpose**: How much does our representation facilitate VAE training?
  - **Baselines**:
    - \{C,G,SD\}VAE use SMILES (text repr.)
    - JT-VAE assembles molecular components
      - It requires NNs other than VAE for scoring
  - **Tasks**:
    - VAE reconstruction
    - Valid prior ratio
    - Global molecular optimization

Image from [Jin+, 18]
Combination with VAE

Our grammar-based representation achieves better scores. This result empirically supports the effectiveness of our approach.

### Result

<table>
<thead>
<tr>
<th>Method</th>
<th>% Reconst.</th>
<th>Valid prior</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>50th</th>
<th>Top 50 Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVAE</td>
<td>44.6%</td>
<td>0.7%</td>
<td>1.98</td>
<td>1.42</td>
<td>1.19</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>GVAE</td>
<td>53.7%</td>
<td>7.2%</td>
<td>2.94</td>
<td>2.89</td>
<td>2.80</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>SD-VAE</td>
<td>76.2%</td>
<td>43.5%</td>
<td>4.04</td>
<td>3.50</td>
<td>2.96</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>JT-VAE</td>
<td>76.7%</td>
<td>100%</td>
<td>5.30</td>
<td>4.93</td>
<td>4.49</td>
<td>3.48</td>
<td>3.93</td>
</tr>
<tr>
<td>GCPN</td>
<td>–</td>
<td>–</td>
<td>7.98</td>
<td>7.85</td>
<td>7.80</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Ours</strong></td>
<td><strong>94.8%</strong></td>
<td><strong>100%</strong></td>
<td>5.56</td>
<td>5.40</td>
<td>5.34</td>
<td>4.12</td>
<td>4.49</td>
</tr>
</tbody>
</table>

\[
f(m) = \log P(m) - \text{SA}(m) - \text{cycle}(m)
\]

- **Penalty to a ring larger than six**
- **Synthetic accessibility score**
- **Water solubility**
A graph grammar can be a building block for a graph generative model

- Classify constraints into hard ones and soft ones
  - ML for the soft ones, rules for the hard ones

- Define a language by encoding hard constraints
  - E.g., valence conditions

- Design a grammar for the language
  - Sometime, w/ an inference algorithm

**Takeaways**

- Code is now public on Github
  - https://github.com/ibm-research-tokyo/graph_grammar
References


