Graph Transformation

Rasmus Nuko Jørgensen
Supervisor: Jakob Lykke Andersen

Bachelor Thesis
2022

Dansk resumé

Denne bachelorrapport er et studie i algebraisk graftransformation. Vi vil forklare den bagvedliggende teori bestående primært af kategoriteori, Double Pushout (DPO), og forskellige morfier. Vi vil bruge denne teori til at forklare vores egen Python implementation af DPO. Vi vil også definere og implementere grafgrammartikken Formose, og bruge den som et tilbagevendende eksempel på graftransformation gennem rapporten.

Vi vil teste og argumentere for vores implementations korrekthed. Gennem benchmarks kan vi stressteste vores implementation og indsamle data for dens køretid ved forskellige inputs. Vores mål er at finde frem til hvilke processer i vores graftransformationsimplementation der spiller den største rolle ift. køretiden. Dette opnår vha. køretidsanalyse og den indsamlede data fra vores benchmarks. Udfra resultaterne fundet i analysen, kom vi frem til størstedelen af køretiden stammer fra søgen efter delgrafisomorfier og isomorfier mellem grafer.

Teoretisk set virker implementation. Den kan fungere som en hjælp til en læser som vil forstå algebraisk graftransformation ved brug af DPO. I forhold til programmeringssprog som Go, Rust, C og C++ er Python meget langsomt. Hvis man skal bruge graftransformation som et reelt værktøj, foreslår vi at tage andre teknologier i brug.

Contents

1 Introduction

A graph can be an intuitive representation of a complex system in many cases. Graphs are widely used in computer science, software engineering, chemistry, and other scientific fields to model many systems.

In addition to nodes and edges, graphs can also have node- and edge labels. An example of labelled graphs is flow networks, which can be used to model and solve complex problems. Problems like the travelling salesman problem [Fin84]. In software engineering, UML diagrams can be used for automatic code generation [UN09].

Labelled graphs could be used to represent molecules, e.g. node labels for chemical symbols and labels for bond types for the edges between them. Moreover, a chemical system can be modelled with these graphs by performing graph transformations, i.e. this allows for the simulation of chemical reactions [And15].

Graph transformation can be used as a common framework to be able to create new graphs from existing graphs [Bal+99, p. 6]. Productions are constructed in order to define the desired behaviour of the transformations. A production specifies some precondition and some postcondition of the transformation. When performing the production on an input graph, we find the precondition as a subgraph in the input graph. We then replace this subgraph with the postcondition, giving us the output graph. This is a simple explanation, and we will see we are faced with some problems we need to solve. We will use the algebraic approach to graph transformation [GK06]. More precisely, the Double Pushout approach (DPO) has its roots in category theory.

A Formose reaction is a reaction discovered by Alexander Butlerov in 1861 [But61]. For example, a typical reaction might be the addition of formaldehyde to glycolaldehyde [SWS81]. The Formose reaction can be modelled as a graph grammar, called the Formose Grammar.

The goal of this thesis is to introduce graph transformations using DPO, use this knowledge to explain our Python implementation of DPO and the Formose grammar, analyse the running time of the implementations, and through discussion of the findings from the analysis, come to a conclusion on the most contributing factors to the overall running time.

2 Theory

This section introduces the theoretical subjects of interest when understanding graph transformation. More precisely, algebraic graph transformation using the double pushout approach.

If an instance of a structure, e.g. a set, can be defined as an object S in a category C, we can reason about the relations between S and other objects of C using morphisms. [Awo10]. Category theory is interesting regarding graph transformation, as it allows us to reason about the relationship between graphs using graph morphisms.

We introduce four types of morphisms: monomorphism, monomorphisms,

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[rr, bend left, "g\circ f"] \arrow[rrr, bend right, "h\circ(g\circ f)=(h\circ g)\circ f"'] & B \arrow[r, "g"] \arrow[rr, bend left, "h\circ g"] & C \arrow[r, "h"] & D
\end{tikzcd}$$

Figure 2.1: $\mathcal{A}$ consists of A, B, C, D and $\mathcal{M}$ consists of $f\colon A \to B,\ g\colon B \to C,\ h\colon C \to D, \dots$. Notice the two compositions described under the arc going from A to D. Let's call them $x\colon h \circ (g \circ h)$ and $y\colon (h \circ g) \circ h$. Due to associativity of the composition of morphisms, x and y are equivalent morphisms.

subgraph isomorphisms and isomorphisms. *Subgraph isomorphisms* are the mappings used in Double Pushout (DPO). Isomorphisms will be used later when we look at chained graph transformation. Isomorphisms are used to make sure we only keep one of multiple isomorphic graphs.

2.1 Category Theory

This section introduces categories, objects, and morphisms, as these are needed to explain the algebraic approach to graph transformation. The main sources for the theory put forth in this section have been [Awo10; Mit65]. Steve Awodey refers to category theory as "the abstract algebra of functions"; a function being any assignment, procedure, process, or expression which can be read in a functional way [Awo14].

A *category* C is defined as a collection of objects $\mathcal{A}$, and a collection of morphisms $\mathcal{M}$. For a morphism $f\colon A \to B$, we call A the domain, and B the co-domain, where $A, B \in \mathcal{A}$.

The morphisms are closed under composition. $f\colon A \to B,\ g : B \to C$ can be composed as $g \circ f\colon A \to C$. This goes for any morphisms f, g, where the co-domain of f is the domain of g.

There are two axioms of categories [Mit65, Chapter 1]. The first one being associativity of compositions. Whenever we can compose three morphisms, it must hold $\gamma \circ (\beta \circ \alpha) = (\gamma \circ \beta) \circ \alpha$.

When depicting categories, we use commutative diagrams, like the one seen in Figure 2.1.

The second axiom of categories is the existence of identities. An identity is defined as: $1_A\colon A \to A,\ \forall A \in \mathcal{A}$. Both axioms are represented in Figure 2.2.

We have the following requirements for categories: objects, morphisms with objects as their domain and co-domain, composition as an operator to perform on the morphisms, associativity of the compositions, and the existence of identity morphisms. Anything that adheres to this definition can be considered as a *category*.

$$\begin{tikzcd}
A \arrow[r, "1_A"] \arrow[dr, "f\circ 1_A"'] & A \arrow[d, "f"] \arrow[dr, "1_B\circ f"] & \\
& B \arrow[r, "1_B"'] & B
\end{tikzcd}$$

Figure 2.2: A morphism $f\colon A \to B$, shown in figure as the vertical arc. The two horizontal arcs are the identity morphisms, $1_A, 1_B$ for A and B respectively. The diagonal arcs are the composition of an identity and f. Notice $f \circ 1_a = 1_B \circ f = f$. The reason 1_A is on the RHS of the composition $f \circ 1_a$, and 1_B is on the LFS of the composition $1_B \circ f$, is because they are the domain and co-domain of $f\colon A \to B$ respectively. Figure is from [Awo10, p. 4].

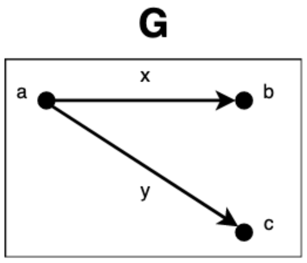


Figure 2.3: A graph G with nodes $V_G = \{a, b, c\}$ and edges $E_G = \{x, y\}$. Using morphisms $s_G\colon E_G \to V_G$ to map source nodes, we get $s_G(x) = a,\ s_G(y) = a$ since a is the source node of both edges. For the morphism $t_G\colon E_G \to V_G$ we get $t_G(x) = b,\ t_G(y) = c$ as the two target of edges x, y.

Graphs as categories Looking at directed graphs as categories, we have two objects and two morphisms from each graph. The objects are the nodes V and the edges E. The morphisms are $s\colon E \to V$ and $t\colon E \to V$, where s and t is used to map edge onto the nodes. s and t map an edge to the source- and target node, respectively. Please see Figure 2.3. For undirected graphs, we can still use s and t, but we have both directions $(u, v), (v, u)$ in the set of edges. [GK06, Chapter 2].

We can use this new knowledge, to reason about morphisms between graphs. If we have two graph $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$, and two morphisms $f_V\colon V_1 \to V_2$ and $f_E\colon E_1 \to E_2$. We map nodes of G_1 to nodes in G_2 using f_V, and edges using f_E. This is visualized in Figure 2.4.

A graph morphism $f\colon G_1 \to G_2$ consist of two morphisms $f = (f_V, f_e)$, where $f_V\colon V_1 \to V_2$ and $f_E\colon E_1 \to E_2$. The morphism f_V and and f_E preserve the source and target functions, that is $f_V \circ s_1 = s_e \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$.

As stated earlier, for a category to be well defined it has closed under composition. The proof for this has been found in [GK06, Chapter 2]. Given three graphs $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2, 3$ and two graph morphisms $f = (f_E, f_V)\colon G_1 \to G_2$, $g = (g_E, g_V)\colon G_2 \to G_3$. Using the associativity

$$\begin{tikzcd}
E_1 \arrow[r, shift left, "s_1"] \arrow[r, shift right, "t_1"'] \arrow[d, "f_E"'] \arrow[dr, phantom, "="] & V_1 \arrow[d, "f_V"] \\
E_2 \arrow[r, shift left, "s_2"] \arrow[r, shift right, "t_2"'] & V_2
\end{tikzcd}$$

Figure 2.4: E_1, E_2 and V_1, V_2 are the edges and nodes of G_1, G_2 respectively. The morphisms s_1, s_2 map source nodes of the edges in G_1, G_2, and t maps target nodes of the edges. f_V/f_E maps the nodes/edges of G_1 to nodes/edges in G_2. Figure from [GK06, Chapter 2].

of composition of morphisms, and knowing f and g preserve the source and target functions, we can conclude the following:

$$g_V \circ f_V \circ s_1 = g_V \circ s_2 \circ f_E = s_3 \circ g_E \circ f_E \tag{1}$$

$$g_V \circ f_V \circ t_1 = g_V \circ t_2 \circ f_E = t_3 \circ g_E \circ f_E \tag{2}$$

This shows the source (1) and target (2) functions are preserved when performing the composition $g \circ f$.

A identity morphism, will also preserve the source and target functions, as all nodes or edges will be mapped to themselves.

We have now shown graphs meet the definition of a category. They have objects (nodes and vertices), morphisms (source and target function), identity morphisms exist, that the composition of morphisms is well defined and have associativity as a property.

Labelled graphs There is a type of graph that has labels on the nodes and edges. In category theory we can represent labelled graphs by introducing two object L_V, L_E, and extending the formulation for directed graphs $G = (V, E, s, t)$, to $G = (V, E, s, t, l_V, v_E)$. L_V and L_E are the node alphabet and edge alphabet, respectively. l_V and l_E are the mappings from the nodes/edges in the labelled graph to the node-/edge alphabets.[GK06, Definitions 2.6, 2.8]. A commutative diagram for labelled graph morphisms shown in Figure 2.5.

Types of graph morphisms Homomorphisms, monomorphisms, subgraph isomorphisms and isomorphisms will now be introduced. The primary source for the definitions is [And15, Chapter 3].

In the following definitions, V_X is the set of nodes, E_X is the set of edges of graph X. m is a *graph homomorphism* if $e = (u, v) \in E_G \Rightarrow (m(u), m(v)) \in E_H$.

- m is a *graph monomorphism* if it is injective, that is
  $\forall u, v \in V_G,\ u \neq v \Rightarrow m(u) \neq m(v)$.

$$\begin{tikzcd}
 & & E_1 \arrow[rr, shift left, "s_1"] \arrow[rr, shift right, "t_1"'] \arrow[dll, "l_{1,E}"'] \arrow[dd, "f_E"] & & V_1 \arrow[drr, "l_{1,V}"] \arrow[dd, "f_V"] & & \\
L_E & = & & = & & = & L_V \\
 & & E_2 \arrow[rr, shift left, "s_2"] \arrow[rr, shift right, "t_2"'] \arrow[ull, "l_{2,E}"] & & V_2 \arrow[urr, "l_{2,V}"'] & &
\end{tikzcd}$$

Figure 2.5: This figures is an extension of Figure 2.4. Here the node alphabet L_V and edge alphabet L_E have been added. Notice the nodes and edges from G_1, G_2 are all mapped to their respective alphabets. The fundamental structure of the graph morphisms has not been changed, but new elements have been added. Figure from [GK06, p. 24].

- m is a *subgraph isomorphism* if it is a graph monomorphism, and $(u, v) \in E_G \Leftrightarrow (m(u), m(v)) \in E_H$.
- m is an *isomorphism* when m is a subgraph isomorphism, and is a bijection of the nodes. This means $\forall x \in V_G, \exists! y \in V_H$ such that $m(x) = y$, and $\forall y \in V_H, \exists! x \in V_G$ such that $y = m(x)$.

2.2 Algebraic Graph Transformation

The main idea of graph transformation is to have a source graph G and a production $p = (L, R)$. We create the target graph H by finding L in G and replacing L in G with R.

This is a straightforward explanation, and it leaves us with three fundamental questions we need to answer.

First off, how do we find L as a subgraph in G. This is the problem known as subgraph matching, and we give a short description of this in Appendix B.

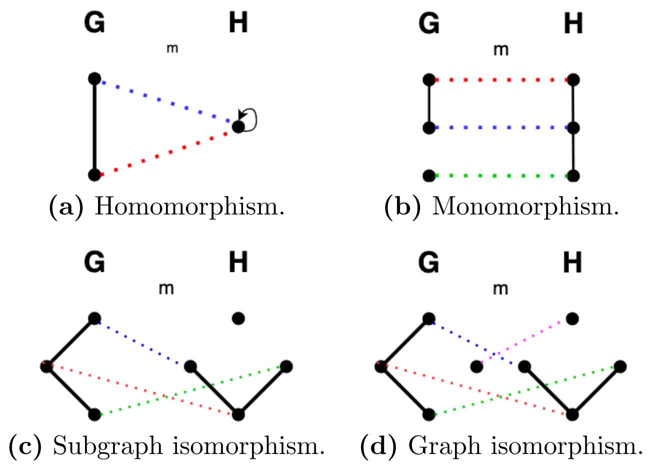
The subsequent two problems we face are: deleting L in G after finding L using subgraph matching and connecting R with the context of the target graph H. In this thesis, we use the double pushout approach described in subsection 2.3 for these purposes.

Pushouts “The algebraic approach is based on pushout constrictions where pushouts are used to model the gluing of graphs” [GK06, Section 1.2]. We will explain pushouts based on [GK06, Definition 2.16].

Given a category C and two morphisms $f\colon A \to B$, $g\colon A \to C$, a *pushout* $(D.f', g')$ over f and g is defined by:

- A pushout object D.
- The morphisms $f'\colon C \to D$ and $g'\colon B \to D$, where $f' \circ g = g' \circ f$.

For all objects X and morphisms $h\colon B \to X$ and $k\colon C \to X$, where $k \circ g = h \circ f$, there is a unique morphism $x\colon D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.



(a) Homomorphism. **(b)** Monomorphism. **(c)** Subgraph isomorphism. **(d)** Graph isomorphism.

Figure 2.6: The coloured, dotted lines show the mapping of nodes in G to H, using $m\colon G \to H$. Notice in **(a)** both nodes in G are mapped to the same node in H. This is possible since homomorphisms are not injective. In **(b)**, even though there is not an edge between the two bottom nodes in G, they can still be mapped to the bottom two nodes of H, since the definition states that only edges in G also have to be present in H. In **(c)** all nodes of G can be mapped to a subgraph of H, as mapping with all the nodes and respective edges can be mapped to H. In **(d)** we can map the entirety of G to the entirety H. Be aware that the mappings shown are not the only possible mappings.

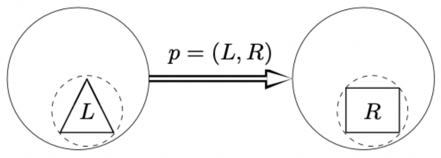


Figure 2.7: The left circle is the source graph G, and the right circle is the target graph H. We find L in G, replace L with R, resulting in H. Figure from [GK06, Figure 1.1]

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"] \arrow[rd, phantom, "="] & B \arrow[d, "g'"] \arrow[rdd, "h"] & \\
C \arrow[r, "f'"] \arrow[rrd, "k"] & D \arrow[rd, "x"] & \\
 & & X
\end{tikzcd}$$

Figure 2.8: For a category with two morphisms f and g that share the same domain A, if the co-domains B and C both are domains in a morphism (B in g', and C for f') that go to the same co-domain D, then this object D is a pushout object. There exists an unique morphism x, with the pushout object D as the domain, to any object X, if X is the co-domain of two morphisms with B and C as the domains (f' and g'). Please notice $x \circ f' = k$ and $x \circ g' = h$, as stated in the definition. Figure from [GK06, p. 29].

In [GK06, p. 29], it is stated that we get pushout objects by glueing two objects along a common subobject. Now, what does this mean?

We can explain this by looking at the second part of graph transformation, where we want to glue two graphs together. Using the objects and morphisms from Figure 2.8, we can think of A as a common subgraph that is present in both B and C. If we wanted to glue B onto C, assuming f and g are subgraph isomorphisms, we can know where nodes from B should be glued onto nodes from C. We know this from using the mappings from their common subgraph A, as all nodes in A are mapped to nodes in B and C via f and g, respectively.

All nodes in C that are mapped using g are where we want to glue the nodes in B that are mapped using f. All nodes in B that are not mapped from A to B via f, are the nodes that need to be added to the output graph (pushout object) D.

2.3 Double Pushout (DPO)

In Figure 2.7 we show a simple example of a graph transformation using a production $p = (L, R)$, and in Section 2.2 we also explain that there are two problems with this simple explanation. This simple explanation did not tell us how we would glue the graphs together to achieve the target graph. DPO accomplishes this.

The main source for the theory presented in this subsection comes from [GK06, pp. 11, 12].

Where the simple explanation in Section 2.2 used a production $p = (L, R)$, DPO uses productions with three graphs and two morphisms $p = (L \overset{\mathrm{l}}{\leftarrow} K \overset{\mathrm{r}}{\rightarrow} R)$. This new graph K is a common interface between L and R, which is in fact the intersection of L and R. We still have L as the as the precondition, and R as

$$\begin{tikzcd}
L \arrow[d, "m"'] & K \arrow[l, hook', "l"'] \arrow[r, "r"] \arrow[d, "d"'] & R \arrow[d, "m'"'] \\
G & D \arrow[l, hook', "l'"] \arrow[r, "r'"'] & H
\end{tikzcd}$$

Figure 2.9: This is the commutative diagram of DPO. All of the morphisms are subgraph isomorphisms. As presented in this section, K is the intersection of L and R, and we see that it is subgraph isomorphic to L, R and D. In graph transformation, we want to find L as a subgraph in G, and we see L is subgraph isomorphic to G in this figure. Likewise, R is subgraph isomorphic to H, since the output graph contains all of R after the graph transformation. The 'intermediate' graph D is subgraph isomorphic to both G and H because D is contained in both G and H. D is the part of G that need to be kept after removing L from G. D is the part of H that needs to be glued with R.

the postcondition of the production p.

In DPO, we use six graphs L, K, R, G, D, H. We have already introduced L, K, R, and we introduce the last three graphs before getting further into the theory of DPO. G is the input graph, which is the graph being transformed, i.e. the graph the production is being performed on. D can be thought of as an intermediate graph, where G has been used as a basis, and the parts of L that need to be removed from G have been removed, and the parts of R that need to be added have not been added yet. Think of this as the graph we glue R onto to get the output graph H. H is the output graph, which is the resulting graph after performing the production $p = (L, K, R)$ on G.

All of the morphisms in DPO are subgraph isomorphisms, meaning every node in the domains are mapped to nodes in the co-domain. Figure 2.9 shows the subgraph isomorphisms between the six graphs.

Comparing Figure 2.9 to Figure 2.8, (K, L, D, G) and (K, R, D, H) in Figure 2.9 correspond to (A, B, C, D) in Figure 2.8. This is where DPO gets the name from. In (K, L, D, G) G is the gluing of graphs L and D. In (K, R, D, H) H is the gluing of graphs R and D. In both pushouts, the common subgraph is K.

When performing a DPO graph transformation, we need an input graph G, a production $p = (L, K, R)$, and a subgraph isomorphism $m \colon L \to G$. The transformation is performed in two steps: the removal step and the glueing step.

In the removal step, nodes and edges in $K \setminus L$ are removed from G, resulting in the graph D. The set of nodes and edges in $K \setminus L$ is not necessarily a legal graph, but the construction $D := ((G \setminus m(L)) \cup m \circ l(K))$, must not be any dangling edges. *Dangling edges* are edges that only are connected to one node. Without dangling edges, we can reconstruct G, by glueing $L \setminus K$ and D together.

In the glueing step, we construct the output graph H by glueing $R \setminus K$ to D. Since D contains K, and R contains K, the nodes and edges of R not already

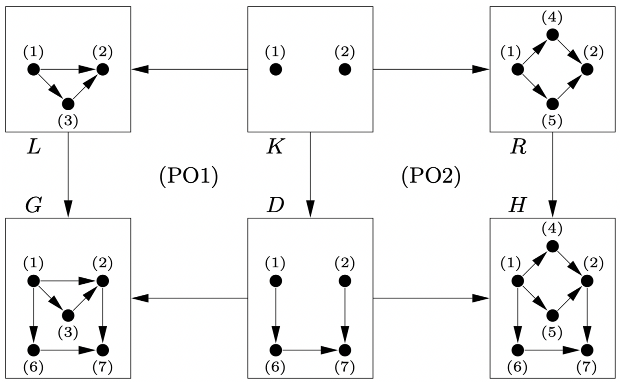


Figure 2.10: The caption from Figure 2.9 explains the different subgraph isomorphisms, so it will be left out for this figure. We see that K is the intersection of L and R. Notice D consists $G \setminus (L \setminus K)$, in this case G without node 3 and it's edges, and the edge $(1, 2)$. G can be constructed by gluing L onto D, in this figure nodes $1, 2$ from L are glued onto $1, 2$ in D and the node 3 is added, alongside all edges present in L. H is constructed by gluing R onto D, by gluing the nodes $1, 2$ in R on to the nodes $1, 2$ in D, the nodes $4, 5$ alongside all edges present in R are also added. Figure from [GK06, p. 13].

present in D is added when gluing $R \setminus K$ onto G. Using the mappings r and d, we know where the nodes of R have to be mapped onto D. The mapping is also used for the edges in R that have one node in K and the other node in $R \setminus K$, to ensure they are connected properly.

For a commutative diagram for DPO containing graphs as object can be seen in Figure 2.10.

Labelled graphs In the implementation, we would like to be able to perform DPO graphs transformations on labelled graphs. This opens up the opportunity to perform graph transformations on molecules in skeletal structure. For a commutative diagram for DPO containing molecules as objects, please see Figure A.2.

For this section, [And15, Section 6.3] has been used as the primary source. Transformation of labelled graphs using DPO might not only result in structural changes of the graphs, which is what has been described thus far. For the production $p = (L, K, R)$ to affect the node- and edge labels, we define Ω as the alphabet L and R are labelled over. The label for K consists of an ordered pair from the Cartesian product $\Omega \times \Omega$. Suppose we have a node u in K with the label $\langle x, y \rangle$, x is the label of the node u maps to in L, and y is the label of the node u maps to in R. We use the function $fst\colon \Omega \times \Omega \to \Omega$ to retrieve the first element in the ordered pair, and $snd\colon \Omega \times \Omega \to \Omega$ for the second element, i.e. $fst(\langle x, y \rangle) = x$, $snd(\rangle x, y \langle) = y$. Please see Figure 2.11 for a visual

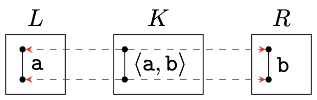


Figure 2.11: In this figure, we have a production $p = (L, K, R)$ where the edges of the graphs are labelled. Notice that the edge in L has the label a, and the edge in R has b as its label. This is because the label for the edge in K has a as the left element in the ordered pair, since this is the edge label from L. The same goes from the right element in the ordered pair, which comes from the edge label in R. Figure from [And15, Figure. 6.8].

representation.

The intermediate graph D, just like K, has ordered pairs as labels. For D the function $fst\colon \Omega \times \Omega \to \Omega$ takes a label (ordered pair) in D, and returns the label for the appropriate node in G. The same goes for the function *snd*, but for the label of the appropriate node in H.

3 Implementation of Graph Transformation

We have implemented DPO graph transformation using Python and NetworkX. The main goal has been to arrive at a working, and not necessarily efficient implementation. We see Python as the programming language we are most proficient in, which is why Python has been chosen.

NetworkX allows us to create and manipulate directed and undirected, labelled and unlabelled graphs. Other than graph creation/manipulation, NetworkX has many useful features built-in, such as subgraph matching. We know this is a crucial part of graph transformation. A small introduction to the subgraph matching algorithm used, VF2 Algorithm, can be found in the appendix Section B.

Python and NetworkX are far from being efficient compared other programming languages/libraries that could have been utilized. The syntax of Python and NetworkX should, hopefully, make it intuitive to follow along when code examples are presented throughout this section.

We assume the reader has read the previous sections. Therefore, we will continue using the names L, K, R, G, D, H for graphs, and the names for mappings as seen in Figure 2.9. All code can the found in the folder */code/*, located in the attached zip file.

3.1 Implementing DPO in Python

This section aims to present how we have chosen to implement the main parts of DPO. We will not explain every minute detail of the implementation. Instead, we will explain how we chose to implement productions, the construction of D by removing $L \setminus K$ in G, and the construction of H by adding $L \setminus K$ to D.

We assume the reader has a basic understanding of standard Python operations and object-oriented programming in Python.

When referencing NetworkX graph objects, lists and dictionaries, it will be written in the font: `some_object`. While graphs and morphisms introduced in Section 2 will be referenced using mathematical notation: G or l. We may mix code examples with mathematical notation where it makes sense, i.e. `I_edge`$=E_L \cap E_R$. This means `I_edge` is the intersection of the edges of L and R.

Production Production has been chosen to be the turning point for the entire implementation. In the file */code/production.py* the class `Production` is defined. Working with graph transformation in our implementation is a matter of specifying a `Production` and performing it on an input graph `G`, using the mapping m that maps nodes in the LHS `L` of the production to nodes in the input graph `G`.

The subgraph isomorphisms m, as it is called throughout Section 2, is called `LG_mapping` in the code. This goes for all mappings in the code. Their name has been changed to `XY_mapping`, where `X` is the domain-, and `Y` is the co-domain of the mapping.

In Section 2.3 we needed to provide three graphs L, K, R and two morphisms l, r in order to define a DPO production $p = (L \overset{\mathrm{l}}{\leftarrow} K \overset{\mathrm{r}}{\rightarrow} R)$. Since K is the intersection of L, R, we decided that K could be computed, removing the need to specify K for each production. This is done in the function `intersection(self, A, B, AB_mapping)`, where the NetworkX graph `A` is copied into a new graph `I`. We proceed to remove every node and edge that is not shared between `A` and `B`. This results in the `I` being a legal graphs consisting of the shared nodes/edges in `A` and `B`, i.e. `I` is the intersection of `A` and `B`. When using the function `intersection` to find graph K, `A`$= L$,`B`$= R$,`I`$= K$.

`I` uses the same name for indices as `A` used, and the mapping `KL_mapping` is therefore a dictionary `{'v': 'v', 'u': 'u'}`, if `v` and `u` are the nodes of `A` in the intersection between `A` and `B`.

The mapping `KR_mapping`, (r), is a dictionary
`{'v': LR_mapping('v'), 'u': LR_mapping('u')}`, since all nodes in the K uses the index names from L, and all nodes in the intersection map to nodes in R. If no mapping for `LR_mapping` is specified, the implementation defaults to map nodes with the same index name together, i.e. $V_L = \{v, u, s\}$, $V_R = \{v, u, t\}$, `LR_mapping = {'v': 'v', 'u': 'u'}`. Notice nodes that are not mapped from L to R and not kept in the mapping `LR_mapping`.

The implementation deviates from the theory when the graphs are labelled. When looking at Figure 2.11, K has labels containing ordered pairs. In the implementation, K only has a node/edge if the labels are the same across L and R.

We have now found graph K for the production, given graphs L, R, and a mapping (`LR_mapping`) from a subset of nodes in L to a subset of nodes in R. In a object of type `Production`, we store L, R, K in `self.L` and `self.R` respectively. And the mappings l, r in `self.KL_mapping` and `self.KR_mapping` respectively. The inverse mappings of l, r are stored in `self.LK_mapping`, `RK_mapping`. These mappings will be used later when constructing the output graph H.

When creating an object of type `Production`, we make sure that the `LR_mapping` is a valid mapping. This is done by asserting: that all nodes in L map to at most one node in R, all entries in the mapping map to unique nodes, and all nodes are present in their respective graphs.

Mapping L onto G In this part we utilize different objects and functions provided by NetworkX in `networkx.algorithms.isomorphism`, namely the type of object `GraphMatcher` and the functions `subgraph_isomorphisms_iter` for finding subgraph isomorphisms, `categorical_node_match` and `categorical_edge_match` for specifying what node/edge attributes the `GraphMatcher` should compare in order to include/exclude matches. Our implementation uses the attribute name `attr` for both node- and edge labels.

We provide `GraphMatcher` with four arguments, the graphs G and L, and the functions mentioned for node- and edge matching. We want to match nodes and edges on the label `'attr'`, an attribute name we chose ourselves. `subgraph_isomorphisms_iter` finds all mappings of L as subgraphs in G. The mappings we get are from G to L. A function `inverse_mapping` is used to inverse the mappings, so we have them going from L to G.

Constructing graph D The next two steps, constructing graph D and constructing graph H, will both take place in the function `perform_transformation`. This function takes a NetworkX graph `G` as the input graph, and a valid `LG_mapping`. `perform_transformation` returns the output graph `H` if the production is legal, otherwise `None` is returned.

We find $L \setminus K$ using the function `diff`. This function takes two NetworkX graph `A`, `B`, and the mapping `AB_mapping`. `diff` returns two dictionaries `diff_nodes` and `diff_edges`, where `diff_nodes` $= V_A \setminus V_B$ and `diff_edges` $= E_A \setminus E_B$. The keys of the dictionaries are the nodes/edges, and the values are the labels associated to those nodes/edges.

Using `diff` and passing the two NetworkX graphs `L` and `K`, and the mapping `LK_mapping`, we get `LK_diff_nodes` and `LK_diff_edges` as the nodes and edges present in L that are not present K.

If nodes and edges have labels, we also include nodes and edges in `LK_diff_nodes` and `LK_diff_edges` if their labels do not match. To achieve this, we can break the procedure into two parts. *1)* Creating two dictionaries using the functions `get_node_attributes` and `get_edge_attributes` from NetworkX, and specifying the name of the attributes as `'attr'`. *2)* Remove all nodes/edges that map from L to K if they have the same labels in both L and K.

Before removing nodes from G to work our way towards D, we need to make sure no nodes are removed that cause dangling conditions in G. We wrote the function `dangling_nodes` for this purpose. This function takes `L, G, LR_mapping` as arguments and returns a list of nodes that would cause dangling conditions if they were to be removed. 'Removed' in this context means that the node does not exist in `LR_mapping`. We use the `diff` function to find all edges in G that are not in L. If any of these edges (u, v) have a node u, which is mapped to L, but is not mapped from L to R, u is added to the list of nodes that would cause dangling conditions. The removal of u would leave the edge (u, v) dangling. We return this list of nodes as `dangling_nodes`.

We now have the `LK_diff_nodes` $= V_L \setminus V_K$, `LK_diff_edges` $= E_L \setminus E_L$, and `dangling_nodes`. We start off by removing the edges in `LK_diff_edges` from G, as removing nodes in a NetworkX graph will remove all edges to and from that node. This is to make sure we do not get an error trying to remove a node that has already been removed. We remove the nodes in `LK_diff_nodes` and ensure no nodes in `dangling_nodes` are removed. If we try to remove a node in `dangling_nodes`, we return `None`, as this is an illegal graph transformation according to DPO.

Here the implementation deviates from the theory. In the implementation, D only has a node/edge if both labels are the same. In the theory, D would have an ordered pair $\langle x, y\rangle$ as its labels, where x is the label in G and y is the label in H. This means that nodes/edges that change label are removed when constructing D and added back with the new label when adding nodes/edges to construct H.

Constructing graph H The graph D has been constructed, and the gluing of $R \setminus K$ onto D is the next step. Similarly to when we had to find $L \setminus K$, we use the function `diff` to find `RK_diff_nodes` $= V_R \setminus V_K$ and `RK_diff_edges` $= E_R \setminus E_K$. If the graphs are labelled, we make sure to copy node/edge labels from R, when adding nodes/edges to D.

For adding the nodes, we add every node in `RK_diff_nodes` to D. Afterwards, we have all the nodes that need to be in H.

We construct the mappings `GLR_mapping` and `RLG_mapping` using the mappings `LR_mapping` and `GL_mapping`. `GLR_mapping` are the nodes in G map to in R, via L. `RLG_mapping` is the inverse mapping of `GLR_mapping`.

When adding an edge (u, v) from `RK_diff_edges`, if either u or v is in not in `RK_diff_nodes`, we map that node via `RLG_mapping`. All nodes in this mapping is where R and D overlap, and should be glued accordingly. Suppose u is not in `RK_diff_nodes` but v is, u is in D and v is a newly added node. The edge (`RLG_mapping[u]`$, v$) is added, where `RLG_mapping` makes sure the edge is connect properly. If both u, v are in `RK_diff_nodes`, u, v are newly added nodes, and the edge (u, v) should not be connected to D. The reason `RLG_mapping` works for nodes mapped from R to D, is because D has been constructed from G. D is just a subset of G.

Suppose a node v that maps to t in `LR_mapping` does not have to same label in L and R. v will not be in the K. Thus v will be in `LK_diff_nodes`. Therefore v will be removed from G when constructing D, and t is added with it's new label (from R) when adding nodes from `RK_diff_nodes`. Because the original node v in L was removed from G, the edges connected to that node have been removed. When glueing graphs together, we must reconnect these edges to the new nodes added from `RK_diff_nodes`.

When reconnecting these edges, we go through the edges of G. Suppose an edge (u, v) from G, where u in `GLR_mapping`, v not in `GLR_mapping`, and `GLR_mapping[u]` in `RK_diff_nodes`. Originally (u, v) was an edge in G, but u was removed when constructing D. To make up for this, we add the edge back as (`GLR_mapping[u]`, v) along with it's label. Vice versa for an edge (u, v) where v is removed. In this case we add the edge (u, `GLR_mapping[v]`).

We now have the output graph H. Nodes and edges have been removed by removing the `LK_diff_nodes` and `LK_diff_edges`. Nodes and edges from `RK_diff_nodes` have been added. Edges from `RK_diff_edges` have been added and glued onto D when needed.

When chaining graph transformation, as in Section 3.2, H is used as the input graph G. Since common index names (names of nodes) between G and R can cause the implementation to malfunction. For this reason all nodes in H are relabelled to combat this behaviour. This is achieved by using the function `nx.relabel_nodes`, where all index names are appended the name of the production. Be aware 'relabel' in the context of `nx.relabel_nodes` does not change the node/edge labels, but just changes the index names of the nodes, e.g. a node `'u'` in a NetworkX graph `G` is accessed as `G.nodes['u']`, if we relabel this node to `'v'` with `G = nx.relabel_nodes(G, {'u': 'v'})`, it would be accessed as `G.nodes['v']`.

3.2 Formose Grammar

As seen in Figure A.2, molecules can be represented as graphs when the molecules are in skeletal structure. We can construct a production p such that it mimics a chemical reaction on one or more molecules.

The Formose reaction converts formaldehyde into sugars. The chemistry of Formose can be described as a graph grammar, known as the Formose grammar [And15, Chapter 12]. The Formose reaction was first described by Alexander Butlerov in 1861 [But61]. This thesis will not go into great detail about the chemical aspect of Formose reactions. However, the article [Ben+10] might be of interest when looking for more information about the Formose grammar.

The Formose grammar has been implemented as it provides a known set of productions we can perform with our implementation. It will also be used to benchmark the implementation in Section 4.3.

We were told about the Formose grammar by our supervisor Jakob Lykke Andersen from IMADA at SDU Odense. Be aware that this section does not simulate a Formose grammar, as described in [And15, Chapter 12]. A simulation of the real Formose grammar can be found by visiting [And20a] and loading the 'Repetition' example.

The real Formose grammar has one formaldehyde, one glycolaldehyde, and adds a glycolaldehyde molecule at each iteration. Where each iteration performs all possible productions from the Formose grammar [And15, Chapter 12]. The Formose grammar in MØD is based on [Ben+10, Figure 9].

Our Formose grammar uses a graph with fixed number n of glycolaldehyde molecules and m formaldehyde molecules. We perform the possible productions on this starting graph, and keep track new (non-isomorphic) graphs. All non-isomorphic graphs get added to a list of graphs. We then perform the four productions on all graphs in the list. All graphs that are non-isomorphic to the entire list of earlier graphs are added to the list. This procedure is repeated until no new graphs are found.

The language of our Formose grammar is a finite set of graphs, as no new atoms are added. In contrast, the language of the real Formose grammar is infinite.

The simple explanation for the language of our Formose grammar being finite is that no new atoms are added. Therefore, the atoms (nodes of the graph) can only be connected in a finite amount of ways.

Formose Grammar Implementation The four productions of the Formose grammar have been constructed in the file */code/formose/formose_grammar.py*. In this file the function `generate_start_graphs()` uses the Python library PySMILES to load n glycolaldehyde and m formaldehyde molecules graphs into a single graph. We will refer to this graph as the starting graph. The starting graph is tweaked to fit our labelling convention.

In the function `perform_formose(n_glyco, m_formal)` from */code/formose/formose.py* the following steps are done:

1. Generate starting graph from `n_glyco` and `m_formal`, and add it to the list `generated_graphs`.
2. Perform all four productions on the graph(s) in `generated_graphs` and store the output graphs in `new_graphs`.
3. Some graphs in `new_graph` might be isomorphic with one another. When we have two or more isomorphic graphs, we only keep one of them. We store all non-isomorphic graphs in the list `new_non_iso`.
4. When then remove all graph from `new_non_iso` that are isomorphic to a graph in `generated_graphs`.
5. We add the graphs from `new_non_iso` to `generated_graphs`. Repeat step 2. to 5. until no new graphs are added in step 5.

During step 3., we reduce the number of isomorphism checks by only checking for isomorphism between two graphs G_i, G_j if $i < j$. Where i, j can be considered as indices in the list `new_graphs`. This still ensures that all graphs are checked against each other, but only once. If this was not in place, we would check all pairs of graphs against each other twice, e.g. G_1, G_3 and G_3, G_1.

In the isomorphism checks we use `networkx.algorithms.isomorphism`, where the function `is_isomorphic` takes two graphs, `categorical_node_match`, and `categorical_edge_match`. Node and edge label matching is set to check the attribute `'attr'`.

`perform_formose` prints the following information after execution: Total running time, total number of iterations, total number of graphs, total number of non-isomorphic graphs, the ratio of non-isomorphic to the total number of graphs, time per graph, time per non-isomorphic graph, total number of glycolaldehyde, total number of formaldehyde, total atom count.

Be aware that 'total number of graphs' is not the total number of graphs that could have been generated. This is because we prune graphs with the isomorphism checks for each iteration.

In */code/formose/formose.py* there is a variable `silent`. If `silent=False`, the number of non-isomorphic graphs generated and running time is printed for each iteration.

Our implementation of the Formose grammar has been tested using results from MØD in Section4.1.

3.3 How to use the implementation

To perform a graph transformation using this implementation, the required Python libraries are NetworkX, Matplotlib, and PySMILES. The latter is only needed if one wishes to load SMITES strings as a graph that follows the implementation's conventions.

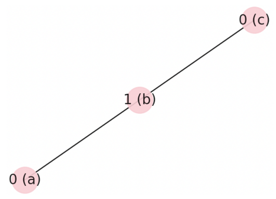
At first, one will need to construct a `Production`. For this two graphs, commonly named `L` and `R`, is need. As the names suggests, graph L and R in a DPO production $p = (L, K, R)$. As explained in Section 3, K and the morphisms l, r does not need to be provided. If no mapping from nodes in L to nodes in R is provided, the nodes are mapped according to their index names, e.g. $V_L = \{1, 2, 3, 4\}$, $V_R = \{2, 3\}$ would generate `LR_mapping = {2: 2, 3: 3}`.

An input graph of type `nx.Graph`, commonly named `G`, has to be constructed. If chained graph transformations are desired like shown in Section 3.2, be sure not to use index names in `R` that already have been used in `G`. That is, do not `R.add_node('x')` if `'x'` already has been added to `G`. Please avoid using index names that contain underscores in R, as this can cause collisions.

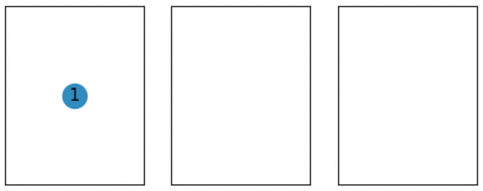
Node/edge labels can be added by using the parameter `attr=LABEL` when making new nodes/edges. A second option is using the functions `nx.set_node_attributes` or `nx.set_edge_attributes`, by passing the graph, a dictionary with the nodes as keys and labels as values, and argument the `name='attr'`. Both approaches can be seen in the code example in Appendix D, and is also available as */code/example.py*.

The example assumes the script is written in the */code/* directory, such that `Production` can be imported as shown.

The `debug=True` parameter for `Production` will print out information during the graph transformation. Giving the graphs and productions names, e.g. `L = nx.Graph(name='L')` will make the debug prints more readable. The debug print from the code example can be found in Appendix E.



(a) Input graph G.



(b) Production $p = (L, K, R)$. The left box contains L, the middle box contains K, the right box contains R.

Figure 4.1: The production p deletes a node since we have a node in L and no nodes in R. Suppose the node in L is mapped to the middle node in G. When p would remove that node, p would cause a dangling condition. This production can not be performed on this input graph.

If one wishes to load SMILES strings as NetworkX graphs that follow our labelling convent, be sure to `import graph_gen`. To load SMILES string use `graph_from_smiles(SMILES)`, where `SMILES` is a valid SMILES string. To convert a graph to a SMILES string, use `smiles_from_graph(G)`, where `G` is a NetworkX graph representing a molecule that follows our labelling convention. These functions have been written using the Python library PySMILES, and modifying the outcomes to follow our labelling convention.

4 Analysis of Implementation

4.1 Testing

Six tests have been written in total. Four of the tests are pretty simple. They cover the creation of productions, graph transformations without labels, graph transformations with node labels, and graph transformations with node and edge labels. These are the tests `production_test.py`, `transformation_test.py`, `node_label_test.py`, and `edge_label_test.py`. There is not much to explain regarding these tests, and they will not get covered in further detail. All tests can be found in the directory */code/tests/*.

The two other tests make sure the following are handled correctly: dangling conditions in `dangling.py`, glueing of graphs in `gluing.py`.

The input graph G and the production p from `dangling.py` have been depicted in Figure 4.1. In this test, we try to perform p with G as input graph, and we assert that `None` is returned from `perform_transformation`, as this is an illegal graph transformation.

As stated earlier, the implementation deviates from the theory in that nodes and edges that change labels are removed when constructing D and added back

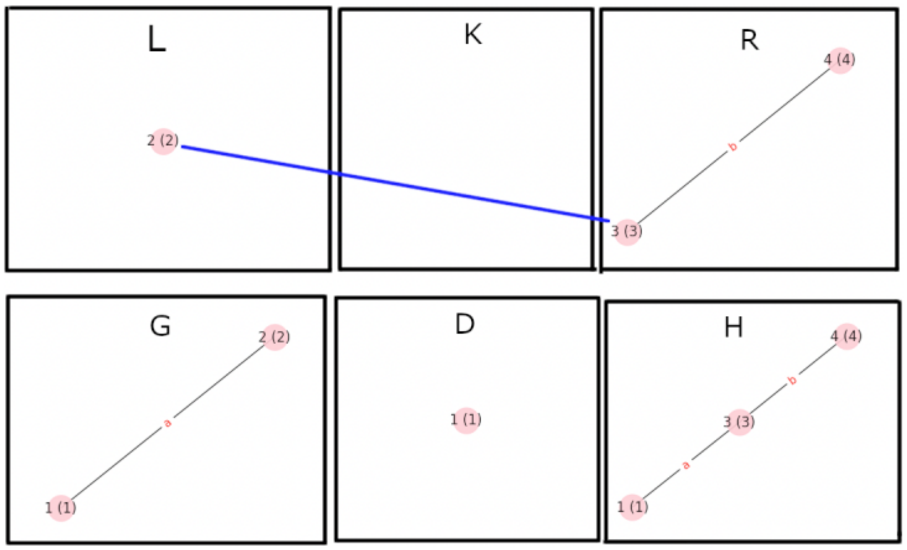


Figure 4.2: All graphs shown have been made using Matplotlib, plotting the graphs during the test script execution. The implementation deviates from the theory, and therefore K does not contain any nodes. The implementation only has nodes in K when the labels are the same across L and R. In the theory K would contain a node u with label $\langle 2, 3\rangle$. The blue line represents the mapping from L to R. Notice that node 2 is removed from D when constructing D. In the theory D would contain a node v with label $\langle 2, 3\rangle$, as well as an edge from 1 to v with the label $\langle a, a\rangle$. When node 3 from R is added to D to construct H, the edge that originally was $(1, 2)$ in G, is reconnected as $(1, 3)$ in H because of the mapping from L to R. The edge label of the removed edge $(1, 2)$ from G is copied to added edge $(1, 3)$ in H. This shows the glueing performed during the graph transformation has the desired outcome.

with new labels when constructing H. This means nodes in $G \backslash L$ that have edges connected to nodes in L might get removed when constructing D. `gluing.py` has been written to show the reconnecting of edge takes place. A representation of `gluing.py` can be seen in Figure 4.2.

Validating our Formose Grammar The Formose grammar has, in part, been implemented as a final test of our production implementation. The Formose grammar allows us to cross-reference the results we get with the results we can get from MØD [And20a] even though MØD simulates a different Formose grammar than the one we have implemented.

Since we start with n number of glycolaldehyde and m formaldehyde, the total number of atoms in all generated graph should be $n \cdot 8 + m \cdot 4$. The molecule glycolaldehyde has 8 atoms, and formaldehyde molecules have 4 atoms.

Our graphs consist of one or more connected components, as we start with n glycolaldehyde and m formaldehyde molecules that are not connected. However, during the iterations, the productions might connect some of the components.

We can use the 'Predicate' example [And20b] to generate all the different connected components our graphs should consist of. If our graphs consist of these connected components, we can validate the results we get from our Formose grammar.

We found the largest valid number of atoms that would display a network a graphs in MØD, to be 20. The Formose simulation in MØD is an iterative process that adds one glycolaldehyde at each iteration. This means the output graphs from MØD with a max of 20 atoms contain the graphs created from any smaller number of atoms.

In the file */code/formose/MOD_formose_graphs.py*, we have all graphs created from the 'Predicate' example [And20b], with max amount of atoms being 20. The graphs have been imported from SMILES strings. These SMILES strings have been attained using the online tool [Che]. A screenshot of this process can be seen in Appendix F.1.

In */code/formose/validate_formose.py* we import the graphs from */code/formose/MOD_formose_graphs.py* and store them in the list `references`. A dictionary is made `instances` where the keys the number of atoms and the values are lists containing dictionaries. The dictionaries have two keys `n_glyco` and `m_formal`, with the values being the number of that molecule in the start graph.

We have $\{4, 8, 12, 16, 20\}$ as the different keys of `instances`, as these are the valid number of atoms ≤ 20. We then have all the possible `n_glyco` and `m_formal` that give the desired number of atoms.

Please see the code snipped below, showing `instances[4]`, `instances[8]`, `instances[20]`.

```
instances = {}
instances[4] =  [{'n_glyco': 0, 'm_formal': 1}]
instances[8] =  [{'n_glyco': 0, 'm_formal': 2},
                 {'n_glyco': 1, 'm_formal': 0}]
instances[20] = [{'n_glyco': 0, 'm_formal': 5},
                 {'n_glyco': 1, 'm_formal': 3},
                 {'n_glyco': 2, 'm_formal': 1}]
```

In */code/formose/validate_formose.py* the following steps are done:

1. Get `n_glyco` and `m_formal` from the list of `instances[ATOMS]`, where `ATOMS` are the number of atoms currently being validated. If there are no more `n_glyco` and `m_formal` for this number of atoms, go to the next number of atoms.

2. Run `perform\_formose(n_glyco, m_formal)` and store the output graphs in `output_graphs`.

3. Assert that the number of nodes in each graph is equal to `n_glyco*8 + m_formal*4`.

4. Go through every graph in `output_graphs`. Use `nx.connected_components` on each graph to get the connected components of that graph.

5. Assert each connected component is isomorphic with some graph from `references`.

6. Repeat this procedure until all connected components have been asserted to be isomorphic with some graph in `references`. That is all connected components, from all the output graphs, of all different combinations of `n_glyco` and `m_formal`, across the different total number of atoms in $\{4, 8, 12, 16, 20\}$.

The script tries all different combinations of `n_glyco`, `m_formal`, and checks all connected components are isomorphic to a graph in `references`. The Output from
/code/formose/validate_formose.py can be found in Section C.

4.2 Running Time Analysis

We have not been able to find the exact worst case running time for the VF2 algorithm.

In [Cor+01] it is stated "$P(s)$ can be done in a time in the worst case proportional to $|N1| + |N2|$, while the computation of $F(s, n, m)$ can be performed in a time proportional to the number of the branches involving n and m.". The description of $P(s)$ and $F(s, n, m)$ can be found in Appendix B.

The subgraph isomorphism problem is an NP-hard problem (see Appendix B). We assume the worst-case running time is some exponential time in relation to the number of nodes in the graphs being matched.

We will go through the function `perform_transformation` from
/code/production.py and the functions that are called from
`perform_transformation`. We will try to find lower- and upper bounds for the number of operations performed. All operations in `perform_transformation` revolve around adding/removing nodes from G, L, R. The bounds will be based on the number of nodes in the largest of the graphs G, L, R. We will call this number of nodes n. Since $V_L \subseteq V_G$, we can say $n = max\{V_G, V_R\}$.

This list goes over the upper limit for each function called from
`perform_transformation`. We can then refer to the lower/upper bounds from this list when calculating the running time of `perform_transformation`. The different functions in this list have been described in Section 3.1. We assume `if` statements and the removal/addition of nodes/edges to be operations that take constant time, and they will not be considered in lower- and upper bounds.

We may write the lower- and upper bound as $O(n)$ / $O(n^2)$ after an operation. In this example, the lower bound is $O(n)$, and the upper bound is $O(n^2)$.

- `diff(A, B, AB_mapping)`: When removing nodes, we loop through all nodes in `A`. We loop through all edges in `A` when removing edges. The best case is `A` not having any edges. The worst case is `A` being a complete graph. The lower bound is $O(n)$, and the upper bound is $O(n + n^2) = O(n^2)$.

- `dangling_nodes(L, G, LG_mapping)`: We calculate `diff(G, L, GL_mapping)` ($O(n)$ / $O(n^2)$). We loop through the nodes of `L` in a list comprehension, and loop through the nodes in this list ($O(n)$ / $O(n^2)$). We then loop through the edges in `GL_diff_edges`, which is at least 0 edges and at most n^2 edges (($O(0)$ / $O(n^2)$). From these operations, we get the lower bounds $O(n + n + 0) = O(n)$ and the upper bound $O(n^2 + n^2 + n^2) = O(n^2)$

When constructing D in `perform_transformation`, we do the following operations: call `diff` ($O(n)$ / $O(n^2)$), call `dangling_nodes` ($O(n)$ / $O(n^2)$), remove all edges in `LK_diff_edges` from `G` ($O(0)$ / $O(n^2)$), remove all nodes from `LK_diff_nodes` from `G` ($O(0)$ / $O(n)$). The lower bound for constructing D is $O(n + n + 0) = O(n)$ and the upper bound is $O(n^2 + n^2 + n^2) = O(n^2)$.

When constructing H in `perform_transformation`, we do the following operations: call `diff` ($O(n)$ / $O(n^2)$), add all nodes in `RK_diff_nodes` to `H` ($O(0)$ / $O(n)$), add all edges in `RK_diff_edges` to D ($O(0)$ / $O(n^2)$), reconnect edges from `G` that have been removed while constructing D ($O(0)$ / $O(n^2)$). The lower bound for constructing H is $O(n+0+0+0) = O(n)$ and the upper bound is $O(n^2 + n + n^2 + n^2) = O(n^2)$.

By adding up the lower- and upper bounds from constructing D and H, we can then conclude that the lower bound for `perform_transformation` is $O(n + n) = O(n)$ and the upper bound is $O(n^2 + n^2) = O(n^2)$.

With our assumption that VF2 has a worst-case in exponential time, we get the worst-case running time for our graph transformations to be $O(k^n + n^2)$ where k is some unknown constant.

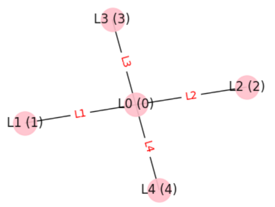
4.3 Benchmark

All benchmarks have been run on a 2020 Macbook Air with 8GB ram. The CPU has 8 cores. The laptop has been connected to its power supply during all tests. Click here for full specification sheet.

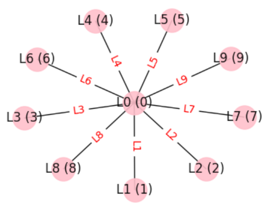
We inspected the CPU usage when the computer was idle and could see it was quite high. When running `formose.py` we inspected the CPU usage percentage in `htop`, which stays between 99.7% and 100%.

The results are used in comparison to one another, and the CPU usage of Python seems relatively stable. Therefore, we conclude that the quite high idle CPU usage does not affect the benchmark results, as the slight variances should even out if run each benchmark a reasonable amount of times.

We have written two benchmarks. In the first benchmark we have a graph G, consisting of n nodes and $n - 1$ edges. We have a node 0 in G, and $E_G = \{(0, v) \mid v \in V_G\}$. The node v and the edge $(0, v)$ both have the label `f'L{v}'`, i.e. the node 5 and edge $(0, 5)$ both have the label `L5`. See Figure 4.3.



(a) $n = 5$.



(b) $n = 10$.

Figure 4.3: Notice that every node has an edge to 0, and each node- and edge label is unique. Two examples are shown, (a) has 5 nodes, and (b) has 10 nodes. Each graph has $n-1$ edges, where n is the number of nodes.

We have two productions, $p = (L, K, R)$ and $q = (R, K, L)$. The graph L is equal to the graph G. The graph R has the same structure as G and L, but uses `R` instead of `L` in the edges, i.e. the node 5 and edge $(0, 5)$ both have the label `R5`. q is the inverse production of p.

All nodes of L map to nodes in G. These nodes and edges are then removed, and the nodes and edges of R. This gives the output graph H, which is equal to R. We then run q on H. All nodes and edges from H, and we add in the nodes and edges from L. Now we are back to where we started.

With this procedure, we can benchmark how long it takes to:

1. Find a `LG_mapping` using the VF2 implementation in NetworkX.
2. Remove n nodes and $n-1$ edges, and add n nodes and $n-1$ edges.

For $n \in \{500, 740, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000, 7500\}$ we find the `LG_mapping` 40 times. For $n \in (10000, 15000, 20000, 25000, 50000)$ we run the mappings $(10, 8, 4, 4, 2)$ times respectively. We perform both productions 500 times for different n values

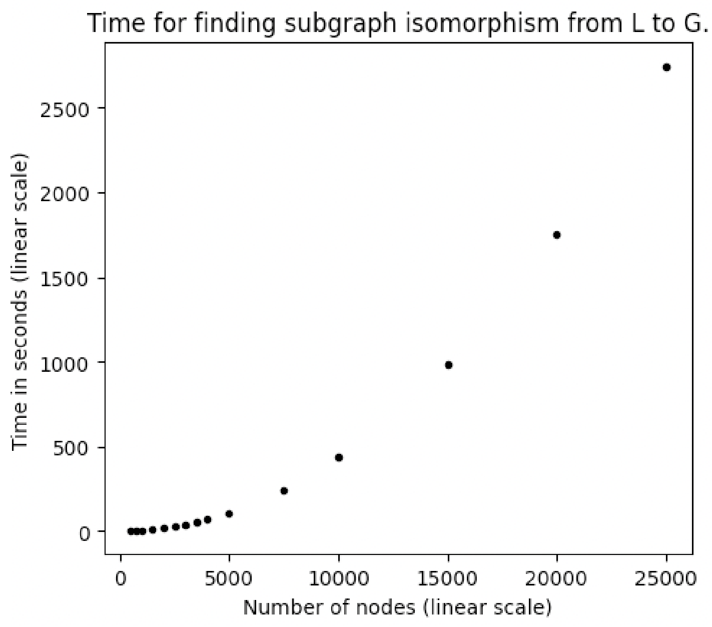
We can find the maximum, minimum and average time of finding an `GL_mapping`, and performing two productions for each graph with n nodes and $n-1$ edges. Be aware that the production times are taken from performing both productions, p and q.

When finding the `LG_mapping` for 50000 nodes, we got a segmentation fault, which indicates we ran out of memory.

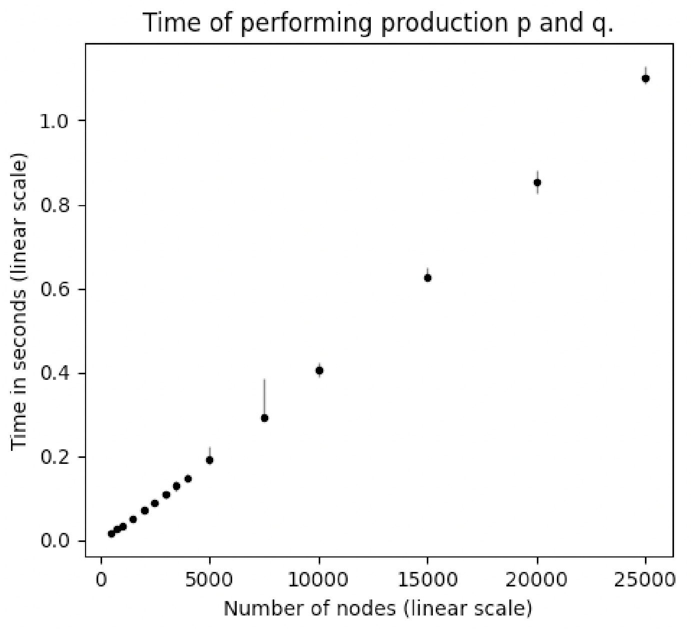
Two graphs have been generated from this benchmark. They can be seen in Figure 4.4

The second benchmark uses our implementation of the Formose grammar. We run the Formose grammar on starting graphs consisting of a different number of glycolaldehyde and formaldehyde molecules.

The different combinations of glycolaldehyde and formaldehyde can be seen in Table G. Please note that the combination of 4 glycolaldehyde 2 formaldehyde only has two iterations. This is the data point in the graphs with 40 atoms and approx. $27 \cdot 10^6$ iterations.

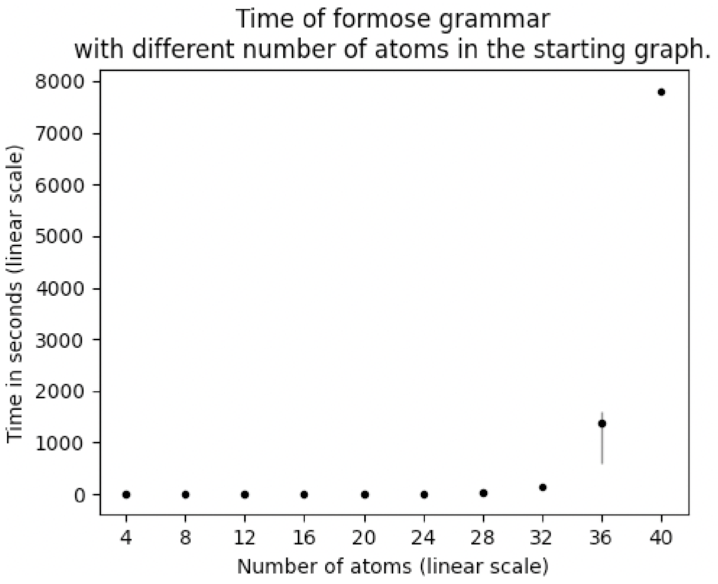


(a)

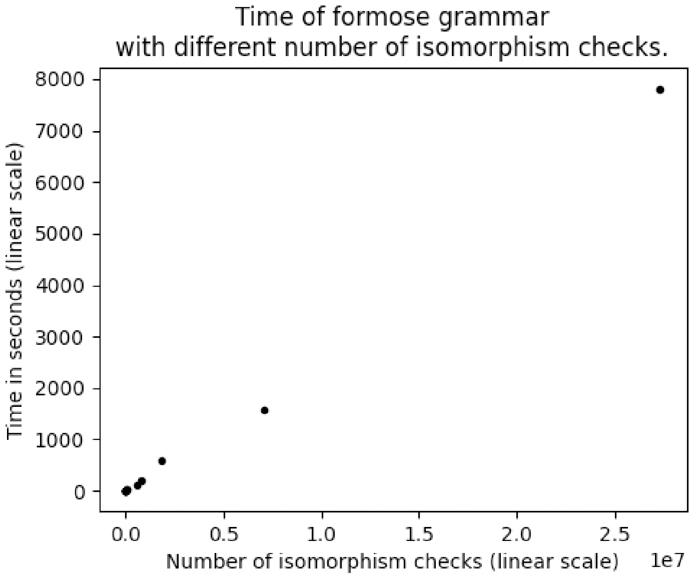


(b)

Figure 4.4: In both (a) and (b), the X-axis has the number of nodes in a graph like the one shown in Figure 4.3. The Y-axes show the time in seconds. Both axes in (a) and (b) are in linear scale. The wicks on the graph mark the min and max values, where the points are the mean values. The table for the data in (a) can be found in Appendix G, in Appendix G for (b).

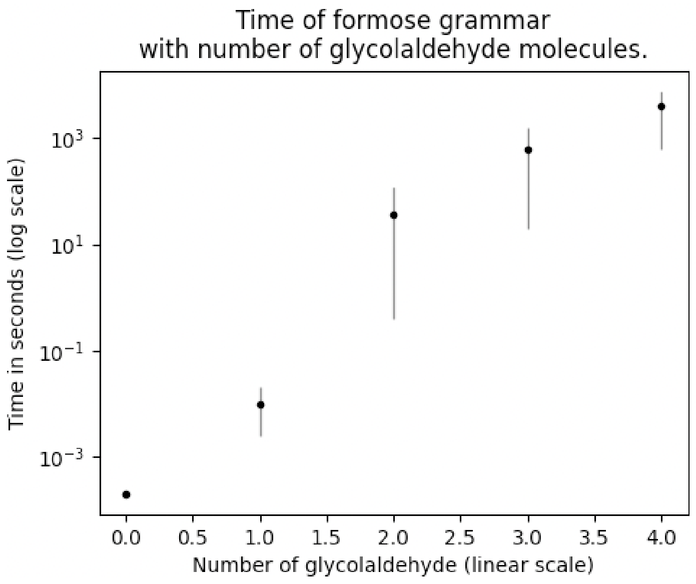


(a)

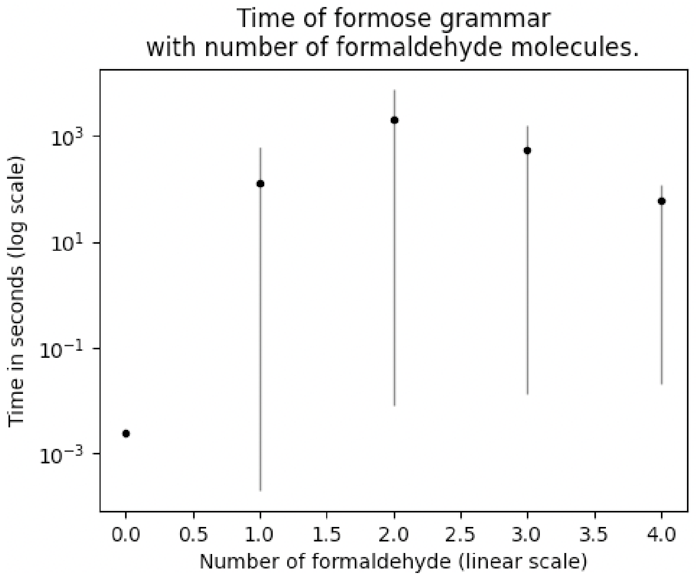


(b)

Figure 4.5: In (a), the X-axis has the number of nodes in the starting graph of our implementation of the Formose grammar. The X-axis in (b) is the number of isomorphism checks performed. Notice the numbers are in $1e7$. The Y-axes show the time in seconds. Both axes in (a) and (b) are in linear scale. The wicks on the graph mark the min and max values, where the points are the mean values. The table for the data in (a) can be found in Appendix G, in Appendix G for (b).



(a)



(b)

Figure 4.6: In (a), the X-axis has the number of glycolaldehyde in the starting graph of our implementation of the Formose grammar. The X-axis in (b) is the number of formaldehyde in the starting graph. The Y-axes show the time in seconds. In (a) and (b), the X-axis is in linear scale, and the Y-axis is in log scale. The wicks on the graph mark the min and max values, where the points are the mean values. The exact times for the different combinations of glycolaldehyde and formaldehyde can be seen in Appendix G.

We kept track of the time it took to perform the Formose grammar throughout the benchmark until no new (non-isomorphic) graphs were found. The data we were interested in is the time corresponding to the total number of nodes in the starting graph, i.e. the total number of atoms for the glycolaldehyde- and formaldehyde molecules.

We tried running the benchmark in parallel, but using four cores did not speed up the process. The time was equal to the performance of one CPU core. We would finish four instances of a case 4 times slower when running 4 cores. So 4 cases would finish at the same time in approx. 320 seconds if the single core could complete a single instance in 80 seconds.

We also kept track of the total number of isomorphism checks performed for each of the combinations of glycolaldehyde and formaldehyde. We did not get to reach a point where the benchmark ended due to too high memory usage like we suspected happened in the first benchmark.

We used the library cProfile in Python to see what function took most of the time. An alteration of the output from cProfile on the test case 3 glycolaldehyde and 2 formaldehyde can be seen in Appendix H. Please note that we removed a lot of the functions that took an insignificant amount of time in order to make the output more readable. The full output, along with the output from the `time --verbose` command, can be found as *cProfile_formose_3glyco_2formal.txt* is the directory */code/benchmark/*.

Plots for the total number of atoms in the starting graph and the total amount of isomorphism checks performed can be seen in Figure 4.5.

We show the time it takes with the different numbers of glycolaldehyde and formaldehyde in Figure 4.6. The goal is to determine whether the number of glycolaldehyde or formaldehyde is the most significant contributor to the overall running time of the Formose grammar.

All benchmark results, raw data and scripts can be found in the directory */code/benchmarks*.

5 Discussion

In this section, we go over the results from Section 4. Then, we will discuss the findings in relation to one another. The goal is the find the main contributors to the overall running time in our DPO implementation and the Formose grammar implementation.

Afterwards, we will go over ways in which our implementation could be improved.

5.1 Results

It is clear to see from the graph (a) in Figure 4.4 that the amount of nodes is non-linear to the time it takes to perform subgraph matching. When looking at TableG, we see a polynomial pattern close to $O(n^2)$. Every time the number of nodes doubles, the mean time approx. grows 4 times.

We assumed the VF2 algorithm to have exponential growth in Section 4.2. We see in Appendix B that the function $P(s)$ computes all the possible permutations (m, n) where m is a node in $G1$ and n is a node in $G2$. The VF2 algorithm then proceeds to loop through these pairings. Since we always have the same number of nodes in both graphs, the number of pairings in this set is always `num_nodes`^{2}.

The reason we see polynomial growth can be derived from the explanation in Appendix B. The VF2 algorithm uses a function to calculate the feasibility function to prune its search tree. The graphs are sparse, and all nodes and edges have unique labels. This allows VF2 to prune most of the branches in the search tree.

We could have a case where the subgraph matching would have a lower asymptotic running time than the construction of D and H. Suppose we have graphs G and L both with a single node. Finding L as a subgraph in G would only require VF2 to loop through a single pairing $(1, 1)$. With a complete graph R with n nodes, we would have to add n nodes and n^2 edges, giving us $O(n + n^2) = O(n^2)$.

These are the interesting observations from the DPO test:

- We are convinced that VF2 performs close to its best-case running time with the graphs and productions from the DPO test. When both graphs used in VF2 have n edges, the best-case running is $O(n^2)$. We assume the worst case to be exponential, as the subgraph isomorphism problem is NP-complete Appendix B.
- With the graphs and productions from the test the function `perform_transformation` runs in some polynomial time $O(n^k)$, where $1 < k \leq 2$ since we found the worst case to be $O(n^2)$.
- We could have special cases where the VF2 has a lower asymptotic running time than `perform_transformation`. But in general the best case for VF2 is $O(n^2)$ and the worst case for `perform_transformation` is $O(n^2)$. This means that the VF2 subgraph matching has the largest impact on the running time out of these two processes.

From Section 4.2 we know the best case and worst case running times for our constructing D and H, i.e. performing a production. The best case and worst case times are $O(n)$ and $O(n^2)$ respectively, where n is the number of nodes in the largest graph of G and R. When looking at the graph (b) in Figure 4.4, the time of performing the productions might seem linear. Table G shows a slight polynomial growth, e.g. going from 2000 nodes to 20000 nodes gives an increase of 12.1 times, where a linear growth would be 10. Similarly, going from 5000 nodes to 25000 nodes increases the mean time by 5.757 times, where linear time would increase the mean time 5 times.

In the graph (a) from Figure 4.5, we see an exponential growth in accordance to how many atoms present in the starting graph. The graph (b) shows signs of linear growth, where the time is linear to the number of isomorphism checks performed.

In Table G we see a trend where a larger number of atoms in the starting graphs most of the time lead to more graphs being generated. Each new graph generated might be isomorphism checked against all other new graphs from the same iteration. The non-isomorphic graphs from that iteration will be isomorphism checked against all previously generated graphs.

In the cProfile output in Appendix H from a test with 3 glycolaldehyde and 2 formaldehyde ($(3, 2)$), we see `perfrom_transformation` was called 7971 times, this aligns with Table G where we see $(3, 2)$ generates 7972 graphs. The reason we have 1 more graph generated than calls to `perform_transformation` is because the starting graph is included in the total graphs. The output from cProfile shows 5.161 seconds were spent in the function `perform_transformation` out of the total 371.809 seconds. Whereas in the same test case 322.163 seconds are spent in
`isomorphvf2.py`.

From this information it becomes evident:

- The number of atoms in the starting graphs gives rise to a larger number of generated graphs.
- A larger number of graphs being generated will lead to more isomorphism checks being performed.
- The time it takes to perform the Formose grammar is linear to the number of isomorphism checks. This is further validated in the cProfile output, where we see most of the time is spent doing isomorphism checks.

We wanted to determine whether the number of glycolaldehyde or formaldehyde contributed the most to the overall running time. In Figure 4.6 we see the times for the different numbers of glycolaldehyde and formaldehyde, ranging from 1 to 4 molecules. The interesting part comes from examining the wick that represents the minimum- and maximum times. In graph (a) we see the wicks are narrow compared to the wicks in graph (b). We see a very noticeable time increase for each increase in the higher the count of glycolaldehyde molecules in the start graph. In contrast, the different numbers of formaldehyde molecules have large, overlapping spans.

Be aware that these are low amounts of glycolaldehyde and formaldehyde molecules. When using our implementation on small combinations of glycolaldehyde and formaldehyde, glycolaldehyde has the biggest impact on the overall running time of our Formose grammar implementation.

5.2 Further Development of Implementation

This first improvement would yield the largest gain but would also require rewriting most of, if not the entire code. It is a know fact that Python is comparatively a very slow programming language. Using a programming language like C or C++, you can achieve much better performance. Porting the entire Python code to C or C++ seems beyond the scope of 'further development' while most likely yielding the best result.

Some Python libraries are written partly in C/C++ and use Python as a wrapper, e.g. a little over a third of the code in NumPy is written in C, or C++ [Num05]. Whereas NetworkX is a pure Python implementation [Net05].

iGraph [iGr06] is a C library that offers many of the same features as NetworkX: undirected/directed unlabelled/labelled graphs, subgraph isomorphism- and isomorphism checks using VF2. iGraph has a Python interface for their implementation [iGr].

While reading anecdotes on online fora, we found people claiming their iGraph code runs upwards of 150 times faster than their NetworkX code. These will not be cited, as they are not credible sources. However, the overall consensus seems to be that iGraph can be much faster than NetworkX. If we were to change out NetworkX for iGraph, we would expect to see a significant reduction in code execution times.

Since iGraph also uses VF2 for subgraph matching, the theoretical running time should stay the same. The boost in performance would come from switching to a C backend.

Another improvement would be a feature to avoid collisions between names of indices in R and H completely. Right now, we mitigate collision by appending the first letter in the name of the production to every node in H. If one was to use names in R that collide with this renaming method, we could not avoid collisions. This feature has not been of high priority because the likelihood of this happening is low, outside of a deliberate attempt to cause a collision.

As the implementation stands, a visual error appears when trying to plot all three graphs of a production using `show_prod` in */code/production.py*. This function should plot all three graphs with their labels in three separate subplots. Nodes, edges, and node labels get displayed properly, but all the edge labels end up in the third graph for R. A workaround has been to plot each graph by itself with `show_graph`. Getting the `show_prod` function to work with labelled graphs would improve the implementation, making it easier to use.

6 Conclusion

In this thesis, we have presented different topics regarding algebraic graph transformation:

- Category theory as a way of reasoning about the relations between objects of similar type. This has been used to show graphs as objects in categories and graph morphisms as a way to relate these graphs.
- The main idea of a graph transformation, which gave rise to the problems of subgraph matching and glueing of graphs.
- The Double Pushout (DPO) approach as a possible solution to these problems.
- The usage these principles on labelled graphs.

We have shown how these methods can be used to implement DPO and the Formose grammar. Furthermore, through benchmarks, we gathered data, which has been used to find the most significant contributors to the overall running time.

From the data from the DPO implementation, we can conclude that in almost every case, the most significant contributor to the overall running time is the process of subgraph matching. The running time of the VF2 algorithm and the construction of the graph D and H are tied to the number of nodes in the graphs. But increasing the number of nodes has the most significant impact on the subgraph matching.

Similarly, the running time of the Formose implementation is tied to the number of atoms in the starting graph. However, increasing the number of atoms has the most significant impact on the number of graphs being generated. This increases the number of isomorphism checks, which we found to be linear to the total running time of the Formose grammar.

A Graph examples

$$\begin{tikzcd}
& A \arrow[r, "4"] \arrow[dr, "4"] & D \arrow[dr, "4"] & \\
S \arrow[ur, "7"] \arrow[r, "3"] \arrow[dr, "5"] & B \arrow[r, "2"] \arrow[dr, "1"] & E \arrow[r, "6"] & T \\
& C \arrow[r, "4"] & F \arrow[ur, "5"] &
\end{tikzcd}$$

Figure A.1

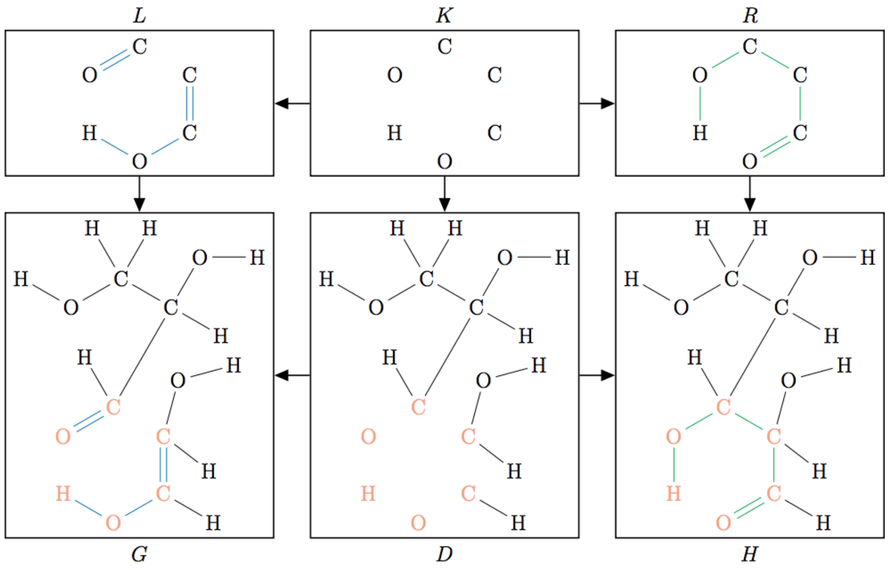


Figure A.2: From [Mer19].

B VF2 algorithm

Subgraph matching is interesting in regards to graph transformation as it is used to find L, from production $p = (L, K, R)$, in an input graph G. That is, subgraph matching is used to find the morphism we have referred to as m in Section 2.3.

Subgraph matching, also know as the subgraph isomorphism problem, is not a trivial problem. This thesis will not go into the complexity theory of the problem. This is in fact an NP-complete problem, as shown in the paper [Coo71].

The VF2 algorithm solves the subgraph isomorphism problem, and is mentioned as it is used through NetworkX in the Python implementation.

VF2 works by iteratively extending a partial solution. It uses a set of feasibility criteria to decide whether it should extend in a certain direction or backtrack [ER12]. More information can be found in the paper [Cor+01].

VF2 Pseudo Code Everything is directly from [Cor+01].

```
PROCEDURE Match(s)
  INPUT: an intermediate state s; the initial state s0 has M(s0)=∅
  OUTPUT: the mappings between the two graphs

  IF M(s) covers all the nodes of G2 THEN
    OUTPUT M(s)
  ELSE
    Compute the set P(s) of the pairs candidate for inclusion in M(s)
    FOREACH (n, m) P(s)
      IF F(s, n, m) THEN
        Compute the state s´ obtained by adding (n, m) to M(s)
        CALL Match(s )
      END IF
    END FOREACH
    Restore data structures
  END IF
END PROCEDURE
```

where $F(s, n, m)$ is a boolean function (called feasibility function) that is used to prune the search tree. If its value is true, it is guaranteed that the state s obtained adding (n, m) to s is a partial isomorphism if s is; hence the final state is either an isomorphism between $G1$ and $G2$, or a graph-subgraph isomorphism between a subgraph of $G1$ and $G2$. Moreover, F will also prune some states that, albeit corresponding to an isomorphism between $G1(s)$ and $G2(s)$, would not lead to a complete matching solution.

C Output from validate_formose.py

```
$ python3 validate_formose.py
----------------------------------
Total time: 0:00:00.000222
Total iterations: 1
Total number of graphs: 1
Total number of non-isomorphic graphs: 1
Unique ratio: 1.0
Time per graph: 0:00:00.000222
Time per unique graph: 0:00:00.000222
Total number of glycolaldehyde: 0
Total number of formaldehyde: 1
Total atom count: 4
----------------------------------
Total time: 0:00:00.000359
Total iterations: 1
Total number of graphs: 1
Total number of non-isomorphic graphs: 1
Unique ratio: 1.0
Time per graph: 0:00:00.000359
Time per unique graph: 0:00:00.000359
Total number of glycolaldehyde: 0
Total number of formaldehyde: 2
Total atom count: 8
----------------------------------
Total time: 0:00:00.002421
Total iterations: 2
Total number of graphs: 5
Total number of non-isomorphic graphs: 2
Unique ratio: 0.4
Time per graph: 0:00:00.000484
Time per unique graph: 0:00:00.001210
Total number of glycolaldehyde: 1
Total number of formaldehyde: 0
Total atom count: 8
----------------------------------
Total time: 0:00:00.000508
Total iterations: 1
Total number of graphs: 1
Total number of non-isomorphic graphs: 1
Unique ratio: 1.0
Time per graph: 0:00:00.000508
Time per unique graph: 0:00:00.000508
Total number of glycolaldehyde: 0
Total number of formaldehyde: 3
Total atom count: 12
----------------------------------
Total time: 0:00:00.005100
Total iterations: 2
Total number of graphs: 9
Total number of non-isomorphic graphs: 2
Unique ratio: 0.2222222222222222
Time per graph: 0:00:00.000567
Time per unique graph: 0:00:00.002550
Total number of glycolaldehyde: 1
Total number of formaldehyde: 1
Total atom count: 12
----------------------------------
Total time: 0:00:00.000660
Total iterations: 1
Total number of graphs: 1
Total number of non-isomorphic graphs: 1
Unique ratio: 1.0
Time per graph: 0:00:00.000660
Time per unique graph: 0:00:00.000660
Total number of glycolaldehyde: 0
Total number of formaldehyde: 4
Total atom count: 16
----------------------------------
Total time: 0:00:00.014313
Total iterations: 3
Total number of graphs: 18
Total number of non-isomorphic graphs: 3
Unique ratio: 0.16666666666666666
Time per graph: 0:00:00.000795
Time per unique graph: 0:00:00.004771
Total number of glycolaldehyde: 2
Total number of formaldehyde: 0
Total atom count: 16
----------------------------------
Total time: 0:00:00.008430
Total iterations: 2
Total number of graphs: 13
Total number of non-isomorphic graphs: 2
Unique ratio: 0.15384615384615385
Time per graph: 0:00:00.000648
Time per unique graph: 0:00:00.004215
Total number of glycolaldehyde: 1
Total number of formaldehyde: 2
Total atom count: 16
----------------------------------
```

```
Total time: 0:00:00.000791
Total iterations: 1
Total number of graphs: 1
Total number of non-isomorphic graphs: 1
Unique ratio: 1.0
Time per graph: 0:00:00.000791
Time per unique graph: 0:00:00.000791
Total number of glycolaldehyde: 0
Total number of formaldehyde: 5
Total atom count: 20
----------------------------------
Total time: 0:00:00.013955
Total iterations: 2
Total number of graphs: 17
Total number of non-isomorphic graphs: 2
Unique ratio: 0.11764705882352941
Time per graph: 0:00:00.000821
Time per unique graph: 0:00:00.006978
Total number of glycolaldehyde: 1
Total number of formaldehyde: 3
Total atom count: 20
----------------------------------
Total time: 0:00:00.403765
Total iterations: 10
Total number of graphs: 233
Total number of non-isomorphic graphs: 21
Unique ratio: 0.09012875536480687
Time per graph: 0:00:00.001733
Time per unique graph: 0:00:00.019227
Total number of glycolaldehyde: 2
Total number of formaldehyde: 1
Total atom count: 20
================================================
Tested graphs with num atoms: [4, 8, 12, 16, 20].
All graphs/components match reference graphs generated from MØD.
```

D Code Example of a Simple Graph Transformation

```python
import networkx as nx
from production import Production

### Input Graph
G = nx.Graph(name='G')
# Adding nodes and node labels
G.add_nodes_from([1, 2, 3])
nx.set_node_attributes(G, {1: 'A', 2: 'B', 3: 'C'}, name='attr')
# Adding edges and edge labels
G.add_edges_from([(1,2),(2,3)])
nx.set_edge_attributes(G, {(1,2): 'X', (2,3): 'Y'}, name='attr')

### Graph L
L = nx.Graph(name='L')
# Adding nodes and node labels
L.add_node(2, attr='B')
L.add_node(3, attr='C')
# Adding edge and edge label
L.add_edge(2, 3, attr='Y')

### Graph R
R = nx.Graph(name='R')
# Adding nodes and node labels
R.add_node('u', attr='D')
R.add_node('v', attr='E')
# Adding edge and edge label
R.add_edge('u', 'v', attr='Z')

### Production
# Specifying LR_mapping, as L and R do not share name indices.
LR_mapping = {2: 'u', 3: 'v'}
prod = Production(L, R, LR_mapping=LR_mapping, debug=True)

# Get list of possible LG mappings
LG_mappings = prod.possible_LG_mappings(G)

# Perform graph transformation
H = prod.perform_transformation(G, LG_mappings[0])
```

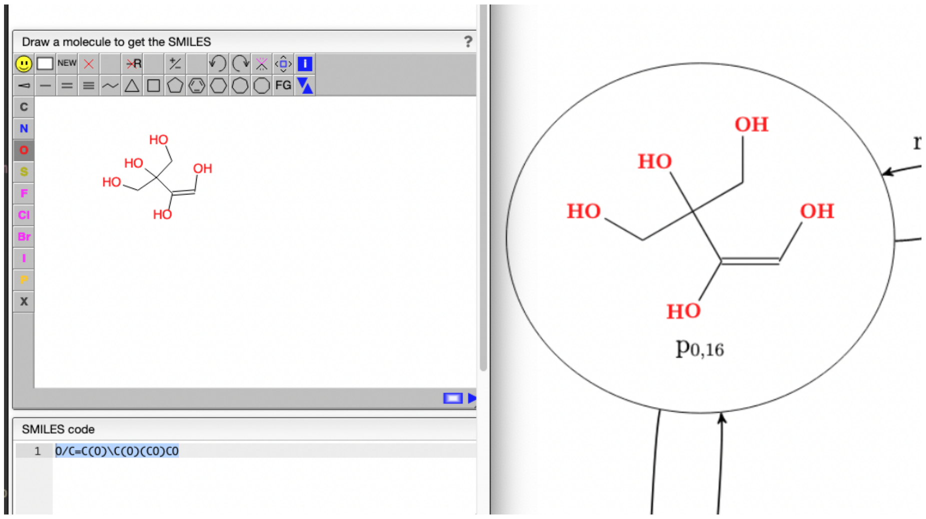
Code example can be found as */code/example.py*.

E Debug Print Example

```
+------------------+
|Intersection: L, R|
+------------------+
I.nodes: []
I.edges: []
+-----------+
|LG Mappings|
+-----------+
LG_mapping: {2: 2, 3: 3}
+---------------------+
|Single transformation|
+---------------------+
+----------+
|diff: L, K|
+----------+
L.nodes: [2, 3]
L.edges: [(2, 3)]
L(2, 3): {'attr': 'Y'}
K.nodes: []
K.edges: []
diff_nodes: {2: 'B', 3: 'C'}
diff_edges: {(2, 3): 'Y'}
diff_L_K edge (2, 3): Y
+----------+
|diff: G, L|
+----------+
G.nodes: [1, 2, 3]
G.edges: [(1, 2), (2, 3)]
G(1, 2): {'attr': 'X'}
G(2, 3): {'attr': 'Y'}
L.nodes: [2, 3]
L.edges: [(2, 3)]
L(2, 3): {'attr': 'Y'}
diff_nodes: {1: 'A'}
diff_edges: {(1, 2): 'X'}
diff_G_L edge (1, 2): X
removing edge: (2, 3)
removing node: 2
removing node: 3
+----------+
|diff: R, K|
+----------+
R.nodes: ['u', 'v']
R.edges: [('u', 'v')]
```

```
R('u', 'v'): {'attr': 'Z'}
K.nodes: []
K.edges: []
diff_nodes: {'u': 'D', 'v': 'E'}
diff_edges: {('u', 'v'): 'Z'}
diff_R_K edge ('u', 'v'): Z
Added node to H: u
Added node to H: v
Added edge to H: [('u', 'v'), 'Z']
Added edge to H: [(1, 'u'), 'X']
H.nodes: {'1_pn': 'A', 'u_pn': 'D', 'v_pn': 'E'}
H.edges: {('1_pn', 'u_pn'): 'X', ('u_pn', 'v_pn'): 'Z'}
```

F SMILES from Skeletal Structure Online Tool



```
O/C=C(O)\C(O)(CO)CO
```

Figure F.1: On the left we see the tool, where a molecule has been drawn and the SMILES string is displayed at the bottom. On the right we see a graph of a molecule from the output of MØD 'Predicate' example [And20b] with the max number of atoms set to 20. Click this for link.

G Benchmark Tables

Table of matching times in label benchmark.
Used in Figure 4.4.

	Time (seconds)			
# Nodes	Mean	Min	Max	#Iterations
500	1.076	1.073	1.082	20
750	2.427	2.419	2.447	20
1000	4.321	4.313	4.326	20
1500	9.707	9.686	9.731	20
2000	17.293	17.264	17.342	20
2500	27.027	26.969	27.151	20
3000	38.935	38.859	39.079	20
3500	53.118	52.937	54.296	20
4000	69.195	69.05	69.313	20
5000	108.915	108.655	109.254	20
7500	245.02	244.124	245.751	20
10000	434.957	432.585	436.242	10
15000	984.496	979.663	991.088	8
20000	1752.428	1751.096	1754.402	4
25000	2745.066	2742.36	2747.276	4

Can be found in */code/benchmarks/results/label_benchmark_matching_results.csv*
Raw data: */code/benchmarks/raw_data/label_benchmark_matching.csv*

Table of production times in label benchmark.
Each iteration performs two productions, p and q.
Used in Figure 4.4.

	Time (seconds)			
# Nodes	Mean	Min	Max	#Iterations
500	0.0162	0.016	0.023	500
750	0.0245	0.024	0.032	500
1000	0.0342	0.032	0.041	500
1500	0.0515	0.049	0.059	500
2000	0.07	0.065	0.076	500
2500	0.089	0.082	0.095	500
3000	0.108	0.1	0.117	500
3500	0.128	0.118	0.141	500
4000	0.148	0.145	0.157	500
5000	0.192	0.182	0.224	500
7500	0.293	0.285	0.386	500
10000	0.404	0.388	0.423	500
15000	0.625	0.618	0.651	500
20000	0.854	0.826	0.881	500
25000	1.103	1.089	1.128	500

Can be found in */code/benchmarks/results/label_benchmark_production_results.csv*
Raw data: */code/benchmarks/raw_data/label_benchmark_production.csv*

Table of the time it took to run Formose grammar, and how many atoms present in the starting graph.
Used in Figure 4.5.

			Time (seconds)			
Glycol	Formal	Total #atoms	Mean	Min	Max	#Iterations
1	0	8	0.002	0.002	0.01	100
1	1	12	0.0051	0.005	0.014	100
1	2	16	0.008	0.008	0.016	100
1	3	20	0.013	0.013	0.013	100
1	4	24	0.020	0.02	0.028	100
2	1	20	0.407	0.404	0.418	100
2	2	24	4.514	4.488	4.582	100
3	1	28	19.876	19.835	20.163	100
2	3	28	21.771	21.741	21.855	100
2	4	32	120.877	120.646	121.41	40
3	2	32	193.242	192.932	195.15	40
4	1	36	604.697	604.105	606.507	5
3	3	36	1588.492	1586.662	1590.554	20
4	2	40	7804.243	7775.921	7832.564	2

Can be found in */code/benchmarks/results/formose_benchmark_results.csv*
Raw data: */code/benchmarks/raw_data/formose_benchmark.csv*

Table of the number of atoms in the Formose grammar, and how many graphs were generated in total.

Glycol	Formal	Total #atoms	Total #graphs
1	0	8	5
1	1	12	9
1	2	16	13
1	3	20	17
1	4	24	21
2	1	20	233
2	2	24	1135
3	1	28	2011
2	3	28	2657
2	4	32	6626
3	2	32	7972
4	1	36	12245
3	3	36	28317
4	2	40	53529

Can be found in */code/benchmarks/results/formose_benchmark_results.csv*
Raw data: */code/benchmarks/raw_data/formose_benchmark.csv*

Table of the time it took to run Formose grammar, and how many isomorphisms checks were made.
Used in Figure 4.5.

			Time (seconds)			
Glycol	Formal	Total #Iso. checks	Mean	Min	Max	#Iterations
1	0	4	0.002	0.002	0.01	100
1	1	13	0.0051	0.005	0.014	100
1	2	23	0.008	0.008	0.016	100
1	3	33	0.013	0.013	0.013	100
1	4	43	0.020	0.02	0.028	100
2	1	1130	0.407	0.404	0.418	100
2	2	18553	4.514	4.488	4.582	100
3	1	71616	19.876	19.835	20.163	100
2	3	100962	21.771	21.741	21.855	100
2	4	558010	120.877	120.646	121.41	40
3	2	829690	193.242	192.932	195.15	40
4	1	1871755	604.697	604.105	606.507	5
3	3	7052588	1588.492	1586.662	1590.554	20
4	2	27296362	7804.243	7775.921	7832.564	2

Can be found in */code/benchmarks/results/formose_benchmark_results.csv*
Raw data: */code/benchmarks/raw_data/formose_benchmark.csv*

H Altered cProfile Output from Formore

Full output in:
/code/benchmark/cProfile_formose_3glyco_2formal.txt.

```
2141899612 function calls (2130409384 primitive calls) in 371.809 seconds

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  371.809  371.809 <string>:1(<module>)
408463516   20.503    0.000   20.503    0.000 coreviews.py:44(__init__)
175304139   16.601    0.000   22.768    0.000 coreviews.py:50(__iter__)
 41205912    3.516    0.000    3.516    0.000 coreviews.py:53(__getitem__)
208586281   31.781    0.000   41.815    0.000 coreviews.py:81(__getitem__)
        1    0.003    0.003  371.809  371.809 formose.py:89(main)
        1    0.697    0.697  371.805  371.805 formose.py:9(perform_formose)
   145135    0.069    0.000  322.163    0.002 isomorphvf2.py:289(isomorphisms_iter)
5568705/369890   12.006    0.000  329.241    0.001 isomorphvf2.py:296(match)
    12947    0.004    0.000    7.274    0.001 isomorphvf2.py:379(subgraph_isomorphisms_iter)
 24223249   56.674    0.000  162.326    0.000 isomorphvf2.py:395(syntactic_feasibility)
  6291407   39.967    0.000  108.270    0.000 isomorphvf2.py:855(__init__)
 35072090    6.209    0.000    6.209    0.000 isomorphvf2.py:902(<listcomp>)
 35072090    5.907    0.000    5.907    0.000 isomorphvf2.py:912(<listcomp>)
  4947395    8.827    0.000    9.312    0.000 isomorphvf2.py:918(restore)
  1977424    0.312    0.000    0.397    0.000 matchhelpers.py:76(categorical_node_match)
 10266669    2.477    0.000    3.469    0.000 matchhelpers.py:79(match)
     7971    0.163    0.000    5.161    0.001 production.py:100(perform_transformation)
     7971    0.037    0.000    0.037    0.000 production.py:181(<dictcomp>)
     4976    0.012    0.000   12.512    0.003 production.py:197(perform_all_transformations)
     7971    0.011    0.000    1.371    0.000 production.py:211(dangling_nodes)
    23913    0.308    0.000    2.142    0.000 production.py:229(diff)
    23913    0.217    0.000    0.530    0.000 production.py:231(<listcomp>)
    23913    0.031    0.000    0.089    0.000 production.py:232(<listcomp>)
    23913    0.144    0.000    0.214    0.000 production.py:253(<listcomp>)
    31884    0.054    0.000    0.165    0.000 production.py:275(translate_edges)
     4976    0.011    0.000    7.339    0.001 production.py:73(possible_LG_mappings)
     4976    0.002    0.000    7.276    0.001 production.py:83(<listcomp>)
     4976    0.002    0.000    0.005    0.000 production.py:86(<listcomp>)
      2/1    0.000    0.000  371.809  371.809 {built-in method builtins.exec}
  4729069    0.243    0.000    0.246    0.000 {built-in method builtins.hasattr}
 15889503    0.819    0.000    0.819    0.000 {built-in method builtins.isinstance}
178582674    6.348    0.000    6.348    0.000 {built-in method builtins.iter}
95018026/88726619    3.474    0.000    4.338    0.000 {built-in method builtins.len}
  5288111    1.482    0.000    1.482    0.000 {built-in method builtins.min}
   138528    0.027    0.000  322.190    0.002 {built-in method builtins.next}
  1967472    4.450    0.000   20.922    0.000 {built-in method builtins.sorted}
        2    0.000    0.000    0.000    0.000 {method 'groupdict' of 're.Match' objects}
  2226256    0.139    0.000    0.326    0.000 {method 'update' of 'dict' objects}
 70144180    4.713    0.000    4.713    0.000 {method 'update' of 'set' objects}
```

Can be found in */code/benchmarks/results/formose_benchmark_results.csv*
Raw data: */code/benchmarks/cProfile_formose_3glyco_2formal.txt*

References

[And15] Jakob Lykke Andersen. “Analysis of Generative Chemistries”. In: (2015).

[And20a] Jakob Lykke Andersen. *MØD - Algorithmic Cheminformatics Group*. 2020. URL: `https://cheminf-live.imada.sdu.dk/mod` (visited on 05/29/2022).

[And20b] Jakob Lykke Andersen. *MØD - Predicate Example*. 2020. URL: `https://cheminf-live.imada.sdu.dk/mod/?example=0212_dgPredicate.py` (visited on 05/29/2022).

[Awo10] Steve Awodey. *Category theory*. Oxford university press, 2010.

[Awo14] Steve Awodey. *Category theory foundations 1.0 — Steve Awodey*. Youtube. 2014. URL: `https://youtu.be/BF6kHD1DAeU?t=146`.

[Bal+99] Paolo Baldan et al. “Concurrent semantics of algebraic graph transformation”. In: *Handbook of Graph Grammars and Computing by Graph Transformation* 3 (1999), pp. 107–187.

[Ben+10] Steven A. Benner et al. “Planetary organic chemistry and the origins of biomolecules”. English. In: *Cold Spring Harbor perspectives in biology* 2.7 (2010), a003467–a003467.

[But61] A. Butlerov. “Formation synthétique d’une substance sucrée”. In: *Compt. rend. Acad. Sci.* 53 (1861). Butlerov used the French spelling of his name in the paper., pp. 145–147.

[Che] Cheminfo. *SMILES Generator Checker*. URL: `http://www.cheminfo.org/flavor/malaria/Utilities/SMILES_generator___checker/index.html` (visited on 05/22/2022).

[Coo71] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: `10.1145/800157.805047`. URL: `https://doi.org/10.1145/800157.805047`.

[Cor+01] L. P. Cordella et al. “An improved algorithm for matching large graphs”. In: *In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*. 2001, pp. 149–159.

[ER12] Hans-Christian Ehrlich and Matthias Rarey. “Systematic benchmark of substructure search in molecular graphs-From Ullmann to VF2”. In: *Journal of cheminformatics* 4.1 (2012), pp. 1–17.

[Fin84] Gerd Finke. “A two-commodity network flow approach to the traveling salesman problem”. In: *Congresses Numeration* 41 (1984), pp. 167–178.

[GK06] U Prange G Taentzer and H Ehrig K Ehrig. *Fundamentals of algebraic graph transformation. With 41 Figures (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

[iGr] iGraph. *Python-iGraph - GitHub Repository*. URL: `https://github.com/igraph/python-igraph` (visited on 05/30/2022).

[iGr06] iGraph. *iGraph - GitHub Repository*. 2006. URL: `https://github.com/networkx/networkx` (visited on 05/30/2022).

[Mer19] Daniel Merkle. *A Double Pushout Diagram*. 2019. URL: `https://imada.sdu.dk/~daniel/posts/dpo` (visited on 05/08/2022).

[Mit65] Barry Mitchell. *Theory of categories*. Academic Press, 1965.

[Net05] NetworkX. *NetworkX - GitHub Repository*. 2005. URL: `https://github.com/networkx/networkx` (visited on 05/30/2022).

[Num05] NumPy. *NumPy - GitHub Repository*. 2005. URL: `https://github.com/numpy/numpy` (visited on 05/30/2022).

[SWS81] R.F. Socha, A.H. Weiss, and M.M. Sakharov. "Homogeneously catalyzed condensation of formaldehyde to carbohydrates: VII. An overall formose reaction model". In: *Journal of Catalysis* 67.1 (1981), pp. 207–217. ISSN: 0021-9517. DOI: `https://doi.org/10.1016/0021-9517(81)90272-4`. URL: `https://www.sciencedirect.com/science/article/pii/0021951781902724`.

[UN09] Muhammad Usman and Aamer Nadeem. "Automatic generation of Java code from UML diagrams using UJECTOR". In: *International Journal of Software Engineering and Its Applications* 3.2 (2009), pp. 21–37.