Analysis of Generative Chemistries

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Doctor rerum naturalium
(Dr. rer. nat.)

im Rahmen eines binationalen Verfahrens
mit der Syddansk Universitet, Odense, Dänemark

im Fachgebiet
Informatik

vorgelegt
von Cand.Scient. Jakob Lykke Andersen
geboren am 27. August 1986 in Assens, Dänemark.

Die Annahme der Dissertation wurde empfohlen von:
1. Professor Dr. Jørgen Bang-Jensen (Syddansk Universitet, Odense, Dänemark)
2. Professor Dr. Peter Dittrich (Universität Jena)
3. Professor Dr. Martin Middendorf (Universität Leipzig)

Die Verleihung des akademischen Grades erfolgt mit Bestehen der Verteidigung am 19.11.2015 mit dem Gesamtprädikat summa cum laude.

Abstract

For the modelling of chemistry we use undirected, labelled graphs as explicit models of molecules and graph transformation rules for modelling generalised chemical reactions. This is used to define artificial chemistries on the level of individual bonds and atoms, where formal graph grammars implicitly represent large spaces of chemical compounds. We use a graph rewriting formalism, rooted in category theory, called the Double Pushout approach, which directly expresses the transition state of chemical reactions. Using concurrency theory for transformation rules, we define algorithms for the composition of rewrite rules in a chemically intuitive manner that enable automatic abstraction of the level of detail in chemical pathways. Based on this rule composition we define an algorithmic framework for generation of vast reaction networks for specific spaces of a given chemistry, while still maintaining the level of detail of the model down to the atomic level. The framework also allows for computation with graphs and graph grammars, which is utilised to model non-trivial chemical systems. The graph generation relies on graph isomorphism testing, and we review the general individualisation-refinement paradigm used in the state-of-the-art algorithms for graph canonicalisation, isomorphism testing, and automorphism discovery.

We present a model for chemical pathways based on a generalisation of network flows from ordinary directed graphs to directed hypergraphs. The model allows for reasoning about the flow of individual molecules in general pathways, and the introduction of chemically motivated routing constraints. It further provides the foundation for defining specialised pathway motifs, which is illustrated by defining necessary topological constraints for both catalytic and autocatalytic pathways. We also prove that central types of pathway questions are NP-complete, even for restricted classes of reaction networks. The complete pathway model, including constraints for catalytic and autocatalytic pathways, is implemented using integer linear programming. This implementation is used in a tree search method to enumerate both optimal and near-optimal pathway solutions.

The formal methods are applied to multiple chemical systems: the enzyme catalysed β-lactamase reaction, variations of the glycolysis pathway, and the formose process. In each of these systems we use rule composition to abstract pathways and calculate traces for isotope labelled carbon atoms. The pathway model is used to automatically enumerate alternative non-oxidative glycolysis pathways, and enumerate thousands of candidates for autocatalytic pathways in the formose process.

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angeführten Quellen und Hilfsmittel benutzt und sämtliche Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, und alle Angaben, die auf mündlichen Auskünften beruhen, als solche kenntlich gemacht. Ebenfalls sind alle von anderen Personen bereitgestellten Materialen oder erbrachten Dienst- leistungen als solche gekennzeichnet.

Tokyo, den 13. April 2016

Jakob Lykke Andersen

Contents

List of Figures

List of Tables

Preface

This thesis was jointly supervised (cotutelle) by Daniel Merkle, University of Southern Denmark, Denmark and Peter F. Stadler, Leipzig University, Germany. The research has been conducted also in close collaboration with Christoph Flamm, University of Vienna, Austria.

The implementation of the methods in this thesis was initially based on the Graph Grammar Library (GGL) [Mann *et al.* 2013a, Mann *et al.* 2013b], but has turned into a new software package called MedØlDatschgerl. As of March 2016, part of this package has been publicly released as MedØlDatschgerl 0.5 [Andersen 2016, Andersen *et al.* 2016].

Contributory Publications

This thesis is based primarily on the following publications and manuscripts.

- Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Inferring chemical reaction patterns using rule composition in graph grammars* [Andersen *et al.* 2013b], see Chapter 7.

- Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *50 Shades of Rule Composition: From Chemical Reactions to Higher Levels of Abstraction* [Andersen *et al.* 2014b], see Chapters 7 and 11 and Sections 12.3 and 13.1.

- Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Generic Strategies for Chemical Space Exploration* [Andersen *et al.* 2014c], see Chapters 9 and 14

- Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Chemical Transformation Motifs — Modelling Pathways as Integer Hyperflows* [Andersen *et al.* 2015a], see Chapter 10.

- Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Maximizing output and recognizing autocatalysis in chemical reaction networks is NP-complete* [Andersen *et al.* 2012], see Section 10.3.

Additional related material can be found in [Andersen *et al.* 2014a, Andersen *et al.* 2013a, Andersen *et al.* 2015b].

Acknowledgements

A large number of people have in some way contributed to making my time as a Ph.D. student great. First and foremost an enormous thanks to Daniel Merkle for a lot of things, but above all for being an awesome advisor. I also sincerely thank Christoph Flamm and Peter F. Stadler for all the discussions and supervision (whether it be official or unofficial).

Throughout most of 2013 I was at the university in Leipzig and in the spring of 2014 I was at the university in Vienna, and I wish to thank the friendly people in both places for making my visits an enjoyable experience. I also want to thank all of IMADA, and especially my former and current office mates, for a wonderful environment. For proofreading I would also like to thank Uffe Thorsen, Rojin Kianian, and Anders Skovgaard Knudsen. Last but not least I am grateful for all the help and support from my family and friends.

Chapter 1

Introduction

The modelling of chemical systems is the basis for many areas of research, for example the study of living organisms [Savinell & Palsson 1992], the efficient synthesis of drugs [Grzybowski *et al.* 2009], and the question of the origin of life [Ruiz-Mirazo *et al.* 2014]. Chemical reaction networks are central components in this modelling, and they are available from various databases such as KEGG [Kanehisa *et al.* 2012] that focus on metabolic networks. Each of the databases are however very concrete models of certain chemistries, e.g., limited to the metabolism of specific organisms. If we wish to study chemical systems outside their scope, for example hypothetical prebiotic chemistries, we must have different formalisms to construct the networks we are interested in.

Several formalisms for working with artificial chemistries have been developed [Dittrich *et al.* 2001], e.g., using rewriting techniques on terms or strings to generate molecules using a rule-based approach. Though, in many of the formalisms the molecules are modelled as quite abstract entities, such as λ-expressions. It is common to model molecules as undirected graphs, where atoms and bonds are directly expressed as vertices and edges. In fact, the term "graph" first appeared in the context of describing molecules [Sylvester 1878]. Formal models for the transformation of graphs has been developed since the early 1970s [Corradini *et al.* 1997], but it is only recently that graph rewriting and the modelling of chemistry have been combined to define artificial chemistries [Benkö *et al.* 2003]. The use of graph rewriting allows for specification of chemistries as concise grammars, from which vast chemical reaction networks can be generated. In [Yadav *et al.* 2004] the potential of a chemical graph transformation system was highlighted, and libraries has since been developed [Mann *et al.* 2013a, Rosselló & Valiente 2005].

In this thesis we investigate the details of using graph grammars for modelling chemistry, where we develop the fundamental methods necessary to construct a comprehensive chemical graph transformation system. While graphs are commonly modelled as molecules, there is no common model, and widely used formats such as SMILES [Weininger 1988] and InChI [Heller *et al.* 2015] do not even have a published formal specification for the specific models they use. We define a simple molecule model, suitable for generative chemistries, and briefly contrast it with the published material on the SMILES and InChI models. With the graph grammar approach and the formal molecule model we define a framework for automatic generation of large reaction networks for

spaces within given chemistries.

A reaction network, whether it be from a database or automatically generated from a graph grammar, simply represents the set of potential molecules and reactions in a chemical system. Several computational methods have been developed for analysing the topology of such networks, e.g., by finding different notions of chemical pathways. Well-known examples of pathway models are Flux Balance Analysis (FBA) [Orth *et al.* 2010], Elementary Flux Modes (EFM) [Schuster *et al.* 2000], and Extremal Pathways (ExPa) [Klamt & Stelling 2003]. However, EFM and ExPa only characterises certain basic pathways and while FBA can represent general pathways it uses real valued flux vectors which can not be used for low-level reasoning of the flow of molecules, without fundamentally changing the method. We introduce a comprehensive general pathway model that allows for this type of reasoning, and we make a detailed comparison to FBA. This mechanistic reasoning of the molecule flow in our model can be extended to the atomic level when the networks are generated with graph grammars, making it even possible to trace individual atoms throughout pathways. The methods presented here thus provide a foundation for a computational counterpart to the isotope labelling experiments from the wetlab [Sauer 2006, Zamboni 2011].

Certain pathways indicate self-replicating behaviour, i.e., autocatalysis [Bissette & Fletcher 2013], which in particular is interesting when studying living systems or prebiotic chemistries [Eschenmoser 2007a]. Autocatalytic subsystems has even been hypothesised to be a key part of the origins of life [Kauffman 1995]. In our pathway model we introduce necessary topological constraints for pathways to be considered autocatalytic. This is the first step towards automatic characterisation of pathways that implement higher levels of organisation, such as chemical hypercycles [Eigen & Schuster 1977].

Overview

The thesis is arranged into four parts; the first three parts contains the main description of algorithms and models of chemistry. In the fourth part we illustrate how the methods can be applied to analyse chemical systems.

Part I: Graphs and Molecule Modelling In Chapter 2 we elaborate on this issue and define a specific molecule model based on labelled graphs. An integral part of graph transformation is to perform pattern matching on graphs (monomorphism enumeration) and decide when two graphs represent the same molecule (graph isomorphism). In Chapter 3 we introduce the formalisations of these two problems, and briefly review the known algorithms for solving them. Further, in Chapter 4 we describe an algorithmic framework for finding symmetries and deriving canonical representations of graphs. In cheminformatics there are multiple interchange formats for molecules of which SMILES and InChI are prominent examples. As the formats are essential for

precise communication of data we briefly describe the two formats and the lack of formality in their published descriptions.

Part II: Graph Transformation and Chemical Reactions The formal graph rewriting approach we use, called the Double Pushout (DPO) approach, is based on category theory. In Chapter 6 we introduce basic elements of category theory and define a variation of the DPO approach, suitable for modelling chemistry. Contrary to the usual setting of rewriting where a single object is transformed into another object, the chemical rewriting requires rewriting of a collection of objects in order to model the merging and splitting of molecules. The formal description of rewriting with chemical graph grammars is described in Section 6.5. For working with transformation rules, e.g., to create efficient algorithms we in Chapter 7 define methods for composing rules in a chemically intuitive manner. We then use rule composition to define an alternative method for transforming graphs.

Part III: Chemical Reaction Networks In Chapter 8 we introduce basic terminology for chemical reaction networks as modelled by directed multi-hypergraphs, and discuss the ambiguities of defining paths and cycles in hypergraphs. Chapter 9 builds on Part II, by defining an algorithmic framework for computing with graph transformations and the simultaneous generation of chemical reaction networks. In Chapter 10 we describe a general model for chemical pathways. An integer linear programming formulation of the pathway definition is presented, and we prove that certain pathway queries are NP-complete. In Section 10.4 we compare our pathway model with Flux Balance Analysis.

Part IV: Applications The combination of generative chemistries on an atomic level and the inference of pathways gives the potential for *in silico* simulation of isotope labelling experiments. In Chapter 11 we describe the first step towards this goal, where we use rule composition to trace atoms through a sequence of transformations. The method is in the same chapter used on an enzyme catalysed multi-step reaction, while we in the following chapter use it in sugar chemistry and on variations of a metabolic pathway. A model of the formose process is analysed in Chapter 12, where we enumerate autocatalytic cycles, calculate carbon traces, and model the borate inhibited chemistry. In Chapter 13 we analyse the carbon traces of two variations of the glycolysis pathway, and enumerate alternative non-oxidative pathways. As an illustration of the versatility of our methods we apply them to a non-chemical graph transformation game in Chapter 14.

The modelling frameworks and algorithms in this thesis have been implemented in a software package. In Appendix A we briefly outline the features of

this software. It is currently in preparation for release as an open source project. Finally, in Chapter 15 we discuss future extensions of the work presented in this thesis.

1.1 Notation

In the following sections we introduce notation that will be used throughout the thesis. Additional notation is introduced as needed.

1.1.1 Multisets

Since we need to use both normal sets and multisets we introduce the following notation for multisets. When constructing a multiset we use double curly brackets (i.e., $\{\!\{\dots\}\!\}$) to distinguish them from normal set constructors, $\{\dots\}$. We use $|Q|$ to denote the cardinality of Q that include duplicate objects. In iteration contexts we introduce a multi-membership operator, $\in_m$, that yields all elements of a multiset. Thus, the cardinality function could be defined as $|Q| = \sum_{q \in_m Q} 1$. For single element q we use $m_q(Q)$ to denote the number of occurrences of q in Q, which can also be described as $m_q(Q) = |\{\!\{a \in_m Q \mid a = q\}\!\}|$.

1.1.2 Graphs

An (un)directed graph $G = (V, E)$ is an ordered pair of the vertex set V and the edge set $E \subseteq V \times V$. For multigraphs the edge collection E is a multiset. We may refer to the vertices as V_G and similarly for the edges as E_G.

For a directed graph $G = (V, E)$ and an edge subset $E' \subseteq E$ we use $\delta^+_{E'}(v)$ (respectively $\delta^-_{E'}(v)$) to denote the subset of out-edges (in-edges) of vertex v contained in E'. If G is a multigraph the return value of δ^+ (δ^-) is still a set of edges, and not a multiset of edges. When E' is the complete edge set the simpler notation $\delta^+(v)$ and $\delta^-(v)$ is used.

Vertex degrees are written with "d" instead of "δ", i.e., $d^+_{E'}(v)$ and $d^-_{E'}(v)$ for out- and in-degrees of vertices in directed graphs, and $d_{E'}(v)$ for undirected graphs. For directed graphs we also use $d_{E'}(v) = d^+_{E'}(v) + d^-_{E'}(v)$.

For a vertex $v \in V$ and a vertex subset $V' \subseteq V$ we use $d(v, V')$ to denote the degree of v restricted to adjacent vertices in V'.

1.1.3 Chemical Reactions

A chemical reaction is a transformation of one collection of molecules into another, e.g., transforming 2 G3P and 1 H_2O into 1 F6P and 1 P_i. We can formally say that a reaction is an ordered pair (E, P) of multisets of molecules, e.g., $(\{\!\{\text{G3P}, \text{G3P}, \text{H}_2\text{O}\}\!\}, \{\!\{\text{F6P}, \text{P}_\text{i}\}\!\})$. We call the molecules in E *educts* and

those in P *products*. Reactions are usually written as chemical equations, instead of using the multiset notation, e.g.,

$$2\,\mathrm{G3P} + \mathrm{H_2O} \longrightarrow \mathrm{F6P} + \mathrm{P_i}$$

1.2 First-order Terms and Unification

In generic models of (abstracted) chemistry we find it useful to label graphs with not just character strings but with variables and more complex structures. For the encoding of such structures we use first-order terms in the style known from Prolog, and the associated algorithm techniques such as syntactic unification. General material on logic programming and Prolog can be found in [Nilsson & Maluszynski 1995], and for various types of unification see, e.g., [Baader & Snyder 2001] and [Knight 1989]. In the following we briefly state the definitions and terminology needed for the following chapters.

Let $\mathcal{F}$ denote a set of functor symbols and $\mathcal{V}$ a set of variable symbols. The set of first-order terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is then defined as the smallest set such that:

- all variables are terms, i.e., $V \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$,
- and for all arities $n \in \mathbb{N}_0$, functor symbols $f \in \mathcal{F}$, and terms $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$

A substitution σ is a mapping $\{v_1 \mapsto t_1, \ldots, v_{|\sigma|} \mapsto t_{|\sigma|}\}$ of variables to terms. It can be applied to a term t by replacing all occurrences of each v_i in t with the corresponding term t_i. In short we write the application of a substitution as $\sigma(t)$. If a substitution only maps variables into variables such that it is a bijection, then it is called a *renaming*. In algorithmic contexts we also refer to a substitution as a set of *variable bindings*.

Between two terms t_1 and t_2 we use the following relations:

- Equality of t_1 and t_2 is written $t_1 = t_2$. For example $\mathtt{f}(X) = \mathtt{f}(X)$, but $\mathtt{f}(X) \neq \mathtt{f}(Y)$ when X and Y are different variable symbols.
- t_2 is *at least as general* as t_1, written as $t_1 \preceq t_2$ or $t_2 \succeq t_1$, if there exist a substitution σ such that $\sigma(t_2) = t_1$. For example, $\mathtt{f}(A, B) \succeq \mathtt{f}(X, X)$ because the terms become equal when A and B are replaced with X in the left term.
- t_1 and t_2 are *isomorphic*, written $t_1 \cong t_2$, if both $t_1 \preceq t_2$ and $t_2 \preceq t_1$. Equivalently, $t_1 \cong t_2$ if there exists a renaming σ such that $t_1 = \sigma(t_2)$. Thus $\mathtt{f}(A, B) \cong \mathtt{f}(X, Y)$ and $\mathtt{f}(A, B) \cong \mathtt{f}(B, A)$, but $\mathtt{f}(A, B) \ncong \mathtt{f}(Z, Z)$.
- t_1 and t_2 are *unifiable*, written $t_1 \overset{u}{=} t_2$, if there exist a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. Such a substitution is called a *unifier* of t_1 and t_2.

For example, a unifier for $\mathtt{f}(X, \mathtt{g}(Y))$ and $\mathtt{f}(Z, Z)$ is $\{X \mapsto \mathtt{g}(Y), Z \mapsto \mathtt{g}(Y)\}$.

The *most general unifier* (mgu) of two terms t_1 and t_2 is the unifier σ such that for any other unifier σ', we have $\sigma(t_1) \succeq \sigma'(t_1)$. That is, the unifier σ produces the most general terms of all unifiers. For example, is $\sigma' = \{X \mapsto \mathtt{g}(\mathtt{a}), Y \mapsto \mathtt{a}, Z \mapsto \mathtt{g}(\mathtt{a})\}$ is not the mgu of $t_1 = \mathtt{f}(X, \mathtt{g}(Y))$ and $t_2 = \mathtt{f}(Z, Z)$ because there is another unifier $\sigma = \{X \mapsto \mathtt{g}(Y), Z \mapsto \mathtt{g}(Y)\}$, and $\sigma(t_1) = \mathtt{f}(\mathtt{g}(Y), \mathtt{g}(Y))$ which is more general than $\sigma'(t_1) = \mathtt{f}(\mathtt{g}(\mathtt{a}), \mathtt{g}(\mathtt{a}))$.

Deciding if the three relations $t_1 \cong t_2$, $t_1 \preceq t_2$, and $t_1 \overset{u}{=} t_2$ hold can be seen as different levels of pattern matching with isomorphism for exact matching, specialisation/generalisation for one-sided matching, and unification for two-sided matching. Assuming we have an algorithm for computing the most general unifier σ of t_1 and t_2, if it exists, we can from σ also see if the terms are isomorphic by checking if σ is a renaming. A variant of such a unification algorithm can also decide if $t_1 \succeq t_2$ by performing the unification but disallowing binding of variables in t_2.

1.2.1 Implementation

We use first-order terms as labels on graphs, such that variables are shared throughout the same graph object but disjoint from variables in other graphs. Functor symbols are however globally shared, so for a graph G we have the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V}_G)$. In our implementation we use the basic design of the Warren Abstract Machine (WAM) [Warren & Center 1983], where we associate a term heap $Heap(G)$ with each graph G. For the full details of term heaps and the WAM we refer to [Aït-Kaci 1991], and in the following we only describe the basic features that we use.

A term heap is an array where each element is in one of three formats:

- $\mathtt{f}/n$: a functor cell, indicating a term $\mathtt{f}(t_1, t_2, \ldots, t_n)$. The next n cells represent the subterms $t_1, t_2, \ldots, t_n$.
- `STR` a: a pointer to a functor cell located at address a in the heap.
- `REF` a: a variable cell, pointing to an address a in the heap. If `REF` a is located at address a, then the cell represents an unbound variable. Otherwise, the cell represents a variable bound to the term at address a.

Figure 1.1 shows an example of this term representation scheme. Note that variable names do not exist in this scheme, and limiting variable scopes to individual heaps is thus trivially fulfilled. Labelling a graph with terms amounts to associating addresses of terms in the heap with each vertex and edge.

For certain algorithms, e.g., subgraph matching, we need to bring the terms from two graphs into the same scope before syntactic unification can

1	g/2	
2	REF	2
3	a/0	
4	h/1	
5	REF	2
6	f/2	
7	STR	1
8	STR	4

Figure 1.1: Representation of the term $\mathtt{f}(\mathtt{g}(X, a), \mathtt{h}(X))$ in a term heap, following the design of the Warren Abstract Machine. The term is rooted at address 6.

proceed. For example, when finding a term-labelled graph monomorphism $m\colon G \to H$ we merge $Heap(G)$ and $Heap(H)$ by letting a copy of $Heap(H)$ play the role of the heap in a WAM and let a copy of $Heap(G)$ act as the temporary register bank X in the same WAM (see [Aït-Kaci 1991]). After unification the resulting heap and register bank, potentially with additional variable bindings, is then used for the label mapping in the morphism m.

Part I

Graphs and Molecule Modelling

Chapter 2

Molecules as Labelled Graphs

Molecules can be described in many different ways, depending on the specific properties of interest. A common method is to describe molecules as graphs, where vertices play the role of atoms and edges model chemical bonds. In this chapter we first describe an abstract model of molecules as graphs, suitable for generative chemistry. Even though this model may be adequate for the explicit modelling of molecules in this work, we however do not limit the discussion of methods to this model. We therefore next describe how the molecule model is represented in the more general class of graphs labelled with first-order terms or character strings. From a practical point of view, this enables abbreviation of large uninteresting molecule fragments (e.g., in coenzyme A and ATP) into smaller subgraphs with special labels. A generic modelling framework also allows for easy encoding of higher-level systems, such as DNA computation [Andersen *et al.* 2014a], or even systems without relation to chemistry.

2.1 Molecule Model

The choice of molecule model must be influenced by how it is going to be used. In this work we use graph rewriting to instantiate reactions, which is done by finding specific substructures of molecules and transform them independently of the surroundings. A guiding principle for the model is therefore that chemical properties must arise from locally encoded data, which in particular have an effect on how aromaticity is modelled.

The model is also influenced by the fact that we already have a mechanism in mind for specifying the change of a molecule. For example, we regard tautomers as separate molecules because we can model their inter-conversions as transformation rules, whereas, e.g., InChI as discussed in Section 5.2, instead use a layered approach to represent the tautomers either as a single compound or as multiple molecules.

We frequently need to represent substructures of molecules, as well as proper complete molecules, but we rarely need to distinguish between the two concepts. Therefore we use the following common definition, without lower bounds on atom valences.

Definition 2.1 (Molecule)**.** Let Ω_B be the set of bond types $\{\textsc{Single}, \textsc{Double}, \textsc{Triple}, \textsc{Aromatic}\}$, and $\Omega_Z = \{\mathrm{H}, \mathrm{He}, \mathrm{Li}, \mathrm{Be}, \mathrm{B}, \mathrm{C}, \mathrm{N}, \mathrm{O}, \ldots, \mathrm{Uuo}\}$ the set of chemical symbols [Wieser *et al.* 2013]. A molecule is a labelled, connected,

Z	Charge	Valid neighbourhoods
H	0	{{Single}}
	+1	{{}}
C	0	{{Single, Single, Single, Single}}, {{Single, Single, Double}}, {{Single, Triple}}, {{Double, Double}}, {{Single, Aromatic, Aromatic}}, {{Aromatic, Aromatic, Aromatic}}
N	0	{{Single, Single, Single}}, {{Single, Double}}, {{Triple}}, {{Aromatic, Aromatic}}, {{Single, Aromatic, Aromatic}}
	+1	{{Single, Single, Single, Single}}
O	−1	{{Single}}
	0	{{Single, Single}}, {{Double}}
P	0	{{Single, Single, Single, Double}}

Table 2.1: Partial table of chemically valid multisets of incident bond types for atoms with a given symbol and formal charge.

simple, and undirected graph $G = (V, E, l^V, l^E)$, with $l^V : V \to \Omega_Z \times \mathbb{Z}$ as the function labelling vertices with a chemical element and a charge, and $l^E : E \to \Omega_B$ as the bond type function.

We have also omitted upper bounds on atom valences due to the complexity of correctly defining such constraints. In the following chapters we see that only the algorithms for inter-conversion with external molecule formats (Chapter 5) require explicit neighbourhood constraints.

Even though we do not enumerate all chemical correct neighbourhoods we have tabulated a subset of them in Table 2.1, and we informally refer to a molecule as being *complete* if all vertices have a neighbour that a chemist would consider valid. A molecule that is not complete is *partial*, which we also refer to as a *molecule fragment*.

The molecule model also ignores the embedding of molecules in 3D space, meaning it can be used to distinguish between structural isomers and but not stereoisomers. However, it seems likely that the model can be extended with stereochemical data (see Section 15.1). An introduction of stereochemical properties to the molecule model would additionally entail a full specification of the neighbourhood constraints, such that a *complete molecule* can be formally defined.

2.1.1 Bond Types and Connectedness

We mostly consider biochemistry, and thus limit our model to the four bond types in Ω_B. The Single, Double, and Triple bonds are local bonds that

can be said to consist of respectively 1, 2, and 3 pairs of electrons, while the AROMATIC bond is used for modelling the non-local aromatic property (see Section 2.1.3).

The definition of a molecule uses normal undirected graphs, and it is therefore not possible to directly model three-centre bonds found for example in the molecule diborane. However, such bonds could possibly be modelled by a 3-cycle of the participating vertices, using a new bond type.

We assume that any bonding is modelled as edges, which means that every molecule is a connected graph. If we wish to consider the ions Na^+ and Cl^- as a single compound, we would have to introduce a new bond type for ionic bonds. We also assume that an atom never self-interact or interact with another atom in multiple independent manners, which makes the molecule graphs simple.

2.1.2 Notation and Visualisation

We adopt the usual chemical notation for charged atoms, i.e., N, N^+, and N^{2+}, for a nitrogen atom with charge 0, +1, and +2, respectively. For visualisations of molecules we mostly follow the normal chemical schemes, though we always depict aromatic bonds explicitly with a solid and a dashed line (see also Table 2.4). This means that we do not use the Kekulé form of molecules in figures, and thus Figure 2.3a and Figure 2.3b show two distinct molecules. As usual in chemical depictions we may omit uncharged hydrogens with only a SINGLE bond as incident edge, and instead annotate the neighbour with an 'H' when a single such neighbour was omitted, or 'H_k' for k omissions. When a vertex models an uncharged carbon we may depict it without the explicit 'C' label, and further if such a carbon vertex adheres to the neighbourhood constraints in Table 2.1 we may omit depicting both the neighbouring hydrogens and the vertex label itself. Figure 2.2 shows an example of the resulting four different depiction schemes for hydrogens and carbons.

2.1.3 Aromaticity

The definition of aromaticity is quite complicated, if such a definition even exist [Stanger 2009]. However, aromaticity can be found in important biochemical molecules, e.g., the nucleotides, and so it must be included in our molecule model in some form. In the following we describe a simplified view of aromaticity and various modelling issues.

When part of a compound is aromatic there are electrons that are delocalised, and they can be said to participate in multiple bonds at the same time. A classical example of an aromatic compound is benzene (see Figure 2.3a) where the six carbon-carbon bonds in some sense are all hybrid single-double bonds. Benzene, and other aromatic compounds, are often visualised as *Kekulé structures* which do not use a special aromatic bond type, but alternating single

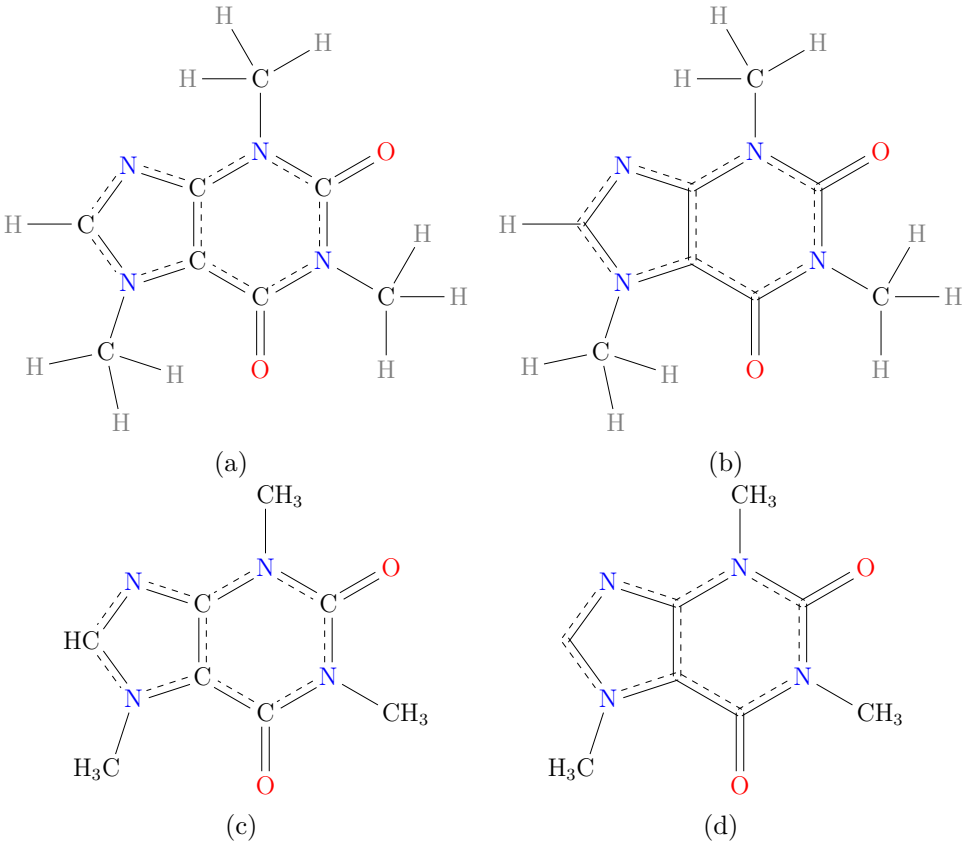


Figure 2.2: Visualisation of caffeine using different schemes for hydrogen and carbon atoms: (a) explicit hydrogens and carbons, (b) vertex labels for uncharged carbons omitted, (c) hydrogens annotated on their neighbours, and (d) the usual chemical depiction with both omitted carbon labels and special hydrogen annotation. It is assumed that carbon atoms without a label have a neighbourhood conforming to Table 2.1.

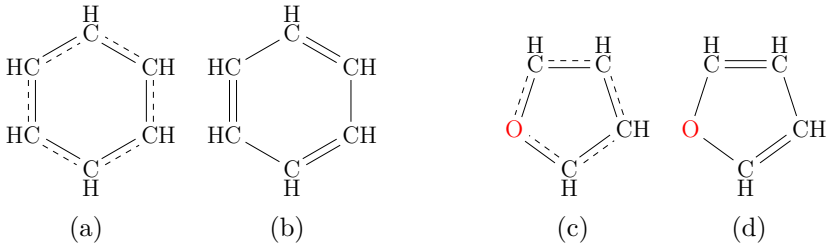


Figure 2.3: The aromatic compounds benzene (a) and furan (c), visualised using a special aromatic bond type (see Section 2.1.2). Commonly they are instead visualised using their Kekulé structure, (b) and (d). Note that some compounds have multiple Kekulé structures, such as derivatives of benzene.

and double bonds. The Kekulé structure for Benzene is shown in Figure 2.3b. In terms of graph theory it could seem that finding a Kekulé structure can be done by finding any maximum cardinality matching among the aromatic bonds, assigning DOUBLE to the edges in the matching, and SINGLE to the remaining edges. However, a simple counterexample for this hypothesis is the molecule furan, visualised in Figure 2.3c, where the vertex modelling oxygen can not be incident to the matching.

The furan molecule also indicates that the opposite conversion, from a Kekulé structure to explicit aromatic bonds may not be trivial, as we can not simply search for ring systems with alternating single and double bonds. Additionally, as cyclooctatetraene and cyclobutadiene illustrate, not all alternating bond systems are aromatic. A widely use rule to avoid this problem is Hückel's rule where the number of certain electrons must satisfy a formula. However, this rule does not work in general [Roberts *et al.* 1952]. Machine learning methods has been developed [Mann *et al.* 2013a, Mann *et al.* 2013b] for the prediction of aromaticity, though these naturally depend on an existing database of compounds.

Double bonded atoms are usually closer together than single bonded atoms, but in benzene all six carbon-carbon bonds have the same length. Additionally, the reactivity for aromatic compounds is in general not the same as if the compound was actually the Kekulé structure. Due to these behavioural differences we chose to explicitly model aromaticity through the AROMATIC bond type. We therefore regard the Kekulé structures as different molecules from their aromatic counterpart. As previously outlined we do not impose valence constraints on the molecule model, and for aromaticity we likewise do not decide the validity of how the aromatic bond type is used. An extension of the model with stereochemical properties, Section 15.1, could potentially give rise to more precise formal constraints for aromaticity.

2.2 Representation as String- and Term-labelled Graphs

To allow for non-chemical models we generalise to the class of connected, undirected graphs with first-order terms as labels on both vertices and edges. We suggest two encoding schemes; the simple standard encoding which only uses simple character strings, and the extended encoding which uses more complex terms that allow for elaborate modelling.

2.2.1 Term Encoding

In the term encoding we explicitly denote atoms and bonds with functors, such that they can be easily distinguished from non-chemical vertices and edges during matching in mixed models. The chemical symbols and charges

	Encoding		
Bond	String	Term	Visualisation
Single	`-`	`bond(-)`	———
Double	`=`	`bond(=)`	═══
Triple	`#`	`bond(#)`	≡≡≡
Aromatic	`:`	`bond(:)`	- - - - -

Table 2.4: Encoding and visualisation schemes for chemical bonds.

are additionally separated, making it possible to do general pattern matching on the data.

An atom with chemical symbol $z \in \Omega_Z$ and charge $n \in \mathbb{Z}$ is a vertex with the label `atom`(z, n), where `atom` is a functor symbol, and each chemical symbol and charge are distinct constants. Similarly is each bond mapped into a term `bond`(b) for individual bond type constants b. The specific encoding is shown in Table 2.4). Matching, for example, a nitrogen with any charge can thus be done by searching for a vertex where the label unifies with `atom`$(\texttt{N}, K)$, for some free variable K.

2.2.2 String Encoding

Usually we do not need the flexibility offered by the term encoding, and we instead use a simpler encoding scheme that maps all labels into simple character strings. Bonds are encoded according to Table 2.4, while atoms are encoded in the following manner, similar to the standard chemical notation:

- Atoms with charge 0 are encoded as the chemical symbol, e.g., Na as `Na`.
- Atoms with charge $+1$ (resp. -1) are encoded as the uncharged version with `+` (resp. `-`) appended, e.g., Na^+ as `Na+` and Cl^- as `Cl-`.
- Let k be the charge of an atom with $|k| > 1$, the atom is then encoded as the uncharged version with $|k|$`+` (resp. $|k|$`-`) appended for $k > 0$ (resp. $k < 0$). For example N^{2+} as `N2+`.

Chapter 3

Graph Morphisms and Structure Comparison

One of the core operations during graph transformation is to find subgraphs to transform. For chemical compounds this is similar to the problem of searching for substructures. Another essential operation is to decide whether two graphs actually contain the same information, e.g., deciding if two molecules are duplicates of each other. In this chapter we first define different types of graph morphisms, which can model relations between graphs, and secondly we describe details of defining morphisms for graphs labelled with first-order terms. Finally we briefly review the computational complexity of finding various types of morphisms.

In this work we are mostly concerned with transforming and comparing the graphs used for modelling molecules. The following definitions are therefore simplified to suit the category of simple, undirected graphs, optionally with labels. In the following definitions, let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be two such graphs.

Definition 3.1 (Graph Morphisms)**.** A *graph (homo)morphism* $m\colon G \to H$ is a structure-preserving mapping of vertices and edges. That is if $e = (u, v) \in E_G$ then $m(e) = (m(u), m(v)) \in E_H$. Furthermore (see also Figure 3.1)

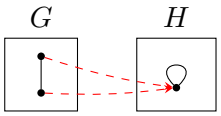
- m is a *monomorphism* if it is injective: $\forall u, v \in V_G, u \neq v \Rightarrow m(u) \neq m(v)$. When a monomorphism exists we may simply write it as $G \subseteq H$ or in the reverse order $H \supseteq G$.

- m is a *subgraph isomorphism* if m is a monomorphism and $(u, v) \in E_G \Leftrightarrow (m(u), m(v)) \in E_H$.

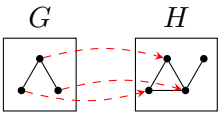
- m is an *isomorphism* if it is a subgraph isomorphism, and is a bijection of the vertices. When an isomorphism exists we say G and H are *isomorphic* and write it as $G \cong H$.

- m is an *automorphism* if G and H refers to the same graph, and m is an isomorphism. We say that m is the *trivial* automorphism when it is the identity morphism id_G.

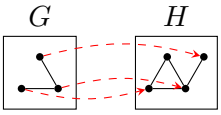
In the following chapters we use detection of isomorphisms for discarding duplicate graphs, and enumeration of monomorphisms for subgraph matching.



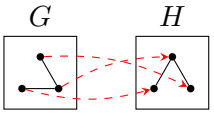
(a) A morphism.



(b) A monomorphism.



(c) A subgraph isomorphism.



(d) An isomorphism.

Figure 3.1: Examples of the graph morphisms from Definition 3.1, visualised as explicit vertex mappings in red. (a) a morphism which is not a monomorphism, (b) a monomorphism which is not a subgraph isomorphism, (c) a subgraph isomorphism which is not an isomorphism, and (d) an isomorphism.

A Note on Terminology

We have chosen the terms "monomorphism" and "subgraph isomorphism", but unfortunately the literature is not consistent in terminology [Conte *et al.* 2004]. Some authors use "subgraph isomorphism" to denote monomorphisms and then "induced subgraph isomorphism" for what we call "subgraph isomorphism".

3.1 Labelled Graph Morphisms

To distinguish among vertices modelling different atom types, and edges modelling different bond types, we label each vertex and edge with auxiliary data. A labelled graph is a tuple $G = (V_G, E_G, l_G^V, l_G^E)$, where (V_G, E_G) is the underlying graph, $l_G^V \colon V_G \to \Omega_V$ is the function labelling vertices with elements from some set Ω_V, and $l_G^E \colon E_G \to \Omega_E$ is the function labelling edges with elements from some set Ω_E. A graph morphism $m \colon G \to H$ on labelled graphs induces the label associations

$$A_V(m) = \{\langle l_G^V(v), l_H^V(m(v))\rangle \mid v \in V_G\}$$
$$A_E(m) = \{\langle l_G^E(e), l_H^V(m(e))\rangle \mid e \in E_G\}$$

Depending on the structure of Ω_V and Ω_E we can then define different kinds of morphisms. For example, if the labels are character strings we simply require that all the associated labels from the morphism are equal, i.e, $s_1 = s_2, \forall(s_1, s_2) \in A_V(m) \cup A_E(m)$.

As a generalisation we label vertices and edges with first-order terms, and direct equality of the terms is not always desired. Given two graphs G and H labelled with first-order terms as defined in Section 1.2, and a morphism $m \colon G \to H$. Let t_G and t_H be the aggregate terms

$$t_G = \texttt{assoc}(l_V(v_1), l_V(v_2), \ldots, l_V(v_{|V_G|}),$$

$$t_H = \texttt{assoc}(l_V(m(v_1)), l_V(m(v_2)), \ldots, l_V(m(v_{|V_G|})), \\ l_E(e_1), l_E(e_2), \ldots, l_E(e_{|E_G|})) \\ l_E(m(e_1)), l_E(m(e_2)), \ldots, l_E(m(e_{|E_G|})))$$

for some arbitrary ordering of V_G and E_G, and a new functor symbol `assoc`. If m is an isomorphism and $t_G \cong t_H$, then G and H not only have the same graph structure, but the labelling is the same, except for renaming of variables. However, if m is a monomorphism we can define different levels of pattern matching by checking if either $t_G \cong t_H$, $t_G \succeq t_H$, or $t_G \overset{u}{=} t_H$. In Chapter 6 we describe how we transform graphs, and the initial step in the transformation is to find a subgraph in the host graph to transform, i.e., to do pattern matching with labelled graphs. Requiring $t_G \cong t_H$ can be interpreted as a check for exact substructure, while $t_G \succeq t_H$ can be used to check if a pattern is less restrictive than another. As noted in Section 1.2, deciding the relation $\succeq$ can be done via one-sided unification. For transforming graphs we, as default, use the full unification requirement, $t_G \overset{u}{=} t_H$, for matches.

3.2 Representational Equality

There is yet another relation between two graphs, which has relevance for isomorphism detection. For two graphs G and H we say that they are *representationally equal*, $G \overset{r}{=} H$, when they are represented in the exact same way. For example, if G and H are stored as adjacency matrices then $G \overset{r}{=} H$ when those matrices are equal. With adjacency lists we can similarly compare the two lists of lists of vertices directly. For both representations we can clearly decide equality in linear time of the size of the data structures. We extend the notion of representational equality to labelled graphs in the natural manner, where we further test the associated labels for representational equality in a suitable manner.

Assume that G and H are represented as adjacency lists, and let the first vertex of each graph be incident to both the second and the third vertex of their corresponding graphs. In G this could be represented as $1\colon 2, 3$, while in H is could be represented as $1\colon 3, 2$. The two graphs are therefore not representationally equal, but as we already have an intrinsic total order of the vertices we can sort the lists of incident edges. We say that an adjacency list on this form is a *globally ordered adjacency list*.

When two graphs are representationally equal then they are necessarily isomorphic, and a specific isomorphism can be constructed simply by mapping vertices with the same index to each other. Note that we can then consider a graph G and a graph G' obtained by permuting the indices of G while changing the underlying representation. If their representations are equal, then we have found an isomorphism between G and G', meaning that the permutation we used to construct G' represents an automorphism on G. In Chapter 4 we

describe an algorithm for computing a canonical representation of a graph, which uses permutations of indices. The canonical form can then be used for subsequent isomorphism testing which can then be done only by checking representational equality.

3.3 Algorithms and Complexity

For this work we are mostly interested in the problems of deciding graph isomorphism and enumerating monomorphisms. Additionally, we are interested in the more specialised problem of deciding if a given graph is isomorphic to any graph in a given set (see Chapter 9 for the context). For an overview of algorithms for both exact and inexact graph matching see [Conte *et al.* 2004].

3.3.1 Monomorphism Enumeration

Given two graphs G and H it is in general NP-complete to decide whether just a single monomorphism $m\colon G \to H$ exists [Cook 1971]. For the restricted case of planar graphs and fixed pattern G the problem can be solved in time $O(n)$ for $n = |G|$ [Eppstein 1999]. All monomorphisms can even be enumerated in time $O(n+k)$, for k monomorphisms. However, we can not in general assume that molecules are planar [III & Maggio 1981], and the algorithm may not efficient in practice.

Due to availability and quality of implementation we use the VF2 algorithm [Cordella *et al.* 2001, Cordella *et al.* 2004] implemented in the Boost Graph Library [Siek *et al.* 2001]. The algorithm is a special variation of tree search algorithms for enumeration of both monomorphisms, subgraph isomorphisms, and isomorphisms. It searches for morphisms by starting from the empty partial morphism and then gradually extending it to a full morphism. An overview of such algorithms can be found in [Solnon 2010], along with a suggestion for a potentially faster variation than VF2. Benchmarks for the family of VF2 algorithms can also be found in [Foggia *et al.* 2001], though only for finding isomorphisms. VF2 is here compared to an algorithm employing bit-vectors [Ullmann 1976], and recently the author has published an improved algorithm [Ullmann 2011] where also specialised methods for molecular graphs are considered.

3.3.2 Isomorphism Testing

Given two graphs G and H it is currently unknown if the problem of deciding whether they are isomorphic can be done in polynomial time. Neither is the problem known to be NP-complete. For graphs with bounded degree, e.g., molecular graphs, the problem is however solvable in polynomial time [Luks 1982]. The result is based on finding the automorphism group of graphs efficiently.

Many of the algorithms for monomorphism enumeration, e.g., VF2, can also be used for deciding isomorphism, but for isomorphism testing there is an additional class of algorithms based on canonicalisation algorithms. With this technique the computational burden is in a pre-computation step on each individual graph, and isomorphism checking can afterwards be done in linear time by checking representational equality (see Section 3.2). This is especially interesting during generation of reaction networks (see Chapter 9), where we iteratively build a collection of non-isomorphic graphs. Each graph in the collection is tested for isomorphism against all new candidates, and pre-computing a canonical form seems desirable. In Chapter 4 we describe a family of canonicalisation algorithms based primarily on [McKay 1981] and [McKay & Piperno 2014b]. The SMILES notation [Weininger 1988] for molecules was originally published with an algorithm [Weininger *et al.* 1989] for canonicalisation. However, as we illustrate in Section 5.1 this algorithm does not work for all molecules in the sense that isomorphic molecules may be assigned different SMILES strings.

In practice we use a combination of VF2 and the incorrect SMILES canonicalisation algorithm. During network generation (see Chapter 9) we check isomorphism for molecules using SMILES, and afterwards we post-process the graphs with VF2 to detect extraneous isomorphisms. For non-molecular graphs we fall back to VF2. We have additionally made initial experiments with a general purpose canonicalisation algorithm, described in Chapter 4.

Chapter 4

Graph Canonicalisation

The graph canonicalisation problem is about finding a canonical order of the vertices in a given graph such that the order is the same no matter how the vertices were ordered initially. As an example, consider Figure 4.1 where a graph G has an initial ordering of the vertices specified by the shown indices. If we reorder the indices of the vertices we can obtain, for example, G_1 or G_2 instead. We have not modified the graph structure so clearly $G \cong G_1 \cong G_2$, but we have implicitly modified the representation of the graph. The globally ordered adjacency lists of the graphs are shown below each graph, and we see that G_1 and G_2 have equal adjacency lists, i.e., they are representationally equal $G_1 \overset{r}{=} G_2$. They are however representationally different from G.

We could now define a total order among adjacency lists and claim that both G_1 and G_2 are "better" than G, e.g., because vertex 1 has fewer neighbours in G_1 than in G, and write $G_1 \overset{r}{<} G$. How specifically the relation $\overset{r}{<}$ is defined is unimportant for the sake of defining a canonicalisation algorithm, but we assume that it defines a total order on the relevant class of graphs. If the graphs are labelled (e.g., molecules) we assume the $\overset{r}{<}$-relation takes the labels into account.

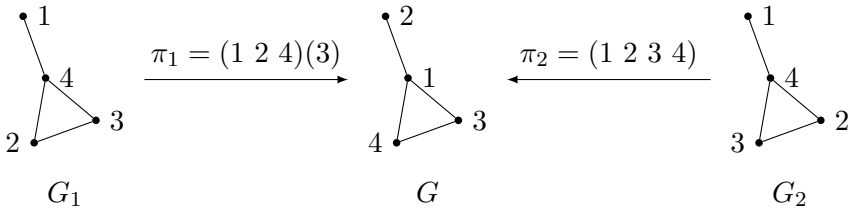


Figure 4.1: Three isomorphic graphs represented as adjacency lists. The underlying indices of the vertices are shown, and the permutations π_1 and π_2 (in cycle notation) describe the relationship between the indices of the graphs. From the adjacency lists we see that G_1 is representationally equal to G_2, and representationally different from G. The permutation $\pi' = \pi_2^{-1} \circ \pi_1 = (2\ 3)(1)(4)$ thus represents an isomorphism from G_1 to G_2.

In general for a graph G with n vertices we could theoretically enumerate all reorderings of the vertices, i.e., $n!$ graph permutations, and select any of the graphs that are smallest according to $\overset{r}{<}$ and use it as the result of canonicalisation. This is clearly a brute-force approach, which in practice is infeasible. It is currently unknown for general graph classes whether a polynomial time algorithm exists. The graph canonicalisation problem has neither been proven NP-hard in general, but if $\overset{r}{<}$ is chosen to order graphs by the lexicographical comparison of their adjacency matrices, then the problem of the finding a smallest graph is NP-hard [Babai & Luks 1983].

In this chapter we describe the individualisation-refinement paradigm, which is the basis of the currently fastest practical algorithms for graph canonicalisation, and the related problem of finding the automorphism group of a graph. The paradigm is additionally the basis of the algorithms used for canonicalisation in the chemical molecule formats SMILES and InChI (see Chapter 5). A selection of fast general algorithms are listed below, with references to descriptions of them.

- **nauty** [McKay 1981, Hartke & Radcliffe 2009, McKay & Piperno 2014b]
- **Traces** [Piperno 2008, McKay & Piperno 2014b]
- **saucy** [Darga *et al.* 2008, Katebi *et al.* 2010]
- **bliss** [Junttila & Kaski 2007]

A short overview of the core ideas is provided by [Hartke & Radcliffe 2009] and [McKay & Piperno 2014a], while the full mathematical details can be found in [McKay 1981], and parts of it in a revised version in [McKay & Piperno 2014b]. The paper [Piperno 2008] notably also includes detailed illustrated examples in its descriptions. The intention in this chapter is to give an unified intuitive understanding of the algorithmic framework, though without the proofs of correctness which can be found in [McKay 1981]. In our description we aim at precisely separating the core of the algorithm from the heuristics used for optimisation.

Throughout we denote the input graph as $G = (V, E)$ where the vertex set is the set of integers $V = \{1, 2, \ldots, n\}$. We generally use the variables u and v to refer to arbitrary vertices, but note that for ease of notation we also use $v_1, v_2, \ldots, v_k$ for an arbitrary sequence of vertices. That is, v_i is not necessarily the vertex with index i, but simply the ith vertex in the context of where it is used.

The graph may be directed or undirected, and may be labelled or not. We assume that a relation $\overset{r}{<}$ is given that induces a total order on the graph class, such that the brute-force strategy described in the beginning correctly gives a canonical form. The relation must thus include comparison of vertex and

edge labels when relevant. In Section 4.3.1 we describe how the labels can be exploited to potentially speed up a canonicalisation algorithm.

The overall idea is to start the algorithm by assuming all vertices are indistinguishable, and then iteratively use local information, e.g., vertex degrees, to partition the vertices and order them. If we reach a point where we can no longer refine the partitioning of the vertices, and some vertices are still indistinguishable, then we forcibly split a partition and again use local information for further refinement. There may be multiple choices when forcibly splitting a partition so we construct a search tree that represents all the choices.

4.1 Preliminary Definitions

4.1.1 Ordered Partitions

For representing an intermediary result of the algorithm we use an *ordered partition* of V. It is a sequence of non-empty sets $\pi = (V_1, V_2, \ldots, V_r)$ such that $V_1, V_2, \ldots, V_r$ forms a partition of V, i.e., $\bigcup_{1 \leq i \leq r} V_i = V$ and $\forall 1 \leq i < j \leq r : V_i \cap V_j = \emptyset$. Each individual V_i is called a *cell*, and cells that only contain 1 vertex are called *trivial* cells. When all cells are trivial the ordered partition is called *discrete*. In the other extreme there is an ordered partition with just a single cell equal to V. This is called the *unit partition*.

The set of all ordered partitions of V is denoted Π. If vertex $v \in V$ is in the ith cell of π we write $cell(v, \pi) = i$. We say that π_1 is *at least as fine* as π_2, and write it $\pi_1 \preceq \pi_2$ [McKay & Piperno 2014b, Section 2.1], if and only if

$$cell(u, \pi_2) < cell(v, \pi_2) \Rightarrow cell(u, \pi_1) < cell(v, \pi_1) \qquad \forall u, v \in V$$

That is, π_1 can be obtained by splitting cells of π_2 while preserving the ordering induced by π_2. When writing examples of ordered partitions we use a special notation, e.g., $[1\ 2 \mid 3]$ which means $(\{1, 2\}, \{3\})$.

The canonicalisation algorithm starts from the unit partition and searches for a discrete partition by splitting cells when some vertices are determined to be "less" than others. A discrete partition can thus represent the canonical form in the sense that if vertex v_i is in cell i, then v_i should have index i in the canonical form. When considering a discrete partition as stored in an array, it will be a map from the canonical indices back to the original indices. For example, consider the discrete partition $\pi_1 = [2 \mid 4 \mid 3 \mid 1]$ as an array of the vertices of the input graph G. At index 1 the vertex 2 is located, indicating it is the first vertex in the canonical form. At index 2 the vertex 4 is located, likewise indicating it is the second vertex in the canonical form. Rewriting π_1 as a permutation in cycle notation we obtain $\pi_1 = (1\ 2\ 4)(3)$. This example is depicted in Figure 4.1.

4.1.2 Permutations

In the following sections we need several concepts from computational permutation group theory, and we refer to [Seress 2003] for a comprehensive treatment of this topic.

Recall that we regard V as the set of integers $\{1, 2, \ldots, n\}$, and let γ be a permutation of V. The image of a vertex v under the permutation γ is denoted v^γ. The application of two successive permutations γ_1, γ_2 on v is written as $(v^{\gamma_1})^{\gamma_2}$ or simply $v^{\gamma_1\gamma_2}$, though note that permutation application is not commutative. The consequence of this notation is that the composition of permutations $\gamma_1\gamma_2$ is interpreted as the application first of γ_1 and then γ_2. We also use the conventional parenthesis notation with explicit composition operators, i.e., $(\gamma_2 \circ \gamma_1)(v) = \gamma_2(\gamma_1(v)) = v^{\gamma_1\gamma_2}$. The inverse of a permutation γ is written as γ^{-1}.

The set of all permutations of V, along with the permutation composition operator, is also known as the symmetric group on n elements, S_n. We say that a permutation $\gamma \in S_n$ *fixes* a vertex $v \in V$, if $v^\gamma = v$. Explicit permutations will generally be written in cycle notation, without fixed elements. For a subset $X \subseteq V$ we define the permutation of X with $\gamma \in S_n$ as $X^\gamma = \{x^\gamma \mid x \in X\}$. For a set of permutations A the group *generated* by A is the smallest group $\langle A \rangle$ that includes A. This group can be created by computing the closure of A under permutation composition. For a vertex $v \in V$ and a permutation group S' we say that the *orbit* of v under S', written $v^{S'}$, is the set of vertices which is the image of v under all permutations in S', i.e., $v^{S'} = \{v^\gamma \mid \gamma \in S'\}$.

In Figure 4.1 we saw how reordering of the vertices can be seen as a permutation of the vertex indices. We extend the notation of permutation application to graphs, such that G^γ, with $\gamma \in S_n$, denotes the permutation of the indices of the vertices in G, and assume the underlying representation to change accordingly. For Figure 4.1, if we assume that the canonicalisation algorithm described below is applied to G, then it may calculate discrete partitions corresponding to π_1 and π_2. During the algorithm it will then further calculate the graphs $G_1 = G^{\pi_1^{-1}}$ and $G_2 = G^{\pi_2^{-1}}$ as candidates for the canonical form.

Recall that a graph automorphism is an isomorphism from a graph to itself, which can be represented by a specific permutation of the indices. This is illustrated in Figure 4.1 where $G_1 \overset{r}{=} G_2$, and the permutation $\pi' = \pi_2^{-1} \circ \pi_1 = (2\ 3)$ is an automorphism on G_1. We are however interested in the corresponding automorphism on G instead. This will be explored in Section 4.3.3. The *automorphism group* of G, denoted $\mathrm{Aut}(G)$, is the subgroup of S_n with all automorphisms of G.

We also extend the application of a permutation to ordered partitions. For a permutation $\gamma \in S_n$ and an ordered partition $\pi = (V_1, V_2, \ldots, V_r)$ we define π^γ to be $(V_1^\gamma, \ldots, V_r^\gamma)$.

4.1.3 Definition of Canonicalisation

In practice we do not necessarily want to compute the actual canonical from of a graph, but rather just the needed permutation of the indices. If we let $\mathcal{G}$ denote the set of graphs, and $\mathcal{S}$ the set of all permutations, a canonicalisation function is then a function $C\colon \mathcal{G} \to \mathcal{S}$ taking a graph G as argument and returning a permutation of the vertex indices. Thus for $\sigma = C(G)$, we can obtain the canonical form as G^σ. The function must fulfil the property described in [McKay & Piperno 2014b, Property C2], that for any permutation $\gamma \in S_n$, where we compute $\sigma' = C(G^\gamma)$, it must hold that

$$G^\sigma \overset{r}{=} G^{\gamma\sigma'}$$

That is, if we canonicalise any permutation of the input graph (which is isomorphic to G) and construct the canonical form, then the two canonical forms are representationally equal. This property is a special case of what is called *isomorphism invariance* of a function. In general we say that a function f involving the vertex set V is isomorphism invariant if a permutation of the vertices results in a corresponding permutation of the output [Piperno 2008, McKay & Piperno 2014b]. This property is required for most of the procedures we use in the canonicalisation algorithm.

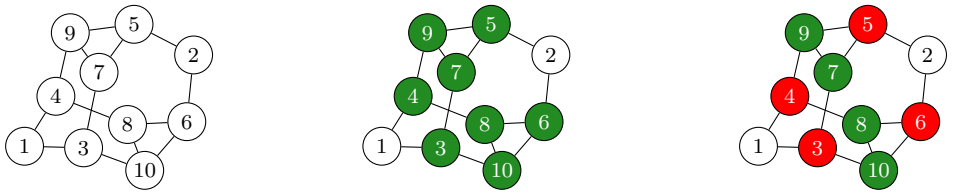
4.2 The Core Algorithm

The core components of the individualisation-refinement approach are partition refinement and vertex individualisation. They will then be used to define a search tree where the canonical form correspond to one of the "best" leaves.

4.2.1 Partition Refinement

In the beginning of the canonicalisation algorithm we assume all vertices are indistinguishable, represented by the unit partition. The idea is now to find properties of the vertices that can be used to iteratively refine the partition, for example degree information as we illustrate here. Note that the choices for refinement can be made completely independent on how the graph relation $\overset{r}{<}$ is defined.

We arbitrarily decide that lower degree is "better" than higher degree, and we can thus trivially refine the unit partition by partitioning the vertices by their degree and ordering the cells from lowest to highest vertex degree. Using the graph in Figure 4.2a as example we start with the unit partition and arrive at the ordered partition illustrated in Figure 4.2b. We can now iterate the degree argument in a more general sense: the vertices in the first cell (white) are adjacent to only some of the vertices in the second cell (green). In general for an ordered partition $\pi = (V_1, \ldots, V_r)$ we can consider two cells V_s and V_t and split V_t according to the values $d(v, V_s)$ for all $v \in V_t$. We say that we



(a) [1 2 3 4 5 6 7 8 9 10] (b) [1 2 | 3 4 5 6 7 8 9 10] (c) [1 2 | 7 8 9 10 | 3 4 5 6]

Figure 4.2: Partition refinement, starting from the unit partition (a), using ascending vertex degree as ordering. (b) the result of shattering the first (and only cell) with itself. (c) the result after shattering the second cell with the first cell of the partition in (b) . This partition is now equitable with respect to the chosen refinement procedure, as no more shattering will refine the partition. The example graph was found in [Piperno 2008].

use V_s to *shatter* V_t [Hartke & Radcliffe 2009]. In Figure 4.2a we had the special case where we used V to shatter itself. Shattering the second cell in Figure 4.2b with the first cell we arrive at Figure 4.2c. Note that in this third ordered partition we can no longer find pairs of cells where shattering results in any cell splits, i.e., $d(v, V_s) = d(u, V_s)$ for all $u, v \in V_t$ for all pairs of cells V_s, V_t. The partition is then said to be *equitable* [McKay & Piperno 2014b].

This particular refinement procedure, shown in [McKay & Piperno 2014b, Algorithm 1], is also known as 1-dimensional Weisfeiler-Lehman refinement [Piperno 2008], which can be seen as a generalisation of Hopcroft's algorithm for DFA minimisation [McKay 1981, Hopcroft 1971]. Other refinement functions can be used instead, e.g., k-dimensional Weisfeiler-Lehman refinement. In general, let $\mathcal{G}$ denote the set of all graphs, then we say that a refinement function must be an isomorphism invariant function $R\colon \mathcal{G} \times \Pi \to \Pi$ that makes the partition strictly finer, or does nothing. Formally, for all graphs $G \in \mathcal{G}$, ordered partitions $\pi \in \Pi$, and permutations $\gamma \in S_n$ we require

$$R(G, \pi) \preceq \pi$$
$$R(G^\gamma, \pi^\gamma) = R(G, \pi)^\gamma$$

This leaves a lot of freedom for defining R, and in the extreme case we find the identity function $R(G, \pi) = \pi$ as a valid choice. In the other extreme it is also valid to use another canonicalisation algorithm as a refinement function, which would guarantee a discrete partition, but does not reduce the problem at hand.

4.2.2 Vertex Individualisation and Target Cell Selection

Assume we are given an ordered partition π where the refinement function can no longer split any cells, and the partition is not discrete. The idea is now to

introduce artificial asymmetry into it by forcibly splitting the cell, and later consider all such artificial splits.

For a graph $G \in \mathcal{G}$, a non-discrete partition $\pi \in \Pi$, and a vertex $v \in V$ belonging to a non-trivial cell of π, we define a strictly finer partition $\pi \downarrow v$ by *individualising* the vertex v. Let π be on the form $(V_1, V_2, \ldots, V_{q-1}, V_q, V_{q+1}, \ldots, V_r)$ with $v \in V_q$. Then

$$\pi \downarrow v = (V_1, V_2, \ldots, V_{q-1}, \{v\}, V_q \backslash \{v\}, V_{q+1}, \ldots, V_r)$$

is the ordered partition where v has been individualised into its own cell, and that cell has arbitrarily been chosen to be smaller than the remainder of V_q. For completeness we also define individualisation for vertices in trivial cells, simply as the identity operation $\pi \downarrow v = \pi$.

After individualisation, if the partition is still not discrete, we can once again use the refinement procedure to obtain a possibly even finer partition. Note that while there is no guarantee that the refinement procedure splits any cells, the individualisation step always splits a cell, unless the partition is already discrete. As every cell must be non-empty there can be at most n cells, and therefore if we repeatedly refine and individualise we eventually obtain a discrete partition.

In the final algorithm we consider the individualisation of every vertex in a specific cell. To find that cell a function called the *target cell selector* is introduced. It is an isomorphism invariant function $Q \colon \mathcal{G} \times \Pi \to 2^V$, that given a graph G and a non-discrete ordered partition $\pi = (V_1, V_2, \ldots, V_r)$, selects a non-trivial cell V_q of π. For example, Q can simply select the first non-trivial cell of π. This is an isomorphism invariant choice as it does not depend on the location of any vertices, but only the structure of π. The target cell selector could also find the first largest non-trivial cell, or the first smallest non-trivial cell. The Traces program uses a yet more complicated target cell selector [McKay & Piperno 2014b, Section 3.2].

For ease of notation we use $\pi_{(v_1, v_2, \ldots, v_k)}$ to denote the ordered partition obtained after repeated rounds of refinement, target cell selection, and vertex individualisation. For the empty sequence we define

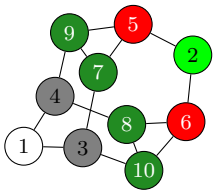
$$\pi_{()} = R(G, \pi)$$

next, if the partition is not discrete, for any $v_1 \in Q(G, \pi_{()})$ we define

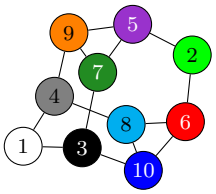
$$\pi_{(v_1)} = R(G, \pi_{()} \downarrow v_1))$$

and generally

$$\begin{aligned} v_{k+1} &\in Q(G, \pi_{(v_1, v_2, \ldots, v_k)}) \\ \pi_{(v_1, v_2, \ldots, v_k, v_{k+1})} &= R(G, \pi_{(v_1, v_2, \ldots, v_k)} \downarrow v_{k+1}) \end{aligned}$$



(a) $\pi_{(1)} = [1 \mid 2 \mid 7\ 8\ 9\ 10 \mid 5\ 6 \mid 3\ 4]$



(b) $\pi_{(1,7)} = [1 \mid 2 \mid 7 \mid 10 \mid 8 \mid 9 \mid 6 \mid 5 \mid 4 \mid 3]$

Figure 4.3: Vertex individualisation and refinement of the ordered partition from Figure 4.2c, using the target cell selector that selects the first non-trivial cell. (a) Individualisation of vertex 1, and subsequent refinement. The first non-trivial cell is $\{7, 8, 9, 10\}$. (b) Individualisation of vertex 7, and subsequent refinement. The partition is discrete and is thus a candidate for creating the canonical form.

As an example, consider the graph from Figure 4.2 and the partition $\pi_{()} = [1\ 2 \mid 7\ 8\ 9\ 10 \mid 3\ 4\ 5\ 6]$ we obtained through refinement. Using the target cell selector that returns the first non-singleton cell we have both vertex 1 and 2 for individualisation. Figure 4.3a shows the result of individualising vertex 1 and the subsequent refinement (see Figure 4.4 for the individualisation of 2). Using the same target cell selector we now have a choice between vertex 7, 8, 9, and 10. Figure 4.3b shows the result when individualising vertex 7. The obtained partition $\pi_{(1,7)}$ is discrete and is a candidate for creating the canonical form. As described in Section 4.1.2 we can interpret the partition as map from the potentially canonical graph back to the input graph. The map is a bijection so we can create the inverse map $\pi^{-1}_{(1,7)}$, which maps vertices in the input graph into the vertices of the potentially canonical graph. The candidate graph is therefore $G^{\pi^{-1}_{(1,7)}}$.

4.2.3 Canonicalisation as a Tree Search

The vertex individualisations are used to introduce artificial asymmetry when the refinement procedure can not split any more cells. However, performing an arbitrary vertex individualisation is not isomorphism invariant so in order to find the canonical form we must consider all individualisations in the chosen cell. Starting from an initial ordered partition this exploration of choices induces a tree of partitions, where the leaves correspond to discrete partitions that are all candidates for constructing the canonical form. An example is shown in Figure 4.4.

Formally, given a graph $G \in \mathcal{G}$ and an initial ordered partition $\pi \in \Pi$ we denote the search tree as $\mathcal{T}(G, \pi)$. Each node in the tree is determined uniquely by the sequence of individualised vertices [McKay & Piperno 2014b, Section 2.3], so we can simply say that the root node is the empty sequence $\tau_r = ()$, which represents the partition $\pi_{\tau_r} = \pi_{()}$. Let $\tau = (v_1, v_2, \ldots, v_k)$ be a node of $\mathcal{T}(G, \pi)$, representing the partition $\pi_\tau = \pi_{(v_1, v_2, \ldots, v_k)}$. If π_τ is discrete then τ is a leaf of $\mathcal{T}(G, \pi)$. Otherwise, let $W = Q(G, \pi_\tau)$ be the target cell of

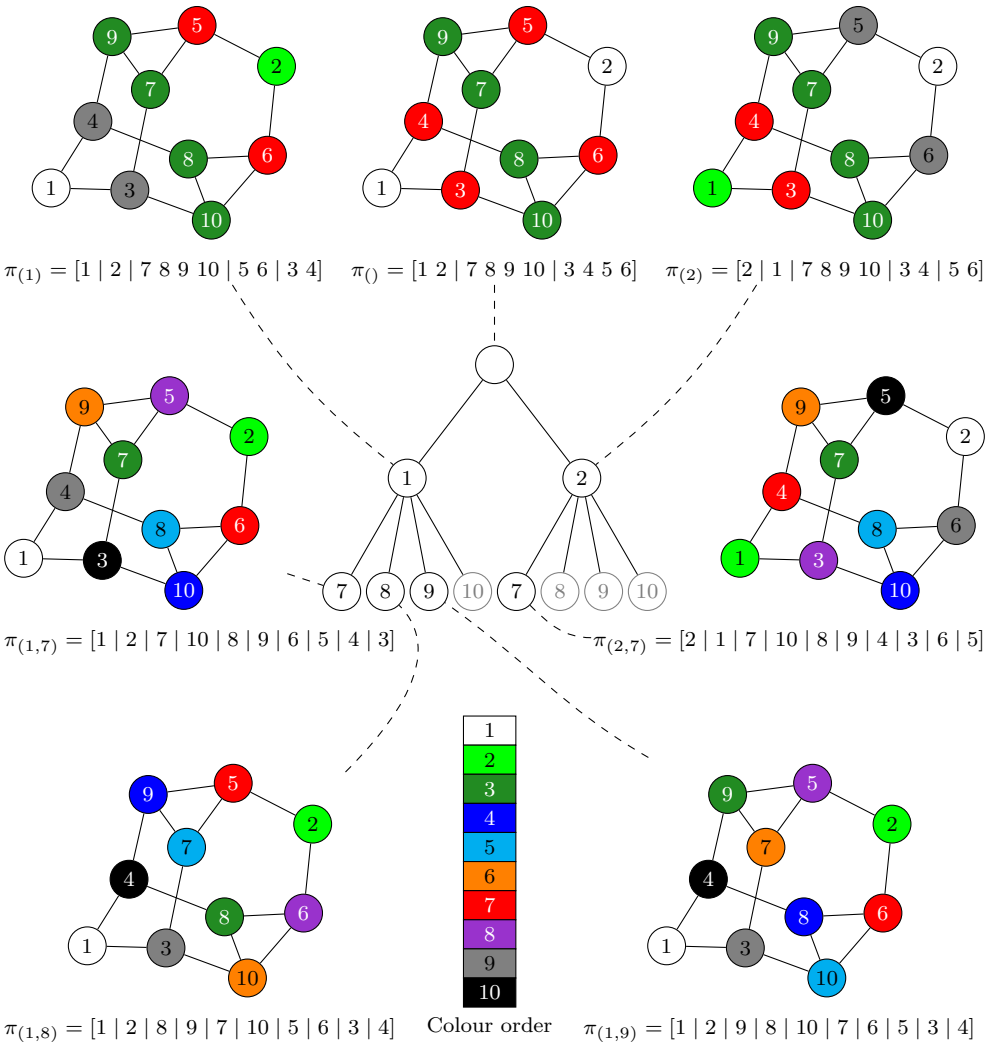


Figure 4.4: A search tree $\mathcal{T}(G, \pi)$ starting with the refinement of the unit partition in the root. Each node in the tree represents a sequence of vertex individualisations, where the latest vertex being individualised is shown in the nodes. For most tree nodes the corresponding partition is shown along with the coloured graph it represents. In the coloured graphs the vertices are labelled with the vertex indices from the input graph, and coloured with "numbers" corresponding to the potential canonical vertex indices. Note that the coloured graphs in the leaves of the left half of the tree (the children of $\tau = (1)$) are all isomorphic. This is also true among the children in the right half of the tree (the children of $\tau = (2)$. However between the halves of the tree, the graphs are not isomorphic. They greyed out nodes correspond to nodes pruned from automorphism discovery (see Section 4.3.3), when depth-first traversal of the tree is used. The example is heavily inspired by [Piperno 2008, Figure 3], though here using different functions for refinement and target cell selection.

π_τ, then the children of τ are

$$\{\tau_{(v_1,v_2,\ldots,v_k,u)} \mid u \in W\}$$

That is, τ has a child representing each choice of vertex individualisation within the selected target cell. In Figure 4.4 most of the tree for our example graph is visualised. Each node of the tree is labelled with the newly individualised vertex. In the following we as a shorthand use $\mathcal{L}(G, \pi)$ to denote the set of leaf nodes of $\mathcal{T}(G, \pi)$.

Recall that we defined a canonicalisation function as taking a graph as an argument and returning the permutation of vertex indices needed to construct the canonical form. Using the search tree defined above we can now define the basic canonicalisation algorithm in the individualisation-refinement approach: Let $R\colon \mathcal{G} \times \Pi \to \Pi$ be a fixed refinement function, and let $Q\colon \mathcal{G} \times \Pi \to 2^V$ be a fixed target cell selector. Further let $\overset{r}{<}\colon \mathcal{G} \to \mathcal{G}$ be a fixed relation that induces a total order of $\mathcal{G}$. The basic canonicalisation function $C\colon \mathcal{G} \to \mathcal{S}$ can then be calculated with the following procedure.

1. Let $G = (V, E) \in \mathcal{G}$ be the input graph to canonicalise.
2. Start from the unit partition $\pi = (V)$, and traverse the tree $\mathcal{T}(G, \pi)$.
3. Find a leaf node τ_c corresponding a $\overset{r}{<}$-smallest graph:
$$\tau_c = \underset{\tau \in \mathcal{L}(G,\pi)}{\arg\min} \left\{ G^{\pi_\tau^{-1}} \right\}$$
4. Let the result be the permutation $\pi_{\tau_c}^{-1}$ mapping vertices of G into the canonical vertices.

In a concrete implementation we naturally also need to define an algorithm to calculate the tree. For example, nauty, saucy, and bliss generates the tree in depth-first order, while Traces uses breadth-first generation combined with *experimental paths* from newly calculated nodes down to a leaf.

4.3 Algorithm Variations and Search Tree Pruning

The description above leaves room for large variations in concrete algorithms, and thereby variations in how fast those algorithms run in practice. For example, we obtain the brute-force algorithm simply by selecting the identity function as the refinement function, i.e., $R(G, \pi) = \pi$. In addition to variations of the core components there are techniques that prune branches from the search while it is being generated, thereby skipping evaluation of the refinement function, which in practice is the most time-consuming component.

The following sections define several optimisation techniques, of which some most of them change the canonical form. For a better understanding

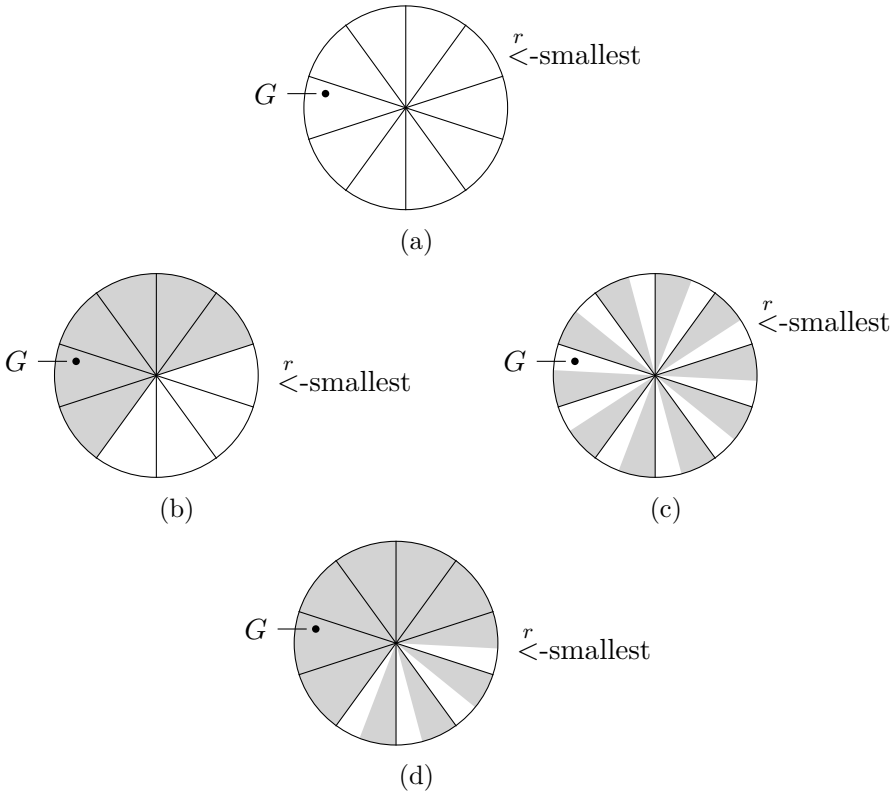


Figure 4.5: Abstract depiction of the set of all permutations of a graph G. Each slice of the set is a subset of permutations that are representational equal. One of the subsets correspond to the $\overset{r}{<}$-smallest representation. (a) If the identity refinement function is used the algorithm explores all $n!$ permutations, and the canonical form is only determined by the $\overset{r}{<}$-relation. (b) With a non-trivial refinement function, some equivalence classes of representations (depicted in grey) will not be considered. The canonical form may now be different, if the global $\overset{r}{<}$-smallest equivalence class was pruned. (c) Using graph automorphisms it is possible to prune among the equivalent graphs, in all classes (see Section 4.3.3). (d) In an efficient implementation both types of pruning would most likely be used.

of how they change the core algorithm we first introduce a different view on canonicalisation. Let G be a graph with n vertices, to be canonicalised in the sense that a "best" permutation from the permutation group S_n must be found. Consider all permuted graphs G^γ for $\gamma \in S_n$, and group them by representational equality as illustrated in Figure 4.5. Using the brute-force approach with the identity function for refinement the search tree will contain all graphs, as indicated in Figure 4.5a. The canonical form will be determined purely by the $\overset{r}{<}$-relation on graphs. A refinement function does however not need to preserve this minimum graph, as depicted in Figure 4.5b.

For example the $\overset{r}{<}$-relation can order vertices by descending degree and the refinement function can order them by ascending degree. Unless the graph is regular this will produce a different canonical form than the brute-force algorithm.

A different type of optimisation is based on detecting automorphisms in the graph and using them to prune redundant subtrees. This pruning situation is depicted in Figure 4.5c, and explore in detail in Section 4.3.3.

4.3.1 Exploiting Vertex and Edge Labels

The core algorithm works both for labelled and unlabelled graphs as the final graph comparison with $\overset{r}{<}$ is assumed to compare labels. However, the labels can be incorporated earlier to provide stronger refinement.

Vertex Labelled Graphs

Let the input graph $G \in \mathcal{G}$ be vertex-labelled with elements from a set Ω_V with the labelling function $l_V \colon V \to \Omega_V$. Additionally let the operators $\overset{r}{<}$ and $\overset{r}{=}$ be defined for pairs of elements Ω_V, in a manner consistent with isomorphism checking between graphs of $\mathcal{G}$.

Instead of starting the algorithm with the unit partition we can construct a finer partition π' by partitioning and sorting V by the labels. That is, we construct π' such that for all $u, v \in V$

$$\begin{aligned} &cell(u, \pi') = cell(v, \pi') && \text{if } l_V(u) \overset{r}{=} l_V(v) \\ &cell(u, \pi') < cell(v, \pi') && \text{if } l_V(u) \overset{r}{<} l_V(v) \end{aligned}$$

Note that using π' instead of the unit partition may change the canonical form calculated by the algorithm.

In the literature of the mentioned algorithms the incorporation of vertex labels is done implicitly by canonicalising both a graph and an ordered partition, instead of just a graph as described here.

Edge Labelled Graphs

Edge labels can be incorporated in at least two different ways. One method is to construct a vertex labelled, bipartite graph $G' = (V \cup E, E')$ where vertices $V \cup E$ are labelled both with their labels in G and with a tag specifying their membership in V and E. The new graph G' is then canonicalised where the labels can be incorporated as described above.

We also suggest another method which does not require the construction of an auxiliary graph, however it may not be feasible (or practical) for all types of labels. Let Ω_E be the set of edge labels, and assume we use the refinement function based on vertex degrees described above. Instead of simply

counting the total degree we can count the occurrence of each unique label. E.g., for chemical molecules we can use a 4-vector to count the vertex degree with respect each bond type. In general the requirement for such generalised counters is that they have a total order defined.

4.3.2 Node Invariants

We can define a pruning scheme based on computing a isomorphism invariant value $\phi(\tau)$, in a totally ordered set, for each tree node τ in the following way (see also [Junttila & Kaski 2007, Section 3.3]). For a node τ in the search tree let $\tau_1, \tau_2, \ldots, \tau_l$ denote the path from the root node $\tau_r = \tau_1$ to $\tau = \tau_l$. Then let $\overline{\phi}(\tau) = (\phi(\tau_1), \phi(\tau_2), \ldots, \phi(\tau_l))$ be the sequence of node invariants for the path, starting from the root. Given a node τ at depth l and a node τ' at depth l' we can now lexicographically compare the first $\min\{l, l'\}$ elements of $\overline{\phi}(\tau)$ and $\overline{\phi}(\tau')$. If they are different we can decide that the node with the smallest invariant prefix is better, and prune the subtree rooted at the node with the larger invariant prefix.

In the literature various invariants have been proposed, e.g., counting the number of cells in the partitions, or the sequence of cell sizes. The name of Traces stems from the introduction of a new invariant which contains the sequence of positions of every cell split. That is, if

$$\pi = (V_1, V_2, \ldots, V_{t-1}, V_t, V_{t+1}, \ldots, V_r)$$

is split into

$$\pi = (V_1, V_2, \ldots, V_{t-1}, V_{t'}, V_{t''}, V_{t+1}, \ldots, V_r)$$

then the position of the split is $\sum_{1 \leq i \leq t'} |V_i|$. When considering partitions to be stored in arrays, the split position is exactly the index of the first vertex in the cell $V_{t''}$. As cells are split during the refinement procedure it is possible with this invariant to abort a partition refinement if it is evaluated to be worse than the currently best trace of cell splits. The pruning by node invariants is also described in general by items (A) and (B) in [McKay & Piperno 2014b, Section 2.4].

4.3.3 Automorphism Discovery

Consider again the search tree in Figure 4.4 and note that the two partitions

$$\pi_{(1,7)} = [1 \mid 2 \mid 7 \mid 10 \mid 8 \mid 9 \mid 6 \mid 5 \mid 4 \mid 3]$$
$$\pi_{(1,8)} = [1 \mid 2 \mid 8 \mid 9 \mid 7 \mid 10 \mid 5 \mid 6 \mid 3 \mid 4]$$

give representationally equal graphs. That is, if $G' = G^{\pi_{(1,7)}^{-1}}$ and $G'' = G^{\pi_{(1,8)}^{-1}}$ then $G' \overset{r}{=} G''$. Therefore there is a trivial isomorphism from G' to G'' that

maps the ith vertex of G' to the ith vertex of G''. We can recast this into an automorphism α on G, which can be described as $\alpha(\pi_{(1,7)}(v)) = \pi_{(1,8)}(v), \forall v \in V$ (see also [McKay 1981, 2.18]). The automorphism is thus

$$\begin{aligned} \alpha \circ \pi_{(1,7)} &= \pi_{(1,8)} && \Leftrightarrow \\ \alpha &= \pi_{(1,8)} \circ \pi_{(1,7)}^{-1} \\ \alpha &= (3\ 4)(5\ 6)(7\ 8)(9\ 10) \end{aligned}$$

Geometrically this automorphism mirrors the graph in the axis through vertex 1 and 2. Such a morphism is called an *explicit automorphism*, as we construct it from explicitly by comparing leaves of the search tree. As described in [McKay & Piperno 2014b, Section 2.4] the set of all explicit automorphisms found during the algorithm generates the automorphism group of G. In some cases it is also possible to find *implicit* automorphisms without comparing leaves, when non-discrete partitions have a certain structure, e.g., see [McKay 1981, 2.24] and [McKay & Piperno 2014b, Sections 3.5 and 3.6].

Any available automorphism can be used to prune the the search tree. Consider again Figure 4.4 and the partition $\pi_{(1)} = [1 \mid 2 \mid 7\ 8\ 9\ 10 \mid 5\ 6 \mid 3\ 4]$, and assume we have discovered the automorphism α as described above. Note that α fixes vertex 1, which is the vertex being individualised to reach $\pi_{(1)}$. When we now individualise respectively vertex 9 and 10 we get

$$\begin{aligned} \pi_{(1)} \downarrow 9 &= [1 \mid 2 \mid 9 \mid 7\ 8\ 10 \mid 5\ 6 \mid 3\ 4] \\ \pi_{(1)} \downarrow 10 &= [1 \mid 2 \mid 10 \mid 7\ 8\ 9 \mid 5\ 6 \mid 3\ 4] \end{aligned}$$

Notice though that the permutation of $\pi_{(1)} \downarrow 9$ with automorphism α gives exactly the partition $\pi_{(1)} \downarrow 10$. The refinement function will use the structure of the graph to split cells, but as an automorphism exactly preserves the graph structure the two resulting partitions must induce the same graph representation. We can therefore skip the subtree $\tau = (1, 10)$ (which happens to just be a leaf), as it contains the same leaves as $\tau' = (1, 9)$. The formal proof of this property requires several new definitions and additional theorems. They can be found in [McKay 1981, McKay & Piperno 2014b].

The general pruning scheme is as follows. Let $A \subseteq Aut(G)$ be a subset of known automorphisms on G, and $\tau = (v_1, v_2, \ldots, v_k)$ an internal node of $\mathcal{T}(G, \pi)$. The children of τ will be given by the set of vertices in the target cell $W = Q(G, \pi_\tau)$. Now construct $A' \subseteq A$ as the set of known automorphisms that fixes each $v_i, 1 \leq i \leq k$. That is, all permutations in A' will map each of the already individualised vertices to them self. For each $w \in W$ we can calculate the orbit of w under the permutations A', as the closure of w when applying A'. In our example this means 7 and 8 are in the same orbit when using only α, and 9 and 10 are in the same orbit. The set W is in this manner partitioned into subsets $W_1, W_2, \ldots, W_p$, and it suffices to explore just one child from each of these subsets.

In Figure 4.4 we have greyed out several nodes due to the automorphism pruning. From the discussion above it is clear why node $(1, 10)$ is pruned. However, notice that α also fixes vertex 2, and the same automorphism can thus be used to prune nodes $(2, 10)$ and $(2, 8)$. The last pruned node, $(2, 9)$, is due to the discovery of an automorphism between $\pi_{(1,7)}$ and $\pi_{(1,9)}$, which fixes both vertex 1 and 2.

The original description of nauty [McKay 1981] used both implicit and explicit automorphisms to perform pruning. However, it only used a list of these automorphisms A, but a permutation group is closed under composition. Therefore all permutations constructed from combinations of permutations in A are also automorphisms. When calculating the automorphisms A' that fixes a sequence of vertices, the algorithm thus may miss some of the combined permutations. For example, two known automorphisms could be $\gamma_1 = (1\ 2)(3\ 4)$ and $\gamma_2 = (1\ 2)(5\ 6)$, of which neither fixes vertex 1 and 2. So when pruning children in a tree node descending from individualisation of vertex 1 and 2 it will not be possible to use γ_1 and γ_2 for pruning. If we calculated their composition $\gamma_1 \circ \gamma_2 = (3\ 4)(5\ 6)$ we see that it would be a usable. The automorphism group of a graph may be very large, and it is therefore not feasible to compute the composition closure of the known automorphisms in general. To alleviate this problem the Traces algorithm in practice introduced the use of Schreier-trees [Seress 2003] that can efficiently represent the closure, and facilitate orbit calculations in subgroups with fixed vertices.

Chapter 5

External Molecule Representation

In the previous chapters we have only dealt with graphs and molecules abstractly and their in-memory representations, while we in this chapter will focus on methods for representing molecules as text strings. Two popular formats are SMILES and InChI, that both are *line notations*, meaning molecules are encoded as single-line ASCII strings. This makes them convenient for communication of data, but despite the popularity of these formats there are no published complete formal specifications for them.

5.1 SMILES

In the Simplified Molecular-Input Line Entry System (SMILES) a molecule is modelled as labelled graph, which is linearised to an ASCII string by recording a depth-first traversal of the graph. The format was first described in [Weininger 1988] and later by Daylight Chemical Information Systems [Daylight Chemical Information Systems 2011], and is to some extent able to encode aromaticity and stereochemical properties. SMILES strings are often human readable and writeable which may have contributed to the formats popularity. However, due to ambiguities in the initial descriptions there are now multiple interpretations of the format [O'Boyle 2012], and no complete formal specification has ever been published. The most comprehensive description seems to be the OpenSMILES [James 2012] effort, which contains additional details on the aromaticity model. As the underlying format is able to represent all molecules using the model described in Section 2.1 we describe a simplified SMILES format and highlight its possible differences with the original description [Weininger 1988].

There can be multiple SMILES strings for a molecule depending on both the chosen root for the depth-first traversal of the molecule graph, and on the order of which neighbours are visited. Using a canonical order of the atoms it is thus possible to compute a canonical SMILES string. Initially an algorithm for canonicalisation was published [Weininger *et al.* 1989], but it did not account for the stereochemical specifications and, as we illustrate below, it is not a proper canonicalisation algorithm. In the following we present the algorithm and compare it to the more general individualisation-refinement algorithm presented in Chapter 4.

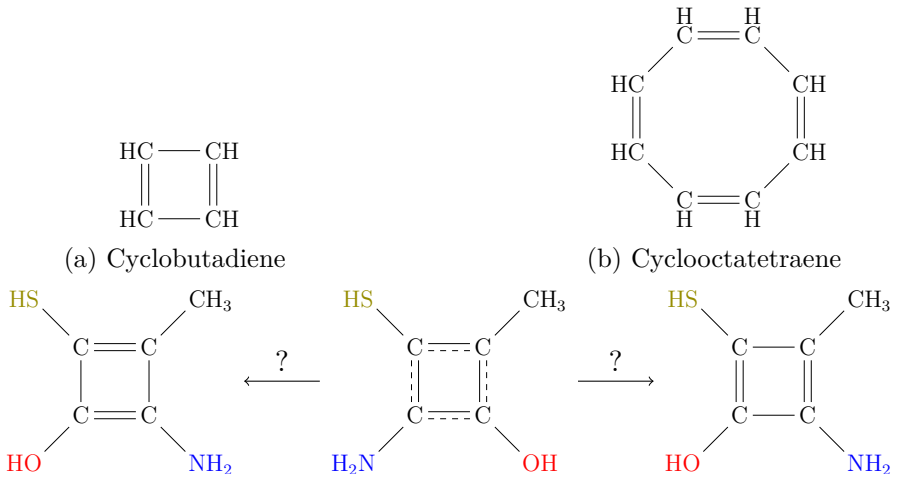
5.1.1 Molecule Model

There is no explicit definition of the molecule model for SMILES in [Weininger 1988], but from the descriptions it seems that the basics are similar to our model, described in Section 2.1: a molecule is a undirected, simple graph labelled with bond types and chemical elements. The vertices are further labelled with a charge and an isotope number, and the graph is allowed to be disconnected. Additionally the model can represent certain types of stereochemical properties, but for simplicity we leave them out of this discussion. Edges are labelled with the same bond type as we use Single, Double, Triple, and Aromatic, though the aromaticity model is more complex. The molecule model does not contain any restrictions on chemical valence or vertex degrees.

From [Weininger 1988, (4) Aromaticity Dectection] is it clear that the goal is to convert Kekulé-like structures into their proper form, using the Aromatic bond type, when the molecules fulfil certain rules. The paper does not give a self-contained definition of these rules, but simply states that for "a ring" to qualify as aromatic all atoms must be sp^2 hybridised and the number of excess π electrons must fulfil the formula $4n + 2, n \in \mathbb{N}_0$, also known as Hückel's rule. There is no explicit constraint that all Aromatic bonds must form a union of cycles in the molecule graph, or that Aromatic bonds must be in any cycle. Multiple examples of aromaticity detection are shown, though none with polycyclic aromatic compounds, leading to uncertainty of whether "a ring" actually means "a ring-system" or a graph theoretic cycle and how the arising conflicts are resolved [Apodaca 2007].

The paper itself [Weininger 1988] does not explicitly state what "sp^2 hybridisation" is and how excess π electrons are counted. A table of bond configurations for this purpose is provided by OpenSMILES [James 2012], though this part of the specification is explicitly noted as being under discussion.

The aromaticity detection algorithm in SMILES is supposed to also work in the inverse direction; if a compound is not aromatic in the model, but specified in a SMILES string as aromatic, then it is supposed to be converted into its proper form without using the Aromatic bond type. This is exemplified by the compunds cyclobutadiene and cyclooctatetraene, illustrated in Figure 5.1, which when specified as aromatic are supposed to be converted into the shown form, with alternating Single and Double bonds. These two molecules are mentioned to be *anti*-aromatic, though this term is not defined in the paper, and it is unclear whether it affects the conversion. In general the conversion is unclear, as it is described only to preserve certain properties, e.g., sp^2 hybridisation, charges, and hydrogen count. Furthermore, when considering derived compounds the conversion is ambiguous, assuming the natural definition of isomorphism. This is illustrated for a derivative of cyclobutadiene in Figure 5.1c.



(a) Cyclobutadiene

(b) Cyclooctatetraene

(c) Ambiguous conversion of non-aromatic compounds

Figure 5.1: According to [Weininger 1988] the compounds cyclobutadiene (a), and cyclooctatetraene (b) can be specified as aromatic, but must be converted into their proper form without AROMATIC bonds. This leads to ambiguity as shown in (c) with a molecule based on cyclobutadiene.

5.1.2 String Encoding

The original SMILES publication [Weininger 1988] did not contain a formal grammar that valid SMILES strings must follow, however a grammar is given by [James 2012]. A grammar for a simplified version of SMILES, e.g., without stereochemical information and isotopes, and only for connected graphs, is shown in Figure 5.2. A molecule is serialised into a character string by traversing a rooted spanning tree of the molecule graph in a depth-first manner, and appending substrings for vertices and edges when they are first encountered. Edges not covered by the chosen tree are each assigned an integer, which is then appended to the substring for both of the incident vertices. The following sections describes the specific semantics of the format.

Atom Representation

An atom is in general represented by the ⟨*bracketAtom*⟩ grammar, where the chemical symbol is enclosed in square brackets. Charges of absolute value $N \neq 0$ are encoded as '`-`⟨N⟩' for negative charges and '`+`⟨N⟩' for positive charges. When $N = 1$ the number can be left out, e.g., for protons, '`[H+]`', or chloride ions, '`[Cl-]`'.

Certain atoms can be marked "aromatic" by writing them in lower case (non-terminal ⟨*aromaticSymbol*⟩), for the purpose of representing AROMATIC bonds implicitly.

Uncharged atoms in the organic subset, ⟨*aliphaticOrganic*⟩, can be writ-

⟨*smiles*⟩ :: ⟨*chain*⟩

⟨*chain*⟩ :: ⟨*branchedAtom*⟩ ⟨*chainTail*⟩

⟨*chainTail*⟩ :: ⟨*bond*⟩ ⟨*chain*⟩ | ϵ

⟨*branchedAtom*⟩ :: ⟨*atom*⟩ ⟨*ringbond*⟩* ⟨*branch*⟩*

⟨*branch*⟩ :: '`(`' ⟨*bond*⟩ ⟨*chain*⟩ '`)`'

⟨*ringbond*⟩ :: ⟨*bond*⟩? (⟨*digit*⟩ | '`%`' ⟨*digit*⟩ ⟨*digit*⟩)

⟨*atom*⟩ :: ⟨*bracketAtom*⟩ | ⟨*aliphaticOrganic*⟩ | ⟨*aromaticOrganic*⟩

⟨*aliphaticOrganic*⟩ :: '`B`' | '`C`' | '`N`' | '`O`' | '`F`' | '`P`' | '`S`' | '`Cl`' | '`Br`' | '`I`'

⟨*aromaticOrganic*⟩ :: '`b`' | '`c`' | '`n`' | '`o`' | '`p`' | '`s`'

⟨*bracketAtom*⟩ :: '`[`' ⟨*symbol*⟩ ⟨*hCount*⟩? ⟨*charge*⟩? '`]`'

⟨*hCount*⟩ :: '`H`' ⟨*digit*⟩?

⟨*charge*⟩ :: ('`-`' | '`+`') ⟨*digit*⟩? ⟨*digit*⟩?

⟨*symbol*⟩ :: ⟨*elementSymbol*⟩ | ⟨*aromaticSymbol*⟩

⟨*elementSymbol*⟩ :: '`H`' | '`He`' | '`Li`' | '`Be`' | '`B`' | '`C`' | '`N`' | '`O`' | ... | '`Uuo`'

⟨*aromaticSymbol*⟩ :: '`b`' | '`c`' | '`n`' | '`o`' | '`p`' | '`s`' | '`as`' | '`se`'

⟨*bond*⟩ :: '`-`' | '`=`' | '`#`' | '`:`' | ϵ

⟨*digit*⟩ :: '`0`' | '`1`' | ... | '`9`'

Figure 5.2: A grammar for simplified SMILES strings, with the starting symbol ⟨*smiles*⟩, based on [James 2012].

ten without brackets when the number of adjacent hydrogen atoms follow specific rules. This can be combined with marking the atom "aromatic" for the atoms described by the non-terminal ⟨*aromaticOrganic*⟩. The details of implicit hydrogen atoms are described in a following section.

Bond Representation

Each of the four bond types, Single, Double, Triple, and Aromatic are represented respectively as '`-`', '`=`', '`#`', and '`:`'. Most Single and Aromatic bonds can be represented implicitly, using the empty string. For example, water can be represented as both '`[H]-[O]-[H]`' and '`[H][O][H]`'. The rule is that an implicit bond is Aromatic if and only if both incident atoms are marked "aromatic" by using lower-case chemical symbols. Thereby '`[C][C]`' is equivalent to '`[C]-[C]`', '`[c][C]`' to '`[c]-[C]`', and '`[c][c]`' to '`[c]:[c]`'.

Branch Representation

The different branches in a subtree are enclosed in parentheses in order to separate them. However, the last traversed branch may be appended without

Element	Normal valences
B	3
C	4
N	3 (and 5 [James 2012])
O	2
P	3 and 5
Non-aromatic S	2, 4, and 6
Aromatic S	3 and 5
F, Cl, Br, and I	1

Table 5.3: Normal valences for the organic subset of elements, as specified by [Weininger 1988] and [James 2012].

a parenthesis. Water can thus be written as both '`[O]([H])([H])`' and '`[O]([H])[H]`'.

Implicit Hydrogen Atoms

Hydrogen atoms with neutral charge and only one SINGLE bond as incident edges can be represented implicitly in two ways. A bracketed atom can specify up to 9 extra neighbouring hydrogen atoms with the ⟨*hCount*⟩ nonterminal. An atom without brackets, i.e., the organic subset specified by ⟨*aliphaticOrganic*⟩ and ⟨*aromaticOrganic*⟩, implicitly defines neighbouring hydrogen atoms such that the atom reaches a so-called "lowest normal valence". The paper [Weininger 1988] describes these valences as noted in Table 5.3, which includes a special case for some cases of "aromatic sulfur". However, the molecule model only specifies that a bond can be aromatic and the use of lower-case letters in SMILES strings for "aromatic atoms" seems only to be a short-hand enabling implicit bonds. It is therefore not entirely clear when sulfur should be considered aromatic.

Another ambiguity is in the method for counting valences when AROMATIC bonds are involved, as nothing is stated in the paper. In [May 2013] this problem is explored in relation to several SMILES implementations. One approach used is to let AROMATIC bonds contribute 1.5 to valence sums, and use special rounding rules. Another is to let them contribute 1 to the sums and add an additional 1 to the sum of each atom incident to an aromatic bond. A third approach is to only add the additional 1 to aromatic atoms that are not incident to double or triple bonds.

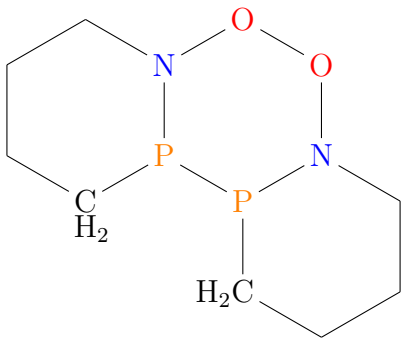


Figure 5.4: The molecule represented by the SMILES string '`N12CCCCP1P3N(OO2)CCCC3`', using 3 ring-closure integers. By reusing ring-closure integers for non-overlapping intervals, only 2 numbers are needed: '`N12CCCCP1P1N(OO2)CCCC1`'.

Cyclic Structures

The use of parenthesis only allows for encoding trees, and thus special notation is used to represent the remaining edges in molecules with cycles. Each of these edges is assigned an integer between 0 and 99, which is appended to the substrings for the incident atoms (see the non-terminals ⟨*branchedAtom*⟩ and ⟨*ringbond*⟩). The bond type for such ring closure edges can be specified explicitly before at least one of the integers, but must of course be in agreement if present at both ends. The molecule 1,3-cyclohexadiene can therefore be represented by all of '`C1CCC=CC=1`', '`C=1CCC=CC1`', and '`C=1CCC=CC=1`'. If no bond type is specified at either end, the bond becomes implicit and is thus either Single or Aromatic depending on the atom specifications.

The ring-closures forms intervals in the SMILES string, as exemplified with the molecule '`N12CCCCP1P3N(OO2)CCCC3`' visualised in Figure 5.4. Before assigning ring-closure integers we can depict the intervals as:

$$\overbrace{\texttt{N}\underbrace{\texttt{CCCCP}}_{i_1}\texttt{P}\underbrace{\texttt{N(OO}}_{i_3}\texttt{)CCCC}}^{i_2}$$

Interval i_1 and i_3 do not overlap so they can unambiguously be assigned the same integer, e.g, giving the SMILES string '`N12CCCCP1P1N(OO2)CCCC1`'.

5.1.3 Canonical Strings

The SMILES format can be combined with graph canonicalisation to obtain canonical strings for molecules. This was attempted in [Weininger *et al.* 1989] where a two-phase algorithm, CANGEN, was presented. In the first phase of the algorithm it finds a supposedly canonical order for the vertices of the molecule graph, using a method similar to the individualisation-refinement

method described in Chapter 4. However, the algorithm does not produce a canonical order for all molecules which we illustrate with an example below. To further explain the problem with the algorithm we describe it using the algorithmic framework from Chapter 4. The second phase of the algorithm uses a canonical vertex order to generate the canonical SMILES string.

Graph "Canonicalisation"

In the description of the SMILES format above we have left out certain aspects, e.g., isotopes and stereochemistry, These aspects are not covered by the algorithm in [Weininger *et al.* 1989] intended for canonicalisation.

In the following description of the algorithm we use the terminology introduced in Chapter 4.

Initial Vertex Partitioning The algorithm constructs an initial ordered partition using vertex labels, as described in Section 4.3.1. Each vertex label is a tuple of the following numbers [Weininger *et al.* 1989, Table I]:

1. number of connections
2. number of non-hydrogen bonds
3. atomic number
4. sign of charge
5. absolute charge
6. number of attached hydrogens

There is however some uncertainty about how these values should be calculated. From the examples it seems that "number of connections" is the vertex degree, when hydrogens that can be made implicit are ignored. The "number of non-hydrogen bonds" is a weighted sum over the same edges, i.e., without implicit hydrogens. There is no description of which weight AROMATIC bonds has. It is clearly possible to map the sign of charge into integers, though no specific mapping is given, meaning different implementations may use different mappings.

Representation of Ordered Partitions The CANGEN algorithm is not described as using ordered partitions, but instead stores the *rank* of each vertex. This is simply a synonym for the cell number of a vertex (see Section 4.1.1).

Refinement The CANGEN algorithm refines a partition by calculating a specific product for each vertex depending on the neighbourhood of the vertex. Let $\rho\colon \mathbb{N} \to \mathbb{N}$ be the function that given a positive integer i returns the ith prime number, and let $r\colon V \to \mathbb{N}$ be the cell number for each vertex. For each vertex $v \in V$ the number $r'(v)$ is calculated as

$$r'(v) = \prod_{(v,u)\in\delta(v)} \rho(r(u))$$

That is, $r'(v)$ is the product of the prime numbers corresponding to the ranks of the neighbours. The list of vertices are then sorted lexicographically according to the tuples $(r(v), r'(v))$, and new cell numbers r'' are assigned such that $r''(v) = r''(u)$ if $(r(v), r'(v)) = (r(u), r'(u))$ and $r''(v) < r''(u)$ if $(r(v), r'(v)) < (r(u), r'(u))$. This procedure is repeated as long as cell numbers change.

It is well-known that the result of prime factorisation of natural numbers is unique, so the product of prime numbers ensures that vertices in the same cell have the same product if and only if they have the same neighbourhood. This refinement procedure is thus equivalent to the one based on vertex degrees outlined in Section 4.2.1, assuming it can be implemented without integer overflow.

Target Cell Selector and Vertex Individualisation CANGEN always selects the first non-trivial cell. A vertex is then individualised by doubling all the cell numbers and reducing the vertex to be individualised with 1. New consecutive cell numbers are then calculated. This clearly performs the same vertex individualisation as described in Section 4.2.2, as the doubling preserves the cell structure, and the reduction by 1 orders the individualised vertex just before its old cell.

Tree Search Aside from the ambiguities in the descriptions in [Weininger *et al.* 1989], the main problem is in the tree search. CANGEN simply does not explore the complete tree, but only a single path. After refinement the algorithm individualises only the first vertex in the target cell. However, a cell must be treated as an unordered set and the "first" vertex is thus determined by the input order of the vertices. CANGEN is therefore not isomorphism invariant, i.e., it is not a canonicalisation algorithm. In Section 5.1.4 we show a small counterexample that further proves this fact.

String Generation

The SMILES format specifies multiple methods to implicitly represent properties, most notable hydrogen atoms. In the string generation we assume that all these methods are used whenever possible, meaning all hydrogen atoms

with charge zero and only a SINGLE bond as incident edges will be made implicit. An exception to this is molecular hydrogen H_2 which is always assigned '`[H][H]`' as its SMILES string.

Recall that a SMILES string is generated by traversing the molecule graph in a depth-first manner starting from an initial vertex. This initial vertex is selected simply to be the one with the lowest canonical index, which is not an implicit hydrogen. By the initial vertex partitioning described above the initial vertex will, if possible, be an "end atom", i.e., an atom with just one edge to vertices that are not implicit hydrogens.

During branching in the depth-first traversal the algorithm selects the next subtree where the root has the lowest canonical index. However, the authors prefer to not have ring-closures with double and triple bonds and thus uses a slightly modified scheme. Unfortunately the description is not entirely clear, and it notably does not mention AROMATIC bonds. The following is therefore our interpretation of the authors intention.

- Let v be the current vertex in the traversal and $N(v) = \{u_1, u_2, \ldots, u_d\}$ the remaining non-visited neighbours of v.
- Mark all edges (v, u_i) if there exists a cycle in the graph using the edge.
- Partition $N(v)$ into two lists: $N_P(v)$ containing all u_i where the edge (v, u_i) is unmarked, and the list $N_C(v)$ corresponding to the neighbours connected with a marked edge.
- Sort $N_P(v)$ and $N_C(v)$ according to the canonical vertex order.
- If $N_P(v)$ is non-empty, visit the subtree corresponding to the first vertex.
- Otherwise, sort $N_C(v)$ such that vertices connected to v with DOUBLE, TRIPLE, and AROMATIC bonds are ordered before SINGLE bonds. The non-SINGLE bonds all compare equal in this procedure, and the sorting must be stable with respect to the previous order.
- Visit the first vertex of $N_C(v)$.

Ring-closure integers must finally be assigned in a canonical manner, and inserted in sorted order on vertices with multiple ring-closures. Recall that the ring-closures induce intervals on the final SMILES string. The paper does not completely state how the assignment should be done, but from the examples it seems that the intervals are first sorted lexicographically by their starting position and end position. They are then assigned the lowest available positive integer when processing them in sorted order. Note that this assignment strategy incidentally ensures that the fewest possible ring-closure integers are used (for the given tree traversal), as the problem is equivalent to graph colouring on interval graphs and the assignment strategy gives optimal solutions [Cormen *et al.* 2001].

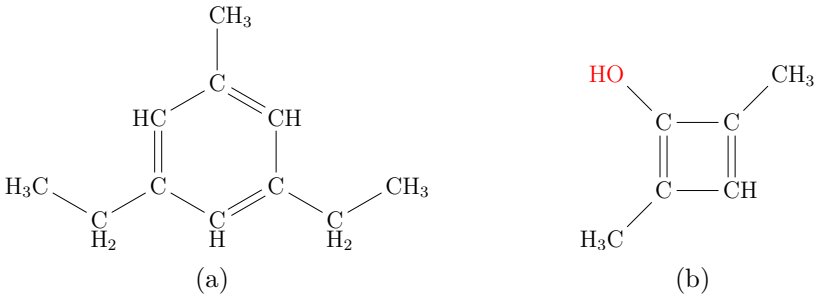


Figure 5.5: Counterexamples for the CANGEN algorithm. (a) A proposed counterexample from [Neglur *et al.* 2005], which however is not a valid molecule in the molecule model of SMILES. Benzene rings in Kekulé form must be converted into a cycle of Aromatic bonds. (b) A counterexample based on cyclobutadiene, which in [Weininger 1988] is explicitly stated to not use Aromatic bonds, but alternating Sinlge and Double bonds. The "canonical" form as generated using CANGEN depends on the input order of the two methyl carbons.

5.1.4 Counterexample for Canonicalisation

Constructing a counterexample to prove that the CANGEN algorithm does not produce a canonical vertex order is not too difficult. However, we must be careful to select a molecule which is covered by the molecule model. For example, in [Neglur *et al.* 2005] a counterexample is presented, which is visualised in Figure 5.5a. Though, this molecule is based on the Kekulé form of a benzene ring, which in [Weininger 1988] is clearly described as not being in the molecule model.

We can instead use cyclobutadiene as a basis, which is similarly clearly described to be modelled with alternating single and double bonds. An example of such a molecule is shown in Figure 5.5b, where the incorrect behaviour can be triggered by two different inputs where the two methyl groups have their order swapped. The first component of the vertex label is the number of connections to non-implicit hydrogens, which orders the carbons in the CH_3 groups and the oxygen in OH group before the rest of the atoms. The second component is the weighted sum of the connections, which is not different between these three atoms. The third component is however the atomic number, which orders the carbons before the oxygen. The refinement procedure in CANGEN does not exploit edge labels, and the first equitable partition thus have the first cell containing exactly the two carbons in the methyl group. As the algorithm only considers the individualisation of the "first" of these vertices we can obtain two different canonical forms, depending on the input order of the methyl carbons.

5.2 InChI

The International Chemical Identifier (InChI) is a molecule format developed as project under the International Union of Pure and Applied Chemistry (IUPAC) [InChI 2015]. The phrase "InChI" is used both for the actual molecule format, the algorithms involved, and a concrete program for encoding and decoding InChI strings, made available by InChI Trust [InChI Trust 2015].

On the webpage of InChI Trust and in a recently published paper [Heller *et al.* 2015] it is stated that the InChI format is intended to be an open standard for representing molecules. There is no single document labelled as being the specification of this standard, but a comprehensive description can be found in [Heller *et al.* 2015] and in the technical manual [Stein *et al.* 2011]. However, in the technical manual it is explicitly stated that "Mathematical details of the algorithms used will not be presented.". The recent paper [Heller *et al.* 2015] contains a short textual description of part of the canonicalisation algorithm, without mathematical details, and with a statement that the algorithm has been based upon [McKay 1981, Hartke & Radcliffe 2009]. As illustrated in Chapter 4 there is a huge amount of flexibility in specifying a concrete canonicalisation algorithm even within the individualisation-refinement paradigm, and different choices lead to different canonical forms. Without a precise specification of the InChI standard it thus becomes extremely difficult to reproduce the format in a compatible manner. In [Heller *et al.* 2015, Software] we however learn that we are not supposed to reimplement the standard, as "InChI, by intention, is assumed to have only a single software implementation". Furthermore, in [InChI Trust 2015, Technical FAQ: 3.2] it is stated that the official InChI software "acts as the final arbiter of the correctness". There is therefore no separation between the specification and the implementation of InChI. This is contrary to the usual practise in software development [Sommerville 2011], and in software standardisation.

Despite the lack of precise documentation for InChI, the format is in widespread use. In the following sections we briefly outline the core ideas for the molecule model and canonicalisation in InChI. The descriptions are in general deduced from [Heller *et al.* 2015].

5.2.1 Molecule Model

The molecule model of InChI contains several submodels, arranged in a layered manner such that they refine each other. For example, the least specific model represents only the number of each atom type in the molecule, while more specific models adds bonds, hydrogen placement, and stereochemical properties. An overview of the layers in the model can be found in [Stein *et al.* 2011, Figure 1], and in the following we provide a more formal description of a few of the layers.

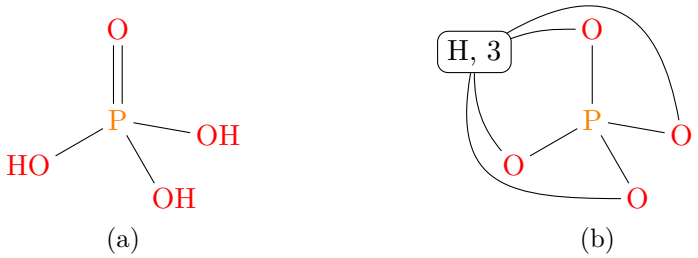


Figure 5.6: An example of shared hydrogens in InChI, for modelling multiple tautomers. (a) Ordinary structural formula for phosphate. (b) Depiction of phosphate in the InChI model up to the hydrogens layer, with the four oxygen atoms sharing three hydrogens.

Empirical Formula Layer In the least specific model a molecule is a multiset of chemical symbols.

Skeletal Connection Layer This layer adds bonds to the molecule, and distinguishes between hydrogens with only one bond and bridging hydrogens with multiple bonds (e.g., found in diborane). A molecule is then a pair $M = \langle G, k \rangle$, where $G = (V, E, l_V)$ is a vertex labelled graph with unlabelled edges, and $k \in \mathbb{N}_0$. The vertices of G are each labelled with one of the chemical symbols, and it seems that if such a vertex is labelled as hydrogen, then it is a bridging hydrogen. The integer k is the number of non-bridging hydrogens in the molecule. The edges are not labelled, and no bond types are therefore modelled.

Hydrogens Layer In the third layer the hydrogen count is incorporated into the graph structure. The details of this incorporation is somewhat unclear, but what is clear is that special vertices are added to the graph from the previous layer. They are labelled as "shared hydrogens" and with a positive integer. Edges are then added between these vertices and the ordinary vertices from the previous layers. A "shared hydrogen" vertex v with count $k_v > 0$, connected to vertices v_1, v_2, v_3 means that v_1, v_2, and v_3 share k_v non-bridging hydrogens. In the special case that a "shared hydrogen" vertex has degree 1, the hydrogens it represents are simply attached to this neighbour. Otherwise the hydrogens represent different tautomers of the molecule, as detected by the patterns in [Heller *et al.* 2015, Table 2]. In Figure 5.6 a phosphate molecule is shown in this model.

Additional Layers The following layers in the model include specification of charges, fixation of positions for the shared hydrogens, stereochemistry, and isotope specification. While short descriptions of these layers are provided, the details of what they model are intermixed with algorithmic descriptions, making their specification rather unclear.

5.2.2 Normalisation

A large part of both [Heller *et al.* 2015] and [Stein *et al.* 2011] is devoted to the normalisation procedures applied to molecules before they are converted into the InChI model. They are introduced in order to obtain a common representation for molecules that ordinarily would be considered the same. Note though that the molecule model that the normalisation is applied to is not explicitly defined in either document, but from the examples it seems similar to the model we have defined in Section 2.1, possibly with extra valence constraints. Some of the normalisation steps seek to move charges while changing bond types, e.g., see [Heller *et al.* 2015, page 17ff], while other steps may add or remove protons [Heller *et al.* 2015, page 19ff].

5.2.3 Canonicalisation

In [Heller *et al.* 2015] the canonicalisation process is described as starting from a canonicalisation of the lowest layer, and then rerunning the canonicalisation with an additional layer while maintaining the canonical form of the previous layer. Formally we can phrase this in the following way. Let $C^{(i)}$ be the function producing the canonical form of molecules in layer i, and let $P^{(i)}$ be the projection function that converts a molecule in layer $j > i$ to the corresponding molecule in layer i. Then for a molecule in the ith layer $M_1^{(i)}$ and molecule in the layer above $M_2^{(i+1)}$, which are isomorphic on layer i, i.e., $M_1^{(i)} \cong P^{(i)}(M_2^{(i+1)})$, the canonicalisation functions must not only satisfy

$$C^{(i)}\left(M_1^{(i)}\right) \overset{r}{=} C^{(i)}\left(P^{(i)}\left(M_2^{(i+1)}\right)\right)$$

but also that the projection and canonicalisation must commute:

$$C^{(i)}\left(P^{(i)}\left(M_2^{(i+1)}\right)\right) \overset{r}{=} P^{(i)}\left(C^{(i)}\left(M_2^{(i+1)}\right)\right)$$

For example, given a molecule in the hydrogens layer, we must obtain the same result no matter if we first canonicalise and then remove the shared hydrogens, or we first remove the shared hydrogens and then canonicalise. Using the terminology from Figure 4.5, this means that the $\overset{r}{<}$-best permutations of molecules on level $i+1$ must correspond to a subset of the $\overset{r}{<}$-best permutations on level i.

A full description of the canonicalisation algorithms naturally depends on a formal specification of the molecule models, including the specification for how to project molecules into lower layers. The canonicalisation process, as implemented in the InChI software, is illustrated with a flow chart in [Heller *et al.* 2015, Figure 10], and a textual description is provided for some layers in the model.

5.2.4 String Encoding

A molecule can after canonicalisation be serialised into an ASCII string. Each string starts with '`InChI=`' followed by a version number for the encoding. Each layer is then appended in order, of which we briefly describe the first few layers.

Empirical Formula Layer Recall that in this layer a molecule is a multiset of chemical symbols. The encoding starts with '`/`' and is then followed by each chemical symbol and the number of its occurrences in the molecule, e.g., '`/C2H6O`'. The symbols are ordered such that carbon is first, hydrogen is second, and the remaining symbols are in alphabetical order. Symbols with count zero are not represented, and symbols with a single occurrence are written without an explicit count.

Skeletal Connection Layer In this layer it is assumed that all atoms that are not non-bridging hydrogens has been assigned consecutive indices starting from 1. The layer starts with the string '`/c`' and is then written as a depth-first traversal of the graph, similar to the structure of SMILES strings, though only using the canonical vertex indices in the string. As an example [Heller *et al.* 2015, page 20], the molecule guanine has empirical formula layer is '`/C5H5N5O`', and the skeletal connection layer is then '`/c6-5-9-3-2(4(11)10-5)7-1-8-3`'. The paper [Heller *et al.* 2015] however does not provide a precise grammar and the semantics for serialisation of this layer.

Hydrogens Layer The string for this layer starts with '`/h`' and contains a specification of each shared hydrogen. No precise specification is given, but examples such as '`/h1H,(H4,6,7,8,9,10,11)`' for guanine indicate that first all shared hydrogens with degree 1 are listed on the form ⟨*vertex id*⟩'`H`'⟨*count*⟩, where ⟨*count*⟩ is the number of hydrognes represented by the "shared hydrogen" vertex. The ⟨*count*⟩ is omitted when it is 1. Next the remaining shared hydrogens are specified on the form '`(H`'⟨*count*⟩'`,`'⟨*vertex ids*⟩'`)`', where ⟨*vertex ids*⟩ is a comma-separated list of vertex indices.

Part II

Graph Transformation and Chemical Reactions

Chapter 6

The Double Pushout Approach

There are several different approaches to defining graph rewriting systems, for example replacement systems where single vertices or edges are replaced with graphs. For the modelling of chemistry we use the Double Pushout (DPO) approach, which is one of the algebraic graph rewriting formalisms. A general overview of graph rewriting can be found in [Corradini *et al.* 1997], and for details on algebraic graph transformation see [Ehrig *et al.* 2006, Ehrig 1979].

6.1 Introduction to Category Theory

To formally define the DPO approach we need a few concepts from category theory. For the full details we refer to [Ehrig *et al.* 2006, Appendix A], while we here only state the definitions and aim at an intuitive understanding of the constructions in categories of graphs.

Definition 6.1 (Category, see [Ehrig *et al.* 2006, Definition A.1])**.** A category $\mathbf{C}$ consists of

- a class of objects $Ob(\mathbf{C})$,
- a class of morphisms $Mor(\mathbf{C})$ where each morphism $f\colon A \to B$ maps $A \in Ob(\mathbf{C})$ to $B \in Ob(\mathbf{C})$, and
- a morphism composition operator $\circ\colon Mor(\mathbf{C}) \times Mor(\mathbf{C}) \to Mor(\mathbf{C})$

such that

- the class of morphisms is closed under composition: for each pair of morphism $f\colon A \to B$, $g\colon B \to C$ there is a morphism $g \circ f\colon A \to C$,
- there is an identity morphism $\mathrm{id}_A\colon A \to A$ for each object $A \in Ob(\mathbf{C})$, and
- morphism composition is associative.

Note that there can be multiple morphisms from one object to another, e.g., if a graph G matches as a subgraph in H in different ways, that correspond to multiple morphisms $m\colon G \to H$.

In the following sections we only use categories where the objects are some class of (labelled) graphs and all morphisms are monomorphisms (see Definition 3.1).

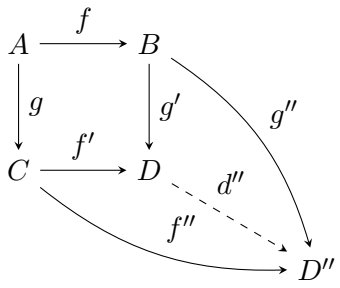


Figure 6.1: Illustration of the definition of a pushout (Definition 6.2). The pair of morphisms f', g' is a pushout of f, g if i) the morphisms commute, and ii) for all morphisms f'', g'' that commute with f, g, there exists a unique morphism d'' that commutes with the other morphisms.

Concepts in category theory are often illustrated using *commutative diagrams*, which are directed graphs where the vertices represent objects and the edges represent morphisms. A diagram, or part thereof, is said to *commute* when composing morphisms along different paths, with the same start and end point, results in the same morphism.

6.1.1 Pushouts and Pullbacks

Definition 6.2 (Pushout, see [Ehrig *et al.* 2006, Definition A.17])**.** Given two morphisms $f\colon A \to B$ and $g\colon A \to C$ in some category, the morphisms $f'\colon C \to D$ and $g'\colon B \to D$ forms a pushout of f and g if and only if (see also Figure 6.1)

i) $g' \circ f = f' \circ g$, i.e., the square commutes, and

ii) for all pairs of morphisms $f''\colon C \to D'', g''\colon B \to D''$ with $g'' \circ f = f'' \circ g$, there exists a unique morphism $d''\colon D \to D''$ such that $f'' = d'' \circ f'$ and $g'' = d'' \circ g'$.

The object D is called the *pushout object*.

An interpretation of a pushout for graphs is that the graph D is the union of B and C with equality specified by common vertices and edges in A. In Figure 6.2 we show an example of a graph pushout, along with candidates for the pushout that satisfy the first condition of the definition, but not the second. The candidate D in Figure 6.2b is not a pushout object because it is too large. The extra vertex and edge makes it possible to find an even larger candidate D'' and two different morphisms, indicated by the choice of either the blue or green mapping. The candidate in Figure 6.2c is on the other hand too small, as we have mapped the extra vertex of B and C to the same vertex in D. When considering a second pushout candidate D'' we are not able to find any morphism from D to D'' such that the diagram commutes.

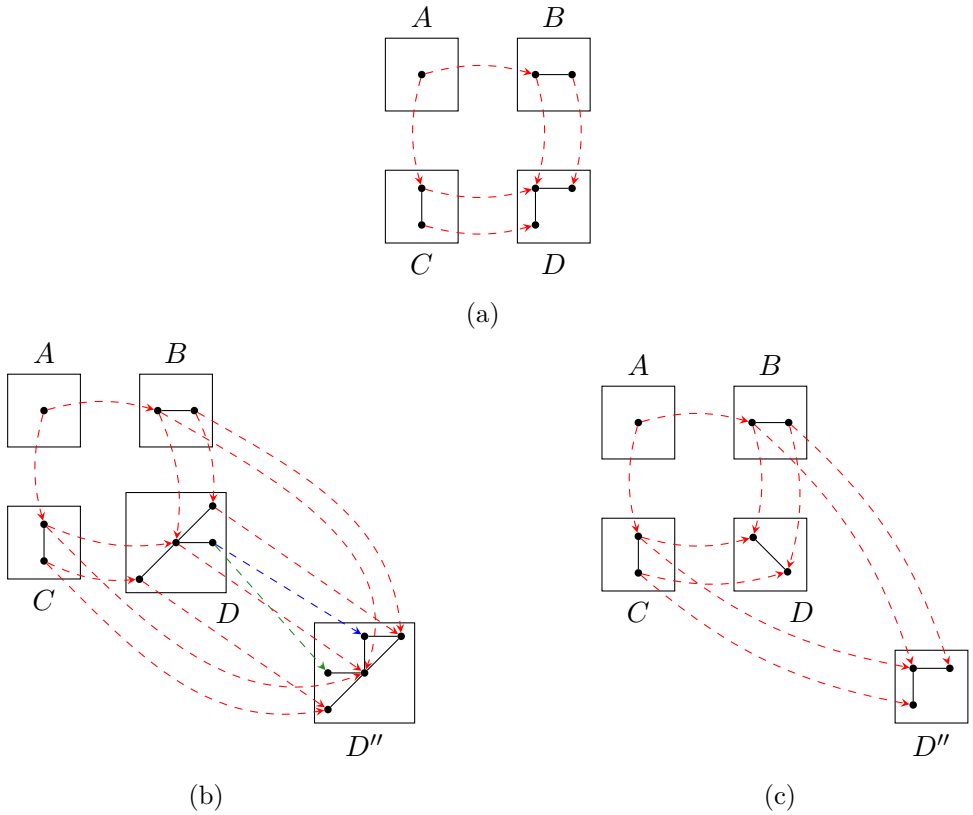


Figure 6.2: Illustration of a pushout and pushout candidates in the category of undirected graphs. (a) a pushout, which "glues" B and C along A to obtain the pushout object D. (b) a pushout candidate which is too large. Two commuting morphisms $D \to D''$ can be found: one using the blue mapping, and one using the green mapping. (c) a pushout candidate which is too small. Unrelated vertices of B and C have been merged, and with a second pushout candidate D'' we can not find any commuting morphisms $D \to D''$.

Suppose we are in the opposite situation of a pushout, i.e., given f', g' and want to find suitable morphisms f, g. This dual construction is a *pullback*:

Definition 6.3 (Pullback, see [Ehrig *et al.* 2006, Definition A.22])**.** Given two morphisms $f' \colon C \to D$ and $g' \colon B \to D$ in some category, the morphisms $f \colon A \to B$ and $g \colon A \to C$ form a pullback of f' and g' if and only if (see also Figure 6.3)

i) $g' \circ f = f' \circ g$, i.e., the square commutes, and

ii) for all pairs of morphisms $f'' \colon A'' \to B, g'' \colon A'' \to C$ with $g' \circ f'' = f' \circ g''$, there exists a unique morphism $a'' \colon A'' \to A$ such that $f'' = f \circ a''$ and $g'' = a'' \circ g$.

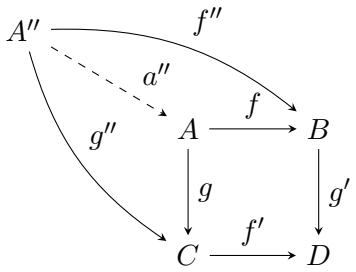


Figure 6.3: Illustration of the definition of a pullback (Definition 6.3), the dual of a pushout. The pair of morphisms f, g is a pullback of f', g' if i) the morphisms commute, and ii) for all morphisms f'', g'' that commute with f, g, there exists a unique morphism a'' that commutes with the rest.

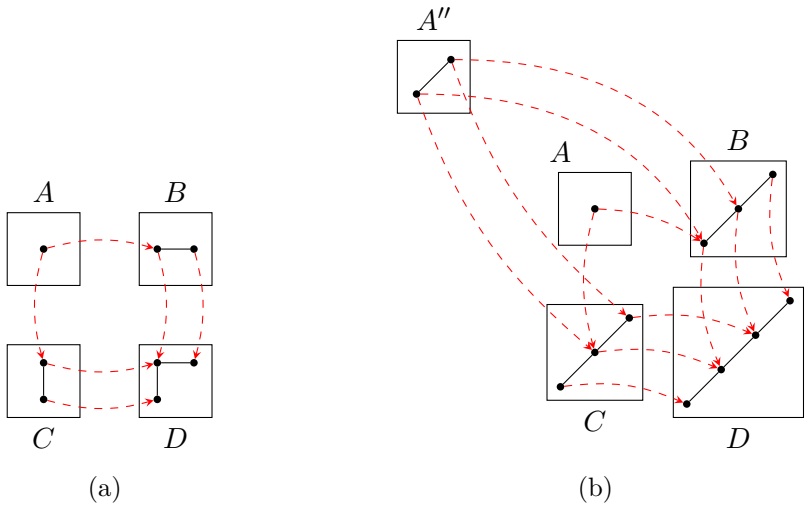
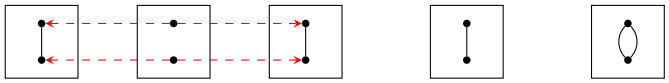


Figure 6.4: Illustration of a pullback and a pullback candidate in the category of undirected graphs. (a) a pullback, which intersects B with C using D to obtain the pullback object A. (b) a pullback candidate which is too small. There is no morphism $A'' \to A$ due to the edge in A''. However, even when ignoring this edge problem the resulting morphism would not commute with the remaining morphisms.

For categories of graphs we can interpret a pullback as the construction of the common subgraph of B and C determined by their embedding in D. In Figure 6.4 we have illustrated a graph pullback, and a candidate that does not fulfil both criteria.

The third variation of the situation we need is the *pushout complement*, where f and g' are given, and we wish to find g and f'.

Definition 6.4 (Pushout Complement, see [Ehrig *et al.* 2006, Definition A.20])**.** Given two morphisms $f\colon A \to B$ and $g'\colon B \to D$ in some category,



(a) (b) (c)

Figure 6.5: The pushout object of (a), in the category of simple graphs is either not existing or is the graph depicted in (b), where the two edges are merged. For multigraphs the pushout object would be the graph depicted in (c).

the morphisms $g\colon A \to C$ and $f'\colon C \to D$ forms the pushout complement of f and g' if and only if f' and g' is the pushout of f, g.

6.1.2 The Category of Simple Graphs

We only consider simple graphs in the following chapters, and this restriction gives rise to multiple definitions of a pushout, pullback, and a pushout complement. Consider the construction of a pushout for the span in Figure 6.5a. For graphs where parallel edges are allowed we can create the pushout object depicted in Figure 6.5c, which conforms to the idea of a disjoint union of the non-common vertices and edges. However, for simple graphs this is not a possibility and we either need to define that no pushout exists for this span, or that non-common edges can be merged in the pushout object and thus let the graph in Figure 6.5b be the pushout object. In [Braatz *et al.* 2011] they chose the latter option, which necessitates the definition of *minimal* pushout complements. For the modelling of chemistry we take the other option, and let no pushout be defined for this case. The span in Figure 6.5a can in the context of transformation of molecules arise if a rule specifies that a bond must be created between two atoms, but they are already bonded. Letting the pushout be defined would mean that one of the bonds either vanish, or we would need to define how any two bonds can be merged in a meaningful way. In both cases the transformation will have a side-effect not explicitly specified by the rule, which we deem highly undesirable.

6.2 Transformation Rules and Derivations

A graph transformation rule in the DPO formalism is a span $p = (L \xleftarrow{l} K \xrightarrow{r} R)$. The left-hand graph L plays the role of a precondition for the application of the rule, while the right-hand graph R is a post-condition. Their relation is specified by the context graph of the rule K, also called the *gluing graph*, together with the morphisms l and r. If the morphisms are unimportant or clear from the context we may write a rule simply as $p = (L, K, R)$.

The transformation of a graph G using p proceeds in the following manner (see Figure 6.6).

1. Find a match morphism $m\colon L \to G$, if it exists.

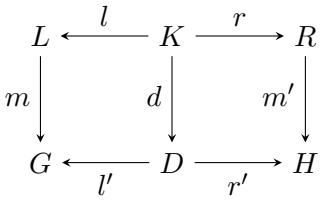


Figure 6.6: The diagram for a derivation $G \overset{p,m}{\Longrightarrow} H$ in the Double Pushout formalism of G to H using the rule p and the matching morphism m.

2. Construct D as the pushout complement of l, m, if it exists.

3. Construct H as the pushout of d, r, if it exists.

We call such a transformation a *derivation* of H from G, using the rule p and the matching morphism m. As a shorthand we write a derivation as $G \overset{p,m}{\Longrightarrow} H$, or as either $G \overset{p}{\Rightarrow} H$ or $G \Rightarrow H$ if the match morphism and rule are unimportant. Assuming m is found, the existence of D can be characterised by what is called the *gluing condition*: G, p, m satisfy the gluing condition if both the *dangling condition* and *identification condition* are satisfied.

***dangling condition*:** there are no edges in $E_G \backslash m(E_L)$ incident to a vertex in $m(V_L \backslash l(V_K))$. That is, if p specifies the deletion of a vertex, it can only be applied if it also specifies the deletion of all the edges incident to the vertex.

***identification condition*:** there are are no distinct vertices $u, v \in V_L$ with $m(u) = m(v)$, $u \in l(V_K)$, $v \notin l(V_K)$, and similarly for distinct edges of E_L. That is, if p specifies the deletion of a vertex or edge, but the preservation of another vertex or edge, then the matching morphism m may not identify those vertices or edges with each other.

A proof for the equivalence of the existence of the graph D and the gluing condition is given in [Ehrig *et al.* 2006, Fact 3.11]. Additionally it states that when D exists, then it is unique up to isomorphism.

Note that a transformation rule has a certain symmetry, in that there is no particular difference between the left and right sides, except for the names we have given them. Thus we can immediately define the inverse derivation $H \overset{p^{-1},m'}{\Longrightarrow} G$, using the inverse transformation rule $p^{-1} = (R \overset{r}{\leftarrow} K \overset{l}{\rightarrow} L)$ and the co-match m'. This is a quite useful property for the modelling of chemistry, where reactions in general are invertible.

We can define different classes of graph transformation by restricting l, r, or m to be monomorphisms, which for example is explored in [Habel *et al.* 2001], though with l always restricted to be a monomorphism such that uniqueness of D is guaranteed. In order to maintain uniqueness when

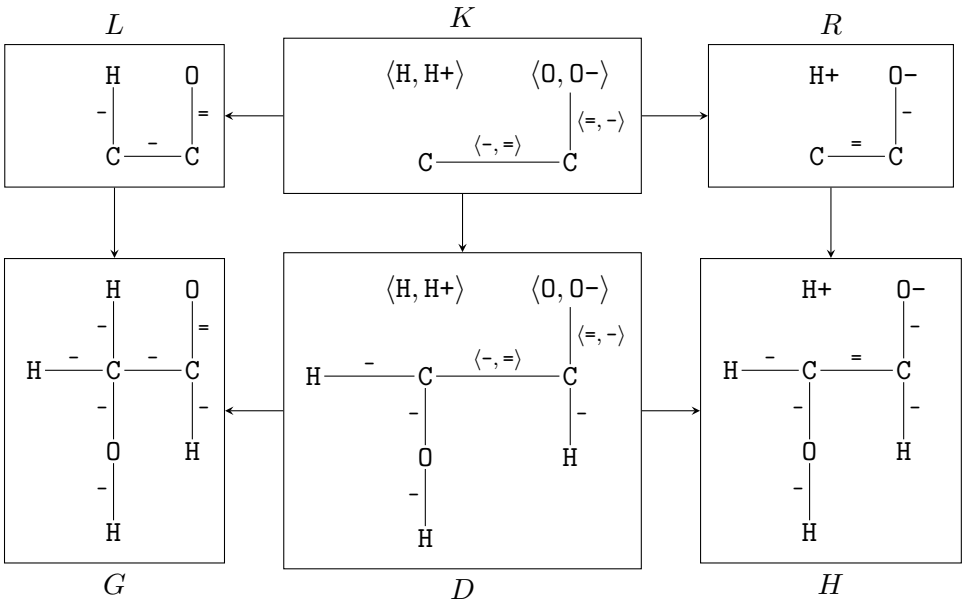


Figure 6.7: Example of a derivation using labelled graphs. The context graph K and intermediary graph D are both labelled with pairs of strings, while the remaining graphs are labelled with strings. To reduce clutter in K and D we depict pairs with equal components $\langle S, S\rangle$ simply as S.

rules are inverted we also restrict r to be a monomorphism. We additionally restrict the matching morphism m to be a monomorphism, as a atoms could otherwise be merged when matched by a rule. Note that with this restriction the identification condition is always fulfilled.

6.3 Labelled Graph Transformation

For our purposes we need that a transformation rule not only specifies a structural change of graphs, but also the possible change of the labels on vertices and edges. This gives rise to the question of how to label the context graph of both rules and derivations. Let Ω be the set of possible labels to be used on graphs to be transformed. We then define that a transformation rule $(L \xleftarrow{l} K \xrightarrow{r} R)$ is labelled such that L and R are labelled graphs over Ω while K is a labelled graph over $\Omega \times \Omega$. The morphism l restricted to the labels is the function $fst\colon \Omega \times \Omega \to \Omega$ returning the first component of an ordered pair, and likewise for r the label restriction is the function $snd\colon \Omega \times \Omega \to \Omega$ returning the second component. The intermediary context graph D of a derivation (see Figure 6.6), is as K a graph with pairs of labels. Figure 6.7 illustrates a derivation where labels are changed. In the graphs K and D we have simplified the visualisation such that a pair where the first and second

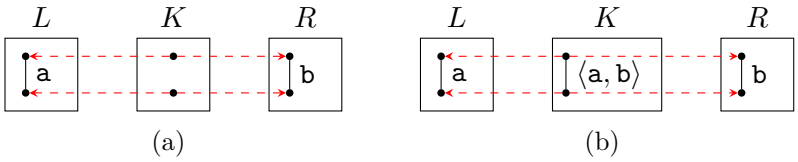


Figure 6.8: The rule in (a) models the removal of an edge with label `a` and the addition of the same edge but with label `b`. We do not represent this kind of transformation, but instead allow rules to change labels as in the rule in (b).

component are the same, are depicted simply as the common label. To further reduce clutter in illustrations in the following chapters, we omit edges from K and D when they change labels.

6.4 Representation of Transformation Rules

A transformation rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ in the category of simple graphs could naturally be represented directly by three graphs and two vertex maps. However, this is arguably a rather verbose representation when restricted to monomorphisms where K is a subgraph of both L and R and thus stored three times. To obtain a more compact representation that also allows for simpler algorithm implementations we disallow certain rules. Consider the rule depicted in Figure 6.8a, which models first the removal and then the addition of an edge, but with a different label. Figure 6.8b show a functionally equivalent rule, which models the label change directly, i.e., in the underlying graph the edge is invariant. Only if we were to attach auxiliary data to edges then the application of these two rules would have observable differences.

To simplify our representation of a rule we opt to not allow rules such as in Figure 6.8a, that is for all rules $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ we require that for all two vertices $u, v \in K$, if $(u, v) \notin E_K$ then $(l(u), l(v)) \notin E_L \vee (r(u), r(v)) \notin E_R$. Now let $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ be a labelled transformation rule as described in the previous section, where Ω is the label set for L and R and $\Omega \times \Omega$ is the label set for K. We can then create a new undirected, labelled graph $\mathcal{C}_p = (V_p, E_p, m_p)$ being the pushout object of $L \xleftarrow{l} K \xrightarrow{r} R$. The vertices and edges are labelled with the function $m_p \colon V_p \cup E_p \to \Omega' \times \Omega'$, where $\Omega' = \Omega \cup \{\textbf{nil}\}$ is the original label set augmented with a distinct new label **nil** that will indicate the absence of a label. Each vertex in V_p and edge in E_p were created because they were in one of $L \backslash l(K)$, $R \backslash r(K)$, or K. For vertices/edges created from $L \backslash l(K)$ with label α in L we attach the new label $\langle \alpha, \textbf{nil} \rangle$. Similarly a vertex/edge in $R \backslash r(K)$ have a label on the form $\langle \textbf{nil}, \beta \rangle$, and a vertex/edge in K on the form $\langle \alpha, \beta \rangle$. Clearly the original rule can be recovered from the placement of the **nil** labels.

Figure 6.9 shows how the rule $p = (L, K, R)$ from Figure 6.7 is represented

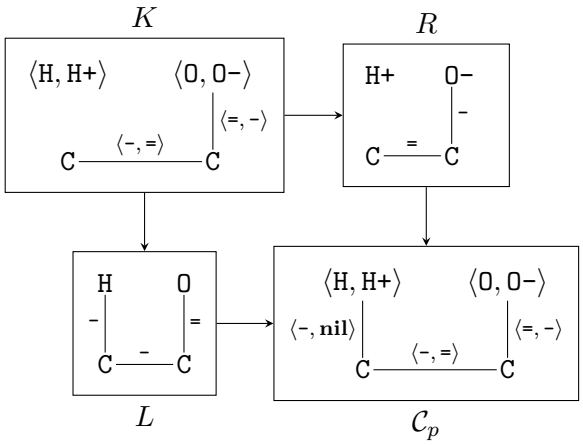


Figure 6.9: Pushout diagram for the construction of the graph $\mathcal{C}_p$ representing the rule $p = (L, K, R)$ from Figure 6.7.

by the pushout object $\mathcal{C}_p$.

Note that the rules that are not representable in this manner are exactly those for which we have previously opted not to define a pushout in the category of simple graphs.

6.5 Chemical Graph Transformation

In the modelling of chemistry we use transformation rules in the Double Pushout formalism to instantiate chemical reactions. The DPO rules specify transformation in a two-step process, first removal and then addition. Labelled graph transformation rules can further specify that the label of a vertex or edge changes. A chemical reaction can similarly be described as sets of bonds to be broken, formed, or changed, e.g., from a single bond to a double bond.

Recall that a reaction is an ordered pair (G, H) of multisets of molecules (see Section 1.1.3), where we here assume that a molecule is a graph. We can view the educts G and products H each as a single disconnected graph, where each connected component corresponds to the individual graphs of the multisets G and H. We can interpret a DPO derivation $G \overset{p,m}{\Longrightarrow} H$ as a chemical reaction, simply by forgetting p and m, and thus we can discover reactions through graph transformation.

In the area of formal grammars we have objects, usually strings, and rewrite rules that transform a single object into a single other object, and the generalisation to graph grammars is straightforward: the objects are graphs and the rewrite rules each transform one graph into another. However, for chemical graph transformation we want to work with connected graphs as the individual objects, but apply transformation rules to multisets of those

graphs, in order to allow for molecules to merge and split. We formally describe this alternate kind of a graph grammar and a graph language in the following manner.

Definition 6.5 (Graph Grammar and Graph Language)**.** For a graph category $\mathbf{C}$ let $\mathbf{C}'$ be the subcategory of $\mathbf{C}$ restricted to the connected graphs. A *graph grammar* over $\mathbf{C}$ is an ordered pair $(\mathcal{G}, \mathcal{P})$, with $\mathcal{G}$ as a finite set of initial graphs with $\mathcal{G} \subseteq Ob(\mathbf{C}')$ and $\mathcal{P}$ as a finite set of Double Pushout transformation rules in $\mathbf{C}$. The *language* of a grammar $L(\mathcal{G}, \mathcal{P})$ is the subset of $Ob(\mathbf{C}')$ derivable from the initial graphs using the given transformation rules:

$$\begin{aligned} \mathcal{G}_0 &= \mathcal{G} \\ \mathcal{G}_i &= \left\{ h \in H \,\middle|\, G \overset{p}{\Rightarrow} H, p \in \mathcal{P}, G = \bigcup_{0 \leq j < i} \mathcal{G}_j \right\} \qquad i > 0 \\ L(\mathcal{G}, \mathcal{P}) &= \bigcup_{i \geq 0} \mathcal{G}_i \end{aligned}$$

From a chemical perspective we can loosely say that a graph grammar is a formalisation of a *chemistry*, consisting of starting compounds and a specification of the types of reactions we assume can take place. We then say that the language of a chemical graph grammar is the *chemical space* or *chemical universe* of the chemistry (see also [Grzybowski *et al.* 2009, Dittrich *et al.* 2001]). In many cases we can not fully describe a chemistry only by a graph grammar, e.g., because the molecules have bounded size. In Chapter 9 we describe an algorithmic framework for generating a graph language with further constraints, and for formal computations with graph grammars.

The special use of connectedness gives rise to a classification of derivations. Consider an arbitrary derivation $\{\!\{g_a, g_b, g_b\}\!\} \overset{p,m}{\Longrightarrow} \{\!\{h_c, h_d\}\!\}$, and let q be another connected graph. Then we can construct an extended derivation $\{\!\{g_a, g_b, g_b, q\}\!\} \overset{p,m}{\Longrightarrow} \{\!\{h_c, h_d, q\}\!\}$ using the same rule and match, but where q is simply added on both sides. In the context of chemistry this corresponds to augmenting a chemical equation with another molecule, even though it does not participate in the reaction. We call the derivations where the left- and right-hand sides are minimal *proper*:

Definition 6.6 (Proper derivation)**.** A derivation $G \overset{p,m}{\Longrightarrow} H$ with the left-hand side $G = \{\!\{g_1, g_2, \ldots, g_n\}\!\}$ is *proper* if and only if

$$g_i \cap \mathrm{img}(m) \neq \emptyset, \forall 1 \leq i \leq n$$

That is, all connected components of G are hit by the match.

With the above definition we now say that only proper derivations should be interpreted as chemical reactions.

6.5.1 Atom Maps and Validity of Adjacency Changes

Chemically valid graph transformation rules have the special property that they neither delete nor create vertices. This would amount to destroying and creating atoms, which is well beyond the scope of biochemistry. For a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ the differences between the three graphs are thus restricted to the edge set, and in a derivation $G \overset{p}{\Rightarrow} H$ we then preserve the vertices. For a derivation we can therefore define a bijection $a\colon V_G \to V_H$, between the vertices of G and H. Using the morphism names of Figure 6.6 we can define the map as $a = r' \circ l'^{-1}$, with the restriction to the vertex sets. In chemistry this map is called the *atom map* of a reaction, and it plays an important role for understanding chemical systems (see Chapter 11). We discover reactions using graph transformation, so the atom maps are explicitly determined by the model. However, atom maps are usually not present in the various chemical databases and it then becomes a challenge to enumerate the chemically valid maps, e.g., see [Chen *et al.* 2013, Mann *et al.* 2013c].

As for molecules we only informally use the notion of chemical validity, and we do not restrict algorithms or discussions to rules with invariant vertex sets. Note though that when no vertices are deleted or created, then the dangling condition of DPO transformation is trivially fulfilled for the rule and its inverse [Andersen *et al.* 2013b].

In our molecule model (see Section 2.1) we have not included formal valence requirements for atoms. This is not only because of the complication of constructing a well-defined rule set, but also because the use of the Double Pushout formalism to a large extend removes the need for such constraints. Recall that during transformation of a graph there are no side-effects, and all changes are explicitly described by the rule. If we assume the input graphs are valid molecules, then we can simply require that for a rule to be chemically valid its transformation must be invariant with respect to the chemical neighbourhood constraints. For example, if a rule specifies the removal of a double bond, incident to a carbon atom, then from Table 2.1 we see that the matching carbon must have either the neighbourhood $\{\!\{\textsc{Single}, \textsc{Single}, \textsc{Double}\}\!\}$ or $\{\!\{\textsc{Double}, \textsc{Double}\}\!\}$. In order for the rule to guarantee the products are valid molecules, then it must add a double bond again or add two single bonds. Similar reasoning can be used for all other cases, but we do not provide a full formalisation in this work. Informally we say that a chemical valid transformation rule must conform to this yet to be defined set of adjacency constraints. In Section 15.1 we briefly sketch the introduction of local geometry into the molecule model, which naturally leads to a potential formalisation.

6.5.2 Comparison of the DPO and SPO Formalism

This section is based on [Andersen et al. 2013b, Appendix A].

In order to model reaction patterns as graph transformation we essentially

just need a formal method for specifying the transformation of a subgraph L into a subgraph R when matched on a graph G. The Double Pushout formalism provides such a method, but so does the *Single Pushout* (SPO) formalism. In SPO a rule is specified as a partial morphism $p\colon L \rightarrowtail R$, meaning the formalism works in a different graph category than DPO, where the morphisms do not need to be total functions, and not preserve edges in the usual manner. A detailed comparison of SPO and DPO can be found, e.g., in [Ehrig *et al.* 1997] and [Löwe 1993], while here we briefly state their difference when considering chemical transformation.

Even though the rules in the two formalisms are differently specified we can covert between the two. Given an SPO rule $p\colon L \rightarrowtail R$ we construct the an equivalent DPO rule $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ by letting K be the domain of p, with r as the restriction of p to its actual domain, and l as the monomorphism that embeds K into L. Conversely, given a DPO rule $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, with l injective, we can construct the equivalent SPO rule $p\colon L \rightarrowtail R$ as $p = r \circ l^{-1}$. In the following descriptions we use the DPO specification format.

The main difference between the two formalisms is in how a rule is used to transform graphs. Where the DPO formalism rejects a candidate transformation when the match do not fulfil the gluing condition, the SPO formalism still has a well-defined transformation by performing extra deletions: The dangling condition is concerned with the edges $E_G \backslash m(E_L)$ that are incident to a vertex in $m(V_L \backslash l(V_K))$. These edges are deleted in an SPO transformation to resolve the problem. Recall that the identification condition states (for vertices) that we can not have distinct vertices $u, v \in V_L$ with $m(u) = m(v)$, such that $u \in l(V_K)$, $v \notin l(V_K)$, because it means the vertex $m(u) = m(v)$ must be both deleted and preserved. In the SPO formalism such conflicts are resolved for both vertices and edges by deleting the offending vertex/edge.

Another way to view the difference between the formalisms is that DPO rules are pure, in the sense that they specify all changes, while the changes of SPO rules may extend beyond what the rules specify. However, if we are only concerned with chemical transformations where the matching morphism is injective and no vertices are created or deleted, then the gluing condition is trivially fulfilled. Thus for chemical transformation the SPO and DPO formalisms are equivalent.

We primarily focus on chemical transformations but when convenient, e.g., in certain rule compositions, we introduce non-chemical rules. In these cases we find it easier to work with DPO rules which do not have side effects, and are based on total morphisms instead of partial morphisms. That DPO rules do not have side effects additionally means that they are always invertible, simply by exchanging the left and right graphs and the morphisms. Further, for modelling chemistry the DPO formalism is particular appealing as it explicitly exposes the context graph, which is related to the concept of transition states in chemical reactions.

Chapter 7

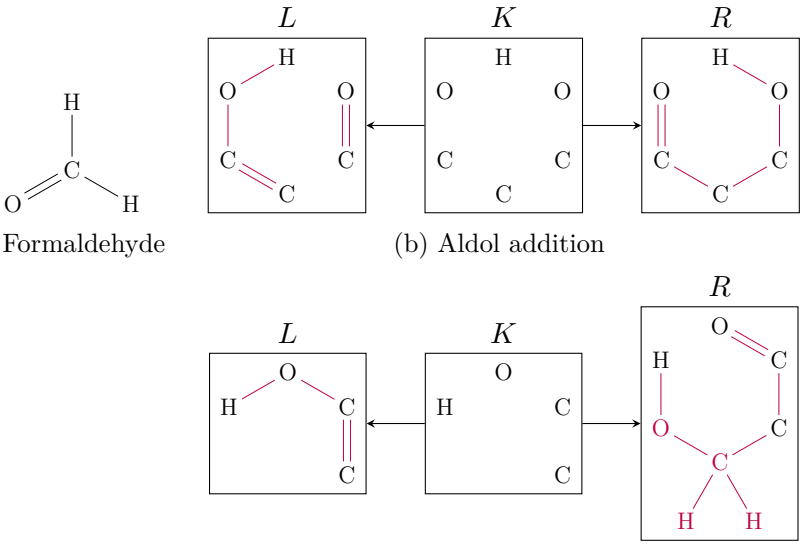
Composition of Transformation Rules

This chapter is an expanded version of the methods described in [Andersen et al. 2013b] and [Andersen et al. 2014b].

For ordinary mathematical functions with multiple arguments there is the concept of *partial application*. For example, if $f(x, y) = x^y$ then we can define a new function $f'(x) = x^2$ by partially applying f to the number 2 in the second position. A specialised form of partial application, called *currying*, is an integral part of the programming language Haskell. With the chemical view on DPO rules, where they are applied not just to a single graph but to a multiset of connected graphs, we can view a rule $p = (L, K, R)$ with k connected components in L as a function of up to k unordered arguments. Similarly to partial function application we can then imagine that a transformation rule can be partially applied to a graph, with the result being a new rule, with fewer connected components in its left side graph. For example, the rule depicted in Figure 7.1b have two connected components in the left-hand graph, and it can therefore be applied to either an enol and a molecule with a carbonyl group, or a single molecule with both features. We can now derive a new rule, Figure 7.1c, where a formaldehyde molecule have been bound to one of the components of the original rule. Suppose we bind another graph to the new rule. The last component would then be used and the resulting rule have the form $(\emptyset, \emptyset, H)$. Such a rule is essentially equivalent to a constant function, implying that a DPO derivation can be calculated by iterated graph binding.

A more general concept than partial application is function composition. Consider two DPO rules $p_1 = (L_1, K_1, R_1)$ and $p_2 = (L_2, K_2, R_2)$ and suppose we have a monomorphism from L_2 to R_1. Then if we first have a derivation $G_1 \overset{p_1}{\Rightarrow} G_2$, then we for sure can find a match of L_2 in G_2 and potentially a derivation $G_2 \overset{p_2}{\Rightarrow} G_3$. If we can define a composition $p_2 \circ p_1$ the two derivations can be combined into $G_1 \overset{p_2 \circ p_1}{\Longrightarrow} G_3$.

The concept of rule composition is in the area of graph transformation related to both *D-concurrency* [Ehrig *et al.* 1991] and *E-concurrency* [Ehrig *et al.* 2006, Golas 2010]. In this chapter we first describe the most general form of composition, similar to D-concurrency, and then several special cases with relevance to the modelling of chemistry. Secondly we describe algorithms for enumerating compositions, and how to use them for enumerating derivations starting from a collection of graphs.



(a) Formaldehyde (b) Aldol addition

(c) Aldol addition with formaldehyde bound

Figure 7.1: An example of partial application of a graph transformation rule. Formaldehyde (a) can be bound to one of the components of the left-hand graph of aldol addition (b). The resulting rule (c) represents the addition of formaldehyde to an enol.

7.1 Classes of Composition

In the general case a composition of two rules $p_i = (L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i)$ for $i = 1, 2$, is done by means of a common subgraph of L_1 and R_2. Formally (see also [Ehrig *et al.* 1991, Section 6.2]) we say that p_1 and p_2 are composed using a graph D with morphisms $d_1 \colon D \to R_1, d_2 \colon D \to L_2$, and write $p = p_1 \bullet_D p_2$ as shorthand for the composed rule if it exists. Note that the order of the operands is the reverse compared to the usual composition operator $\circ$, such that p is equivalent to first applying p_1 and then p_2. The composed rule exist if the diagram in Figure 7.2 exists with the squares (1), (2), (2'), (3), (3'), and (4) all being pushouts. We then have $p = p_1 \bullet_D p_2 = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ with $l = s_1 \circ w_1$ and $r = t_2 \circ w_2$ as the resulting composition. Algorithmically we can describe the construction of p as

1. Construct E as the pushout object of (1).
2. Construct C_1 and C_2 as the pushout complement objects of respectively (2) and (2').
3. Construct L and R as the pushout objects of respectively (3) and (3').
4. Construct K as the pullback object of (4).

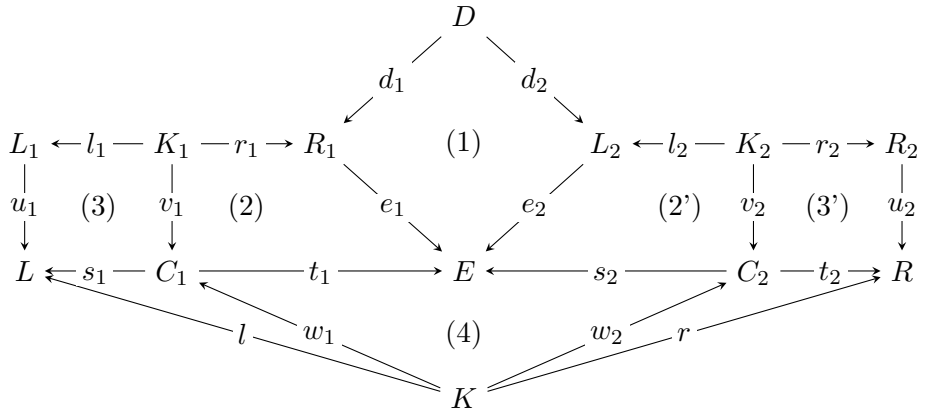


Figure 7.2: Commutative diagram for general rule composition [Ehrig *et al.* 1991, Section 6.2]. The two rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ for $i = 1, 2$ are composed using the common common graph D and the morphisms d_1 and d_2. The resulting rule is $p = p_1 \bullet_D p_2 = (L \xleftarrow{l} K \xrightarrow{r} R)$ with $l = s_1 \circ w_1$ and $r = t_2 \circ w_2$.

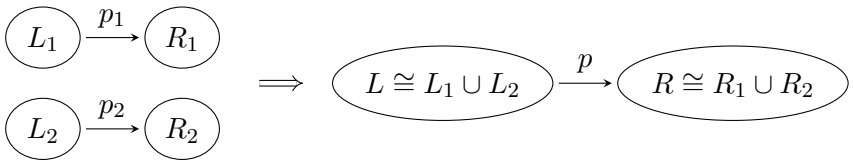


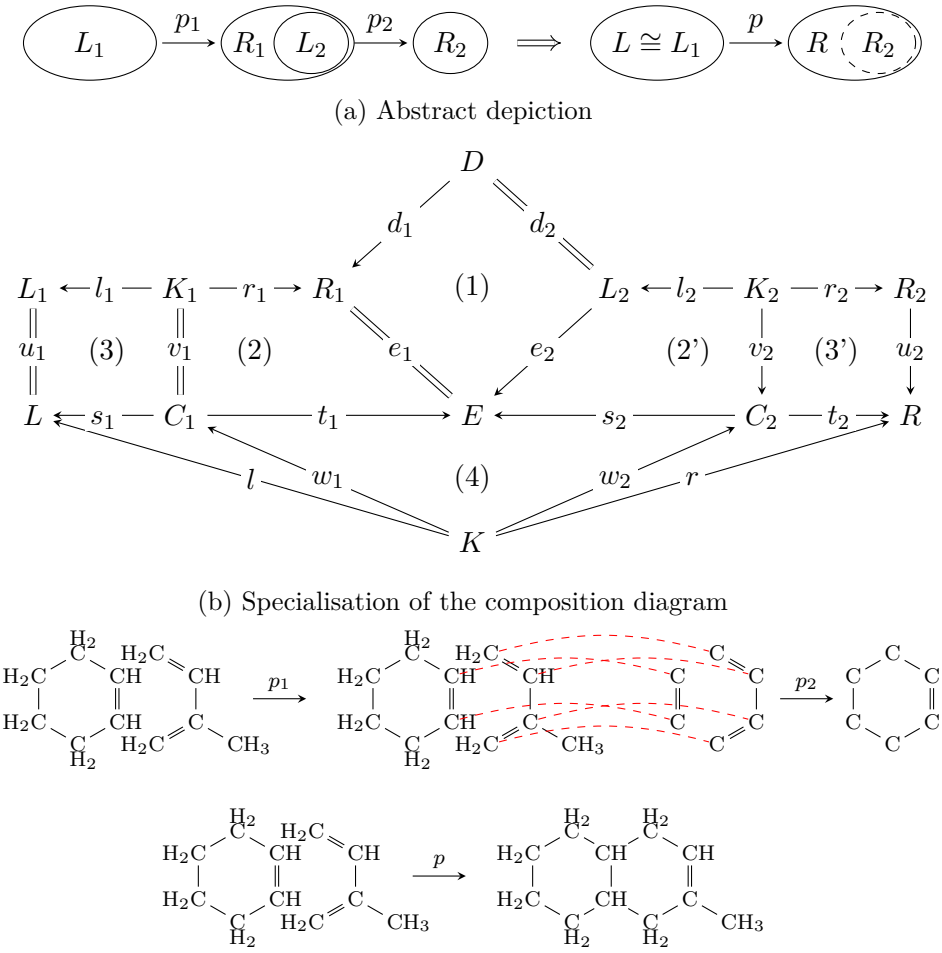
Figure 7.3: Composition $p = p_1 \bullet_\emptyset p_2$ with an empty common subgraph, giving a composed rule which models the parallel transformation using p_1 and p_2.

If any of the constructions are not defined, then the composition is not defined.

In the followings section we describe simplified cases of composition for specific choices of the common subgraph D.

7.1.1 Parallel Composition

For D being the empty graph we can always compose the rules, and obtain a rule which combines the effect of p_1 and p_2, i.e., $p = (L_1 \cup L_2 \xleftarrow{l_1 \cup l_2} K_1 \cup K_2 \xrightarrow{r_1 \cup r_2} R_1 \cup R_2)$. Intuitively we can see this using the diagram in Figure 7.2, where $D = \emptyset$ means that pushout (1) degenerates to a disjoint union, i.e., $E = R_1 \cup L_2$. The image of e_1 is thus disjoint from the copy of L_2, and the completion of pushouts (2) and (3) then propagates L_2 into C_1 and L without modification. Symmetrically R_1 is propagated into R. In short we write this merging of two rules as $p = p_1 \bullet_\emptyset p_2$ and visualise it as in Figure 7.3.



(a) Abstract depiction

(b) Specialisation of the composition diagram

(c) Chemical example

Figure 7.4: Full composition $p = p_1 \bullet_{\supseteq} p_2$ where D is a copy of L_2 and d_2 is an isomorphism. (a) Abstract depiction; L_2 is isomorphic to a subgraph of R_2. (b) Specialised commutative diagram for full composition. As d_2 is an isomorphism so will e_1, v_1 and u_1 be. (c) Chemical example; $p_1 = (G, G, G)$ is the identity rule for a graph G encoding the educts cyclohexene and isoprene. The second rule, p_2, is the transformation rule for the Diels-Alder reaction. The composed rule therefore encodes the overall rule of the Diels-Alder reaction on the input molecules. The context graphs and D are omitted from the drawings for simplicity. The monomorphism $d_1 \circ d_2$ is shown with red dashed lines.

7.1.2 Full Composition

When we see DPO rules as a kind of abstract functions where the left- and right-side graphs respectively are the pre- and postconditions we can look at the case where the precondition of p_2 is fulfilled completely by the post-condition of p_1, as illustrated in Figure 7.4a. We call this special case *full*

composition and formally specify it as when $D \cong L_2, d_2 = \mathrm{id}_{L_2}$ and d_1 being a monomorphism. The effect of d_2 being the identity morphism, and thereby an isomorphism, is illustrated in Figure 7.4b. Because (1) is a pushout we must have e_1 being an isomorphism, which in turn makes both v_1 and u_1 isomorphisms as well.

A chemical example of full composition is shown in Figure 7.4c, where p_1 additionally is a special identity rule that requires a specific graph and does not change it. The effect of this choice of p_1 is discussed in Section 7.2. For full composition we note that due to the complete embedding of L_2 in R_1 we have $L \cong L_1$, meaning the resulting rule have the same precondition as p_1. As shorthand we may write $p_1 \bullet_{\supseteq} p_2$ to denote an arbitrary full composition or the enumeration of all such compositions, depending on the context.

We can additionally define the symmetric kind of full composition, $p_1 \bullet_{\subseteq} p_2$ where d_1 is an isomorphism and d_2 a monomorphism. Further we have the special case $p_1 \bullet_{\cong} p_2$ where both d_1 and d_2 are isomorphisms. However, we reserve the term *full composition* for $p_1 \bullet_{\supseteq} p_2$ which particularly is useful for a method for calculating atom traces, see Chapter 11.

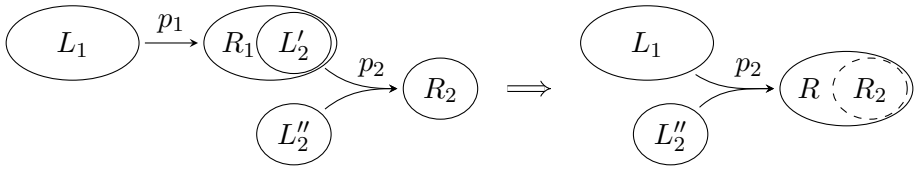
7.1.3 Partial Composition

For chemical graph transformation we have the perspective that graphs can be interpreted as multisets of their connected components, and if the rule p_2 models the merging of two molecules then L_2 will have two connected components. In a full composition we had both of those components embedded in R_1, but we can generalise to the case where a subset of the components are embedded in R_1, as visualised in Figure 7.5a.

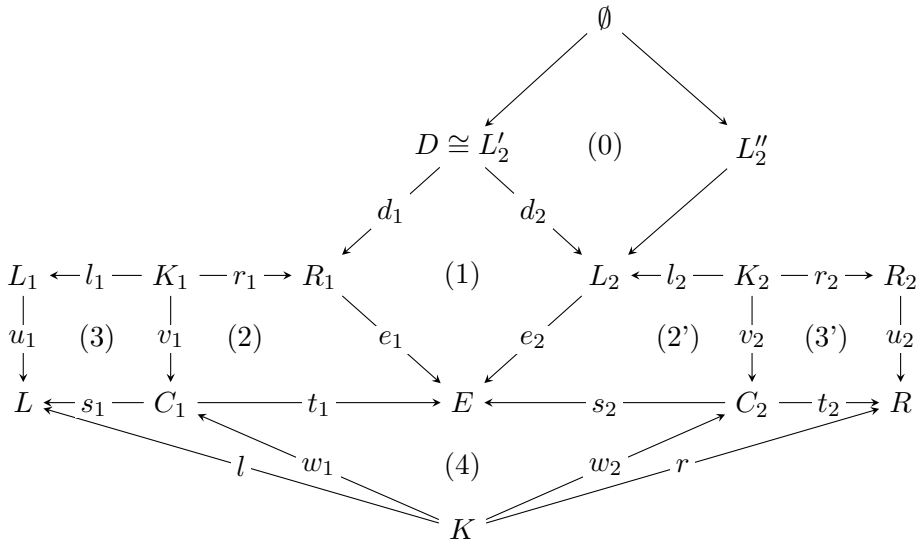
Formally we can describe this type of composition by letting $D \cong L_2'$ and introducing the graph L_2'' such that L_2 is the disjoint union of L_2' and L_2'', illustrated in Figure 7.5b. As for full composition we require d_1 to be a monomorphism. Note that this specification generalises both parallel and full composition, but we however refer to parallel composition explicitly and do not include it in partial composition, i.e., we require $D \ncong \emptyset$. As shorthand we write $p_1 \bullet^{c}_{\supseteq} p_2$ for an arbitrary partial composition, or all of them, due to the splitting of L_2 into components and requiring $R_1 \supseteq D$.

Figure 7.5c shows a chemical example with partial composition, where L_2 has two connected components. The smaller component is used as the common subgraph D is thus not a precondition in the resulting rule, while the other component is preserved in L.

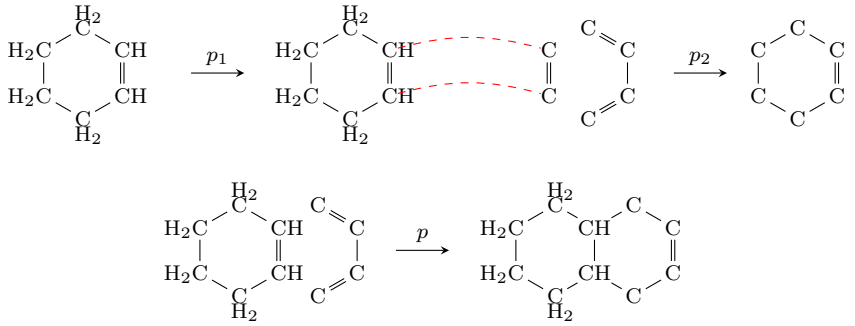
In the symmetric case of partial rule composition we can instead require components of R_1 to be a subgraph of L_2 and write $p_1 \bullet^{c}_{\subseteq} p_2$.



(a) Abstract depiction

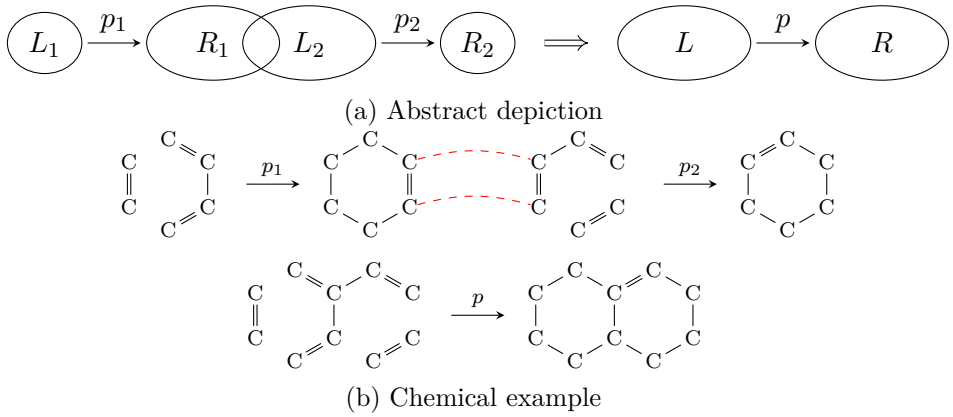


(b) Specialisation of the composition diagram



(c) Chemical example

Figure 7.5: Partial composition $p = p_1 \bullet^{c}_{\supseteq} p_2$ where D is a copy of a non-empty subset of the connected components of L_2, and d_2 is the inclusion morphism back into L_2. (a) Abstract depiction; connected components of L_2 are either completely matched into R_2 or not at all. (b) Specialised commutative diagram for partial composition. The selection of connected components of L_2 to form D is specified by pushout (0), that degenerates to a disjoint union. To exclude parallel composition from partial composition we require $D \ncong \emptyset$. (c) Chemical example; p_1 is the identity rule for cyclohexene and p_2 is the Diels-Alder transformation rule. The composed rule encodes the partial application of the Diels-Alder reaction to the molecules, leaving the diene to be instantiated at a later stage. The context graphs and D are omitted for simplicity. The partial morphism $d_1 \circ d_2$ is shown with red dashed lines.



(a) Abstract depiction

(b) Chemical example

Figure 7.6: General rule composition where D is a common subgraph of both R_1 and L_2. The commutative diagram is shown in Figure 7.2. (a) Abstract depiction, with the common subgraph D shown as the intersection of R_1 and L_2. (b) Chemical example; the rule for the Diels-Alder reaction is composed with itself. The context graphs and D are omitted for simplicity. The partial morphism $d_1 \circ d_2$ is shown with red dashed lines.

7.1.4 General Composition

We now briefly return to the most general case of composition where D simply must be a common subgraph of R_1 and L_2. As shorthand notation we use $p_1 \bullet_{\cap} p_2$, and visualise the relation abstractly as in Figure 7.6a, while a chemical example is shown in Figure 7.6b.

Several further subclasses of composition can be defined, for example requiring D to be maximal with respect to inclusion in R_1 and L_2, but we have not encountered a natural use of further classes in the context of chemistry.

7.2 Binding, Unbinding, and Identification of Graphs

In the beginning of this chapter we argued that given a graph and a rule we can define how to bind the graph onto the rule and obtain a new rule modelling the partial application. To formalise graph binding we now consider a rule $p_1 = (\emptyset, \emptyset, G)$ to be equivalent to the graph G, since it models the unconditional creation of G. The creation of H in the derivation $G \overset{p_2,d_1}{\Longrightarrow} H$ is then equivalent to the full composition $p_1 \bullet_{\supseteq} p_2 = (\emptyset, \emptyset, G) \bullet_{\supseteq} p_2 = (\emptyset, \emptyset, H)$ (see Figure 7.4b). Consider now a division of L_2 into the disjoint subgraphs L_2' and L_2'', and assume we want to model the binding of G to L_2'. We can then see this as the partial composition $p = (\emptyset, \emptyset, G) \bullet_{L_2'} p_2$, illustrated in Figure 7.7. From the division of L_2 into disjoint graphs and the definition of pushouts we have that the resulting composed rule have L_2'' as its left-side graph, which

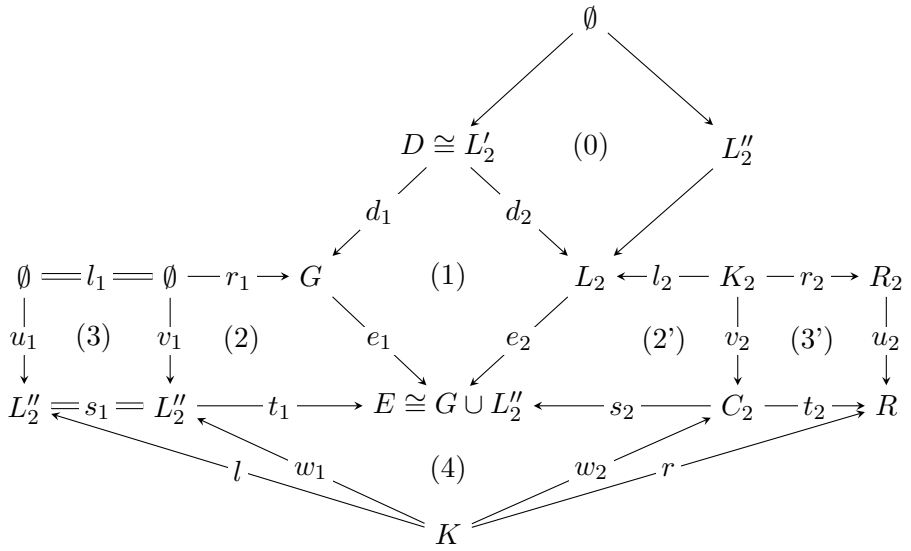


Figure 7.7: Commutative diagram for binding a graph G to a rule p_2 by reduction to a partial composition $(\emptyset, \emptyset, G) \bullet^c_{\supseteq} p_2$. Pushout (0) makes L_2 the disjoint union of L'_2 and L''_2, while the morphism d_1 embeds L'_2 into G. For (1) to be a pushout we thus have E being the disjoint union of G and L''_2. Pushouts (2) and (3) then propagates L''_2, the unbound part of L_2, to be the left-side graph of the resulting rule.

exactly was the part of L_2 we did not want to bind onto.

In Section 7.4 we use graph binding to define an algorithm for enumerating derivations from a set of input graphs, but graph binding also have direct use in a chemical setting. Many reactions in biochemistry have compounds, such as water, as educts and products that are present in a high quantity such that we can argue they are always present. Reaction patterns, in form of DPO rules, can then be simplified by binding the abundant compounds to obtain the rules modelling only the "interesting" part of the reaction pattern. Graph binding can only simplify the educt side of the reaction pattern, while for the product side we need to use the symmetric composition $p_1 \bullet^c_{\supseteq} (H, \emptyset, \emptyset)$, which we then refer to as *graph unbinding*.

The third trivial manner to create a rule from a graph G is to create the identity rule (G, G, G). In Chapter 11 we describe how we use such identity rules together with full composition to calculate atom traces. The property we use is that the result of the composition $(G, G, G) \bullet_{\supseteq} p_2$ is a rule (G, D, H) corresponding the lower half of the usual DPO diagram for the derivation $G \overset{p_2}{\Longrightarrow} H$.

Based on their use we refer to the rules $(\emptyset, \emptyset, G)$, $(G, \emptyset, \emptyset)$, and (G, G, G) respectively as the *bind rule*, *unbind rule*, and the *identity rule* for the graph G.

	R_1^1	R_1^2	R_1^3
L_2^1	1		2
L_2^2		1	1

(a) Match matrix

	R_1^1	R_1^2	R_1^3	$R_1^\emptyset$
L_2^1	1		2	1
L_2^2		1	1	1

(b) Extended match matrix

Figure 7.8: Example of (a) a match matrix and the same matrix with (b) its virtual extension to model unmatched connected components. The top row specifies 1 possibility for $L_2^1 \subseteq R_1^1$ and 2 for $L_2^1 \subseteq R_1^3$. The extended matrix further specifies that L_2^1 can be unmatched. The bottom rows can be interpreted similarly. We display the number of morphisms instead of a representation of the morphisms themselves.

7.3 Enumeration of Partial and Full Compositions

In the previous sections we defined classes of composition by different constraints on the common subgraph D and the morphisms d_1 and d_2. For composition of two rules p_1 and p_2 we may wish to enumerate all possible compositions within a certain class. In the case of full composition we can clearly enumerate all morphisms $d_1 \colon L_2 \to R_1$ using the algorithms described in Section 3.3, while for parallel composition there is only a single composition as D must be the empty graph. Partial composition is somewhat more complicated as we need to enumerate all partitions of L_2 into $\{\!\{L_2', L_2''\}\!\}$ and for each of those enumerate all monomorphisms $d_1 \colon L_2' \to R_1$. In practice we wish to avoid creating the intermediary graph L_2' and instead work with partial morphisms $\mu \colon L_2 \rightarrowtail R_1$ such that $\mathrm{dom}(\mu) = \mathrm{img}(d_2)$. Note that μ must be injective as $d_1 \colon D \to R_1$ is required to be a monomorphism. In the following we describe an algorithm which enumerates all such μ for partial composition.

Let k_1 and k_2 be the number of connected components of respectively R_1 and L_2, and then let $R_1 = \left\{\!\!\left\{ R_1^1, R_1^2, \ldots, R_1^{k_1} \right\}\!\!\right\}, L_2 = \left\{\!\!\left\{ L_2^1, L_2^2, \ldots, L_2^{k_2} \right\}\!\!\right\}$ for an arbitrary ordering of the components. As a first step we compute all sets of monomorphisms $M_{i,j} = \{L_2^i \to R_1^j\}$ and arrange them in a $k_2 \times k_1$ matrix. Figure 7.8a illustrates an example of a match matrix, with $|M_{i,j}|$ written in each cell. Only a subset of the components of L_2 need to be selected for the final morphism, so we extend the match matrix with a virtual column $R_1^\emptyset$ as illustrated in Figure 7.8b, where we imagine a single morphism to exist. All selections of 1 morphism in each row are now candidates for combination into a final partial morphism $\mu \colon L_2 \rightarrowtail R_1$. We have explicitly defined partial composition such that it does not include parallel composition, meaning the choice of the virtual morphism in column $R_1^\emptyset$ for all rows at the same time is not a candidate. Additionally, the same component of R_1 can be selected by multiple components of L_2, e.g., in Figure 7.8 both L_2^1 and L_2^1 can select R_1^3. If the images of the selected morphisms are not disjoint then the combined

morphism μ is not injective, and must therefore be excluded.

By enumeration of all morphism selections under the mentioned conditions we clearly enumerate all viable partitions $L_2 = \{\!\{L_2', L_2''\}\!\}$, with L_2'' consisting of the components where the $R_1^\emptyset$ column is selected. As all componentwise monomorphisms are represented in the matrix, the enumeration of selections will enumerate all monomorphisms for each partition of L_2.

Even though full compositions can be enumerated by direct enumeration of monomorphisms $L_2 \to R_1$, we in practice reuse the algorithm described above, though without the virtual column $R_1^\emptyset$.

A complete example for enumeration of all partial rule compositions can be found in [Andersen *et al.* 2013b, Appendix B].

7.4 Derivation by Repeated Graph Binding

In Section 7.2 we argued that a derivation $G \overset{p}{\Rightarrow} H$ is equivalent to a full composition $(\emptyset, \emptyset, H) = (\emptyset, \emptyset, G) \bullet_\supseteq p$, and we thus do not need a dedicated algorithm for computing derivations. We can now consider the following enumeration problem: given a set of graphs $\mathcal{G}$ and a rule $p = (L, K, R)$, compute all proper derivations $G \overset{p}{\Rightarrow} H$ with $G \subseteq \mathcal{G}$. This is a simplified subproblem used in the framework described in Chapter 9, and arguably a subproblem of generating the language of a graph grammar (Definition 6.5). The problem could theoretically be solved using the match matrix approach described in the previous section, with $p_1 = (\emptyset, \emptyset, \mathcal{G})$ and special handling to recover the actual derivations. However, for the framework in Chapter 9 the set of graphs $\mathcal{G}$ is growing dynamically and can become very large, meaning it may not be feasible to pre-compute all individual morphisms. In a future implementation the match matrix algorithm and the approach described below could potentially be unified.

Let k be the number of connected components of L, then we can solve the problem by enumerating all multisets of graphs from $\mathcal{G}$ with size 1 through k, and then enumerate all derivations that are proper using each multiset as G. Note that the number of multisets of cardinality k from a ground set of size n, also known as the number of multicombinations or k-combinations with repetition, is given by the multichoose function

$$\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$$

In the extreme case each multiset gives a unique derivation, but in practice we observe that some graphs can never be matched by the rule and thus many multisets are never viable.

In order to adapt to the cases of unmatchable graphs we use repeated graph binding to compute derivations. The details of this method is presented in Algorithm 1. The main idea is that all derivations $G \overset{p}{\Rightarrow} H$ with $G =$

Algorithm 1: Computing derivations by repeated graph binding.

Input: $\mathcal{G}$, a list of unique graphs, in arbitrary order.
Input: $p = (L, K, R)$, a transformation rule.
1 $k \leftarrow$ number of connected components of L
2 $Q_0 \leftarrow \{\langle p, \{\!\{\}\!\}, 1\rangle\}$
3 **for** $i = 1$ **to** k **do**
4 $Q_i \leftarrow \emptyset$
5 **foreach** $\langle p', G', m\rangle \in Q_{i-1}$ **do**
6 **for** $j = m$ **to** $|\mathcal{G}|$ **do**
7 $g \leftarrow \mathcal{G}[j]$
8 **foreach** *partial composition* $(\emptyset, \emptyset, g) \bullet^{c}_{\supseteq} p'$ **do**
9 $(L'', K'', R'') \leftarrow$ result of the composition
10 $G'' \leftarrow G' \cup \{\!\{g\}\!\}$
11 **if** $L'' \cong \emptyset$ **then**
12 **yield** *derivation* $G'' \overset{p}{\Rightarrow} R''$
13 **else**
14 $Q_i \leftarrow Q_i \cup \{\langle (L'', K'', R''), G'', j\rangle\}$

$\{\!\{a, b, b, c\}\!\}$, can be calculated by first binding a to p, then binding b to each result, and so forth. Each intermediary result can then be specified by a multiset of already bound graphs G' and a rule p' modelling p with G' bound. Additionally we keep track of the next graph we should try to bind, meaning the multisets implicitly will be tried in a total order determined by the order of the input graphs $\mathcal{G}$. If a graph binding results in a rule with an empty left-side graph it can bind no further graphs, and the right-side graph corresponds to the right-side of a proper derivation.

It should be noted that in the algorithm we do not directly compute the matching morphism for the derivations. In some cases the algorithm additionally do duplicate computation; for a derivation $\{\!\{a, a\}\!\} \overset{p}{\Rightarrow} H$ it first binds a once to p. As we know that one further a can be bound we get at least two intermediary results with the rules p_1 and p_2 that can bind another copy of a. In the next round we exactly do this binding and the algorithm will, at least, twice yield the derivation. However, in practice we rarely encounter derivations with the same graph multiple times in the left side, and thus the duplicate computation is not a performance problem.

It should be noted that the algorithm used in the framework described in Chapter 9 is slightly different, as we require additional control on the candidate multisets, where only a subset of graphs from $\mathcal{G}$ is used in the initial round of graph binding.

Part III

Chemical Reaction Networks

Chapter 8

Reaction Networks as Directed Hypergraphs

A directed hypergraph is an ordered pair $\mathcal{H} = (V, E)$ of vertices and hyperedges. In contrast to a normal graph, in each edge $(e^+, e^-) \in E$ of a hyper-digraph the tail e^+ and head e^- is not just a single vertex, but a set of vertices. Further, in a directed *multi*-hypergraph the edges are pairs of *multi*sets of vertices. A chemical reaction network is essentially such a multi-hypergraph [Zeigarnik 2000], where each vertex models a molecule and each edge models a reaction. The tail of an edge models the multiset of educts and the head models the products. In the remainder of this work we refer to directed multi-hypergraphs simply as hypergraphs.

Notationally we use the convention that objects with a superscripted plus $^+$ refer to "out"-related elements relative to vertices (e.g., out-edges), and that a superscripted minus $^-$ refer to "in"-related elements.

For visualising hypergraphs and reaction networks we use schemes similar to the one illustrated in Figure 8.1. Vertices are depicted as rounded nodes and the hyperedges as boxes. However, for the simple edges (e^+, e^-) with $|e^+| = |e^-| = 1$ we omit the box and draw an arrow directly from the single tail vertex to the single head vertex. Note that the multiplicity in a head or tail is depicted explicitly with parallel arrows between the vertex and the box.

Hypergraph representations of reaction networks can be recreated from a multitute of sources, e.g., via observations from experiments [Bishop *et al.* 2006, Fialkowski *et al.* 2005, Grzybowski *et al.* 2009], from various databases [Karp & Caspi 2011] such as KEGG [Kanehisa *et al.* 2012] and MetaCyc [Altman *et al.* 2013], or from published SBML files [Hucka *et al.* 2004] and stoi-

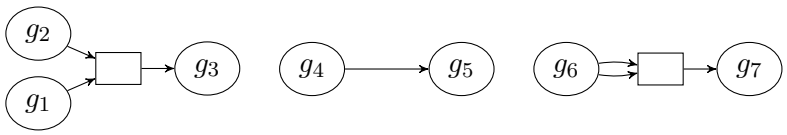


Figure 8.1: Visualisation scheme for directed multi-hypergraphs and chemical reaction networks, exemplified by a hypergraph with the three edges: $(\{\!\{g_1, g_2\}\!\}, \{\!\{g_3\}\!\})$, $(\{\!\{g_4\}\!\}, \{\!\{g_5\}\!\})$, and $(\{\!\{g_6, g_6\}\!\}, \{\!\{g_7\}\!\})$. Vertices are depicted as rounded nodes and hyperedges as boxes, though for edges with singleton tail and head multisets we replace the box by a direct arrow. Multiplicity in the tail and head sets is explicitly depicted with parallel arrows.

chiometric matrices. The various sources naturally can not cover all possible chemistries, and may not even be complete within their chemical scope. In the next chapter we describe an algorithmic framework for automatic generation of reaction networks from models of chemistry based on graph grammars. These networks are labelled with the exact molecule graphs and reaction patterns used to discover them in the first place.

Reaction networks can for example model a set of reactions available for drug synthesis or the reactions in a living cell, and it then becomes interesting to find (optimal) conversion routes, usually called *pathways*, within the networks. It does not seem like there is a universal, exact definition of what a pathway actually is, but in Chapter 10 we suggest a formal definition based on a generalisation of network flows from digraphs to directed hypergraphs. This definition is similar to the well-known *chemical fluxes* from Flux Balance Analysis (FBA) [Kauffman *et al.* 2003], but with differences that makes it possible to analyse pathways from a mechanistic point of view. In Section 10.4 we compare our pathway modelling framework to FBA. For the pathway model we briefly discuss different notions of *autocatalysis* (Chapter 10), and define basic models of *catalytic* and *autocatalytic* pathways.

8.1 Basic Definitions

For the formal discussion of directed multi-hypergraphs, and especially of the computational complexity of algorithms, we need a few definitions, all based on a directed multi-hypergraph $\mathcal{H} = (V, E)$. In some settings it becomes useful the reinterpret $\mathcal{H}$ as a bipartite digraph, which in [Zeigarnik 2000] is referred to as the "bipartite digraph of mechanisms" in the context of reaction networks. Here we simply call it the *underlying digraph*:

Definition 8.1 (Underlying Digraph)**.** The underlying digraph of $\mathcal{H}$ is a directed, bipartite multi-graph $UDG(\mathcal{H}) = (V', E')$ with

$$\begin{aligned} V' =& V \cup E \\ E' =& \left\{\!\!\left\{ (v, e) \mid e = (e^+, e^-) \in E, v \in_m e^+ \right\}\!\!\right\} \\ & \cup \left\{\!\!\left\{ (e, v) \mid e = (e^+, e^-) \in E, v \in_m e^- \right\}\!\!\right\} \end{aligned}$$

For discussing the complexity of algorithms we use the size of hypergraphs as the basic measurement of instance sizes:

Definition 8.2 (Size)**.** The size of $\mathcal{H}$, denoted $size(\mathcal{H})$, is the number of vertices and edges in the underlying digraph. I.e., $size(\mathcal{H}) = |V| + |E| + \sum_{e \in E}(|e^+| + |e^-|)$.

The characteristic that distinguishes hypergraphs from normal digraphs is that the cardinality of at least one tail or head is larger than 1. As a short-hand we call this cardinality the *edge degree*:

Definition 8.3 (Edge Degree)**.** The *edge degree* of an edge $e = (e^+, e^-) \in E$ is the maximum cardinality of the head and tail. The edge degree of a hypergraph $\mathcal{H}$ is the maximum edge degree of all edges.

The hypergraphs where either all heads or tails are singleton sets has particular interest in chemistry.

Definition 8.4 (Forward and Backward Edges/Hypergraphs [Gallo *et al.* 1993])**.** An edge $(e^+, e^-) \in E$ is a backward-edge (B-edge) if $|e^-| = 1$, and it is a forward-edge (F-edge) if $|e^+| = 1$. The whole hypergraph $\mathcal{H}$ is a B-hypergraph if all edges are B-edges. Similarly $\mathcal{H}$ is an F-hypergraph if all edges are F-edges.

8.1.1 Paths and Cycles

In directed hypergraphs there are multiple choices for defining what a "path" and a "cycle" is. For example, [Zeigarnik 2000] describes that a cycle in the underlying digraph could be called a hypercycle, but also suggests a more complex definition where a hypercycle is a minimal subset of the hyperedges $E' \subseteq E$ such that for all vertices the in- and out-degree, restricted to E', is the same: $\forall v \in V : d^-_{E'}(v) = d^+_{E'}(v)$. While the first definition has an obvious generalisation to a hyperpath, the second can lead to multiple kinds of hyperpaths, e.g., does a path start and end in a single vertex, or a set of vertices? For the latter definition it has been proven NP-hard to find such a hypercycle [Özturan 2008]. Another kind of hyperpath can be found in [Fagerberg *et al.* 2015], discussed for B-hypergraphs, but generalisable to hypergraphs with any edge degree. Finally it should be noted that in the chemical literature there is an unrelated concept also called a hypercycle [Eigen 1971, Eigen & Schuster 1977], which involves cycles of cycles in a reaction network, and a notion of autocatalysis (see also Chapter 10 and Section 15.3).

In the following sections we however do not work directly with paths and cycles, but only use a simple characterisation of hypergraphs:

Definition 8.5 (Acyclicity)**.** The hypergraph $\mathcal{H}$ is acyclic if the underlying digraph $UDG(\mathcal{H})$ is acyclic.

8.2 Stoichiometric Matrices

In this work we use almost entirely directed multi-hypergraphs as the mathematical objects for modelling reaction networks, but much literature (e.g., see [Papin *et al.* 2004]) represent networks as *stoichiometric matrices*.

Let $\mathcal{H} = (V, E)$ be a directed multi-hypergraph, and let $V = v_1, \ldots, v_{|V|}$ and $E = e_1, \ldots, e_{|E|}$ be sequences of the vertices and edges in some arbitrary order. We can then accurately represent $\mathcal{H}$ as two matrices; the in-incidence matrix $\mathbf{S}^-$ and the out-incidence matrix $\mathbf{S}^+$, both in the domainn $\mathbb{N}_0^{|V|\times|E|}$. For

each pair of vertices and reactions, v_i, e_j, the matrices are defined as $\mathbf{S}^+_{i,j} = m_{v_i}(e^+_j)$ and $\mathbf{S}^-_{i,j} = m_{v_i}(e^-_j)$. Thus the columns of $\mathbf{S}^+$ represents the tail-multiset of each hyperedge, while $\mathbf{S}^-$ represents the head-multisets. The actual stoichiometric matrix is defined as $\mathbf{S} = \mathbf{S}^- - \mathbf{S}^+$, which in chemical terms is the change of the number of each molecule that each reaction induces. Not every hypergraph can therefore be accurately represented as a stoichiometric matrix, specifically it is those with a hyperedge (e^+, e^-) with $e^+ \cap e^- \neq \emptyset$. For chemistry this means that the stoichiometric matrix can not represent catalysts of single reactions.

Chapter 9

Network Generation

This chapter is based the methods described in [Andersen et al. 2014c].

We have previously defined the notion of a *graph grammar* (Definition 6.5), primarily for modelling the concept of a chemistry without explicitly defining each molecule and reaction. In order to explore the chemical space defined by a grammar we in this chapter describe a domain-specific programming language for computation with graph grammars. Several such frameworks already exists for non-chemical areas such as model checking and verification, proof representation, and modelling control flow of programs. A concrete example is the strategy language [Fernández *et al.* 2012] for PORGY [Pinaud *et al.* 2012, Andrei *et al.* 2011]. However, the existing frameworks are build on the classical model of graph computation where a rewrite rule transforms a single graph into another graph. For modelling chemistry we need the alternate interpretation described in Section 6.5, where multisets of connected graphs are transformed into another multiset of connected graphs. The functional programming language presented here, and in [Andersen *et al.* 2014c], is to our knowledge the first attempt for this type of graph rewriting.

A program in the language, also called an *exploration strategy* or simply a *strategy*, is a function taking a collection of graphs and returning another collection of graphs. Initially the goal of this framework was to generate the space of a chemistry, defined by a graph grammar, so in addition to computation with collections of graphs the execution of a strategy builds a specially labelled directed multi-hypergraph, called a *derivation graph*, which keeps track of all discovered derivations. Before we formally define the language and its semantics we first define derivation graphs and then discuss the simple example of repeatedly applying a transformation rule to a collection of graphs.

9.1 Derivation Graphs

In the evaluation of the rule application strategy we discover derivations. To record these we build a directed multi-hypergraph, labelled with graphs and transformation rules, and call it a *derivation graph*. For practical reasons we omit the matching morphisms for the individual derivations.

Let $\mathbf{C}$ be a category of simple graphs, and recall that the set of graphs is denoted $Ob(\mathbf{C})$. Further, let $DPO(\mathbf{C})$ denote the set of transformation rules in the DPO formalism over the category, as described in the previous

chapters. A derivation graph in the graph category $\mathbf{C}$ is a labelled directed multi-hypergraph $\mathcal{H} = (V, E, l_V, l_E)$, without parallel edges, but with loops allowed. The labelling functions are defined as $l_V \colon V \to Ob(\mathbf{C})$ and $l_E \colon E \to 2^{DPO(\mathbf{C})}$, with the constraint that the derivations indicated by the labels are actually valid. That is, for all edges $(e^+, e^-) \in E$, then for each associated rule $p \in l_E(e^+, e^-)$ there exists a derivation $G \overset{p}{\Rightarrow} H$, where $G = \{\!\{l_V(v) \mid v \in_m e^+\}\!\}$ is the multiset of graphs associated with e^+ and $H = \{\!\{l_V(v) \mid v \in_m e^-\}\!\}$ the multiset from e^-. Additionally, each graph in the vertices must be a connected graph, and all graphs associated with the vertices must be unique up to isomorphism, i.e., for all $u, v \in V, u \neq v$ we have $l_V(u) \ncong l_V(v)$.

The matching morphisms of the derivations are omitted both because we have rarely needed them, and because the derivations are discovered using Algorithm 1, which do not yield the matching morphisms in the first place. When needed, the matching morphisms can be computed from the information in the derivation graph.

For the inference of pathways (Chapter 10) we are usually not interested in pathways that only differ in the transformation rule used to generate reactions. Therefore we have not allowed parallel edges in a derivation graph, but instead annotate each edge with a set of transformation rules.

The allowance of loop edges, i.e., edges (e^+, e^-) with $e^+ = e^-$, may seem strange. The educts and products are the same of such a reaction, but the atom map (Section 6.5.1) may not be the identity morphism, and thus we allow for recording the existence of a derivation of this type.

9.2 Rule Application on Collections of Graphs

The core of the framework is the possibility of applying a rule, but before we formally define the framework we take a look at a simplified situation. Assume we are given a graph grammar with the set of starting graphs $\mathcal{G}$ and a single rule p, and want to compute the set of graphs obtainable after two applications of the rule. That is, if we use p as a function on sets of graphs with the definition

$$p(\mathfrak{U}) = \mathfrak{U} \cup \bigcup_{\mathfrak{U} \overset{p}{\Rightarrow} H} H$$

then we want to compute $p(p(\mathcal{G}))$. We can calculate each application $\mathcal{G}_1 = p(\mathcal{G})$ and $\mathcal{G}_2 = p(\mathcal{G}_1)$ using Algorithm 1. However, note that $\mathcal{G}_1 \supseteq \mathcal{G}$ and thus in the second application we enumerate at least all the derivations we also found in the during the first application.

To avoid duplicate computation we keep track of two sets of graphs: the universe of all graphs $\mathcal{U}$ discovered so far, and the recently discovered graphs $\mathcal{S} \subseteq \mathcal{U}$. We can then redefine rule application for a rule p as

$$p(\mathcal{S}, \mathcal{U}) = (\mathcal{S}', \mathcal{U}')$$

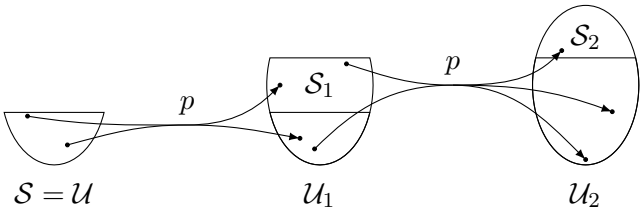


Figure 9.1: Illustration of two applications of a rule p to set of graphs $\mathcal{U}$ with a special subset $\mathcal{S}$, i.e., the computations $(\mathcal{U}_1, \mathcal{S}_1) = p(\mathcal{U}, \mathcal{S})$ and $(\mathcal{U}_2, \mathcal{S}_2) = p(\mathcal{U}_1, \mathcal{S}_1)$. Each derivation must use at least one graph from the subset. Two examples of abstract derivations are shown with the endpoints indicating in which sets the graphs are members.

$$\mathcal{S}' = \bigcup_{\substack{G \overset{p}{\Rightarrow} H \\ G \subseteq \mathcal{U} \\ G \cap \mathcal{S} \neq \emptyset}} H \setminus \mathcal{U}$$
$$\mathcal{U}' = \mathcal{U} \cup \mathcal{S}'$$

where $\mathcal{S}'$ is the set of newly discovered graphs from enumerating all derivations where at least one graph from $\mathcal{S}$ is used. Assuming a second round of rule application we thus only consider derivations where the left side contains at least one new graph. An evaluation of $(\mathcal{U}_2, \mathcal{S}_2) = p(p(\mathcal{U}, \mathcal{S}))$ with this scheme is illustrated in Figure 9.1.

9.3 Language Specification

A *state* is a pair $F = \langle \mathcal{U}_F, \mathcal{S}_F \rangle$ with $\mathcal{S}_F \subseteq \mathcal{U}_F \subseteq Ob(\mathbf{C})$. Let $\mathcal{F}$ be the set of all such states, then a *strategy* is a function $f \colon \mathcal{F} \to \mathcal{F}$, A *program* is a strategy evaluated on the empty state $\langle \emptyset, \emptyset \rangle$. The execution of a program additionally builds a derivation graph $\mathcal{H} = (V, E, l_V, l_E)$ starting from the empty graph.

The sets $\mathcal{U}_F$ and $\mathcal{S}_F$ of a state F will be referred to as respectively the *universe* and the *subset* of the state. Usually we use them as if they are sets, but some strategies use them as lists of unique graphs, i.e., they order the graphs or their result depend on an order.

In the following we describe different kinds of strategies, where we generally assume they are each given a state $F = \langle \mathcal{U}_F, \mathcal{S}_F \rangle$ and returns a new state $F' = \langle \mathcal{U}_{F'}, \mathcal{S}_{F'} \rangle$. Many of the strategies are parametrised, which we note with square brackets around the parameters, i.e., a strategy Q with parameters k and T is written as $Q[k, T]$.

9.3.1 Rule Application

A transformation rule $p \in DPO(\mathbf{C})$ is a strategy, with the semantics that p is applied to a graph state $\langle \mathcal{U}_F, \mathcal{S}_F \rangle$ as explained in the previous section to

obtain a set of derivations

$$D = \{G \overset{p}{\Rightarrow} H \mid G \subseteq \mathcal{U}_F \wedge G \cap \mathcal{S}_F \neq \emptyset\} \tag{9.1}$$

and the resulting sets of graphs are constructed as

$$\mathcal{S}_{F'} = \bigcup_{G \overset{p}{\Rightarrow} H \in D} H \backslash \mathcal{U}_F \tag{9.2}$$

$$\mathcal{U}_{F'} = \mathcal{U}_F \cup \mathcal{S}_{F'} \tag{9.3}$$

The derivation graph $\mathcal{H}$ is augmented such that a new vertex is created for each graph in $\mathcal{S}_{F'}$ not already represented in $\mathcal{H}$, and similarly is the set of hyperedges augmented with the derivations in D. If an hyperedge already exists, then p is added to the associated set of rules. Note that this requires isomorphism checking of each candidate graph from $S_{F'}$ against all previously discovered graphs.

9.3.2 Parallel

A `parallel` strategy is defined in terms of a set of substrategies $\{Q_1, Q_2, \ldots, Q_n\}$. The result of applying a parallel strategy is the union of the results from applying the individual substrategies. For an input state F let $F_i = Q_i(F)$, then

$$\begin{aligned} F' &= \texttt{parallel}[\{Q_1, Q_2, \ldots, Q_n\}](F) \\ U_{F'} &= \bigcup_{1 \leq i \leq n} U_{F_i} \\ S_{F'} &= \bigcup_{1 \leq i \leq n} S_{F_i} \end{aligned}$$

A simple use of parallel strategies is to model the possibility of different reactions happening independently on the same input. As an example, consider modelling the formose chemistry which consists of keto-enol tautomerism and aldol addition, both reversible reactions (see Chapter 12 for the grammar). Let p_2 and p_3 denote the transformation rules respectively for the enol-keto reaction pattern and the aldol addition pattern. The evaluation of the parallel strategy $Q = \texttt{parallel}[\{p_2, p_3\}]$ then tries to apply both rules on the same input, i.e., independently, as illustrated in Figure 9.2.

9.3.3 Sequence

To obtain a more imperative style of writing in the framework, and increase left-to-right readability we introduce special notation for composition of strategies. Let $Q_1, Q_2, \ldots, Q_n$ be a list of substrategies, and let Q be the composed strategy $Q_n \circ \cdots \circ Q_2 \circ Q_1$. We then write Q equivalently as $Q_1 \rightarrow Q_2 \rightarrow$

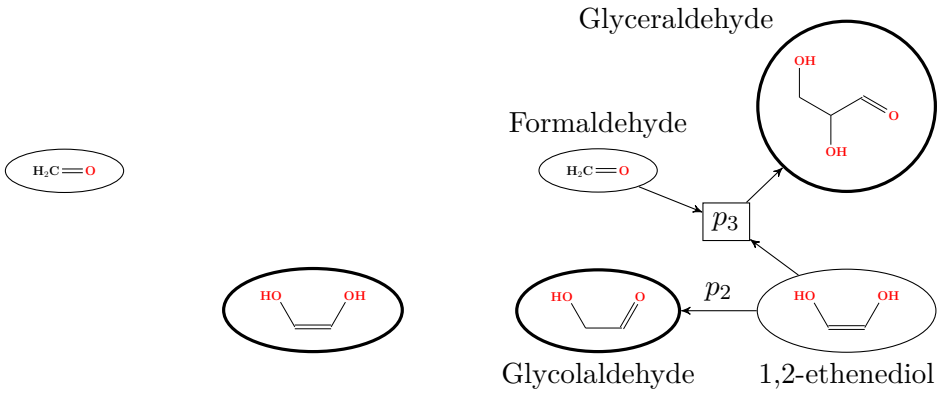


Figure 9.2: Application of a **parallel** strategy $Q = \texttt{parallel}[\{p_2, p_3\}]$ to a state F, with p_2 being the transformation rule for the enol to keto conversion and p_3 being the transformation rule for aldol addition (see Chapter 12). (a) the reaction network in the input state F, with $\mathcal{U}_F = \{\text{formaldehyde}, \text{1,2-ethenediol}\}$ and $\mathcal{S}_F = \{\text{1,2-ethenediol}\}$. (b) the reaction network after evaluation of $Q(F)$, with two new molecules; glycolaldehyde and glyceraldehyde. The resulting state F' has $\mathcal{U}_{F'} = \{\text{formaldehyde, 1,2-ethenediol, glycolaldehyde, glyceraldehyde}\}$ and $\mathcal{S}_{F'} = \{\text{glycolaldehyde}, \text{glyceraldehyde}\}$. In both networks the subset of the states are highlighted.

$\cdots \to Q_n$. Additionally, if $Q_1 = Q_2 = \cdots = Q_n = Q'$, we may use the usual notation for powers of functions, $Q = Q'^n$, for the sequence.

An example of the application of a sequence strategy can be seen in Figure 9.3, in which two sequential steps of the formose chemistry (parallel strategies) are derived starting from a state with $\mathcal{U}_F = \{\text{formaldehyde}, \text{glycolaldehyde}\}$ and $\mathcal{S}_F = \{\text{glycolaldehyde}\}$.

9.3.4 Repetition

The sequencing strategy only allows composition of a fixed number of strategies, whereas the repetition strategy is used to compose a single strategy with itself "as much as possible".

A repetition strategy Q is parametrised by a non-negative integer n and an inner strategy Q', and is written as $\texttt{repeat}[Q', n]$. The inner strategy is sequenced with itself until the subset in the state reaches a fixed point or it is empty, however at most n times: Let $F_i = Q'^i(F)$ for some input state F, then the repetition strategy is formally defined as

$$Q = \texttt{repeat}[Q', n] = Q'^k$$
$$k = \min\{0, 1, \ldots, n\}$$

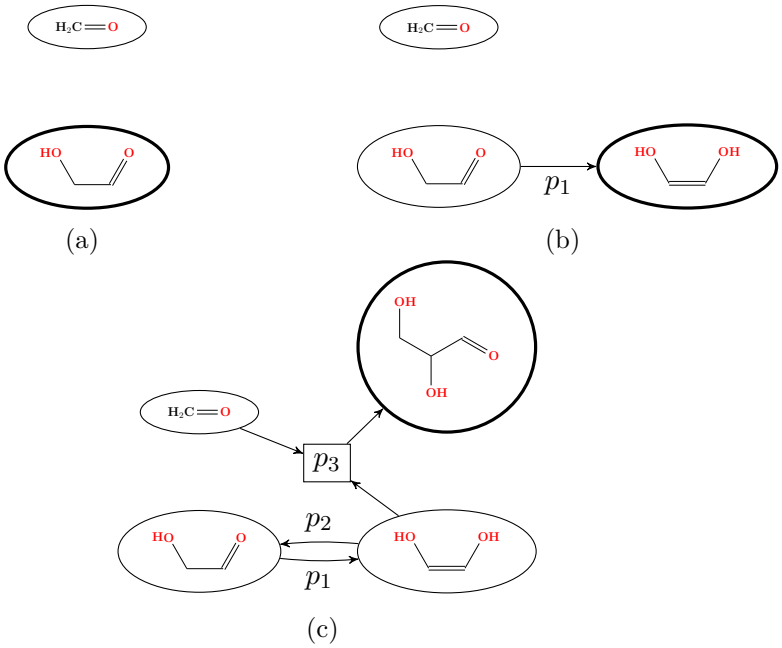


Figure 9.3: Application of the sequence strategy $Q = \texttt{parallel}[\{p_1, p_2, p_3, p_4\}] \rightarrow \texttt{parallel}[\{p_1, p_2, p_3, p_4\}]$ to a graph state F_0, with p_i denoting the transformation rules for keto-enol tautomerism and reversible aldol addition (see Chapter 12). (a) the initial reaction network from F_0 with $\mathcal{U}_{F_0} = \{\text{formaldehyde}, \text{glycolaldehyde}\}$ and $\mathcal{S}_{F_0} = \{\text{glycolaldehyde}\}$. (b) the intermediary reaction network after evaluation of the first step of the strategy. The difference in state is that 1,2-ethenediol is now added to the universe and subset, while glycolaldehyde no longer is in the subset. (c) the reaction network after complete evaluation of $Q(F_0)$. The final state F_2 has all four molecules in the universe and only glyceraldehyde in the subset. Note that in the last step of the strategy the reverse keto-enol reaction is discovered, but glycolaldehyde is already in the universe so it will not be added to the subset of F_2. The subset of the state is highlighted in each network.

such that

$$F_k = F_{k+1} \quad \vee \quad \mathcal{S}_{F_{k+1}} = \emptyset \quad \vee \quad k = n$$

This means that Q' will be executed until no new graphs can be discovered, though limited to at most k iterations. The result of the computation is then the last non-empty state. We motivate this condition of a non-empty subset of a produced state by our definition of rule application, which requires at least one graph from the subset. By returning the last state with non-empty subset the repetition strategy can be used as a pre-computation in a sequence to find a kind of closure under some inner strategy.

Note that for $k = 0$ the strategy becomes the identity strategy, i.e., the resulting state is the same as the input state. If n is set large enough to

not limit the repetition in practice, we write this unbounded repetition as $Q = \texttt{repeat}[Q']$.

The side-effect of executing strategies that contain rule application is the construction of a derivation graph. Even though the last application of Q' in a repeat strategy may not discover new graph it can still discover new derivations among the already known graphs. We therefore define that internally a repeat strategy must execute $k + 1$ repetitions. This allows repetition strategies to be used for calculating the closure of reactions as well as molecules.

In Figure 9.3 the strategy for deriving two steps of the formose network is shown. As a generalisation the strategy $Q = \texttt{repeat}[n, \texttt{parallel}[\{p_1, p_2\}]]$ can be used to derive (at most) n steps of the network. Figure 9.4 shows another example using the repetition strategy, where all isomers of glyceraldehyde 3-phosphate (G3P) are generated.

9.3.5 Revive

Consider the following high-level description of a "if possible"-strategy: Given a single graph g, try to apply the rule p. If the application of p is successful, then let H denote all the produced graphs and return $H \backslash \{g\}$ (all graphs not already known). If the application of p is not successful, then $\{g\}$ should be returned. The simple strategy $Q = p$ applied to $F = \langle \{g\}, \{g\} \rangle$ only partially achieves this, as illustrated in the following. Let $F' \langle \mathcal{U}', \mathcal{S}' \rangle = Q(F)$ be the resulting state after evaluation of the strategy on F. Using the definition of the rule application strategy, Equations (9.1) to (9.3), we get

- $\mathcal{S}_{F'} = H \backslash \{g\}$ and $\mathcal{U}_{F'} = H \cup \{g\}$ if p is successfully applied, and
- $\mathcal{S}_{F'} = \emptyset$ and $\mathcal{U}_{F'} = \{g\}$ if p can not be applied.

However, the desire was to have $\mathcal{S}_{F'} = \{g\}$ in the unsuccessful case. The intention of the revive strategy is to provide a mechanism to model this behaviour.

A rule application strategy discovers a (possibly empty) set of derivations. We say that a graph g is *consumed* during the execution of a rule application strategy if any of the discovered derivations $G \Rightarrow H$ have $g \in G$. In the natural way we extend this and say that a graph g is consumed in the execution of a strategy if it is consumed by any of its substrategies. A revive strategy, $\texttt{revive}[Q']$, is parametrised by a single substrategy, Q', and is defined as:

$$
\begin{aligned}
F' &= \texttt{revive}[Q'](F) \\
\overline{F} &= Q'(F) \\
\mathcal{U}_{F'} &= \mathcal{U}_{\overline{F}} \\
\mathcal{S}_{F'} &= \mathcal{S}_{\overline{F}} \cup \{g \in \mathcal{S}_F \mid g \in \mathcal{U}_{F'} \wedge g \text{ is not consumed in the execution of } Q'(F)\}
\end{aligned}
$$

That is, any graph from the input subset which is still in the output universe and was not consumed, will be added to the output subset. Our motivating

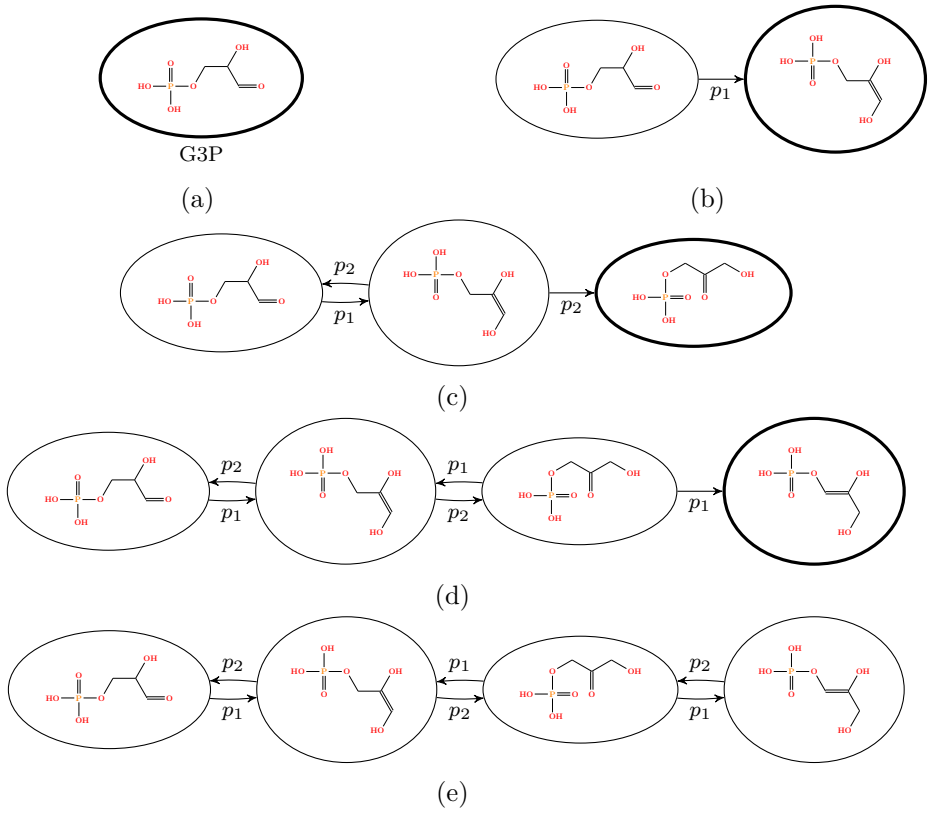
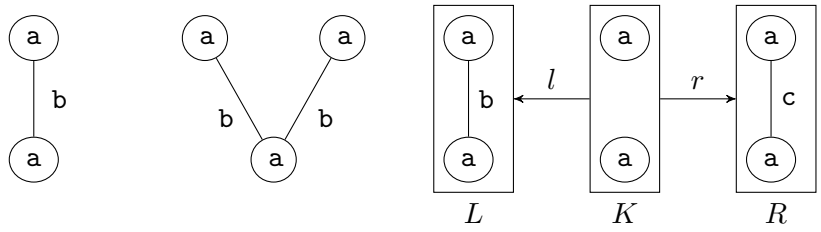


Figure 9.4: The strategy $Q = \texttt{repeat}[\texttt{parallel}[\{p_1, p_2\}]]$ applied to the the initial graph state F_0 with $\mathcal{U}_0 = \mathcal{S}_0 = \{\text{G3P}\}$ (shown in (a)). (b)–(d) the intermediary reaction networks from evaluation of $Q(F_0)$. Each step discovers a new isomer which constitutes the new subset. Additionally, the reaction to the previous isomer is discovered. However, this molecule is already in the universe of the current state and is therefore not added to the subset. (e) the final step in the repetition results in an empty subset as only known molecules (those in the universe) are rediscovered. The state from (d) is therefore the result of the evaluation. In all networks the subset of the current state is highlighted.



(a) g_1 (b) g_2 (c) $p = (L \xleftarrow{l} K \xrightarrow{r} R)$

Figure 9.5: Graphs and transformation rule for the example of the semantics of revive strategies.

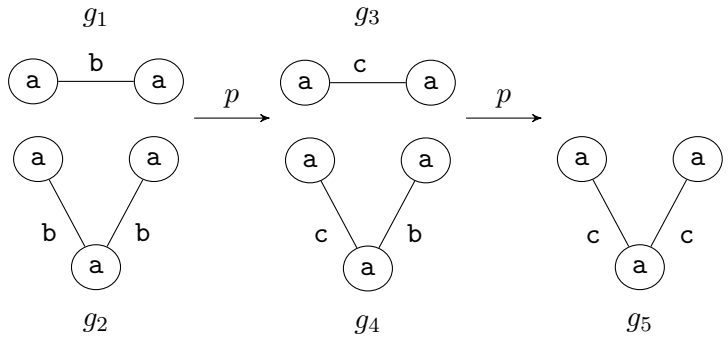


Figure 9.6: Illustration of the application of **repeat**$[p]$ to the state $\langle\{g_1, g_2\}, \{g_1, g_2\}\rangle$. Only the subset of the graph states are shown. The first application of p results in two new graphs, g_3 and g_4, but as p can only be applied to g_4 the final subset is only a single graph, g_5, instead of both g_3 and g_5.

example can now be solved with the strategy $Q =$ **revive**$[p]$. If the application of p is unsuccessful, then g is not consumed and will be added to the resulting subset.

As another example, consider the following problem. Two graphs, g_1 and g_2, and the transformation rule p, as illustrated in Figure 9.5 are given. We wish to develop a strategy to transform all edge labels using rule p, with the intend to use this strategy as a pre-computation for a subsequent strategy. That is, the subset of the graph state after evaluation of the strategy must contain the completely transformed graphs in the subset. The strategy $Q =$ **repeat**$[p]$ may seem like the most intuitive approach to model this process. However, the evaluation of $Q(\langle\{g_1, g_2\}, \{g_1, g_2\}\rangle)$ does not give the intended result, which is illustrated in Figure 9.6. The problem is that the repetition strategy will continue as long as *any* new graph can be discovered, and does not preserve the most derived graphs in the subset. Using the strategy **repeat**[**revive**$[p]$] correctly solves the problem, and we can informally say that **repeat**$[Q']$ implements "as long as possible" with respect to a set of graphs

while `repeat`$[$`revive`$[Q']]$ implements "as long as possible" with respect to individual graphs.

9.3.6 Derivation Predicates

For the purpose of precise modelling and problems with combinatorial explosion it is convenient to limit the set of derivations a transformation rule can produce. We define two variations of the concept of derivation predicates, which both introduce extra constraints in Equation (9.1) to prune unwanted derivations. The strategy `leftPredicate`$[P, Q']$ is defined by the predicate P on a multiset of graphs and a transformation rule, and by the substrategy Q'. A candidate derivation from the graphs G with the rule p found by Q', is only fully calculated and accepted if $P(G, p)$ is true. The other variant is the strategy `rightPredicate`$[P, Q']$ which is also defined by a predicate and a substrategy, though with the predicate P evaluating a complete derivation. Thus, a derivation $G \overset{p}{\Rightarrow} H$ is only accepted if $P(G \overset{p}{\Rightarrow} H)$ is true. Clearly the left-predicate strategy is redundant from the perspective of modelling, however if the desired constraint only depends on the left side of the candidate derivation and the rule, then it can be evaluated earlier in practice.

The predicates can for example be used to model the simple constraint that all derived graphs must be at most a certain size, say 42 vertices. In a chemical context this is highly relevant as molecules in some chemistries can not reach large sizes. This requires a right predicate strategy as information about the right side of the derivation (the products) are needed. Assuming Q' is the inner strategy for deriving graphs we can introduce the constraint as

$$Q = \texttt{rightPredicate}[P, Q']$$
$$P(G \overset{p}{\Rightarrow} H) \equiv \forall h \in H : |V_h| \leq 42$$

As another example, we might want to restrict that some molecule g should not be an educt in any reaction with the transformation rule being p'. This constraint does not require the information of a complete derivation, and may as such be formulated as a left predicate strategy:

$$Q = \texttt{leftPredicate}[P, Q']$$
$$P(G, p) \equiv \neg(p = p' \wedge g \in G)$$

with Q' being an arbitrary strategy.

9.3.7 Filter, Sort, Take and Add

To facilitate more elaborate use of strategies for computing sets of graphs we define several strategies which correspond to functions on lists in other languages. As a graph state is composed of both a universe and a subset, all of these strategies are defined in two variations.

A filter strategy is parametrised by a predicate on a graph and a graph state:

$$\begin{aligned} F' &= \texttt{filterSubset}[P](F) \\ U_{F'} &= U_F \\ S_{F'} &= \{g \in S_F \mid P(g, F)\} \end{aligned} \qquad \begin{aligned} F' &= \texttt{filterUniverse}[P](F) \\ U_{F'} &= \{g \in U_F \mid P(g, F)\} \\ S_{F'} &= \{g \in S_F \mid P(g, F)\} \end{aligned}$$

A sorting strategy is parametrised with a predicate on two graphs and a graph state, used as a less-than predicate in a stable sort of a list of graphs:

$$\begin{aligned} F' &= \texttt{sortSubset}[P](F) \\ U_{F'} &= U_F \\ S_{F'} &= \texttt{stableSort}[P](S_F) \end{aligned} \qquad \begin{aligned} F' &= \texttt{sortUniverse}[P](F) \\ U_{F'} &= \texttt{stableSort}[P](U_F) \\ S_{F'} &= S_F \end{aligned}$$

The choice that the sorting algorithm must be stable is motivated by the desire to allow lexicographical sorting by sequencing several sorting strategies.

A take strategy is parametrised with a natural number:

$$\begin{aligned} F' &= \texttt{takeSubset}[n](F) \\ k &= \min\{n, |S_F|\} \\ U_{F'} &= U_F \\ S_{F'} &= \{S_{F,1}, S_{F,2}, \ldots, S_{F,k}\} \end{aligned} \qquad \begin{aligned} F' &= \texttt{takeUniverse}[n](F) \\ k &= \min\{n, |U_F|\} \\ U_{F'} &= \{U_{F,1}, U_{F,2}, \ldots, U_{F,k}\} \\ S_{F'} &= S_F \cap U_{F'} \end{aligned}$$

An addition strategy appends a given set of graphs to either the universe and optionally also to the subset:

$$\begin{aligned} F' &= \texttt{addSubset}[\{g_1, g_2, \ldots, g_n\}](F) \\ U_{F'} &= U_F \cup \{g_1, g_2, \ldots, g_n\} \\ S_{F'} &= S_F \cup \{g_1, g_2, \ldots, g_n\} \end{aligned} \qquad \begin{aligned} F' &= \texttt{addUniverse}[\{g_1, g_2, \ldots, g_n\}](F) \\ U_{F'} &= U_F \cup \{g_1, g_2, \ldots, g_n\} \\ S_{F'} &= S_F \end{aligned}$$

An example usage of these strategies is the procedure of ranking graphs according to some property and taking the best n graphs for subsequent calculation, i.e.:

$$Q' = \texttt{sortSubset}[P] \rightarrow \texttt{takeSubset}[n]$$

Note that the sorting predicate P can be based on any external data such as results from wet lab experiments. A concrete example of this is shown in [Andersen *et al.* 2013a].

We previously mentioned that the overall strategy implementing a program is evaluated on the empty state, and thus the addition strategies must be used to introduce graphs. Typically a program is then on the form

$$\texttt{addUniverse}[\mathcal{U}] \rightarrow \texttt{addSubset}[\mathcal{S}] \rightarrow Q$$

for starting with a set of passive graphs $\mathcal{U}$ and active graphs $\mathcal{S}$. The addition strategies are however not restricted to this scheme, and can be placed in the middle of a strategy to inject further graphs. As a notational shortcut we use $F := Q$ to say that F is the result of executing the program Q. That is, it means $F = Q(\langle \emptyset, \emptyset \rangle)$.

Chapter 10

Pathways

This chapter is based on [Andersen et al. 2015a].

A method for studying chemical reaction networks is to search for pathways. Well-established theories such as Flux Balance Analysis (FBA) [Papoutsakis 1984, Watson 1984, Fell & Small 1986, Kauffman *et al.* 2003, Orth *et al.* 2010], Elementary Flux Modes (EFM) [Schuster & Hilgetag 1994, Schuster *et al.* 2000, Behre *et al.* 2012, Zanghellini *et al.* 2013], Extremal Pathways (ExPa) [Klamt & Stelling 2002, Klamt & Stelling 2003, Wagner & Urbanczik 2005], and Chemical Organizations (CO) [Kaleta *et al.* 2006, Centler *et al.* 2008, Kaleta *et al.* 2009] have been developed for this purpose. There is thus no single definition of what a pathway actually is, but informally it is usually understood as something that has a well-defined interface of input/output molecules and a specification of how the input is transformed into the output using the reactions available in the network. A similar concept is well-known for normal digraphs, where network flows have been used extensively for modelling and analysis of networks [Ahuja *et al.* 1993, Bang-Jensen & Gutin 2009].

In this chapter we define a pathway model, based on a generalisation of flows from digraphs to directed hypergraphs. Such hyperflows have been formally studied for restricted classes of graphs [Cambini *et al.* 1997, Gallo *et al.* 1998], which however do not include general chemical reaction networks. A hyperflow is essentially the same as a chemical flux, which we describe in more detail in Section 10.4. We define a pathway to be an *integer* hyperflow, meaning a pathway can be interpreted as the number of times each reaction happens. This type of pathway is called a *route* in [Zeigarnik 2000], though in the context of defining cycles in hypergraphs.

The introduction of the integrality constraint for hyperflows means that the linear programming approach used in FBA (see Section 10.4) is no longer applicable, and several natural questions become NP-complete (Section 10.3). We therefore implement the pathway model with integer linear programming. However, an important benefit of our approach is that pathways can be interpreted as a low-level description of what happens to each individual molecule when following the pathway. This makes the integer hyperflows similar to concepts known from Petri net theory [Petri 1962], and we briefly discuss this connection in Section 15.2.

Throughout this chapter we assume a directed multi-hypergraph $\mathcal{H} = (V, E)$ to be given as input for analysis. See Chapter 8 for basic definitions

for hypergraphs.

Catalysis and Autocatalysis

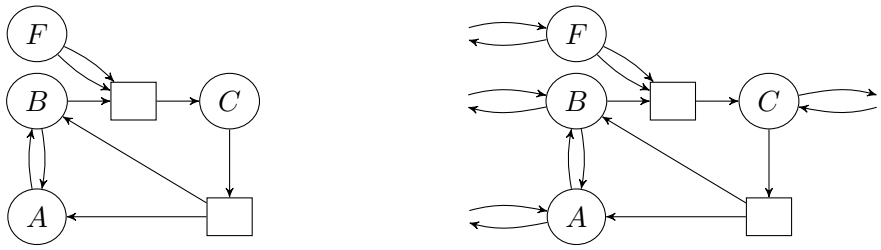
When an overall reaction is catalysed by some compound C, then C is used as an educt and regenerated again as a product. In its simplest form it is thus a reaction $E + C \longrightarrow P + C$, and more generally a pathway with this type of overall reaction. A specialised version of catalysis is called *autocatalysis*, where the catalyst C is produced in a higher quantity than it is consumed, i.e., a pathway where the overall reaction $E + C \longrightarrow 2\,C + W$ can be realised when taking reaction rates into account. In the following we only focus on the topological constraints for autocatalysis.

Autocatalysis is on a more general level the concept of self-replication, which is an integral part of living systems. It has even been hypothesised that autocatalysis is a key ingredient in the origins of life [Kauffman 1995]. There are several formalisms related to autocatalysis, e.g., the *autocatalytic sets* [Kauffman 1986, Hordijk & Steel 2004] where a set of reactions is called autocatalytic when all reactions are catalysed by molecules produced by the reactions in the set itself. It thus related to semi-self-maintaining sets from Chemical Organization theory [Dittrich & Speroni Di Fenizio 2007]. Another set-theoretic notion of autocatalysis can be found in [Kun *et al.* 2008] where breadth-first marking of reaction networks from specified input molecules is used for the analysis. A molecule is then deemed autocatalytic if it is not reachable from the input compounds, but enables the production of further unreachable molecules.

In this work we illustrate that our pathway model enables a formal definition of necessary constraints for both catalytic and autocatalytic pathways. A full-fledged model, that incorporates the inherent causal relationship between the input of an (auto)catalyst and its subsequent production, is not part of the initial model, but will be discussed in Section 15.3. The constraints we introduce relate to the overall reaction of pathways, and we therefore use the term *overall (auto)catalysis*. However, with the aim of approaching a more precise model of autocatalysis we use elaborate local routing constraints. Furthermore we use breadth-first marking, similar to [Kun *et al.* 2008], as a precomputation to obtain a variant of autocatalysis without trivially reachable candidates.

10.1 Model Description

In order to model the input and output of molecules for a pathway we first extend the given network with additional hyperedges, and then formally define the basic pathway model with notions of catalytic and autocatalytic pathways. As the basic model allows for pathways that in some situations can be seen as misleading and having futile branches, we then expand the hypergraph to allow



(a) (b)

Figure 10.1: (a) A small network, $\mathcal{H}$. (b) The extended network, $\overline{\mathcal{H}}$. Note that most of the input/output edges in the extended network will be constrained in the final formulation, and thus for many chemical networks many of these edges will effectively be removed to model specific interface conditions.

for additional routing constraints and expand the definitions of (auto)catalytic pathways.

10.1.1 Integer Flows on Extended Hypergraphs

We need a mechanism to introduce and extract molecules from the network, and we therefore define the *extended hypergraph* $\overline{\mathcal{H}}$ of $\mathcal{H}$ as

$$\begin{aligned} \overline{\mathcal{H}} &= (V, \overline{E}) \\ \overline{E} &= E \cup E^- \cup E^+ \\ E^- &= \{e_v^- = (\emptyset, \{\!\{v\}\!\}) \mid v \in V\} \\ E^+ &= \{e_v^+ = (\{\!\{v\}\!\}, \emptyset) \mid v \in V\} \end{aligned} \tag{10.1}$$

which has additional "half-edges" e_v^- and e_v^+, for each $v \in V$. These explicitly represent potential input and output channels to and from $\mathcal{H}$, see Figure 10.1.

Recall the multiplicity function for multisets, where we for a vertex $v \in V$ and an edge $e \in \overline{E}$ can write $m_v(e^-)$ for the number of occurrences of v in the head of e (and $m_v(e^+)$ for the tail). We use $\delta^+(\cdot)$ and $\delta^-(\cdot)$ to denote a set of incident out-edges and in-edges respectively, and thus use $\delta^+_{\overline{E}}(v)$ as the set of out-edges from v, restricted to the edge set $\overline{E}$, i.e., $\delta^+_{\overline{E}}(v) = \{e \in \overline{E} \mid v \in e^+\}$. Likewise, $\delta^-_{\overline{E}}(v)$ denotes the restricted set of incident in-edges of v.

Definition 10.1. An integer hyperflow on $\overline{\mathcal{H}}$ is a function $f\colon \overline{E} \to \mathbb{N}_0$ satisfying, for each $v \in V$ the conservation constraint

$$\sum_{e \in \delta^+_{\overline{E}}(v)} m_v(e^+) f(e) - \sum_{e \in \delta^-_{\overline{E}}(v)} m_v(e^-) f(e) = 0 \tag{10.2}$$

We mostly speak of integer hyperflows, and will for brevity refer to them simply as flows.

In order to constrain the in- and out-flow to certain vertices we specify a set of inputs (sources) $S \subseteq V$ and outputs (targets/sinks) $T \subseteq V$. Thus

$$f(e_v^-) = 0 \quad \forall v \notin S \quad \text{and} \quad f(e_v^+) = 0 \quad \forall v \notin T \tag{10.3}$$

serve as additional constraints in an I/O-constrained extended hypergraph, which is completely specified by the the triple $(\mathcal{H}, S, T)$.

We adopt the notion of an *overall flow* from the chemical *overall reaction* for a pathway, which is simply a convenient notation for the I/O flow. For a flow f we syntactically write the overall flow as

$$f(e_{v_1}^-)\, v_1 + \cdots + f(e_{v_{|V|}}^-)\, v_{|V|} \longrightarrow f(e_{v_1}^+)\, v_1 + \cdots + f(e_{v_{|V|}}^+)\, v_{|V|}$$

However, as usual for chemistry we omit the terms with zero as coefficient.

Flows are non-negative by definition. While we for reversible reactions could have allowed negative flows (see Section 10.4), we adhere to the usual framework of flow problems. It is therefore necessary to model every reversible reaction by two separate edges $e = (e^+, e^-)$ and $e' = (e^-, e^+)$. This separation of the flow will later allow us to define useful chemical constraints on the flow.

A capacity function $u\colon \overline{E} \to \mathbb{N}_0$, finally limits the flow from above, i.e., $f(e) \leq u(e)$, as in many typical flow problems. We however do not use the capacity function explicitly.

10.1.2 Specialised Flows – Overall (Auto)catalysis

We here define a simple notion of both catalysis and autocatalysis in terms of the I/O flow of a network. As we only constrain the flow of the overall reaction, we call this *overall* (auto)catalysis. Catalysis, in the chemical sense, is when a molecule is consumed by some reaction sequence and is regenerated. Thus, as a basic model of catalysis we say a vertex $v \in V$ is overall catalytic in a flow f if both its input and output flows are non-zero and they are equal:

$$0 < f(e_v^-) = f(e_v^+) \tag{10.4}$$

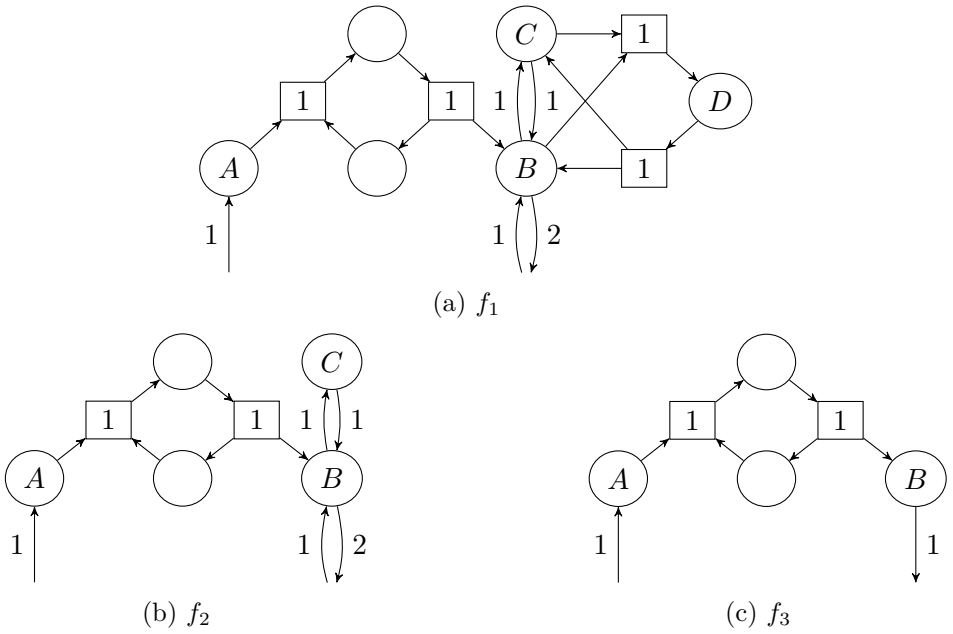
Similarly, autocatalysis is when a molecule is consumed in a reaction sequence, regenerated again, and at least one extra copy is produced. In terms of flow we say a vertex $v \in V$ is overall autocatalytic in a flow f if

$$0 < f(e_v^-) < f(e_v^+) \tag{10.5}$$

We extend the terminology to say that a flow f is overall (auto)catalytic if some vertex is overall (auto)catalytic in f.

10.1.3 Chemically Simple Flows and Vertex Expansion

Modelling reversible reactions as pairs of irreversible ones gives rise to pathways were both an edge e and its inverse e^{-1} have positive flow. Consider the



(a) f_1

(b) f_2

(c) f_3

Figure 10.2: Simplification of a flow f_1 to an equivalent flow f_3, by removal of futile 2-edge subpathways. (a) The molecule D is created through $B + C \longrightarrow D$, but can only be interpreted as being consumed through the reverse reaction. (b) After removal of 1 flow from the reactions $B + C \longleftrightarrow D$, the molecule C now participates in a futile 2-edge flow. (c) Removing 1 flow from $B \longleftrightarrow C$ and the I/O edges $\emptyset \longleftrightarrow B$, we arrive at the simplest flow.

hypergraph annotated with a flow in Figure 10.2a. Here we see three pairs of reversible reactions with positive flow: $B + C \longleftrightarrow D$, $B \longleftrightarrow C$, and the I/O reactions $\emptyset \longleftrightarrow B$. However, we can argue that this flow is not "simple" in the sense that there is no interpretation of the flow without a futile conversion of matter. In the pathway a single copy of D is created, through the reaction $B + C \longrightarrow D$, and it can only be routed into a single reaction, $D \longrightarrow B + C$. The subpathway $B + C \longrightarrow D \longrightarrow B + C$ is thus a futile 2-edge branch that we can simplify away, yielding the equivalent flow in Figure 10.2b. The same reasoning can now be applied to C, and subsequently B, resulting in the flow depicted in Figure 10.2c.

Formally we say that a flow f is not *chemically simple* if there is a vertex $v \in V$ that has only one in-edge $e \in E$ with positive flow and only one out-edge $e' \in E$ with positive flow, where the two edges are each others inverse, $e' = e^{-1}$.

The original flow in Figure 10.2a fulfils the requirement for overall autocatalysis in vertex B (Equation (10.5)), but clearly the in-flow of 1 B is not involved in the extra production of B, which goes against the idea of general autocatalysis. The simplified flow, Figure 10.2c, is however not overall autocatalytic,

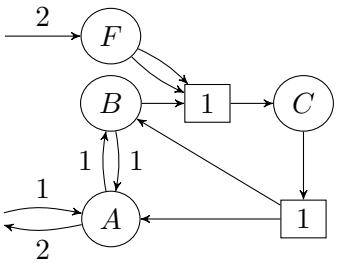


Figure 10.3: Example of a flow with meaningful 2-cycles, in the network from Figure 10.1b. Only edges with non-zero flow are shown.

and it is therefore desirable to constrain the model such that non-simple flows are not possible, in order to further approach a precise characterisation of autocatalysis.

From the shown example it is tempting to simply disallow all 2-cycles of flow. This is the approach effectively used in FBA-related methods (see Section 10.4), and also in flows on normal graphs [Ahuja *et al.* 1993, Bang-Jensen & Gutin 2009]. However, as illustrated in Figure 10.3, this is too strong a constraint. We can interpret this flow such that no flow is directly reversed:

1. $\emptyset \longrightarrow A$
2. $A \longrightarrow B$
3. $\emptyset \longrightarrow F$ twice
4. $B + 2\,F \longrightarrow C$
5. $C \longrightarrow A + B$
6. $B \longrightarrow A$
7. $A \longrightarrow \emptyset$ twice

Since this interpretation is a series of chemically meaningful transformations, it should not be excluded from the pathway model.

To facilitate the constraints that disallow flows that are not chemical simple we expand the extended hypergraph into a larger network. Each vertex is expanded into a subnetwork that represents the routing of flow internally in the expanded vertex. Formally, for each $v \in V$

$$V_v^- = \{u_{v,e}^- \mid \forall e \in \delta_{\overline{E}}^-(v)\} \tag{10.6}$$

$$V_v^+ = \{u_{v,e}^+ \mid \forall e \in \delta_{\overline{E}}^+(v)\} \tag{10.7}$$

$$E_v = \left\{ \left(\left\{\!\!\left\{ u^- \right\}\!\!\right\}, \left\{\!\!\left\{ u^+ \right\}\!\!\right\} \right) \mid u^- \in V_v^-, u^+ \in V_v^+ \right\}$$

That is, v is replaced with a bipartite graph $(V_v^- \cup V_v^+, E_v)$ with the vertex partitions representing the in-edges and out-edges of v respectively, and the edge set being the complete set of edges from the in-partition to the out-partition. We say that E_v is the set of *transit edges* of v.

The original edges are reconnected in the natural way; for each $e = (e^+, e^-) \in \overline{E}$ the reconnected edge is $\widetilde{e}$:

$$\begin{aligned}\widetilde{e} &= (\widetilde{e^+}, \widetilde{e^-}) \\ \widetilde{e^-} &= \left\{\!\!\left\{ u^-_{v,e} \mid v \in_m e^- \right\}\!\!\right\} \\ \widetilde{e^+} &= \left\{\!\!\left\{ u^+_{v,e} \mid v \in_m e^+ \right\}\!\!\right\}\end{aligned}$$

We finally define the expanded hypergraph as

$$\begin{aligned}\widetilde{\mathcal{H}} &= (\widetilde{V}, \widetilde{E}) \\ \widetilde{V} &= \bigcup_{v \in V} V_v^- \cup \bigcup_{v \in V} V_v^+ \\ \widetilde{E} &= \bigcup_{v \in V} E_v \cup \{\widetilde{e} \mid e \in \overline{E}\}\end{aligned}$$

We expand the definition of a flow function to $f\colon \widetilde{E} \to \mathbb{N}_0$ and pose the usual conservation constraints, but on $\widetilde{\mathcal{H}}$: for all $v \in \widetilde{V}$

$$\sum_{e \in \delta^+_{\widetilde{E}}(v)} m_v(e^+) f(e) - \sum_{e \in \delta^-_{\widetilde{E}}(v)} m_v(e^-) f(e) = 0 \tag{10.8}$$

The I/O constraints translates directly to the expanded network. In Section 10.1.4 we formally describe the relationship between flows on the extended and the expanded network.

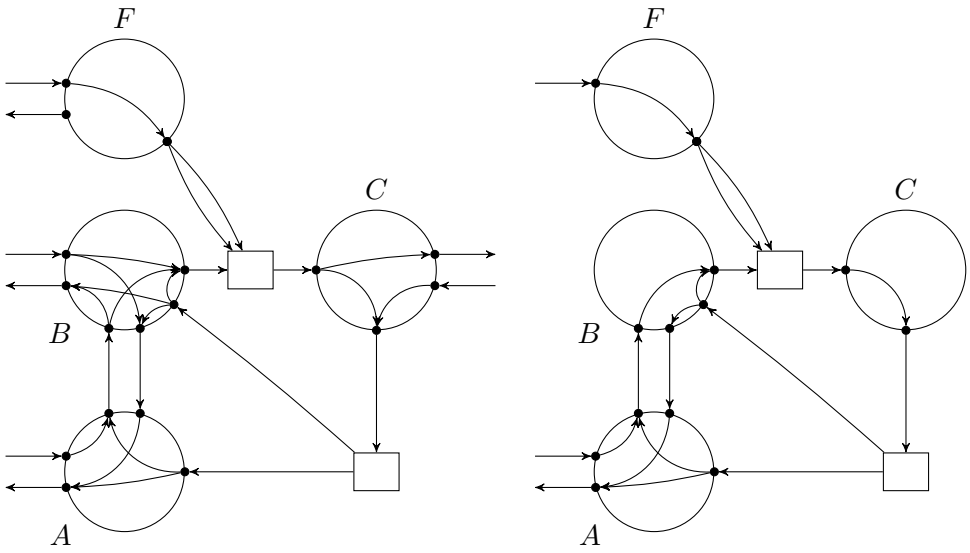
Using the expanded network we can prevent flow from being directly reversed; a flow f must satisfy that for every pair of mutually inverse edges $e = (e^+, e^-), e' = (e^-, e^+) \in \overline{E}$, we have

$$f((u^-_{v,e}, u^+_{v,e'})) = 0 \qquad \forall v \in e^- \tag{10.9}$$

Figure 10.4 shows the expanded version of the network from Figure 10.1b, with these constraints in effect. Note that the expansion of the network also opens the possibility of forbidding other 2-sequences of edges, and in general the possibility of posing constraints on the routing of flow internally in vertices.

When querying for chemical pathways with partially unknown I/O specification we have found it useful to distinguish between reversible reactions that are in the original network $\mathcal{H}$ and the reversible I/O reactions. That is, we may choose to not pose the above constraints on transit edges $(u^-_{v,e}, u^+_{v,e'})$ when $e = e^-_v \wedge e' = e^+_v$, thus allowing excess input-flow to be routed directly out of the network again.

Figure 10.3 showed a valid flow with a meaningful 2-cycle. The expanded flow is shown in Figure 10.5, and we note that no 2-cycles exist in this flow.



(a) (b)

Figure 10.4: The network from Figure 10.1b expanded into $\widetilde{\mathcal{H}}$. The vertices of $\widetilde{\mathcal{H}}$ are the small filled circles, while the large circles, A, B, C and F, only serves as visual grouping of the actual vertices. (a) The expanded network with most transit edges. The transit edges constrained to zero flow in Equation (10.9) are omitted. (b) The expanded network, with the source set $S = \{A, F\}$ and sink set $T = \{A\}$, where edges directly or indirectly constrained to flow zero are omitted.

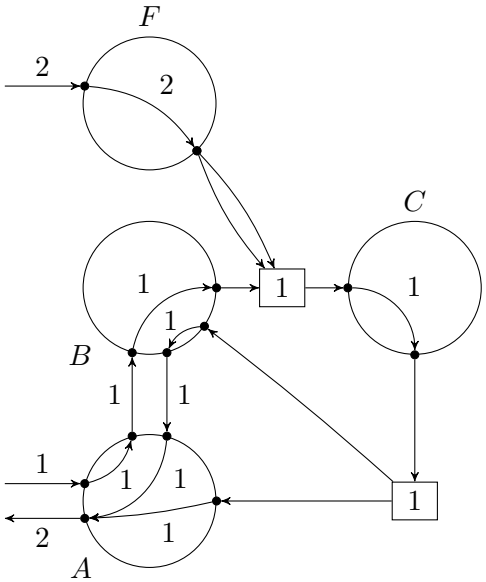


Figure 10.5: The example flow from Figure 10.3 in the expanded network, with only edges with non-zero flow shown. Note that no 2-cycles exist in this flow.

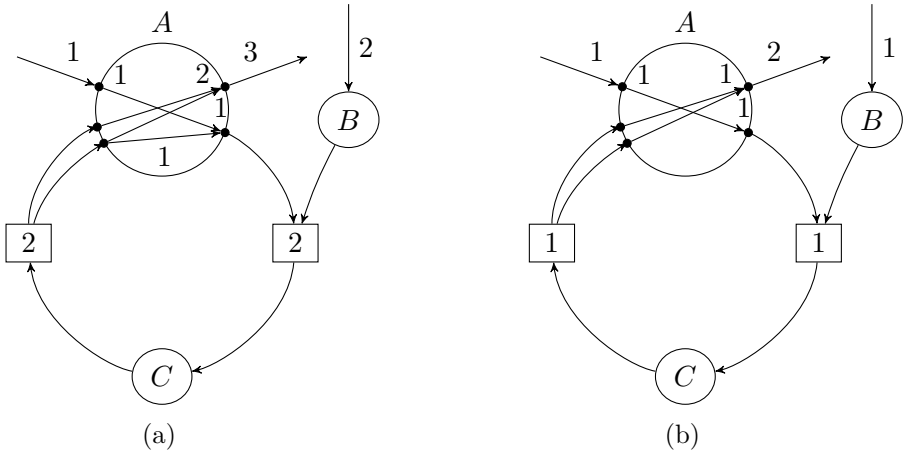


Figure 10.6: A simplified network with an overall autocatalytic flow. (a) The vertex A is both overall autocatalytic and an intermediate vertex in the flow. (b) The same motif for overall autocatalysis, but without A being an intermediate vertex.

Overall Catalysis and Autocatalysis

In Equations (10.4) and (10.5) we defined the I/O constraints for overall catalysis and autocatalysis. These constraints are converted in the obvious manner to the expanded network $\widetilde{\mathcal{H}}$. However, the expanded network reveals another possibility for somewhat misleading flows, exemplified in Figure 10.6a. Vertex A is overall autocatalytic, but is also utilised as an intermediary molecule. The same autocatalytic motif can be expressed by a simpler flow, Figure 10.6b.

In the interest of finding the simplest (auto)catalytic flows, we introduce the following constraints. Let f be a flow and $v \in V$ a vertex satisfying the I/O constraints for overall catalysis Equation (10.4) (resp. overall autocatalysis Equation (10.5)). In the expanded network f must additionally satisfy the transit constraints (note that $\delta_E^{\pm}(v)$ does not include the I/O edges):

$$f((u^+_{v,e'}, u^-_{v,e''})) = 0 \qquad \forall e' \in \delta_E^-(v), e'' \in \delta_E^+(v)$$

That is, all transit flow in an overall (auto)catalytic vertex must flow either from the input edge e_v^- or towards the output edge e_v^+.

Exclusive Autocatalysis

If a compound is overall autocatalytic, it merely means that; if it is available then even more can be produced. However, this does not mean that it can not be produced solely by the other input compounds Solutions can therefore be found that may be surprising. One such solution is illustrated in Figure 10.7. As a variant of our definition of autocatalysis we define that a vertex v, is *exclusively overall autocatalytic* if and only if it is overall autocatalytic and

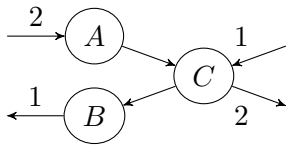


Figure 10.7: The vertex C is overall autocatalytic, but not autocatalytic in the chemical sense.

is not *trivially reachable* from the other input vertices, $S \backslash \{v\}$. A vertex v is trivially reachable from a vertex set S' if it can be marked during a simple breadth-first marking of the hypergraph $\mathcal{H} = (V, E)$. For completeness, the pseudocode is shown in Algorithm 2.

Algorithm 2: Breadth-first marking of a hypergraph

```
Input  : A directed (multi-)hypergraph H = (V, E).
Input  : A set of starting vertices S' ⊆ V.
Output : A marked subset of the vertices.
1 foreach v ∈ S' do mark v
2 while no more hyperedges can be marked do
3     foreach (e+, e−) ∈ E do
4         if all v ∈ e+ are marked then
5             mark e
6             foreach v ∈ e− do mark v
```

Note that breadth-first marking of hypergraphs, and variations thereof, has in the literature also been referred to as finding *scopes* of molecules [Handorf *et al.* 2005, Ebenhöh *et al.* 2004]. Breadth-first marking has in those studies been used alone to analyse metabolic networks, and define set-theoretical notions of pathways and later of autocatalysis [Kun *et al.* 2008]. The methods thus do not have focus on the underlying mechanism of the pathways, but only reachability.

10.1.4 Properties of the Expanded Hypergraph

The expansion of the networks obviously changes the size of the underlying model, and it is therefore necessary to investigate how large the expanded network can get, in order to bound the computational complexity of algorithms.

Proposition 10.1. *The size of the extended network and the expanded network is polynomial in the size of the original network.*

Proof. The size of the extended network is $size(\overline{\mathcal{H}}) = size(\mathcal{H}) + 4 \cdot |V|$, as two half-edges are added to each vertex. For the expanded network, $\widetilde{\mathcal{H}}$, the size depends on the in- and out-degree of the vertices in the extended network. Let $d^-_{\overline{E}}(v)$ denote the in-degree of $v \in V$, and $d^+_{\overline{E}}(v)$ the out-degree. Note that

the degree counts the number of unique incident edges, so for $e \in \overline{E}, v \in V$: $m_v(e^-) > 1$ the size contribution of e to $d^-_{\overline{E}}(v)$ is still only 1. Then the size of the expanded network is

$$\begin{aligned} size(\widetilde{\mathcal{H}}) &= size(\overline{\mathcal{H}}) - |V| + \sum_{v \in V} (d^-_{\overline{E}}(v) + d^+_{\overline{E}}(v)) + 3 \sum_{v \in V} d^-_{\overline{E}}(v) \cdot d^+_{\overline{E}}(v) \\ &\leq size(\overline{\mathcal{H}}) - |V| + 2 \cdot |V| \cdot |E| + 3 \cdot |V| \cdot |E|^2 \end{aligned}$$

where the inequality stems from the fact that at most all vertices are in all head and tail sets, in the original network. □

Translation of Flow

Proposition 10.2. *A feasible flow $f\colon \widetilde{E} \to \mathbb{N}_0$ on $\widetilde{\mathcal{H}}$ can be converted into an equivalent feasible flow $g\colon \overline{E} \to \mathbb{N}_0$ on $\overline{\mathcal{H}}$, with: $g(e) = f(\widetilde{e})$, for all $e \in \overline{E}$.*

Proof. If f is feasible, Equation (10.8) holds for all $\widetilde{v} \in \widetilde{V}$. By the definition of $\widetilde{\mathcal{H}}$, we can say that Equation (10.8) holds for all $\widetilde{v} \in V_v^- \cup V_v^+$ for all $v \in V$. Recall that all transit edges have singleton heads and tails, and $f(e) = f(\widetilde{e}), \forall e \in \overline{E}$. Thus, by addition of Equation (10.8) in each $v \in V$ we get

$$\begin{aligned} \forall v \in V \quad : \quad & \sum_{u^-_{v,e} \in V_v^-} \left(\overbrace{\sum_{u^+ \in V_v^+} f((u^-_{v,e}, u^+))}^{\text{out-flow of } u^-_{v,e}} - \overbrace{m_{u^-_{v,e}}(e^-) f(e)}^{\text{in-flow of } u^-_{v,e}} \right) \\ & + \sum_{u^+_{v,e} \in V_v^+} \left(\overbrace{m_{u^+_{v,e}}(e^+) f(e)}^{\text{out-flow of } u^+_{v,e}} - \overbrace{\sum_{u^- \in V_v^-} f((u^-, u^+_{v,e}))}^{\text{in-flow of } u^+_{v,e}} \right) = 0 \end{aligned}$$

Here, the flow along each transit edge is first added and then subtracted again, so we can simplify the expression to

$$\forall v \in V \quad : \quad \sum_{u^-_{v,e} \in V_v^-} -m_{u^-_{v,e}}(e^-) f(e) + \sum_{u^+_{v,e} \in V_v^+} m_{u^+_{v,e}}(e^+) f(e) = 0$$

Using the definition of V_v^- and V_v^+ (Equations (10.6) and (10.7)) we can verify that these relaxed constraints are exactly those of Equation (10.2), i.e., the constraints on flows in $\overline{\mathcal{H}}$. □

Proposition 10.3. *Let $f\colon \overline{E} \to \mathbb{N}_0$ be a feasible flow on $\overline{\mathcal{H}}$. It can then be decided in polynomial time, in the size of $\mathcal{H}$, if a feasible flow $g\colon \widetilde{E} \to \mathbb{N}_0$ in $\widetilde{\mathcal{H}}$ exists such that $g(\widetilde{e}) = f(e)$ for all $e \in \widetilde{E}$. If it exists it can be computed in polynomial time.*

Proof. The proof proceeds by a reduction to finding a feasible flow in bipartite normal directed graphs, with balance constraints. We refer to [Ahuja *et al.* 1993, Bang-Jensen & Gutin 2009] for a definition of this problem. Recall that the edges of $\overline{H}$ are translated directly into a subset of the edges in $\widetilde{\mathcal{H}}$, and we as such are tasked with finding a feasible flow on all the transit edges, which can be decomposed into finding a feasible flow for each expanded vertex independently. Let $v \in V$, then the hypergraph $(V_v^- \cup V_v^+, E_v)$ only contains edges with singleton head and tail multisets. It is therefore a normal directed, bipartite graph. We then define the flow balance function $b\colon V_v^- \cup V_v^+ \to \mathbb{N}_0$ as $\forall u_{v,e}^- \in V_v^- : b(u_{v,e}^-) = m_v(e^-)f(e)$ and $\forall u_{v,e}^+ \in V_v^+ : b(u_{v,e}^+) = -m_v(e^+)f(e)$. Using the natural lower bound of flow $l \equiv 0$ and infinite upper bound finally gives us the complete specification. A feasible integer flow, if one exists, can be found in polynomial time in the size of the network [Ahuja *et al.* 1993, Bang-Jensen & Gutin 2009]. □

We can define many different pathway problems, depending on which extra constraints we introduce. As we see in Section 10.3 there are classes of constraints that makes the problems strongly NP-hard, even for networks with bounded degree reactions. The last proposition shows a potentially practical algorithmic approach to working with flows in the expanded network. In the next section we however show a simpler approach to directly find the flows in the expanded network, using integer linear programming.

10.2 Implementation using Integer Linear Programming

For analysing reaction networks we are not just interested in a single pathway problem, but a variety of problems with different classes of constraints. As proven in Section 10.3 some of these constraints makes the problem of finding a pathway NP-hard. We therefore use integer linear programming as a basis for finding pathways, which makes it trivial to add custom (linear) constraints when a specific problem calls for it. Using an ILP solver additionally enables the search for optimal pathway using user defined objective functions.

The ILP formulation characterising feasible flows is based on an expanded hypergraph $\widetilde{\mathcal{H}} = (\widetilde{V}, \widetilde{E})$. The flow function is modelled by an integer variable x_e for each edge $e \in \widetilde{E}$, and by constraints for flow conservation. The basic constraints are thus

$$\sum_{e \in \delta_{\widetilde{E}}^+(v)} m_v(e^+) \cdot x_e - \sum_{e \in \delta_{\widetilde{E}}^-(v)} m_v(e^-) \cdot x_e = 0 \qquad \forall v \in \widetilde{V}$$

$$x_e \in \mathbb{N}_0 \qquad \forall e \in \widetilde{E}$$

This definition is similar to an ILP formulation of a classical network flow problem, but with important differences; $\widetilde{H}$ is a hypergraph so an edge $e \in \widetilde{E}$

may be in both $\delta^-_{\widetilde{E}}(u)$ and $\delta^-_{\widetilde{E}}(v)$ (or $\delta^+_{\widetilde{E}}(u)$ and $\delta^+_{\widetilde{E}}(v)$) for $u \neq v$. Additionally, $\widetilde{H}$ is a *multi*-hypergraph, and thus the coefficients $m_v(e^+)$ and $m_v(e^-)$ are introduced, which may be larger than 1.

Additionally, the constraints for chemical flows specified in Equation (10.9) are added in the obvious way. In the following sections we describe constraints for finding catalytic and autocatalytic flows. For the formulation we use M to denote a classical "large enough" constant, and as some parts of the catalysis and autocatalysis models are similar we first describe the formulation of these common parts.

10.2.1 Strict Flow Through Overall (Auto)catalytic Vertices

In our definition of (auto)catalysis we require that if a vertex is (auto)catalytic, then no flow can enter the vertex from the network and exit the vertex to the network again. Let z_v be the indicator variable for the vertex $v \in V$ being (auto)catalytic, then the requirement is trivially enforced by the following constraints:

$$x_e \leq M \cdot (1 - z_v) \qquad \forall e = (u^-_{v,e'}, u^+_{v,e''}) \in V_v^- \times V_v^+ : e' \neq e_v^- \wedge e'' \neq e_v^+$$

10.2.2 Overall Catalysis

We model catalysis by introducing an indicator variable $z^c_v \in \{0, 1\}$ for each $v \in V$ indicating whether v is catalytic or not. Thus we can enforce a solution to be catalytic by posing the constraint

$$\sum_{v \in V} z^c_v \geq 1$$

The actual constraints for the indicator variables are obtained partially by the section above on strictness of flow. Below follows the last requirement, Equation (10.4), which is realised through a set of auxiliary indicator variables, $z^0_v, z^<_v, z^>_v \in \{0, 1\}$

$$x_v^- = x_v^+ = 0 \Leftrightarrow z_v^0 = 1 \equiv \begin{cases} 1 - z_v^0 \leq x_v^- + x_v^+ \\ M \cdot (1 - z_v^0) \geq x_v^- + x_v^+ \end{cases}$$

$$x_v^- < x_v^+ \Leftrightarrow z_v^< = 1 \equiv \begin{cases} x_v^- < x_v^+ + M \cdot (1 - z_v^<) \\ x_v^- \geq x_v^+ - M \cdot x_v^< \end{cases}$$

$$x_v^+ < x_v^- \Leftrightarrow z_v^> = 1 \equiv \begin{cases} x_v^+ < x_v^- + M \cdot (1 - z_v^>) \\ x_v^+ \geq x_v^- - M \cdot z_v^> \end{cases}$$

$$0 < x_v^- = x_v^+ \Leftrightarrow z_v^c = 1 \equiv \begin{cases} z_v^c \geq 1 - z_v^< - z_v^> - z_v^0 \\ z_v^c \leq 1 - z_v^0 \\ z_v^c \leq 1 - z_v^< \\ z_v^c \leq 1 - z_v^> \end{cases}$$

10.2.3 Overall Autocatalysis

As for catalysis we model autocatalysis with a set of indicator variables $z_v^a \in \{0,1\}$ for all $v \in V$, and force a solution to autocatalytic with the constraint

$$\sum_{v \in V} z_v^a \geq 1$$

We use the constraints for strictness of flow and model the remaining constraint, Equation (10.5), using the auxiliary variable set z_v^-, indicating $x_v^- > 0$:

$$0 < x_v^- \Leftrightarrow z_v^- = 1 \equiv \begin{cases} z_v^- \leq x_v^- \\ M \cdot z_v^- \geq x_v^- \end{cases}$$

$$0 < x_v^- < x_v^+ \Leftrightarrow z_v^a = 0 \equiv \begin{cases} z_v^a \leq x_v^- \\ x_v^- < x_v^+ + M \cdot (1 - z_v^a) \\ M \cdot z_v^a + x_v^- \geq x_v^+ - M \cdot (1 - z_v^-) \end{cases}$$

10.2.4 Solution Enumeration

A typical use of solvers for integer programs is to find a single optimal solution. However, from a chemical perspective we are also interested in near-optimal solutions and in some cases even all solutions. The structure of our formulation additionally have influence on when two solutions are considered different. Often we might not consider two solutions different if they only differ in the flow on the transit edges, i.e., those introduced by the vertex expansion. This makes it difficult to use build-in features in solvers, such as the solution pool in IBM ILOG CPLEX, to enumerate solutions.

For finding multiple solutions we therefore explore a search tree based on the domain of the variables; each vertex in the tree represents a restriction of the variable domains, with children representing more constrained domains and the parent representing a less constrained domain. Note that this tree, in theory, is infinite as some variable may have no upper bound. In each vertex we use an ILP solver to find an optimal solution for the sub-problem. If the problem is infeasible the sub-tree is pruned, otherwise a path to a leaf in the tree is constructed to represent the solution found by the ILP solver. The quality of the found solution at the same time acts as a lower bound on the objective function of the sub-tree (when minimising the function). Vertices in the tree are explored in order of increasing lower bound.

If a different value of flow is not to be considered a difference in the solution we simply do not consider the corresponding variables to be part of the branching procedure.

10.3 Computational Complexity

This section is based on parts of [Andersen et al. 2012], but adapted to the pathway model described above. The formulations have additionally been slightly altered to highlight the connection to the synthesis planning problem.

Here we assume all flows are integer, but it should be noted that real-valued flows have also been studied for special classes of networks, e.g., gain-free hypergraphs [Jeroslow *et al.* 1992], and B-hypergraphs [Cambini *et al.* 1997, Gallo *et al.* 1993, Gallo *et al.* 1998].

We can define a wide range of hyperflow problems depending on the specific structure of the input network and on which constraints we impose on the flow. As a basis we assume the that integer hyperflows must be found in an I/O constrained network $(\mathcal{H}, S, T)$ (see Section 10.1.1).

One variation is the synthesis planning problem where the input network is an acyclic B-hypergraph [Fagerberg *et al.* 2015, Hendrickson 1977]. The goal is to produce a molecule $t \in V$ from a set of available chemicals $S \subseteq V$, such that some objective is optimised. For example, minimising a weighted sum of the used chemicals. For a class of objective functions it is possible to enumerate the k best synthesis plans in time polynomial in k and the size of the network.

The synthesis planning problem can be generalised in several ways, but we focus on the introduction of upper bounds on the input flow, which makes the problem strongly NP-complete. We consider variations of the following generalisation:

MAX-Output Given is a distinguished target vertex $t \in T$, a desired output flow $\gamma \in \mathbb{N}_0$, and a bound on input flow $b\colon S \to \mathbb{N}_0$, is there a flow f such that $f(e_t^+) = \gamma$ and $f(e_s^-) \leq b(s), \forall s \in S$.

In the following NP-completeness proofs we reduce from the 3-partition problem:

3PART Given a multiset of integers $\{\!\{a_1, a_2, \ldots, a_{3m}\}\!\}$, can they be partitioned into m triples each with sum $A = \frac{1}{m}\sum_{i=1}^{3m} a_i$.

The 3-partition problem is strongly NP-complete [Garey & Johnson 1975]. This holds even when $A/4 < a_i < A/2$, meaning that the target sum A can only be obtaining by exactly 3 numbers. In the following reductions this assumption will be used to make sure that only triples are created.

We first prove that **MAX-Output** is strongly NP-complete for acylic networks, by a reduction from **3PART**. The reduction is then modified to prove that the result also holds when all hyperedges have bounded degree. Finally we modify the construction to use B-hypergraphs. In [Andersen *et al.* 2012] the same basic reduction is also used to show NP-completeness of **MAX-**

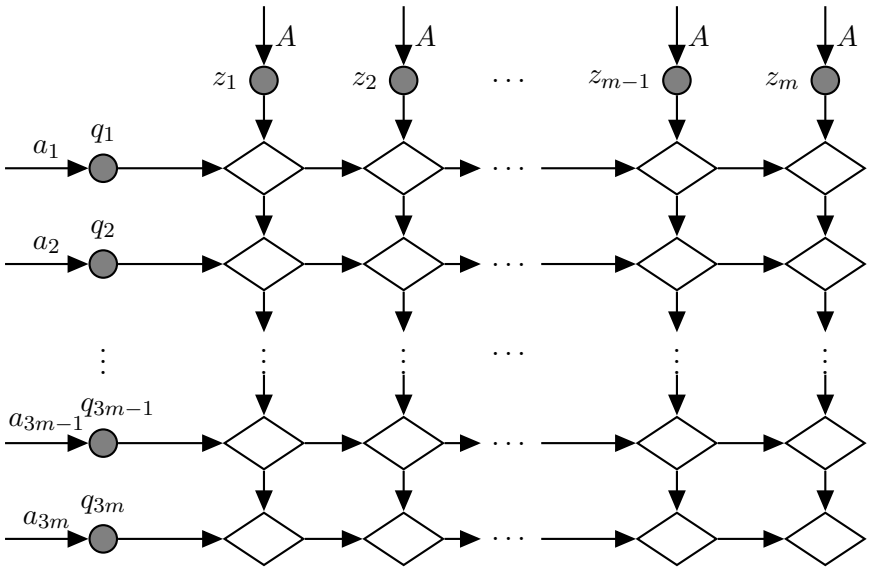


Figure 10.8: Abstract view of the network constructed for the reduction from **3PART**. The diamonds represent the switching graphs which may each consume flow directly from a number-vertex q_i and a triple-vertex z_j. Not shown is a connection from each switching graph and a goal vertex g.

Output restricted to a single input vertex, and for a variation with overall autocatalytic flows.

10.3.1 The Main Reduction

Given a **3PART** instance of $n = 3m$ integers $\{\!\{a_1, a_2, \ldots, a_n\}\!\}$, let $A = \frac{1}{m}\sum_{i=1}^{n} a_i$ be the target sum for each of the m triples. We assume $A/4 < a_i < A/2$, such that the target sum can only be obtained by exactly 3 numbers. Construct a directed multi-hypergraph $\mathcal{H}$ with vertices $q_i, 1 \leq i \leq 3m$ for each number a_i, and vertices $z_j, 1 \leq j \leq m$ for each target triple. The overall idea is to construct an intermediary network with "switches" that each consume flow from a number-vertex and a triple-vertex to model the assignment of a number to a triple. Each switch will, when active, route flow to a special goal vertex. This scheme is visualised in Figure 10.8, where a matrix of switches, depicted as diamond nodes, are connected directly to number-vertices and triple-vertices.

Note that we use the notation from chemical reactions for specifying hyperedges. Aside from the vertices q_i and z_j, we add a goal vertex g. For each pair of numbers and triples $(i, j), 1 \leq i \leq 3m, 1 \leq i \leq m$ we create a switch

with new vertices w_{ij}, v_{ij}, and x_{ij}, and the following hyperedges:

$$\begin{aligned} a_i\, q_i &\longrightarrow w_{ij} \\ a_i\, z_j &\longrightarrow v_{ij} \\ w_{ij} + v_{ij} &\longrightarrow x_{ij} \\ x_{ij} &\longrightarrow a_i\, g \end{aligned} \tag{10.10}$$

Each of these switching networks simply serve to make the overall conversion $a_i\, q_i + a_i\, z_j \longrightarrow a_i\, g$ possible, but we have opted to break the conversion into multiple steps in order to simplify the subsequent proof modifications.

Finally we create an I/O constrained network by specify the source and sink vertices as $S = \{q_1, q_2, \ldots, q_{3m}, z_1, z_2, \ldots, z_m\}$ and $T = \{g\}$.

Note that this construction yields an acyclic hypergraph. Each switch (i, j) contributes $3 \cdot (a_i + 2) + 4 = 3a_i + 10$ to the size of the network, while number-vertices, triple-vertices, and the goal vertex contribute $3m + m + 1$ to the size. The I/O extension (Equation (10.1)) adds 4 for each of the three intermediary vertices in each switch, and another $4 \cdot (3m + m + 1)$ for the remaining vertices. The size of the I/O extended network is thus

$$\begin{aligned} & m \cdot \sum_{i=1}^{3m} (3a_i + 10 + 4 \cdot 3) + (1 + 4) \cdot (3m + m + 1) \\ =& 3m \cdot \sum_{i=1}^{3m} a_i + 66m^2 + 20m + 5 \end{aligned}$$

As **3PART** is strongly NP-complete we can assume that all numbers a_i are bounded by a polynomial in $n = 3m$. The network size is thus polynomial in the size of the **3PART** instance.

Theorem 10.1 ([Andersen *et al.* 2012])**.** ***MAX-Output*** *is strongly NP-complete for acyclic hypergraphs.*

Proof. Clearly **MAX-Output** is in NP, as a candidate flow f can be checked for feasibility in polynomial time of the size of the network. The completeness is proven by a reduction from **3PART**, with the I/O constrained network $(\mathcal{H}, S, T)$ being constructed as described above. We restrict the input flow with $b(q_i) = a_i, 1 \leq i \leq 3m$ and $b(z_j) = A, 1 \leq j \leq m$. The special goal vertex $t \in T$ is set to the goal vertex g, and the required output flow γ is set to $m \cdot A$.

The claim is then that the **3PART** instance is a "yes"-instance if and only if there is a flow f solving the **MAX-Output** instance. Assume the numbers $\{\!\{a_1, a_2, \ldots, a_{3m}\}\!\}$ can indeed be partitioned into triples $T_1, T_2, \ldots, T_m$, each with the sum A. A flow f can then be constructed with

$$\begin{aligned} & f(e_g^+) = \gamma = m \cdot A \\ & f(e_{q_i}^-) = a_i \qquad\qquad 1 \leq i \leq 3m \end{aligned}$$

$$f(e^-_{z_j}) = A \qquad\qquad 1 \leq j \leq m$$

For each triple $T_j = \{a_{j_1}, a_{j_2}, a_{j_3}\}$ we set flow 1 on each hyperedge in the switches (j_1, j), (j_2, j), and (j_3, j). The remaining hyperedges get flow 0.

Each a_i is assigned to exactly one triple T_j, so at most one switch in the ith row has non-zero flow, meaning exactly a_i flow is drained from q_i, fulfilling the conservation constraint for q_i. Each T_j consists of 3 numbers, corresponding to 3 switches with flow 1, draining a total of A flow from z_j as the sum of T_j is A. The conservation constraint is thus also fulfilled for each z_j. Finally, the m columns of switches each routes A flow to g, which balances the flow of g.

In the other direction, assume f is a feasible flow for the **MAX-Output** instance. This means the goal vertex g has out-flow $m \cdot A$, which can only come from the switches. Each of the m columns of switches can contribute at most A flow to g due to the bound on input flow to each z_j. Thus in order to reach $m \cdot A$ flow in g this maximum must be used. A switch can have at most flow 1, as the input flow to each q_i is bounded. For the 3-partition problem we assumed that $A/4 < a_i < A/2$, so the maximum flow of A in a column can only be obtained by exactly 3 switches with non-zero flow. Each q_i is thus paired with exactly one z_j through a switch with flow 1. We can therefore construct a valid partitioning for the **3PART** instance by assigning a_i to triple T_j if switch (i, j) have flow 1. □

10.3.2 Reducing the Edge Degree to Two

The main reduction constructs a hypergraph with a maximum edge degree of $\max\{a_i\}$, while for non-trivial chemical reaction networks this maximum can still be as low as 2. It is therefore important to show that a network with constant edge degree does not make the problem easier.

The only edges with degree more than 2 in the network are three of the edges in each switch:

$$a_i\, q_i \to w_{ij}$$
$$a_i\, z_j \to v_{ij}$$
$$x_{ij} \to a_i\, g$$

The idea is to replace each of these edges with a subgraph with edge degree at most 2, and leave the rest of the network intact. These individual subgraphs must have at most polynomial size in $n = 3m$, but as the new edges will have constant degree, we just have to argue that the number of edges and vertices is polynomial in n.

We first construct a subgraph for expanding a flow of 1 to a flow of k, and reverse all the edges to get a merging of k to 1.

Assume $k = 2^u$. The expansion can easily be implemented by a series of edges

$$x_i \to 2\, x_{i+1} \qquad 0 \leq i < u$$

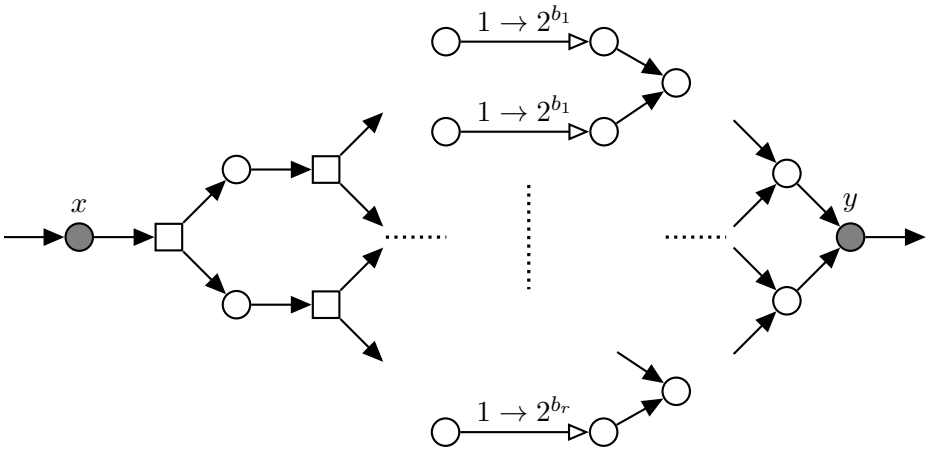


Figure 10.9: An expansion edge $x \xrightarrow{1 \to k} y$, with $k = \sum_{i=1}^{r} 2^{b_i}$. The flow is on the left duplicated to r flows. In the middle they are expanded by repeated duplication to powers of 2. On the right are the r power of 2 flows summed.

with x_0 as the input vertex and x_u as the output vertex of the subgraph. The subgraph can be abstracted to a special edge denoted as $x_0 \xrightarrow{1 \to 2^u} x_u$.

Assume any positive k, in binary representation, with $\lfloor \log k \rfloor + 1$ bits. Let $b_1, b_2, \ldots b_r$ denote the position from the right side of all the 1-bits in k such that $k = \sum_{i=1}^{r} 2^{b_i}$. We can create a general expansion edge, $x \xrightarrow{1 \to k} y$, by combining r instances of $(1 \to 2^{b_i})$-expansion edges as depicted in Figure 10.9. The left and right side of the graph are trees that respectively create r 1-flows from a single 1-flow and combines the r resulting 2^{b_i}-flows to a single k-flow. The size of the complete expansion edge is in $O(\log^2 k)$ as r is at most the number of bits in k, and each of the $(1 \to 2^{b_i})$-expansions are of size $O(\log k)$.

A $(k \to 1)$-merge edge can, as mentioned, be implemented by simply reversing all edges in an expansion edge.

Corollary 10.1 ([Andersen *et al.* 2012])**.** ***MAX-Output*** *is strongly NP-complete for acyclic hypergraphs, with bounded hyperedge degree.*

Proof. Using expansion and merge edges all the switches in the network form the previous reduction can be implemented with a degree of at most 2 and with polynomial size in the individual a_i. As 3PART is strongly NP-complete, the reduction can be carried out in polynomial time. □

10.3.3 Restriction to B-hypergraphs

The network created in the main reduction is not a B-hypergraph, but we can make a slightly modified construction with this restriction.

Corollary 10.2. ***MAX-Output*** *is strongly NP-complete for acyclic B-hypergraphs, with bounded edge degree.*

Proof. Only the hyperedges $x_{ij} \longrightarrow a_i\, g$ in each switch (i, j) violates the B-property. We can replace the switches in Equation (10.10) with the following construction.

$$\begin{aligned} a_i\, q_i &\longrightarrow w_{ij} \\ a_i\, z_j &\longrightarrow v_{ij} \\ w_{ij} + v_{ij} &\longrightarrow x_{ij} \\ x_{ij} &\longrightarrow g \end{aligned} \tag{10.11}$$

That is, instead of creating a_i goal flow in a switch (i, j) we create flow 1. Clearly the network is still polynomial in size. Each column will now create 3 goal flow instead of A goal flow, so modify the output flow constraint to $\gamma = 3m$. As this network is a B-hypergraph we only need merge-edges to reduce the edge degree to 2. Merge edges (see Figure 10.9 with all hyperedges inverted) only use B-hyperedges, to the complete network is a B-hypergraph with edge degree 2. □

If we wish to restrict the output flow to 1 we can attach a merge edge $g \xrightarrow{3m \to 1} g'$, with a new goal vertex g'. Thus, when comparing with synthesis planning, as described in the beginning of this section, we see that the only difference is the bound on the input flow.

10.3.4 Alternative Reduction

For Section 10.4 in the comparison of integer hyperflows to Flux Balance Analysis we explicitly use a reduction of the well-known **Independent-Set** problem to **MAX-Output**. We therefore present the reduction here, along with an ILP formulation for solving **Independent-Set**.

Independent-Set Given an undirected graph G and an integer $k \in \mathbb{N}$, find a set of vertices $V' \subseteq V_G$, of cardinality k, such that no edge in the graph is between vertices of V', i.e., $E_G \cap V' \times V' = \emptyset$.

ILP Formulation for Maximum Independent-Set

Let $G = (V, E)$ be the input graph.

$$\begin{aligned} \max \quad & \sum_{v \in V} x_v \\ \text{s.t.} \quad & x_u + x_v \leq 1 && \forall (u, v) \in E \\ & x_v \in \{0, 1\} && \forall v \in V \end{aligned}$$

The resulting independent set is the vertices $v \in V$ with $x_v = 1$.

Reduction to MAX-Output

Let G be the input graph and k the integer for the **Independent-Set** problem. We first construct a directed multi-hypergraph $\mathcal{H}$:

$$\begin{aligned}\mathcal{H} &= (V_{\mathcal{H}}, E_{\mathcal{H}})\\ V_{\mathcal{H}} &= \{v_g\} \cup \{v_e \mid e \in E_G\}\\ E_{\mathcal{H}} &= \{(\{\!\{v_e \mid e \in \delta(v)\}\!\}, \{\!\{v_g\}\!\}) \mid v \in V_G\}\end{aligned}$$

We thus construct a vertex for each edge in G and an extra "goal vertex". The hyperedges correspond to the vertices of G, with the goal vertex as the head and the rest of the vertices corresponding to the incident edges of G as the tail. As source set we use $S = \{v_e \mid e \in E_G\}$, and let v_g be the only sink vertex. The output flow of v_g is set to $\gamma = k$, while the upper bound on input flow to each v_e is set to 1.

Each hyperedge in $E_{\mathcal{H}}$ can at most have flow 1 due to the input flow bounds, which can be used to signify that the corresponding vertex in G is included in the independent set. Selecting flow 1 on two hyperedges corresponding to adjacency vertices in G is not possible, as this would drain flow from the same vertex in the hypergraph, which bounded to be 1. The hyperedges in $E_{\mathcal{H}}$ with flow 1 thus correspond to an independent set in G.

10.4 Comparison to Existing Methods

The basic pathway model described in Section 10.1.1 is quite similar to the formalism used in FBA, EFM and ExPa, with the latter two methods primarily aiming to categorise specific classes of pathways [Papin *et al.* 2004]. In the following we recast FBA in terms of hypergraphs as the underlying models of reaction networks to clarify the similarities but also the differences with our present approach.

The mathematical development of FBA, EFM, and ExPa is based upon the concepts of the stoichiometric matrix (sse Section 8.2) and *flux vectors*. Recall that the stochiometric matrix $\mathbf{S}$ only describes the original reaction network if all hyperedges have disjoint head and tail. All direct catalysts, however, are cancelled out in the stochiometric matrix, hence the equivalence fails whenever there are reaction hyperedges with $e^+ \cap e^- \neq \emptyset$. This somewhat limits the scope of FBA. Although it is possible in principle to replace reactions with direct catalysts by a sequences of intermediate reactions that consume and regenerate the catalyst, the resulting FBA network is no longer directly equivalent to the original and special care must be taken to ensure equivalence of solutions.

A *flux vector* $f \in \mathbb{R}^{|E|}$ for a network $\mathcal{H} = (V, E)$ models a pathway, and must satisfy the usual conservation constraint, $\mathbf{S} \cdot f = 0$ (cmp. Equation (10.2)). Reversible reactions are modelled in one of two ways:

- Combined: reversible reactions are modelled as a single reaction, but with the flow/flux allowed to be negative. The flow/flux of irreversible reactions is constrained to be non-negative. This is the approach followed when finding EFMs [Schuster & Hilgetag 1994].

- Separate: reversible reactions are modelled as two inverse reactions, and the flow/flux on all reactions must be non-negative. This is the approached followed when finding ExPas [Schilling *et al.* 2000]. We also follow this approach both for mathematical simplicity and because it allows us to make use of the enhanced modelling capabilities offered by the expanded network.

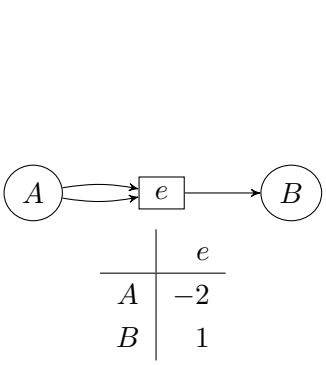
The extension of the stoichiometric matrix $\mathbf{S}$ to incorporate I/O reactions can also be implemented using both the "combined" and the "separate" way of handling reversible reactions. The I/O constraints from Equation (10.3), specified by S and T, translate naturally to the corresponding constraints on the extended flux vector.

Linear Programming versus Integer Linear Programming

With FBA we additionally define a linear objective function to find an optimal flux vector, possibly with additional linear constraints [Savinell & Palsson 1992]. As a flux vector is real-valued, and all the stated constraints are linear, it can be found using linear programming (LP) in polynomial time [Khachiyan 1980, Karmarkar 1984]. Herein lies a major difference to the model presented in this contribution, where we require an *integer* hyperflow. We can thus characterise the linear program from FBA as the LP relaxation of the basic pathway problem presented in Section 10.1.1.

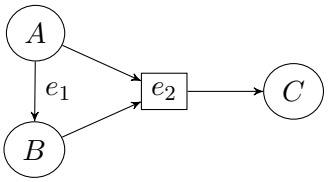
The LP relaxation of an ILP yield an integer solution only under special conditions. The best known sufficient condition is that the matrix of constraint coefficients is totally unimodular (TU), i.e., when all its square submatrices have determinants -1, 0, or $+1$, and thus all entries of the matrix are also -1, 0, or $+1$. This is the case for example for integer flows in graphs [Ahuja *et al.* 1993, Bang-Jensen & Gutin 2009]. As the simple examples in Figure 10.10 shows, this not true in general for stochiometric matrices and hence for hyperflows.

Even though total unimodularity is not a necessary condition, it is not too difficult to construct reaction networks with linear optimisation problems where the integer problem and the LP relaxation have drastically different optimal solutions. As an example consider the carbon rearrangement network described in Section 13.2, and the question: given 1 xylulose 5-phosphate (X5P) and an arbitrary amount of phosphate (P_i), find a pathway that maximises the production of acetylphosphate (AcP). As X5P contains 5 carbon atoms and AcP contains 2, it is clear that the maximum production from a single molecule must be at most 2 AcP. It turns out that the optimal integer



	e
A	-2
B	1

(a)



	e_1	e_2
A	-1	-1
B	1	-1
C	0	1

(b)

Figure 10.10: Examples of reaction networks with not totally unimodular stoichiometric matrices. (a) all entries in a TU matrix must be -1, 0, or $+1$. (b) the submatrix consisting of the top two rows has determinant 2.

Flow	Reaction
1.0	$\mathrm{G3P} + \mathrm{DHAP} \longrightarrow \mathrm{FBP}$
1.0	$\mathrm{G3P} \longrightarrow \mathrm{DHAP}$
0.5	$\mathrm{R5P} \longrightarrow \mathrm{X5P}$
0.5	$\mathrm{E4P} + \mathrm{F6P} \longrightarrow \mathrm{G3P} + \mathrm{S7P}$
1.5	$\mathrm{P_i} + \mathrm{X5P} \longrightarrow \mathrm{AcP} + \mathrm{G3P} + \mathrm{H_2O}$
0.5	$\mathrm{P_i} + \mathrm{F6P} \longrightarrow \mathrm{AcP} + \mathrm{E4P} + \mathrm{H_2O}$
0.5	$\mathrm{P_i} + \mathrm{S7P} \longrightarrow \mathrm{AcP} + \mathrm{R5P} + \mathrm{H_2O}$
1.0	$\mathrm{FBP} + \mathrm{H_2O} \longrightarrow \mathrm{P_i} + \mathrm{F6P}$
Overall	$\mathrm{X5P} + 1.5\,\mathrm{P_i} \longrightarrow 2.5\,\mathrm{AcP} + 1.5\,\mathrm{H_2O}$

Table 10.11: A pathway with maximum production of AcP from 1 X5P. See Section 13.2 for a table of molecule abbreviations.

solution just 1 AcP, by the single reaction $\mathrm{X5P} + \mathrm{P_i} \longrightarrow \mathrm{AcP} + \mathrm{G3P} + \mathrm{H_2O}$. However, the optimal solution to the LP relaxation of the problem yields 2.5 AcP, via the pathway described in Table 10.11. A scaling of this flow with a factor 2 gives an integer solution of course. Since LP solutions with integer-valued constraint matrices and objective functions with integer coefficients are rational, it is mathematically always possibly to scale the LP solution to integer values. The actual numbers, however, may become very large. Taking physiological constraints into account, the number of available individual molecules may be small, as low as 100 copies [Guptasarma 1995], and even smaller for biological macromolecules.

More importantly, however, the example above shows that the ILP frame-

work allows us to phrase questions on the network in a more sophisticated way. In FBA, we are effectively confined asking for overall yield. In the integer hyperflow setting we can just as well ask whether there is a pathway to produces 7 AcP from exactly 3 X5P.

Integrality Gap

In the previous example we maximised the production of a specific molecule, and saw that the ILP solution have objective value 1 and the LP relaxation have objective value 2.5. The ratio between these values is known as the *integrality gap*, and it is known that this gap can scale with the input instance. For a simple example, consider the reaction networks stemming from the polynomial-time reduction described in Section 10.3.4, reducing the well-known **Independent-Set** problem to maximising the production of a single molecule in a reaction network with bounded input. Applying the reduction to complete graphs with n vertices, and formulating the problem in terms of hyperflows, we obtain an integrality gap of $\frac{n}{2}$: for integer flows we can use at most 1 reaction, thus giving a maximum output of 1. When the integrality constraint is removed we can let the flow be 0.5 on all reactions, giving an output of $\frac{n}{2}$. The reaction networks for the complete graphs of size 3 and 4 are shown in Figure 10.12. This illustrates that the use of the LP relaxation is not just a technical detail, but changes the nature of the problem entirely.

Solution Enumeration

A linear program may have an uncountable number of optimal solutions as the variables are in the domain of real numbers. The solutions can however be described by enumerating the, possibly exponentially many, corners of the optimal face of the polyhedron defined by the linear program[Matheiss & Rubin 1980, Swart 1985]. This has also been applied to FBA [Lee *et al.* 2000].

In Section 10.2.4 we described a simple method for enumerating solutions when using integer linear programming. When restricting the solutions to be integer, we only have finitely many candidates with upper bounds on flow values, and it is straight-forward to enumerate not only optimal solutions, but also near-optimal solutions.

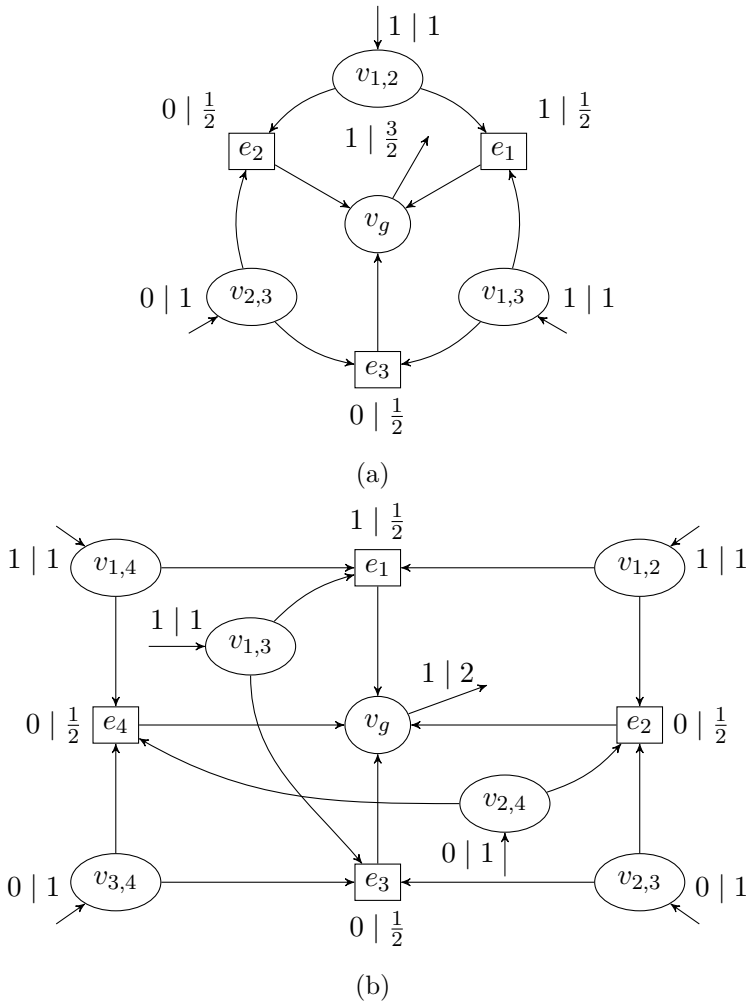


Figure 10.12: Reduction from the independent set problem to the problem of maximising output from a molecule in a reaction network, applied to the two graphs (a) K_3 and (b) K_4. The hyperedges are annotated with both a feasible integer flow and a feasible non-integer flow, written as ⟨integer⟩ | ⟨non-integer⟩. Allowing a maximum inflow of 1 to all vertices $v_{i,j}$ and maximising the outflow of v_g corresponds to finding a maximum independent set in the original graph.

Part IV

Applications

In this part we illustrate how the methods presented in the previous chapters can be used to analyse models of chemistry. The first chapter introduces a methods for calculating atom traces, which is also used for analysis the the subsequent chapters. Those next chapters are however mostly independent and can be read in any order. In the first three chapters we apply the methods to chemical examples, while we in the last chapter illustrate that they can also be used to solve a non-chemical graph transformation problem.

Chapter 11

Atom Tracing

This chapter is based on part of [Andersen et al. 2014b].

One method for studying living organisms is through what is called *isotope-labelling* experiments [Sauer 2006, Zamboni 2011]. In such an experiment the system under consideration are given modified input molecules, where the number of neutrons have been changed in one or more atoms. The molecules in the system are then measured in order to deduce where the modified atoms are located. Thus, combining the results with a reaction network modelling the system it may be possible to reason about the concrete trace of the labelled atoms through the system, and thereby study the detailed function of the organism. For such an analysis it is therefore important to have a model of the reaction network for the organism, annotated with accurate atom maps each reaction, and algorithms for their manipulation. However, atom maps are in most cases not available in chemical reaction databases.

As a subproblem we here consider a method based on rule composition for calculating the overall atom trace through a sequence of reactions modelled by DPO derivations. In Sections 12.3 and 13.1 we use this method for analysing complete pathways, while we in this chapter illustrate it by calculating the overall atom maps for an enzyme-catalysed multi-step reaction called β-lactamase. As a basis for the modelling of this mechanism we used the MACiE (Mechanism, Annotation, and Classification in Enzymes) database [Holliday *et al.* 2005, Holliday *et al.* 2012]. It is a publicly available, hand-curated database of enzymatic reaction mechanisms, where the individual steps of the overall enzyme reaction have been experimentally verified. Detailed stepwise mechanistic information can be accessed, in pictorial form, for more than 300 overall enzyme reactions. However, atom traces for the overall enzyme reactions are not available, and information of the mechanism's flexibility with respect to a reordering of individual steps to achieve a given overall reaction is not included.

11.1 Computing Atom Traces

Recall that reactions modelled by derivations $G \overset{p}{\Rightarrow} H$ can be represented by a rules (G, D, H) obtained through the full composition $(G, G, G) \bullet_{\supseteq} p$ (Section 7.2). The atom map for the reactions can then be obtained by these rules as described in Section 6.5.1. For a more concise notation we simply use $\bullet$

to mean $\bullet_{\supseteq}$, and $\imath_G$ as a shorthand for the identity rule (G, G, G). Additionally we assume that the rule composition operator is left-associative, i.e., $a \bullet b \bullet c$ means $(a \bullet b) \bullet c$.

Given a multiset of educt graphs G and a sequence of transformation rules $p_1, p_2, \ldots, p_k$, possibly modelling complete chemical reactions, we can compute all k-step reactions specified by the rules as $\imath_G \bullet p_1 \bullet p_2 \bullet \ldots \bullet p_k$. If specific target multiset of molecules H is desired, we can extend the composition expression to $\imath_G \bullet p_1 \bullet p_2 \bullet \ldots \bullet p_k \bullet_H \imath_H$ where the last composition implements an isomorphism check of the right-hand side by composing with $\imath_H$ using H as the common subgraph. Assuming H consists only of complete molecules, and that no complete molecule is a proper subgraph of another complete molecule, we can simply use full composition for this last step as well. Using full composition also enables the constraint that H only specifies a sub-multiset of the produced molecules. Such a "check-point constraint" can also be inserted in the middle of a composition sequence if there is a specific requirement for an intermediary state of the system.

11.2 The β-lactamase Mechanism

β-lactamases (MACiE entry 0002, EC number 3.5.2.6) are bacterial enzymes that convey resistance against β-lactame antibiotics such as penicillins by catalysing the overall reaction

$$\underset{\text{(CHEBI:35627)}}{\beta\text{-Lactam}} + \underset{\text{(CHEBI:15377)}}{\text{water}} \rightarrow \underset{\text{(CHEBI:33705)}}{\text{substituted } \beta\text{-amino acid}}$$

by means of a 5-step mechanism, which is detailed in MACiE as follows (see database entry for full details):

1. Lys73 deprotonates Ser70 thereby initiating a nucleophilic addition onto the carbonyl carbon of the β-lactam.
2. The resulting intermediate collapses, cleaving the C-N bond of the β-lactam and the nitrogen deprotonates Ser130.
3. Ser130 deprotonates Lys73.
4. Glu166 deprotonates water, which initiates a nucleophilic addition at the carbonyl carbon.
5. Collapse of this intermediate leads to cleavage of the acyl-enzyme bond and liberates Ser70, which in turn deprotonates the Glu166.

The 5 individual steps were modelled as transformation rules $p_1, \ldots, p_5$ depicted in Figure 11.1. For step (2) an alternative mechanism has been suggested in [Atanasov *et al.* 2000]: protonation of the β-lactam nitrogen occurs as the

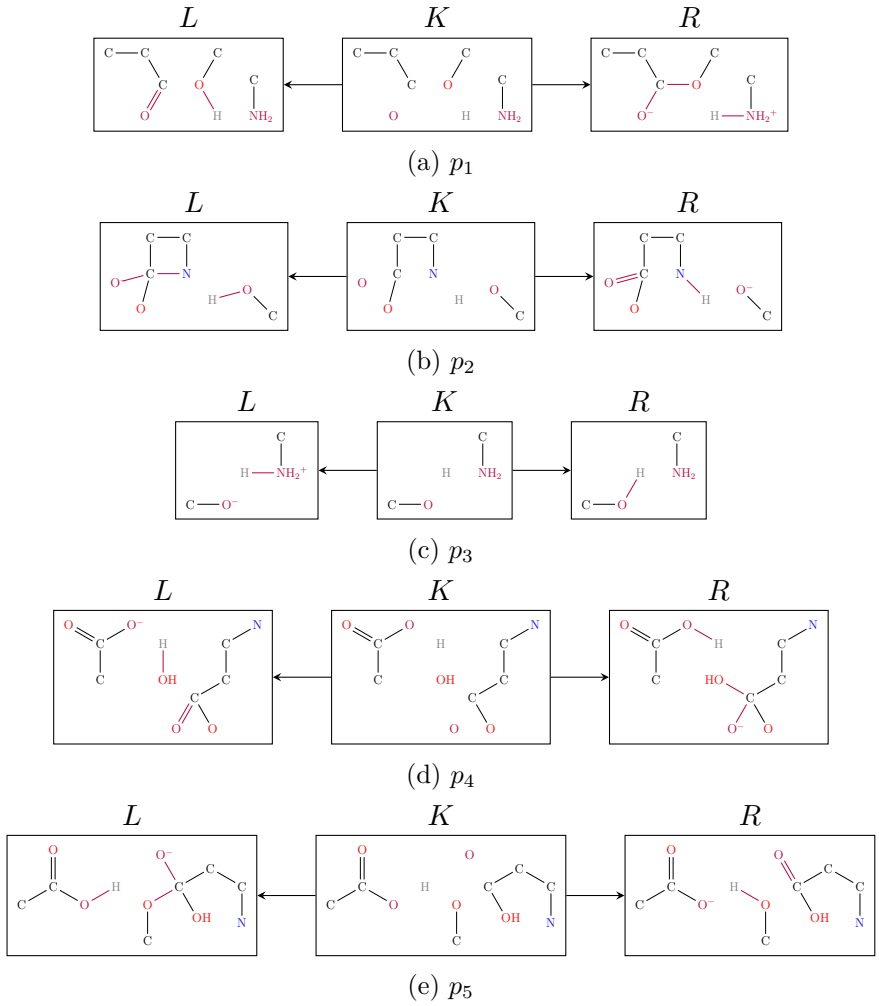


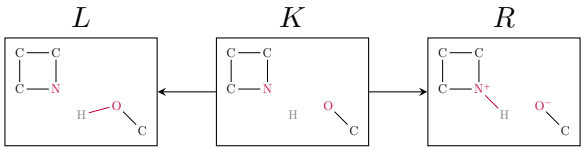
Figure 11.1: Transformation rules for the 5-step enzyme β-lactamase mechanism (MACIE entry 0002, EC number 3.5.2.6).

first step in the reaction as an initiation step and not as a consequence of the C-N bond cleavage. We modelled this alternative as a replacement of rule p_2 by two transformation rules p_{1b} and p_{2b}, depicted in Figure 11.2.

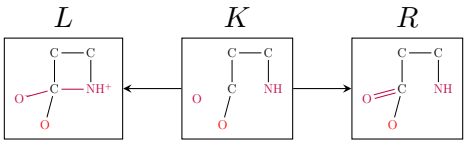
The atom traces for the overall reaction is computed by a composition of the rules $p_1, \ldots, p_5$ with the identity rule for the input compounds, i.e., the β-lactam, water, and the catalysts (Glu166, Lys73, and twice Ser130). Let G and H be the the graph representation of the input and output compounds, respectively. The overall composition

$$\imath_G \bullet p_1 \bullet p_2 \bullet p_3 \bullet p_4 \bullet p_5 \bullet \imath_H \tag{11.1}$$

results in the two overall rules depicted in Figure 11.3. Both are in agreement with the overall mechanism given in MACiE and differ only in their hydrogen

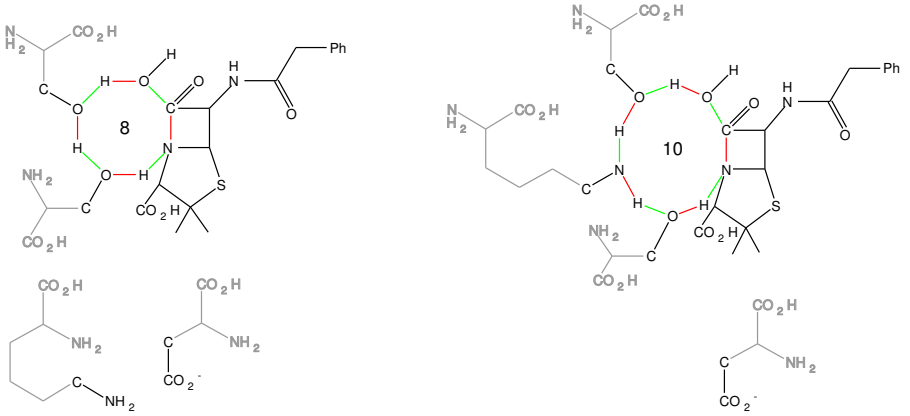


(a) p_{1b}



(b) p_{2b}

Figure 11.2: Transformation rules to replace step p_2 from Figure 11.1, based on the mechanism as suggested in [Atanasov *et al.* 2000].



(a) (b)

Figure 11.3: The two overall reactions resulting from either composition Equations (11.1) and (11.2), using the elementary steps of the β-lactamase (MACiE entry 0002, EC number 3.5.2.6). Red bonds are broken and green bonds are formed during the transformation. While the overall reactions (as typically found in metabolic databases such as KEGG or MetaCyc) are identical, they differ in their hydrogen trace and the size (8 or 10) of the cyclic virtual transition state. Note that the acid/basic catalysts (the two amino acids lysine and glutamine) needed for the reaction to work still show up as precondition in the overall rules. Using partial composition results in two more generic overall reactions. These two rules are depicted as the strict subgraphs resulting from removing the grey parts from the catalysts.

traces. The overall cyclic virtual transition states are an 8 cycle and a 10 cycle, which only differ by the exchange of a hydrogen in the amino group of Glu. The alternative model for step 2, which corresponds to

$$\imath_G \bullet p_1 \bullet p_{1b} \bullet p_{2b} \bullet p_3 \bullet p_4 \bullet p_5 \bullet \imath_H \tag{11.2}$$

results in the same two overall rules.

In order to check the flexibility of the reaction with respect to the order of the individual steps of the enzyme mechanism, we investigated all permutations of the rules for the composition order and verified whether the resulting overall rule produces the substituted β-amino acid as final product. Formally, we compute

$$\imath_G \bullet p_{\sigma(1)} \bullet \ldots p_{\sigma(5)} \bullet \imath_H$$

for all 120 permutations σ. Only the following three compositions are well-defined and result in the expected overall rules: $(p_1, p_2, p_3, p_4, p_5)$, $(p_1, p_2, p_4, p_3, p_5)$, and $(p_1, p_2, p_4, p_5, p_3)$. A detailed inspection shows that step p_3 is the recycling step of the mechanism, which can be applied concurrently to steps p_4 and p_5.

The same experiment based on the rule set $\{p_1, p_{1b}, p_{2b}, p_3, p_4, p_5\}$ shows that eight compositions are possible, all resulting in the same atom traces as given above. The first two steps need to be p_1 and p_{1b}, their relative order however is arbitrary. The subsequent rules p_{2b}, p_4, and p_5 must be in this order. The recycling step p_3 requires the rules p_1 and p_{1b} as prerequisite, but can be performed concurrently to the remaining steps, i.e., it may appear in position 3, 4, 5, or 6, thus accounting for the 8 feasible permutations.

This method allows for an automated analysis of the flexibility of the ordering of individual steps. Note that usually a relatively small number of all possible permutations have to be computed, as most often already the composition of a prefix of an arbitrarily chosen permutation is not possible. For instance, in the previous example, only two of the 30 possible initial two steps are feasible, which prunes most compositions early. The DPO framework provides an inroad to reduce the computational efforts even further. Since each rule is reversible, feasibility can be tested by exploring the space of overall rules from both ends and checking for overlaps at intermediate steps rather then expanding the possible pathways from one end only.

When using full composition we must specify all educt molecules in the initial graph, but we can instead use partial composition to automatically detect the required functionality of the catalysts and the additional compounds (in this case a water molecule). Let G' be the graph representation of β-lactam which is the core compound of the reaction, and H' the corresponding core product molecule. The partial composition of the rules

$$\imath_{G'} \bullet^c_\supseteq p_1 \bullet^c_\supseteq p_2 \bullet^c_\supseteq p_3 \bullet^c_\supseteq p_4 \bullet^c_\supseteq p_5 \bullet^c_\supseteq \imath_{H'}$$

result in the overall rule as depicted highlighted in Figure 11.3, i.e., any grey molecule or edge disappears. The overall rules show the automatic inference of the necessity of the four functional units of the catalysts and the necessity of the water molecule, as they are subsequently added to the left side of the overall rule during the partial rule composition. When defining transformation rules the difficulty often lies in the question of defining the size of the context around a reaction centre: a large context leads to a very specific rule, while a too small context might lead to chemically invalid reactions. Comparing full and partial compositions can be employed as a method to detect the functional units of the catalysts.

The atom mapping of the full composition result shows that in the composed rule with the 8-cycle the acid-base catalysts lysine and glutamic acid are unmodified during the overall process although they are necessary for the mechanism. In the composed rule with the 10-cycle only the acid-base catalyst glutamic acid is unmodified. The other catalysts and the water molecule are modified, however only based on the fact that the hydrogen atom for proton donation is different from the accepting hydrogen.

Chapter 12

The Formose Reaction

The formose reaction, first described in [Butlerov 1861], is a process where formaldehyde is converted into sugars. First the conversion is slow but then rapidly consumes formaldehyde, while turning brown and getting a sweet taste. The sudden conversion hints at an autocatalytic reaction and multiple candidates for its mechanism has been proposed, e.g., in [Breslow 1959] and [Benner *et al.* 2010]. The process is further interesting as it has been hypothesised to have taken place on the prebiotic Earth [Benner *et al.* 2010], and it may thus be a source of complex carbohydrates. A recent review of formose studies can be found in [Delidovich *et al.* 2014].

Based on [Benner *et al.* 2010, Figure 9] we have modelled the chemistry of formose by a graph grammar with starting graphs for formaldehyde and glycolaldehyde, Figure 12.1. The transformation rules, shown in Figure 12.2, model two reversible reactions: keto-enol tautomerism and the aldol reaction. As the initial condensation of formaldehydes is a reaction we leave it out of the model, and instead include glycolaldehyde as a seed molecule. The language of this grammar is clearly infinite, so in order to analyse the network for the chemistry we generally assume that each molecules have at most 9 carbon atoms. We can thus generate the complete network with the following exploration strategy.

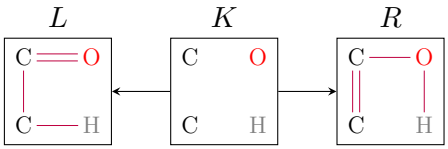
$$
\begin{aligned}
Q = & \texttt{addUniverse}[\text{formaldehyde}] \\
& \rightarrow \texttt{addSubset}[\text{glycolaldehyde}] \\
& \rightarrow \texttt{rightPredicate}[P, \\
& \qquad \texttt{repeat}[\texttt{parallel}[\{p_1, p_2, p_3, p_4\}]] \\
&]
\end{aligned}
$$

$$
P(G \overset{p}{\Rightarrow} H) \equiv \forall h \in H : h \text{ has at most 9 carbon atoms}
$$

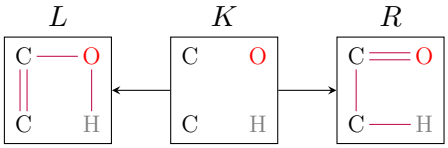
The resulting network consists of 284 molecules and 978 reactions, and can be calculated in less than a minute on a normal desktop computer.

(a) (b)

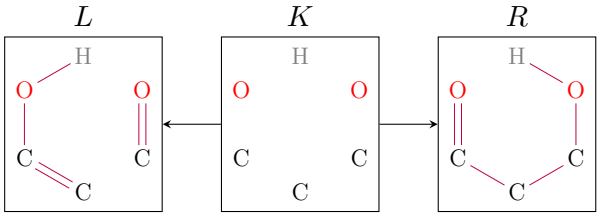
Figure 12.1: Starting graph for the formose chemistry: (a) formaldehyde and (b) glycolaldehyde.



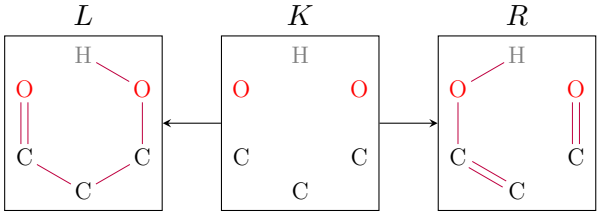
(a) $p_1 = p_2^{-1}$, keto to enol conversion



(b) $p_2 = p_1^{-1}$, enol to keto conversion



(c) $p_3 = p_4^{-1}$, aldol addition



(d) $p_4 = p_3^{-1}$, aldol splitting

Figure 12.2: Transformation rules for the formose chemistry. (a), (b) the rules p_1 and p_2 model keto-enol tautomerism. (c), (d) the rules p_2 and p_3 model aldol addition and aldol splitting.

From the grammar it follows that each molecule has exactly one double bond. As a shorthand we use the naming scheme $\mathrm{C}\langle N\rangle_{\langle t\rangle}$ for each molecule, where $\langle N\rangle$ specifies the number of carbon atoms and $\langle t\rangle$ indicates the position of the double bond. We use $_{\mathrm{a}}$ for aldehydes, $_{\mathrm{e}}$ for enol forms, and $_{\mathrm{k}}$ for ketones. When relevant we may use further annotations, e..g, to specify the position of the enol- and keto-double bonds. Formaldehyde is then referred to as $\mathrm{C1}_a$, or simply C1, while glycolaldehyde becomes $\mathrm{C2}_a$.

12.1 Autocatalytic Pathways

This section is based on part of the results of [Andersen et al. 2015a].

One of the features of the formose reaction is that it is autocatlytic, because

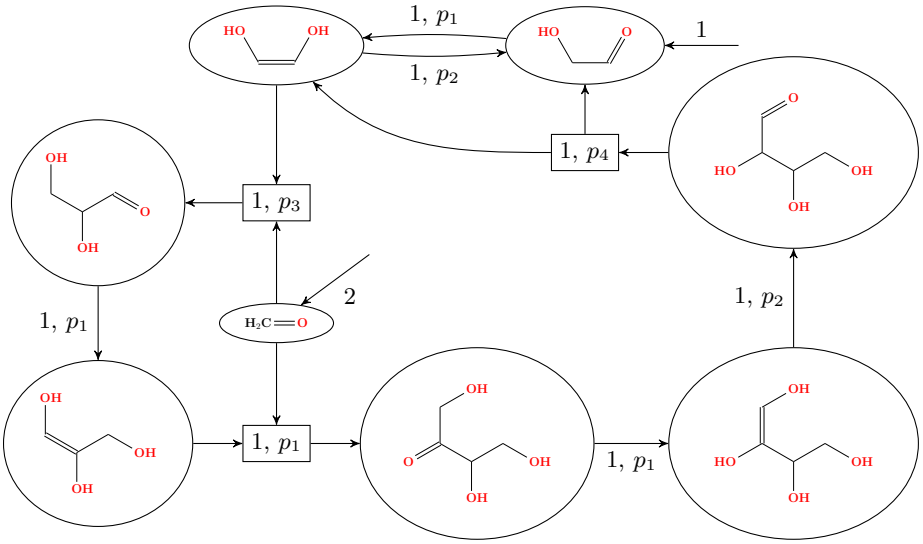


Figure 12.3: An overall autocatalytic pathway in the formose chemistry. The reactions are annotated with the flow of the pathway, and the transformation rule used to generate reaction.

it suddenly converts huge amounts of formaldehyde into larger sugars. With our graph grammar model, based on [Benner *et al.* 2010], we can find a short pathway which is overall autocatalytic in glycolaldehyde, in the sense described in Chapter 10. This pathway, which convert 1 glycolaldehyde ($C2_a$) and 2 formaldehyde (C1) into 2 glycolaldehyde, is depicted in Figure 12.3. However, experimental evidence exists [Ricardo *et al.* 2006, Kim *et al.* 2011] that this base cycle can not account for the massive consumption of formaldehyde. Thus, the question is which other potentially autocatalytic cycles exist in the chemistry.

Here we study this question in a limited form, by enumerating overall autocatalytic pathways with the overall reaction $C2_a + 2\,C1 \longrightarrow 2\,C2_a$. We use the reaction network described in the introduction, containing sugars with at most 9 carbon atoms. To further limit the scope of enumeration we consider two pathways equal if they have non-zero flow on the same set of hyperedges. Technically we add an indicator variable $z_e \in \{0, 1\}$ for each hyperedge e, and constrain it to be 1 if and only if the flow on e is positive. The enumeration algorithm (Section 10.2.4) is then asked to only branch on these indicator variables.

Table 12.4 shows the resulting number of solutions found, grouped by the number of reactions used and the maximum size of molecules involved. The enumeration was split into 6 queries, one for each row of the table, and the combined computation time was approximately 134 hours. We were not able to find all the solutions corresponding to the two unknown entries en the table. Each of those queries, in the current implementation, needs more than

	Maximum #C						
Unique reactions used	4	5	6	7	8	9	Sum
6	0	0	1	1	1	2	5
7	0	0	0	0	0	2	2
8	1	5	7	17	37	68	135
9	0	0	12	12	37	69	130
10	0	12	50	274	849	—	≥ 1185
11	0	5	41	190	738	—	≥ 974

Table 12.4: Overview of the number of overall autocatalytic flows in the formose chemistry. Solutions are grouped by the number of unique reactions used, and by the number of carbon atoms in the largest molecule used. We were not able to compute the missing entries due the demand of computation time (more than 200 hours) and memory (more than 64 GB RAM). The pathway in Figure 12.3 is the single pathway in the first column, where the largest molecule has 4 carbon atoms.

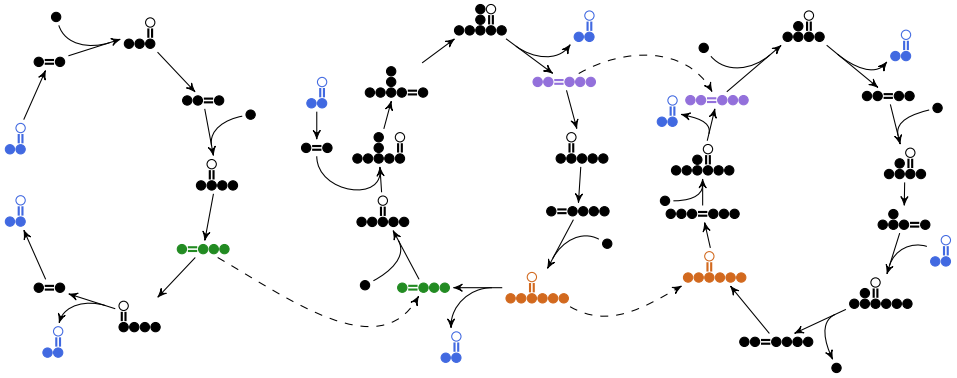
200 hours of computation time and more than 64 GB memory.

This computational analysis of the chemical space of the formose process reveals that there potentially is a very high number of autocatalytic cycles. However, the base cycle from Figure 12.3 has a special property: starting from just 1 glycolaldehyde and at least 2 formaldehyde, and only using the reactions in the quantity specified by the flow, this is the only pathway of those enumerated that can be realised without borrowing a molecule. Informally we can then speak of one pathway "triggering" another by temporarily lending it a molecule. This concept is exemplified in Figure 12.5 with a chain of 3 pathways starting from the base cycle are illustrated. A formal treatment of the concept of when a pathway is "realisable" and what it specifically means that a pathway "trigger" another is out of scope for this work, but in Section 15.2 we outline some ideas in this direction. Here we simply note that the automatic enumeration of pathways conforming to a certain motif, such as autocatalysis, opens for higher-order analysis of chemistries.

12.2 Product Stabilisation by Borate

This section is based on part of the results of [Andersen et al. 2014c].

The formose reaction has been discussed as a prebiotic source of carbohydrates, and especially five-carbon sugars are interesting in this context as they are needed to form nucleotides [Benner *et al.* 2012]. However, the unguided formose reaction results in a mix of sugars of different sizes [Decker *et al.* 1982], and extra conditions must be imposed to select for specific molecules. An ex-



(a) (b) (c)

Figure 12.5: Schematic overview of 3 autocatalytic pathways in the formose chemistry, where carbon-carbon double bonds and carbonyl groups are shown, while hydroxyl groups and hydrogens are implicit. The first pathway can trigger the second pathway via the green molecule, and the second pathway can in turn trigger the third pathway using either the purple or orange molecule. The overall autocatalytic compound, glycolaldehyde, is shown in blue.

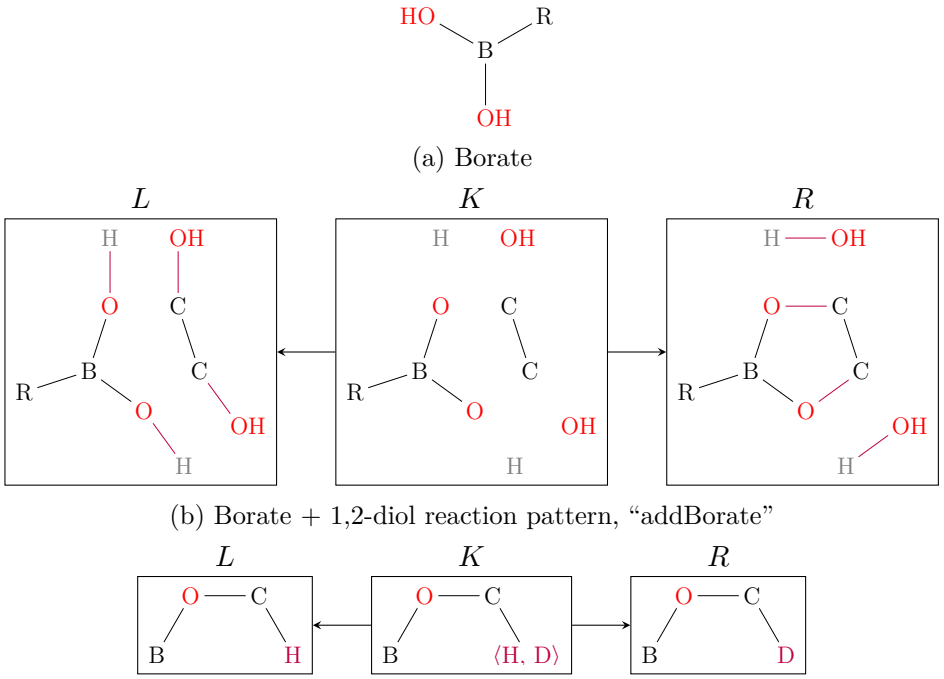
perimentally confirmed candidate for such a condition is the addition of borate to the reaction [Ricardo *et al.* 2004, Benner *et al.* 2010, Kim *et al.* 2011]. The idea is that a borate can form a complex with a sugar by attachment to a 1,2-diols group, i.e., a pair of neighbouring hydroxyl groups on the carbon chain (see [Ricardo *et al.* 2004, Figure 1]). This inhibits the usual keto-enol tautomerisation reactions.

Using the strategy language, described in Chapter 9, we here try to model the borate inhibited formose reaction. The aim is to automatically generate a reaction network consistent the experimentally observed compounds. To keep the model simple we use a borate-like molecule, Figure 12.6a, with just two hydroxyl groups instead of a complete molecule. For enabling the formation of borate complexes we use the transformation rule shown in Figure 12.6b. This rule is additionally equipped with the matching constraint that the two carbons may not have incident double bonds. This reaction pattern is described in [Benner *et al.* 2010] as inhibiting keto-enol tatutomerism by making the hydrogen atoms attached to the carbon atoms non-acidic. To approximate this behaviour we relabel these vertices from `H` to `D`, thereby preventing the reaction pattern of enolisation, p_1, from matching at these locations.

For this proof-of-concept modelling we limit the molecule size to 5 carbon atoms, which we model with a right predicate strategy around the application of the basic formose reaction patterns:

$$\texttt{rightPredicate}[P_{\#C}, \texttt{parallel}[\{p_1, p_2, p_3, p_4\}]]$$

$$P_{\#C}(G \overset{p}{\Rightarrow} H) \equiv \forall h \in H : h \text{ has at most 5 carbon atoms}$$



(a) Borate

(b) Borate + 1,2-diol reaction pattern, "addBorate"

(c) Relabelling of hydrogen to make it non-reactive, "HtoD"

Figure 12.6: Molecules and reaction patterns for borate inhibition of the formose reaction. The borate molecule (a) is modelled with only two hydroxyl groups to simplify the model. (b) the reaction pattern for forming borate complexes with 1,2-diols. This rule additionally has a matching constraint: none of the carbon atoms may not have incident double bonds. To approximate the subsequent non-reactivity of the hydrogens on the carbon atoms we relabel them to `D` using the reaction "HtoD" (c).

As a reference, we generate the non-inhibited reaction network with the strategy Q_{BFS}:

$$
\begin{aligned}
Q_{\text{BFS}} = {} & \texttt{addUniverse}[\{\text{formaldehyde}\}] \\
& \rightarrow \texttt{addSubset}[\{\text{glycolaldehyde}\}] \\
& \rightarrow \texttt{repeat}[\\
& \qquad \texttt{rightPredicate}[P_{\#C}, \texttt{parallel}[\{p_1, p_2, p_3, p_4\}]] \\
&]
\end{aligned}
$$

Not all molecules can actually bind with borate and must therefore be preserved while the other molecules form complexes. This is modeled with a revive strategy around the actual complex forming reaction pattern, "addBorate". After the potential forming of a borate complex, the relevant hydrogen atoms must be made inactive using the rule "HtoD". The number of relevant hydrogens may not be the same for all molecule and therefore the relabelling

strategy is embedded in both a repeat and revive strategy. This models the notion of "as many times as possible" on a collection of molecules. The reaction network with borate inhibition can thus be calculated by the following strategy:

$$\begin{aligned} Q_{\text{borate}} = & \texttt{addUniverse}[\{\text{formaldehyde}, \text{borate}\}] \\ & \rightarrow \texttt{addSubset}[\{\text{glycolaldehyde}\}] \\ & \rightarrow \texttt{repeat}[\\ & \qquad \texttt{revive}[\text{addBorate}] \\ & \qquad \rightarrow \texttt{repeat}[\texttt{revive}[\text{HtoD}]] \\ & \qquad \rightarrow \texttt{rightPredicate}[P_{\#C}, \texttt{parallel}[\{p_1, p_2, p_3, p_4\}]] \\ &] \end{aligned}$$

Let $\mathcal{G}$ denote the set of molecules used and generated by the evaluation of Q_{borate} on the empty graph state. This set of molecules contain both borate complexes and simple carbohydrates without boron. To obtain the equivalent set of molecules with the borate removed we can use the strategy

$$\begin{aligned} Q_{\text{canon}} = & \texttt{addSubset}[\mathcal{G}] \\ & \rightarrow \texttt{repeat}[\texttt{revive}[\text{DtoH}]] \\ & \rightarrow \texttt{repeat}[\texttt{revive}[\text{removeBorate}]] \end{aligned}$$

with "removeBorate" being the inverse transformation rule of "addBorate", and "DtoH" being the inverse of "HtoD". Note that "removeBorate" requires water molecules as educts, but if "addBorate" was ever used in Q_{borate} these molecules must be in $\mathcal{G}$.

As a variant of the network, we also calculate the network with a an extra molecule, dihydroxyacetone, in the subset:

$$\begin{aligned} Q^{+}_{\text{borate}} = & \texttt{addUniverse}[\{\text{formaldehyde}, \text{borate}\}] \\ & \rightarrow \texttt{addSubset}[\{\text{glycolaldehyde}, \text{dihydroxyacetone}\}] \\ & \rightarrow \texttt{repeat}[\\ & \qquad \texttt{revive}[\text{addBorate}] \\ & \qquad \rightarrow \texttt{repeat}[\texttt{revive}[\text{HtoD}]] \\ & \qquad \rightarrow \texttt{rightPredicate}[P_{\#C}, \texttt{parallel}[\{p_1, p_2, p_3, p_4\}]] \\ &] \end{aligned}$$

In Figure 12.7 the reference reaction network created with Q_{BFS} is shown. Reactions in black are active only in the basic formose reaction with formaldehyde and glycolaldehyde as the set of input molecules. If borate is added to the input set of molecules, the reactions highlighted in blue are active, while the rest of the network is inactive. Finally if dihydroxyacetone is added to

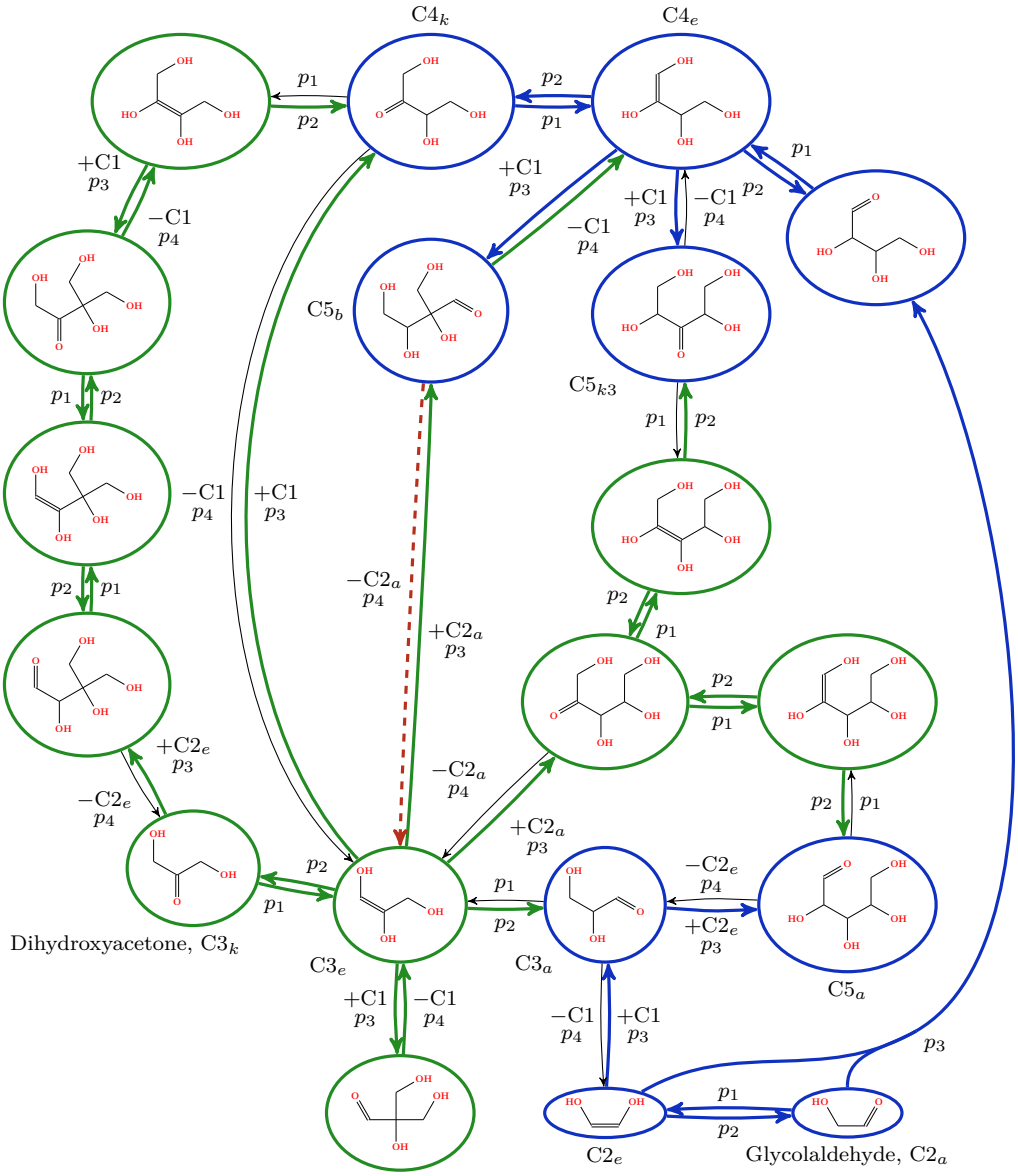


Figure 12.7: The reaction network of the formose chemistry as calculated with the strategy Q_{BFS}. The blue subnetwork correspond to the borate inhibited network calculated with Q_{borate}. The green and blue networks together with the red reaction (C5$_b$ to C3$_e$) correspond the network calculated with Q^{+}_{borate}, i.e., with dihydroxyacetone as an input compound. Each reaction is annotated with the reaction pattern, p_i, used to realise the concrete reaction. For the aldol reactions, p_3 and p_4, the secondary educt (+) or product (−) is additionally shown. The addition of borate in Q^{+}_{borate} is done with the strategy **revive**[addBorate], meaning that at most 1 borate is added in each iteration. The red reaction is no longer available if the addition is done with the strategy **repeat**[**revive**[addBorate]], meaning "add as many as possible".

the input set of molecules the reactions highlighted in green are activated in addition to the blue part of the network. The evaluation of Q_{borate} leaves only the blue reactions, which are selective pathways from glycolaldehyde ($C2_a$) to five-carbon sugars ($C5_b$, $C5_3$, $C5_{l,2}$) active, while the rest of the network is shut down via borate inhibition. These pathways rely on a constant replenishment of glycolaldehyde. Here dihydroxyacetone ($C3_k$) comes into play. $C3_k$ can only be formed from within the formose network via retro-aldol reaction from higher carbohydrates. If added to the reaction network an catalytic loop is activated (sub-network in green: $C3_k$, $C3_e$, $C4_k$, $C4_e$, $C5_b$, retro-aldol red dashed arrow to $C3_e$ and $C2_a$) supporting the blue sub-network since $C2_a$ ends up as some five-carbon sugars in the blue sub-network. $C3_e$ enters another round in the cycle to construct another $C2_a$. These computational results are in very good agreement with the experimental results presented in [Kim *et al.* 2011].

12.3 Carbon Tracing

This section is based on part of the results of [Andersen et al. 2014b].

As evident from Section 12.1 there are many theoretical possibilities for autocatalysis in the formose chemistry. In [Breslow 1959] a cycle is suggested which utilises dihydroxyacetone, depicted as the black pathways in Figure 12.8. However, using our model of the chemistry, based on [Benner *et al.* 2010], we see a slightly shorter pathway bypassing dihydroxyacetone, depicted in green in Figure 12.8. In this section we analyse the possible carbon traces in these pathways, using the method described in Chapter 11.

The input graph G is comprised of two formaldehyde and one glycolaldehyde molecule, and the goal H consists of two copies of glycolaldehyde. Both are represented by their corresponding identity rules $\imath_G = (G, G, G)$, $\imath_H = (H, H, H)$. The two proposed pathway are represented as the following composition expressions, where we use $\bullet$ to denote full composition $\bullet_{\supseteq}$.

$$\imath_G \bullet p_1 \bullet p_2 \bullet p_1 \bullet p_2 \bullet p_1 \bullet p_1^{-1} \bullet p_2^{-1} \bullet p_1^{-1} \bullet \imath_H \tag{12.1}$$

and

$$\imath_G \bullet p_1 \bullet p_2 \bullet p_1 \bullet p_1^{-1} \bullet p_1 \bullet p_2 \bullet p_1 \bullet p_1^{-1} \bullet p_2^{-1} \bullet p_1^{-1} \bullet \imath_H \tag{12.2}$$

Based on a prefix composition of Equations (12.1) and (12.2), the traces for the intermediates can be computed. The carbon traces are summarised in Figure 12.9. Note that in this figure sequences of isomerisation and aldol-addition steps are depicted as one step in order to minimize clutter.

The rule composition based on Equation (12.2) results in six non-isomorphic composed overall rules, each having a different carbon trace for the 4 carbon atoms of G. One of those rules is depicted in Figure 12.10. While 4! carbon

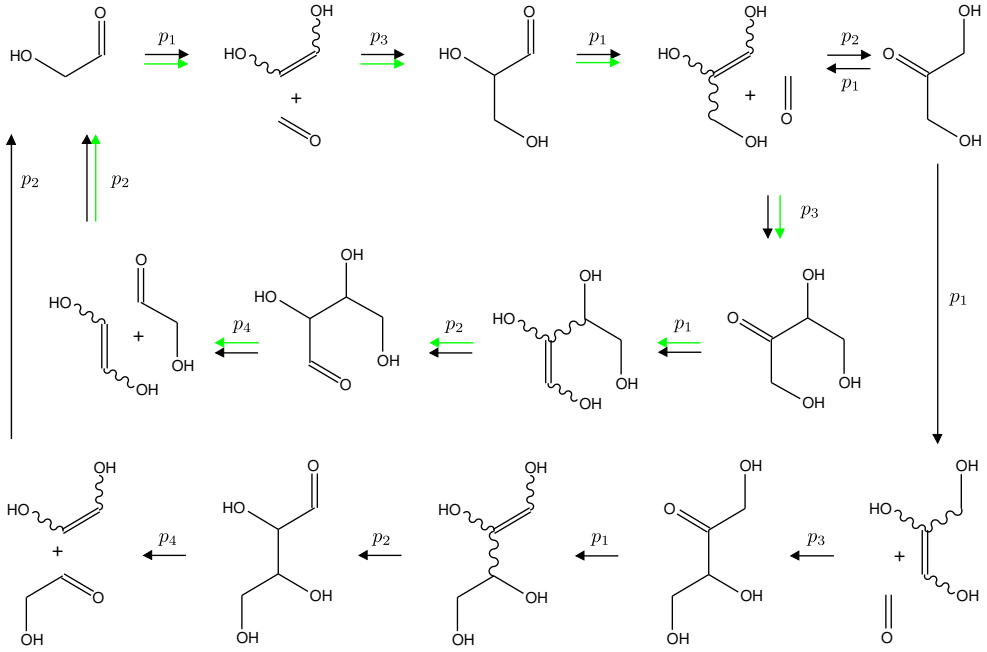


Figure 12.8: Detailed mechanism for suggestions for the formose process. The labels p_1 and p_2 indicate the keto-enol tautomerisation (forward and backward), p_3 and p_4 refer to aldol- and retro-aldol reaction. The pathways indicated by the black arrows have dihydroxyacetone as an intermediate, while the pathway with green arrows are shorter and do not use dihydroxyacetone. In order to allow and easy visual tracking of carbon atoms, the sequence of compounds is duplicated after the second aldol addition.

traces is a trivial upper bound, the mechanism allows only for six of them as the carbon from the second added formaldehyde cannot end up as the carbonyl carbon in the resulting glycolaldehyde. If a labelling experiment could be performed with all carbons uniquely labelled and if the glycolaldehydes after exactly one instantiation of the reaction cycle would be analysed, then not twelve but only nine different glycolaldehydes could be observed. If the mechanism follows Equation (12.1) (i.e., dihydroxyacetone is not an intermediate of the mechanism), the two input formaldehydes never combine into the same glycolealdehyde, reducing the set of overall reactions to four rules. Using the same labelling experiment as above, only six different glycolaldehydes could be observed.

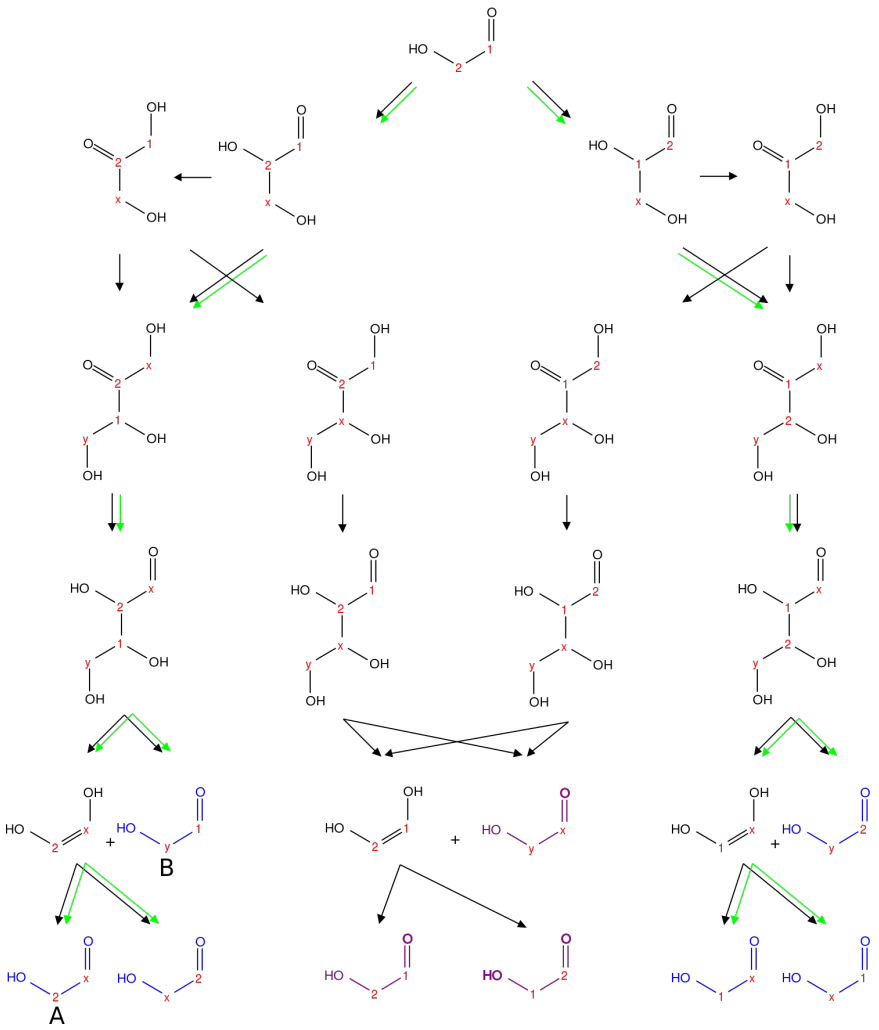


Figure 12.9: Carbon atom traces for one round of the formose process based on the mechanisms in Figure 12.8. Green reaction arrows indicate possible carbon atom traces following the shorter formose cycle, and black reaction arrows indicate possible traces following the cycles having dihydroxyacetone as intermediate. The carbonyl (resp. alcohol) carbon of the starting molecule glycolealdehyde is labelled 1 (resp. 2). After condensation of this molecule with two formaldehydes labelled x and y, the intermediate molecule decomposes into two glycolealdehydes. Depending on the mechanism, the labelled carbon atoms end up in 9 (resp. 6) different positions of the resulting glycolealdehydes (shorter cycle: blue molecules, longer cycles: blue and purple molecules). Note that the carbon from the second formaldehyde (y) can not end up as the carbonyl carbon in the resulting glycolealdehydes. The shorter cycle allows for a strict subset of carbon traces only, the two formaldehydes never recombine into a glycolealdehyde. From the 6 (resp. 4) possible composed overall reactions of the longer (resp. shorter) cycle, the one that results in the two blue glycolaldehydes A and B is depicted in Figure 12.10.

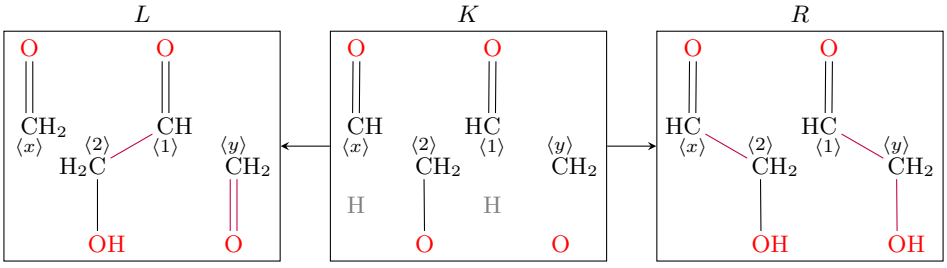


Figure 12.10: One of six possible overall rules based on the composition Equation (12.2) including carbon atom trace information. The mapping of the atoms corresponds to creation of the glycolaldehydes denoted A and B in Figure 12.9.

Chapter 13

The Glycolysis Pathway

Glycolysis is one of the central pathways in carbon metabolism, which converts 1 glucose molecule (a C6 sugar) into 2 pyruvate molecules (a C3 acid) while creating energy-rich ATP molecules. There are multiple variations of the glycolysis process (see [Bar-Even *et al.* 2012] for a review), with the most common being the Embden-Meyerhof-Parnas (EMP) pathway. An alternate pathway is the Entner-Doudoroff (ED) pathway [Entner & Doudoroff 1952], which only creates 1 ATP molecule as opposed to the EMP which creates 2 ATP molecules for each glucose. In the first section we illustrate how the methods described in Chapter 11 can be used to automatically create accurate atom maps the overall EMP and ED pathways.

After the glycolysis pathway the two resulting pyruvates can be converted into C2 molecules which can then be used in the citric acid cycle. However, also 2 CO_2 molecules are created during this conversion, meaning only 4 of the original 6 carbon atoms actually end up as C2 molecules. Recently it has been found [Bogorad *et al.* 2013] that it is possible to engineer a *non-oxidative* glycolysis pathway with perfect conversion. It thus converts 1 glucose molecule into 3 C2 compounds, but without the creation of ATP. In the second half of this chapter we use enumerate many possible implementations of such a pathway, using the pathway model described in Chapter 10.

13.1 Carbon Tracing the EMP and ED Pathways

This section is based on part of the results of [Andersen et al. 2014b].

Isotope labelling experiments in glycolysis are commonly used to analysis the activity of the different pathway variations (e.g., see [Borodina *et al.* 2005]). It is known that the EMP and ED pathways lead to different carbon traces, but since atom maps are usually not available in databases it can be tedious and error prone to analyse trace data manually. We here demonstrate that a chemistry model based on DPO transformation rules enables the automatic inference of atom traces for complete pathways, illustrated with the EMP and ED pathways. The two pathways have been modelled using the following transformation rules:

p_1 Pyranose-furanose	p_3 Ketose-aldose	p_5 ATP-dephosphorylation
p_2 Furanose-linear	p_4 ATP-phosphorylation	p_6 NAD+-phosphorylation

p_7 Phosphomutase	p_{10} NAD+-oxoreductase	p_{13} Reverse aldolase
p_8 Enolase	p_{11} Lactonohydrolase	
p_9 Keto-enol	p_{12} Hydrolyase	

The rules were manually created using information from MACiE database (described in Chapter 11) to ensure accurate atom maps for the individual reaction patterns. They are visualised in Figures 13.1 and 13.2.

We use $G(EMP)$ to denote the graph of educts to the EMP pathway, consisting of 1 glucose, 2 ATP, 2 ADP, 2 phosphates, and 2 NAD^+. For the ED pathway we likewise have $G(ED)$, consisting of 1 glucose, 1 ATP, 1 ADP, 1 phosphate, and 2 NAD^+. Correspondingly we have $H(EMP)$ as the output graph of the EMP pathway, with 2 pyruvates, 4 ATP, 2 NADH, 2 water, and 2 H^+. In the case of the ED pathway $H(ED)$ models 2 pyruvates, 2 ATP, 2 NADH, 2 water, and 2 H^+. Note that the same approach as presented in the β-lactamase example (Section 11.2) for automatically inferring the necessary functional groups could be applied, and an explicit definition of the catalysts in $G(\cdot)$ and $H(\cdot)$ would not be necessary.

For EMP we compute the composition

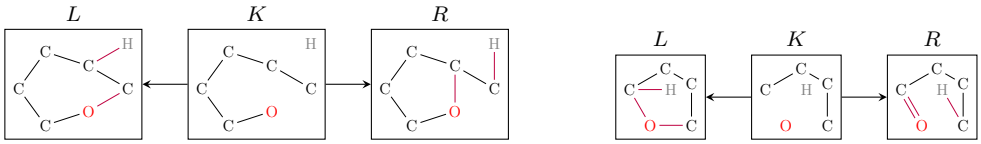
$$\imath_{G(EMP)} \bullet \overbrace{p_4 \bullet p_1 \bullet p_4 \bullet p_2 \bullet p_{13} \bullet p_3}^{\text{Glucose} \to \text{2 G3P}} \bullet \underbrace{(p_6 \bullet_\emptyset p_6) \bullet (p_5 \bullet_\emptyset p_5) \bullet (p_7 \bullet_\emptyset p_7) \bullet (p_8 \bullet_\emptyset p_8) \bullet (p_5 \bullet_\emptyset p_5) \bullet (p_9 \bullet_\emptyset p_9)}_{\text{2 G3P} \to \text{2 Pyruvate}} \bullet \imath_{H(EMP)}$$

and for ED we compute

$$\imath_{G(ED)} \bullet \underbrace{p_4 \bullet p_{10} \bullet p_{11} \bullet p_{12} \bullet p_{13}}_{\text{Glucose} \to \text{G3P + Pyruvate}} \bullet \underbrace{p_6 \bullet p_5 \bullet p_7 \bullet p_8 \bullet p_5 \bullet p_9}_{\text{G3P + Pyruvate} \to \text{2 Pyruvate}} \bullet \imath_{H(ED)}$$

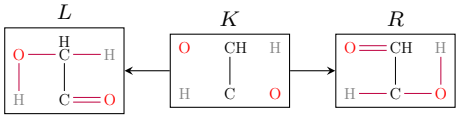
The resulting rules are depicted in Figure 13.3. To reduce clutter we only draw the glucose and pyruvate components. Formally this can be achieved by composing with rules on the left and right that binds and unbinds the unwanted components. Clearly, the carbon traces of the two rules differ. Such an approach can be used for an automated design of labelling experiments to detect the activity of pathway alternatives.

The prefixes of the rule composition expressions allows to infer all the intermediate compounds and their corresponding atom traces relatively to the input compounds. The summary of this analysis is depicted in Figure 13.4 for both pathways, though only with the traces for carbon atoms shown. The black reaction arrows show the EMP pathway, while the green arrows show the ED pathway. The six carbon atoms from glucose are converted into two pyruvate molecules in two different ways depending on whether EMP or ED was used to catabolise glucose. While the EMP pathway has a fructose 1,6-bisphosphate as an intermediate, in which a pentose ring is cleaved, in the ED pathway the hexose ring of the glucose 6-phosphate is cleaved. The carbon

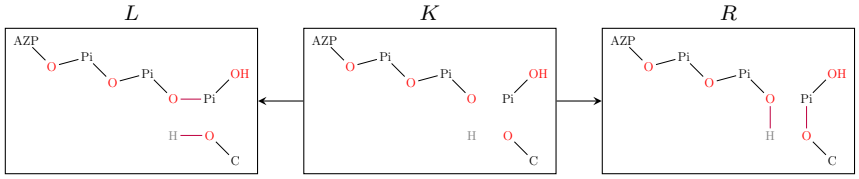


(a) p_1, Pyranose-furanose

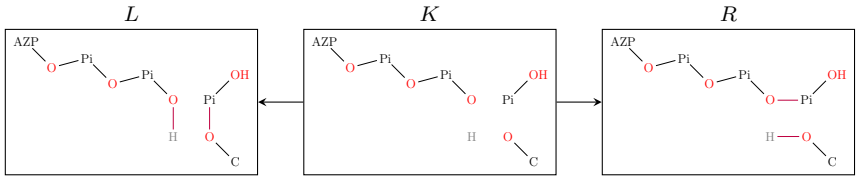
(b) p_2, Furanose-linear



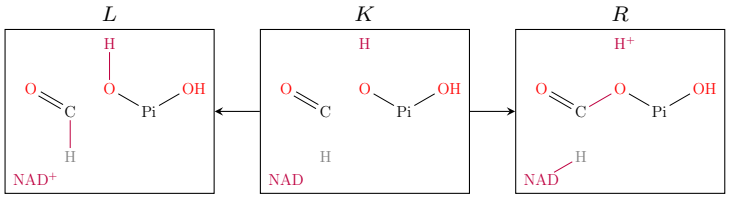
(c) p_3, Ketose-aldose



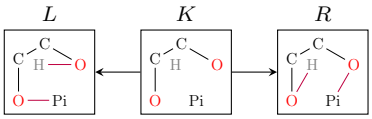
(d) p_4, ATP-phosphorylation



(e) p_5, ATP-dephosphorylation



(f) p_6, NAD+-phosphorylation



(g) p_7, Phosphomutase

Figure 13.1: Transformation rules for modelling the reactions in the EMP and ED pathways for glycolysis. In the modelling we have abbreviated certain groups into graphs with non-chemical labels. For example, part of phosphate groups are represented by vertices with the label `Pi`, and adenosine is represented by the label `AZP`.

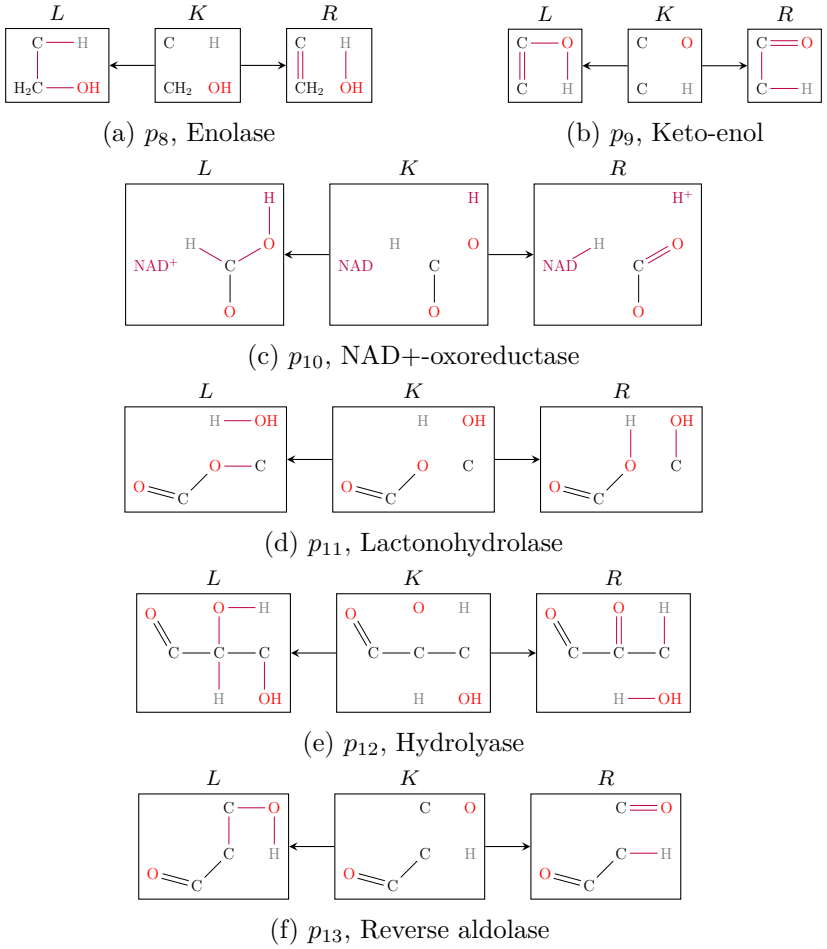


Figure 13.2: More transformation rules for modelling the reactions in the EMP and ED pathways for glycolysis.

trace of one of the two pyruvates is identical, while it is inverted in the other pyruvate.

13.2 Enumeration of Non-oxidative Pathways

This section is based on part of the results of [Andersen et al. 2015a].

Metabolic pathways have also been studied with the aim of using modified microbes to produce biofuel, or other compounds of commercial interest (e.g., see [Tao *et al.* 2001, Causey *et al.* 2003]). In this setting the glycolysis pathways can play a key role, by breaking glucose into smaller carbohydrates. However, the natural pathways based on EMP and ED only creates 2 C2 molecules from each C6 sugar, while the remaining two carbon atoms are expelled as CO_2.

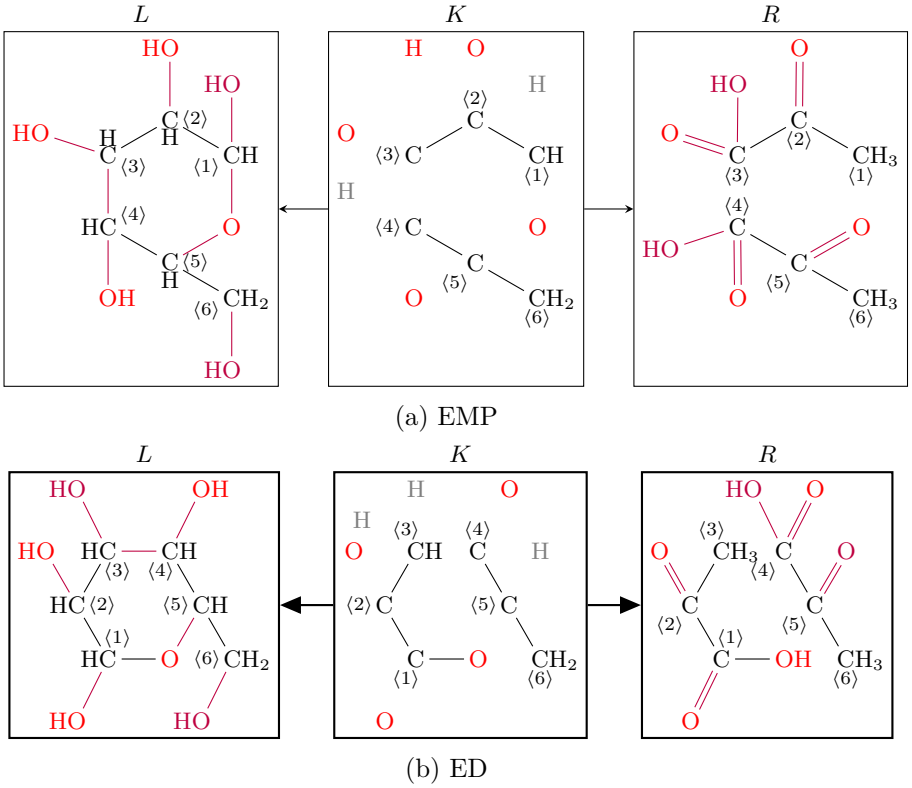


Figure 13.3: Simplified transformation rule for the overall (a) EMP pathway and (b) ED pathway, with each carbon atom labelled. Some hydroxyl-groups appear to be destroyed and created due to the simplification such that only glucose and pyruvate is depicted.

A recent study [Bogorad *et al.* 2013] addresses this attacks the aforementioned problem by hand-crafting a non-oxidative glycolysis (NOG) pathway, which fully converts C6 molecules into C2 molecules. The general logic of this designed pathway is to couple the splitting reaction that produces the desired C2 body and a C4 body as putative wast, to a carbon rearrangement network, which then recycles the C4 body into molecules, that can be fed back into the NOG as educts. With this strategy NOG achieves a 100 % carbon atom economy.

The paper [Bogorad *et al.* 2013] discusses several sources of variation for the structure of the NOG pathway. First the splitting reaction can be performed by two types of phosphoketolase (PK) enzymes, differing only in the their input sugar preference; either fructose (F) or xylulose (X). Second the carbon rearrangement network can go either via fructose 1,6-bisphosphate (FBP, a C5 sugar) or sedoheptulose 1,7-bisphosphate (SBP, a C7 sugar). This freedom allows for different structural designs for the NOG pathway, of which 3 suggestions are shown in [Bogorad *et al.* 2013, Figure 2]. In this section

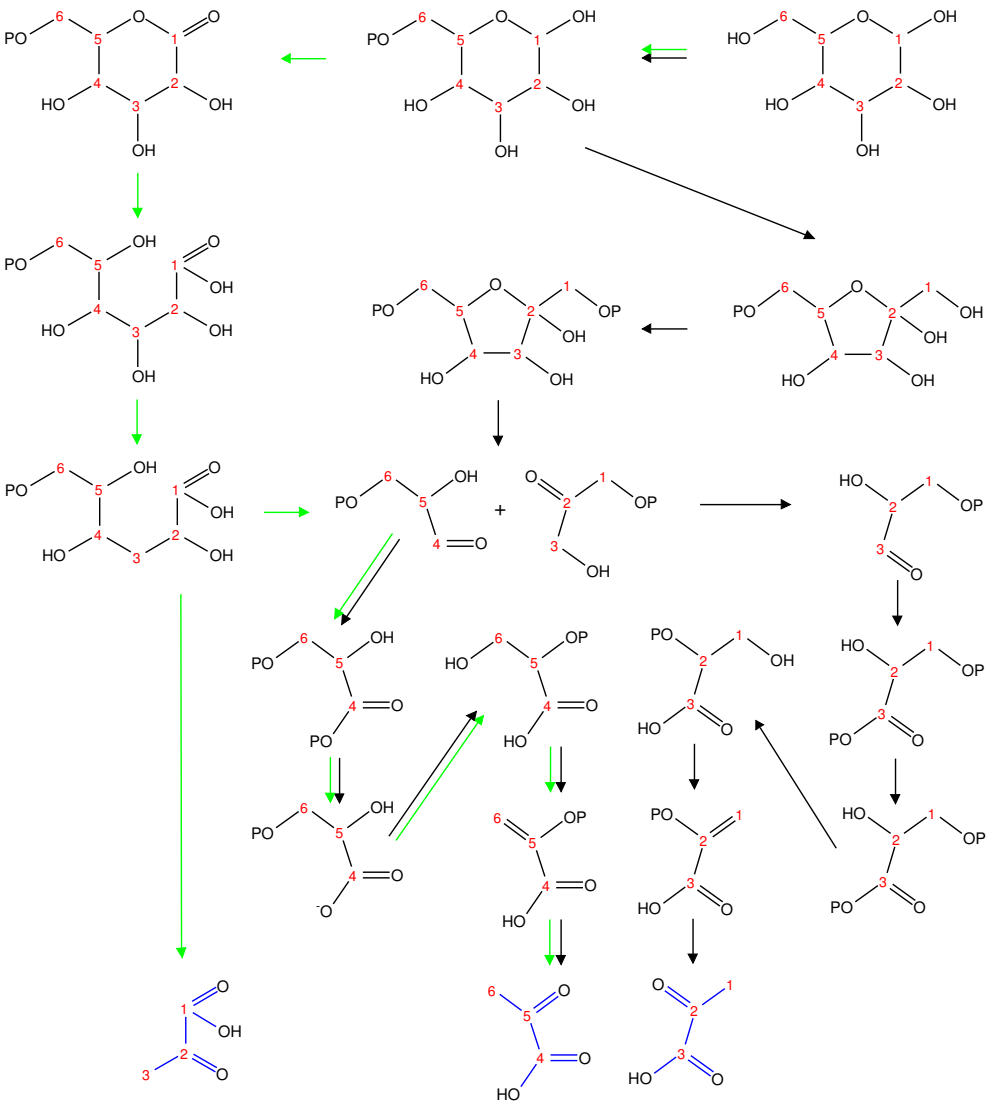


Figure 13.4: Carbon trace of two glycolysis pathways. The Embden-Meyerhof-Parnas pathway (EMP) is depicted with black reaction arrows, and the Entner-Doudoroff pathway (ED) is depicted with green reaction arrows. The six carbon atoms from glucose are converted into two pyruvate molecules, highlighted in blue, in two different ways depending on whether EMP or ED was used to catabolise glucose. The trace for one pyruvate overlaps in the pathways, while the sequence of carbons is inverted in the other pyruvate.

Abbr.	Name	Description	Example Reaction
AL	Aldolase	A generic aldol addition.	G3P + DHAP ⟶ FBP
AlKe	Aldose-Ketose	Aldehyde to ketone conversion.	R5P ⟶ Ru5P
KeAl	Ketose-Aldose	Ketone to aldehyde conversion.	Ru5P ⟶ R5P
PHL	Phosphohydrolase	Use water to cleave off phosphate.	FBP + H_2O ⟶ F6P + P_i
PK	Phosphoketolase	Break C-C-bond and add phosphate.	F6P + P_i ⟶ AcP + E4P
TAL	Transaldolase	Move C3-end.	F6P + E4P ⟶ G3P + S7P
TKL	Transketolase	Move C2-end.	G3P + S7P ⟶ X5P + R5P

Table 13.5: List of generic transformation rules for modelling the non-oxidative glycolysis chemistry.

we illustrate that many more equivalent solutions can be found automatically. We use a generic model of the chemistry in order to explore related reactions for which concrete enzymes may not yet exist.

Most of the molecules in the chemistry are sugar phosphates, where we use the naming scheme C$\langle N \rangle$P for a sugar with $\langle N \rangle$ carbon atoms and a single phosphate group. The bisphosphates, sugars with two phosphate groups, are abbreviated as C$\langle N \rangle$P2. However, for the molecules with a well-known name we use the following abbreviations.

P_i**:** Phosphate

AcP**:** Acetyl phosphate

G3P**:** Glyceraldehyde 3-phosphate

DHAP**:** Dihydroxyacetone phosphate

E4P : Erythrose 4-phosphate

Ru5P**:** Ribulose 5-phosphate

X5P**:** Xylulose 5-phosphate

R5P**:** Ribose 5-phosphate

G6P**:** Glucose 6-phosphate

F6P**:** Fructose 6-phosphate

FBP**:** Fructose 1,6-bisphosphate

S7P**:** Sedoheptulose 7-phosphate

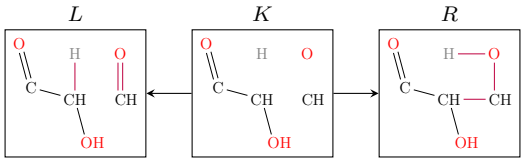
Modelling

The molecules are encoded as graphs in the straight-forward manner, though without stereochemical information. This implies that certain classes of molecules are represented as a single molecule, e.g., Ru5P and X5P.

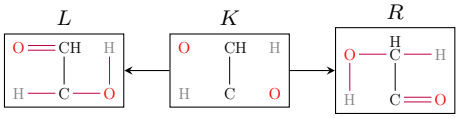
We have modelled the generic transformation rules listed in Table 13.5, and shown in Figure 13.6. In [Bogorad *et al.* 2013] the use of phosphoketolase is associated with specific names for the specific reactions:

- XPK for X5P + P_i ⟶ AcP + G3P
- XPK for F6P + P_i ⟶ AcP + E4P

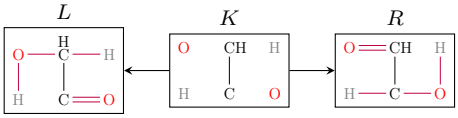
We extend the naming scheme to cover educts with 7 and 8 carbons:



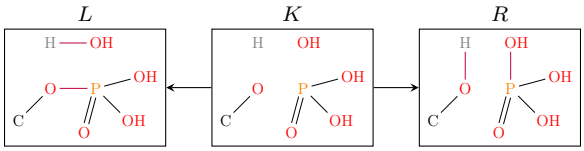
(a) Al, Aldolase



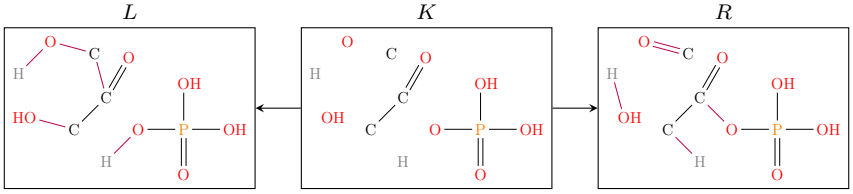
(b) AlKe, Aldose-Ketose



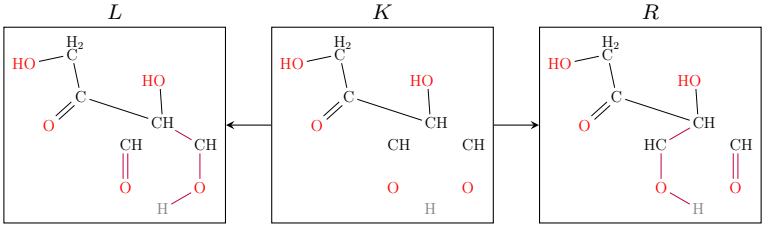
(c) KeAl, Ketose-Aldose



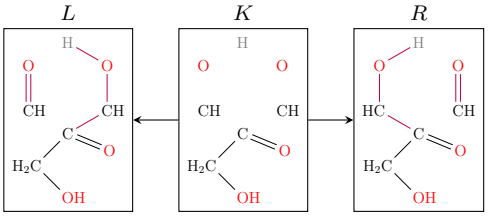
(d) PHL, Phoshohydrolase



(e) PK, Phosphoketolase



(f) TAL, Transaldolase



(g) TKL, Transketolase

Figure 13.6: Transformation rules for modelling the non-oxidative glycolysis chemistry.

- SPK for $\mathrm{S7P} + \mathrm{P_i} \longrightarrow \mathrm{AcP} + \mathrm{X5P}$
- OPK for $\mathrm{C_8P} + \mathrm{P_i} \longrightarrow \mathrm{AcP} + \mathrm{G6P}$

Network

To create the reaction network we use the starting molecules $\mathrm{P_i}$, AcP, G3P, DHAP, E4P, R5P, Ru5P, F6P, S7P, and FBP. The network is created by iterating the application of all the transformation rules listed in Table 13.5, until no new molecules are discovered. This chemistry is theoretically infinite, so we impose the restriction that no molecule with more than 8 carbon atoms may be created. We can express this precisely with the following exploration strategy.

$$\begin{aligned} Q_{\mathrm{NOG}} = {} & \texttt{addSubset}[\{\mathrm{P_i}, \mathrm{AcP}, \mathrm{G3P}, \mathrm{DHAP}, \mathrm{E4P}\}] \\ & \to \texttt{addSubset}[\{\mathrm{R5P}, \mathrm{Ru5P}, \mathrm{F6P}, \mathrm{S7P}, \mathrm{FBP}\}] \\ & \to \texttt{rightPredicate}[P_{\#\mathrm{C}}, \\ & \qquad \texttt{repeat}[\texttt{parallel}[\{\mathrm{Al}, \mathrm{AlKe}, \mathrm{KeAl}, \mathrm{PHL}, \mathrm{PK}, \mathrm{TAL}, \mathrm{TKL}\}]] \\ & \quad] \end{aligned}$$

$$P_{\#\mathrm{C}}(G \overset{p}{\Rightarrow} H) \equiv \forall h \in H : h \text{ has at most 8 carbon atoms}$$

The execution of the strategy results in a network with 81 molecules and 414 reactions.

Pathways

For the overall reaction $\mathrm{F6P} + 2\,\mathrm{P_i} \longrightarrow 3\,\mathrm{AcP} + 2\,\mathrm{H_2O}$ we have enumerated all solutions using at most 8 unique reactions and with at most 11 reactions happening in total. To further constrain the chemistry we excluded all sugars without phosphate groups, and disabled the reactions generated with PK and AL that acts only on C2 molecules. In total we find 263 different pathways, which were computed within a few minutes on a normal desktop computer. In Table 13.7 we categorise the pathways according to the properties

- number of unique reactions used,
- number of reactions used (i.e., sum of flow on internal hyperedges),
- whether the only bisphosphate used is FBP,
- the histogram of different PK reactions (see the modelling above).

The table shows the number of solutions in each combination. Interestingly it turns out that the solution space where FBP is the only bisphosphate used is quite similar to the space where other bisphosphates are allowed. The solutions are distributed the same except for a 1-shift in the number of reactions.

	Only FBP				Other Bisphosphates								
	8 Unique Reactions				7 Unique Reactions					8 Unique Reactions			
PK Type	Reactions				Reactions					Reactions			
X, F, S, O	8	9	10	11	7	8	9	10	11	8	9	10	11
0, 0, 0, 3	–	–	–	–	–	–	–	–	–	–	–	4	16
0, 0, 1, 2	–	–	–	–	–	–	–	–	–	–	3	2	–
0, 0, 2, 1	–	–	–	–	–	–	–	–	–	–	4	–	–
0, 0, 3, 0	–	–	1	2	–	–	1	2	–	–	–	9	20
0, 1, 0, 2	–	–	–	–	–	–	–	–	–	–	4	4	–
0, 1, 1, 1	–	–	–	–	–	–	–	–	–	3	–	–	–
0, 1, 2, 0	–	1	–	–	–	1	–	–	–	–	8	2	–
0, 2, 0, 1	–	–	–	–	–	–	–	–	–	–	6	–	–
0, 2, 1, 0	–	1	–	–	–	1	–	–	–	–	9	–	–
0, 3, 0, 0	–	–	2	4_a	–	–	2	4	–	–	–	14	24
1, 0, 0, 2	–	–	–	–	–	–	–	–	–	–	2	4	–
1, 0, 1, 1	–	–	–	–	–	–	–	–	–	1	–	–	–
1, 0, 2, 0	–	1	–	–	–	1	–	–	–	–	6	2	–
1, 1, 0, 1	–	–	–	–	–	–	–	–	–	2	–	–	–
1, 1, 1, 0	1	–	–	–	1	–	–	–	–	3	–	–	–
1, 2, 0, 0	–	2	–	–	–	2	–	–	–	–	10	–	–
2, 0, 0, 1	–	–	–	–	–	–	–	–	–	–	4	–	–
2, 0, 1, 0	–	1	–	–	–	1	–	–	–	–	7	–	–
2, 1, 0, 0	–	2_c	–	–	–	2	–	–	–	–	10	–	–
3, 0, 0, 0	–	–	2_b	4	–	–	2	4	–	–	–	12	20

Table 13.7: Overview of number of NOG pathways, listed by the histogram of the number of phosphoketolase reactions used. The numbers are further categorised by whether FBP is the only bisphosphate used or not, and by the number of reactions and unique reactions used. Categories marked with subscript $_a$, $_b$, and $_c$ refer to [Bogorad *et al.* 2013, Figure 2], and we see that not only are there alternate solutions in the exact same categories, but in the case of $_a$ we even find a shorter pathway with the same properties. Note that the left block is similar to the middle block of categories, but with 1 less flow. This is due to a replacement of part of the pathway with a shorter pathway, see Table 13.8. The pathway shown in Figure 13.9 is from the framed blue category, and the pathway in Figure 13.10 is from the framed green category. Their counterparts with the replacement from Table 13.8 are the unframed shaded numbers.

$FBP + H_2O$	$\xrightarrow{PHL}$	$P_i + F6P$
$E4P + F6P$	$\xrightarrow{TAL}$	$G3P + S7P$
$G3P + DHAP$	$\xrightarrow{AL}$	FBP
$DHAP + E4P + H_2O$	$\rightarrow$	$S7P + P_i$

(a)

$DHAP + E4P$	$\xrightarrow{AL}$	C_7P_2
$H_2O + C_7P_2$	$\xrightarrow{PHL}$	$S7P + P_i$
$DHAP + E4P + H_2O$	$\rightarrow$	$S7P + P_i$

(b)

Table 13.8: Two pathways with the same overall reaction. (a) a 3-reaction pathway using FBP as bisphosphate. This subpathway is highlighted in Figure 13.9. (b) a 2-reaction pathway using a bisphosphate with 7 carbons. This subpathway is highlighted in Figure 13.10.

By subsequent analysis we found a 1-to-1 mapping between these two solution sets such that the only difference in the pathway is the subpathways described in Table 13.8.

In Figure 13.9 one of the solutions are illustrated in detail. This solution has similar properties to the solution shown in [Bogorad *et al.* 2013, Figure 2a]: the phosphoketolase reactions all has F6P as educt, and the only bisphosphate used is FBP. However, this solution can be regarded as being shorter than [Bogorad *et al.* 2013, Figure 2a] as it uses fewer reactions, though the number of unique reactions is the same. Allowing other bisphosphates than FBP to be used enables even shorter solutions to be found. Figure 13.10 shows the shortest solution, which uses a bisphosphate with 7 carbon atoms. Its use of phosphoketolase is also different, as it uses both XPK, FPK, and SPK.

Our investigation nicely illustrates how vast the design space for a non-oxidative pathway is. Even if the reaction chemistry was restricted to only a handful of enzyme functionalities, systematic exploration of the space without computational approaches is inefficient and many interesting solutions may be missed.

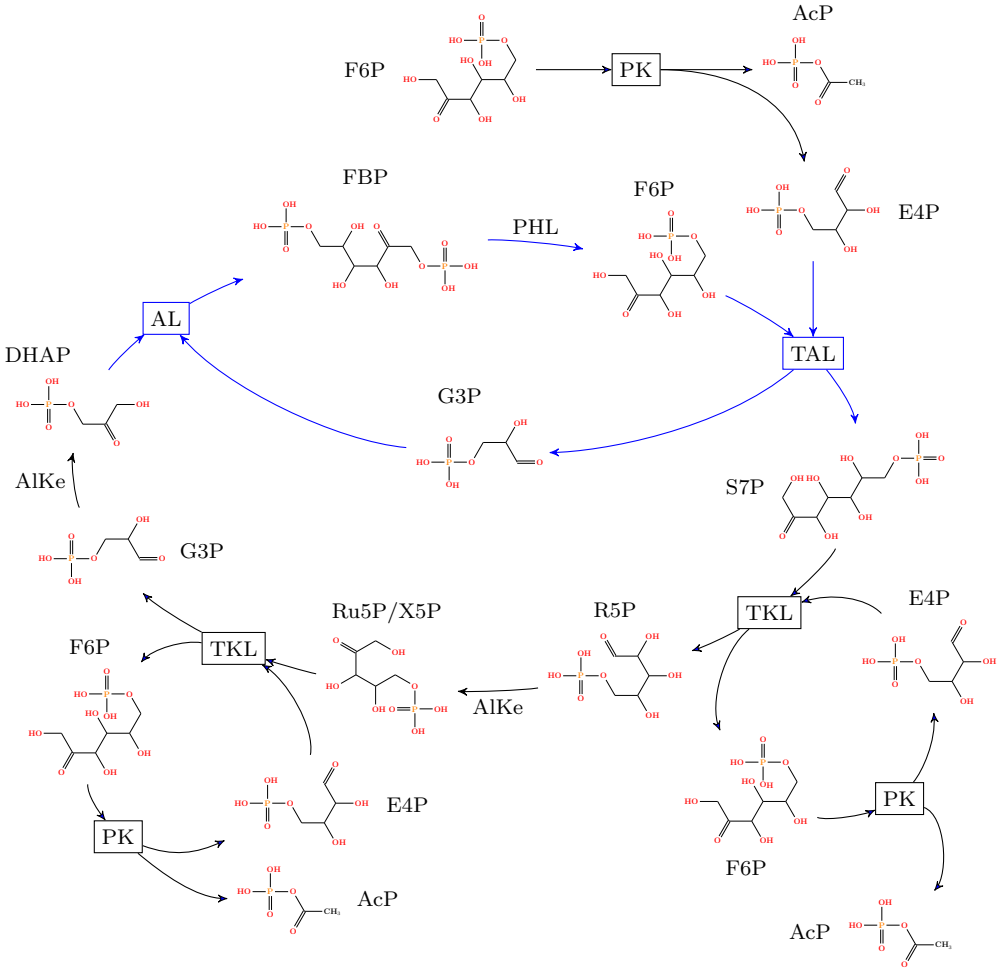


Figure 13.9: Illustration of a pathway similar to the one depicted in [Bogorad *et al.* 2013, Figure 2a], but using fewer reactions. This solution category is the framed blue cell of Table 13.7. The highlighted subpathway is the pathway from Table 13.8a.

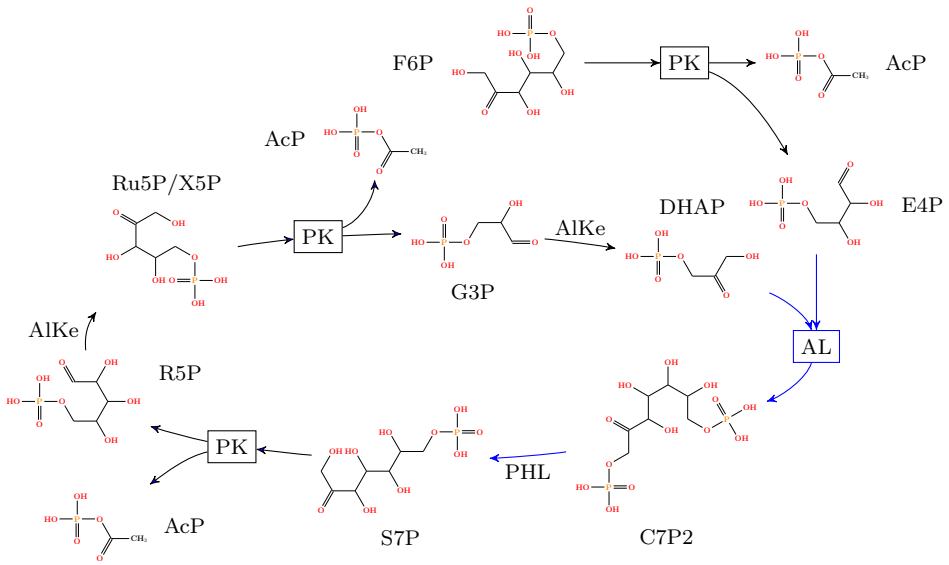


Figure 13.10: Illustration of the shortest NOG pathway, denoted by the framed green cell in Table 13.7. It uses three different phosphoketolase reactions: XPK, FPK, and SPK. The highlighted subpathway is the pathway from Table 13.8b.

Chapter 14

Solving the Catalan Game

This chapter is based on parts of [Andersen et al. 2014c].

While the topic of this work is to formally model chemistry, the presented methods can be used in non-chemical settings as well. We illustrate this fact here by modelling a small puzzle game called Catalan [increpare games 2011]. In each level the player is presented with a simple undirected graph without labels. The goal is to transform the graph into a single vertex using the following rewriting rule; given a vertex v with degree exactly 3, identify v with its neighbours and preserve simpleness of the graph by identifying parallel edges and deleting loops. Figure 14.1 shows level 1 with the intermediary graphs towards the goal graph with a single vertex.

The transformation rule in the game can not be formulated as a single rule in the DPO formalism, because such rules must explicitly match the vertices and edges that are changed, while the Catalan transformation rule needs to change arbitrarily many edges. In the following we show how the strategy framework described in Chapter 9 can be used to implement a move in the game, using only DPO rules.

Let g be the graph for a Catalan level, with all edge labels set to the empty string and all vertex labels set to the arbitrarily chosen label "`0`". For the implementation using strategies we can describe a move as:

1. Find a vertex v with at least 3 neighbours and mark it by changing the label to "`A`". Mark the 3 matched neighbours with the label "`R`".
2. If possible, find another fourth neighbour of v and mark v with "`FAIL`".
3. Discard all graphs with a vertex with the label "`FAIL`".

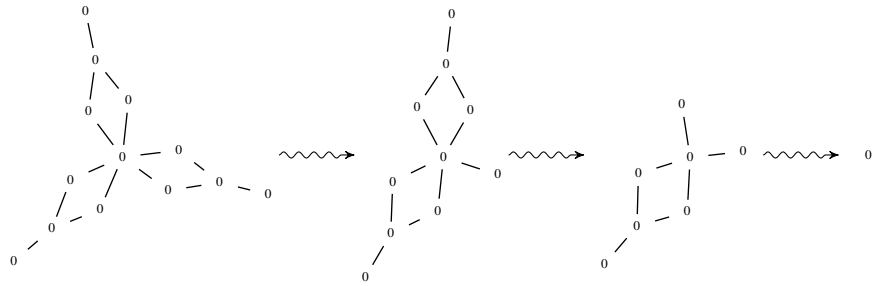


Figure 14.1: Level 1 of the Catalan game and the intermediary graphs during transformation to a graph with a single vertex.

4. For all edges e with both end-vertices having label "R", remove e.
5. For all edges (u, r) with u having label "0" and r having label "R", add (u, v) if it does not exist already and then remove (u, r).
6. For all edges (u, r) with u having label "0" and r having label "R", remove (u, r).
7. Remove all neighbours of v having label "R".
8. Unmark v by changing the label to "0".

Step 3 can be implemented with a filtering strategy while the other steps each require a transformation rule. The following strategy can be used to solve a level, in the sense that if a graph with a single vertex with label "0" is found, then a path to that graph is equivalent to a solution.

$$
\begin{aligned}
Q_{\text{catalan}} = {} & \texttt{addSubset}[\{g\}] \to \texttt{repeat}[\\
& \quad \text{mark} \to \texttt{revive}[\text{markForFail}] \to \texttt{filterUniverse}[P_{\text{fail}}] \\
& \quad \to \texttt{repeat}[\texttt{revive}[\text{removeInterR}]] \\
& \quad \to \texttt{repeat}[\texttt{revive}[\text{reattachExternal}]] \\
& \quad \to \texttt{repeat}[\texttt{revive}[\text{removeAttached}]] \\
& \quad \to \text{removeR} \to \text{unmark} \\
&] \\
P(g', F) \equiv {} & \text{no vertex of } g' \text{ has the label "FAIL"}
\end{aligned}
$$

The details of the transformation rules (mark, markForFail, removeInterR, reattachExternal, removeAttached, removeR and unmark) are shown in Figure 14.2.

With strategy Q_{catalan} all 56 levels of Catalan could be solved, all but one level took less than 10 minutes of computation time. Figure 14.3b shows the derivation graph created when executing the strategy with g encoding level 25 of the game, and Figure 14.3a shows g. In contrast to chemical reaction networks, the derivation graphs from this strategy degenerates into normal digraphs. This is simply due to the connectedness of all left- and right-hand sides of the used rules.

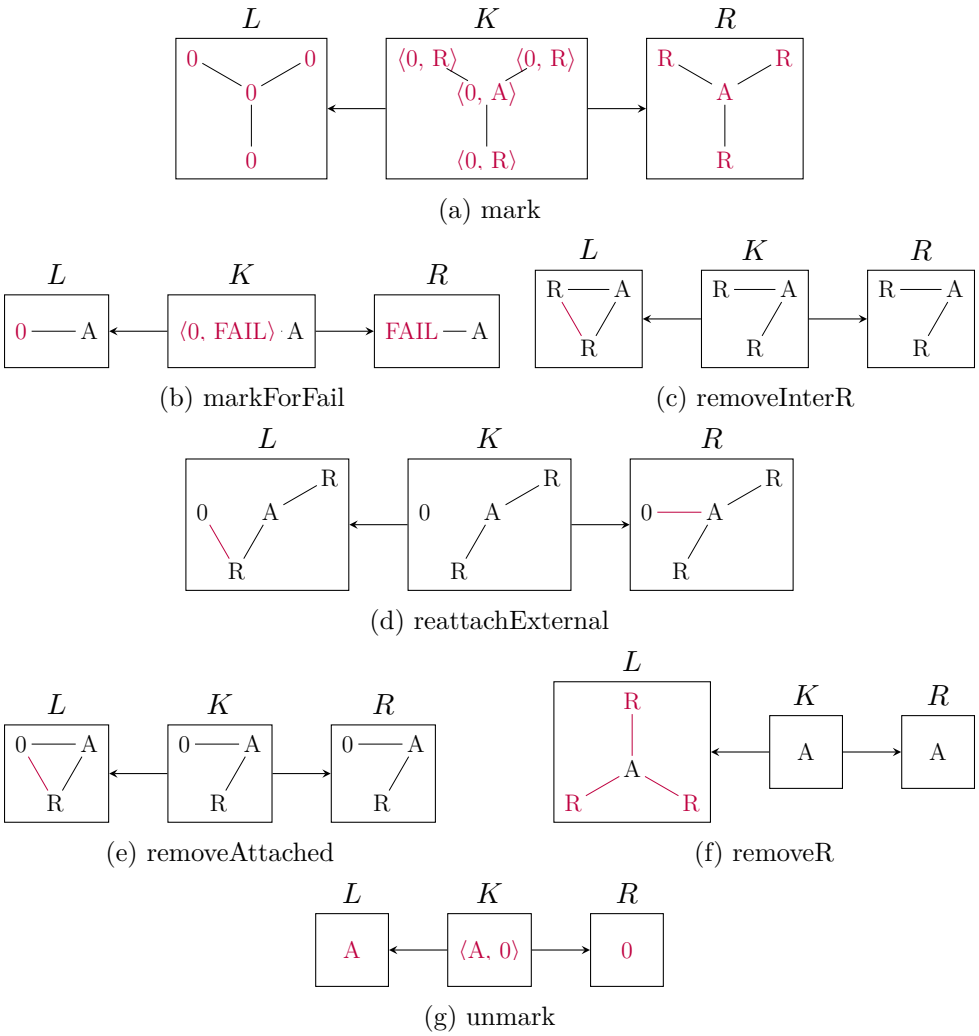
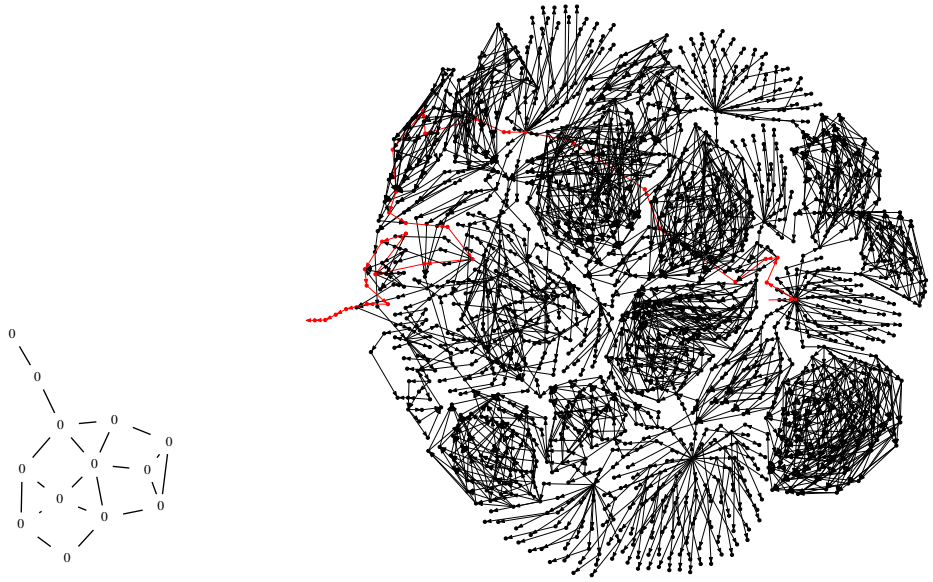


Figure 14.2: Transformation rules for solving the Catalan game.



(a) (b)

Figure 14.3: (a) Level 25 of the Catalan game. (b) The derivation graph created during execution of Q_{catalan} of level 25 of the Catalan game. Vertices are depicted as black dots, and a path equivalent to a solution is highlighted in red.

Chapter 15

Summary and Future Work

In this thesis we have presented a wide range of algorithms and modelling techniques for generative chemistry. Among them are:

- A molecule model where chemical properties are encoded locally, and an encoding scheme for representing molecules as graphs labelled with character strings or first-order terms.
- A model for generic reaction patterns using an adapted version of the Double Pushout approach for graph rewriting.
- Definitions of rule composition, based on concurrency theory, that are relevant for working with chemical reactions. These definitions forms the basis of the core algorithms for generating molecules.
- A strategy framework for automatic generation of chemical reaction networks, which can also be used for graph grammar computation with connected graphs.
- A coherent formal model for chemical pathways in reaction networks, that enables detailed reasoning of the flow of molecules.
- A simplistic model for catalytic and autocatalytic pathways.
- An implementation of the pathway model using integer linear programming, with a tree search algorithm for enumerating both optimal and near-optimal solutions.

The methods have been used to define an algorithm for calculating atom traces for reaction sequences. We have also used the methods to analyse models of the formose chemistry, several versions of the glycolysis pathway, and the enzyme catalysed β-lactamase reaction. In addition we have used the strategy framework in [Andersen *et al.* 2014a] to implement a virtual machine for DNA-templated computing, and in [Andersen *et al.* 2013a] to generate a reaction network for hydrogen cyanide biased towards mass spectrometry results from a wet lab experiments. Furthermore, in [Andersen *et al.* 2015b] we have used the methods to create an *in silico* model of Eschemoser's glyoxylate scenario [Eschenmoser 2007b], which provides a hypothesis for the prebiotic creation of core metabolites from hydrogen cyanide, using a series of autocatalytic cycles.

In Appendix A we provide a brief overview of the software package we have developed, which implements the presented models and algorithms. This software has been used to perform the practical applications, and it is currently in preparation for release as an open source project.

During the development of the methods we have found many possibilities for future research, of which preliminary results have been obtained for several of them. In the following sections we briefly outline a few of the possibilities.

15.1 Modelling of Stereochemistry

The molecule model described in Section 2.1 defines molecules in a very broad sense, in that it does not formally constrain the amount of bonds an atom can participate in. Realistically only a finite set of neighbourhoods are chemically valid, as outlined by Table 2.1, which is related to molecular geometry and stereochemistry. For example the Valence-Shell Electron-Pair Repulsion (VSEPR) theory [Gillespie 1963] can be used as a basis for enumerating valid geometries, and thereby refine the molecule model. An incorporation of local geometry will additionally enable precise modelling of several types of stereoisomerism, e.g., chirality and cis-trans isomerism.

The extension of the graph model can possible be done with a variant of what is called the *ordered list* method [Petrarca *et al.* 1967, Wipke & Dyott 1974], which we briefly outline in the following. When using adjacency lists to represent the molecule graphs then the collection of incident edges to a vertex is conceptually just a set of edges. In the ordered list method these collections are represented as ordered lists, where the order of the edges represents a configuration in a local geometry. How a configuration is encoded in the list may thus be different for different geometries.

As an example consider the tetrahedral geometry, that carbon atoms can exhibit. We can decide that the first edge in the list points "up", and the last three edges are ordered counter-clockwise when seen from "above". This is illustrated in Figure 15.1 where all 24 permutations are partitioned according to which configuration they encode. Note that the lists that are equivalent can be characterised by the permutation group $G_{\equiv} = \langle (1)(2\ 3\ 4), (1\ 2)(3\ 4) \rangle$.

When deciding isomorphism between molecules we must take into account that each of the two configurations can be represented by 12 different permutations of the incident edges. Given a candidate morphism, which is an isomorphism in the underlying labelled graph, we can create the permutation induced by the morphism as illustrated in Figure 15.2. This permutation can be checked for membership in $G_{\equiv}$, the permutation group of equivalence. In the shown example the permutation is not a member, and the candidate morphism is invalid.

Preliminary investigations show that similar encoding schemes can be used for the geometries common in biomolecules. One of the challenges is that the

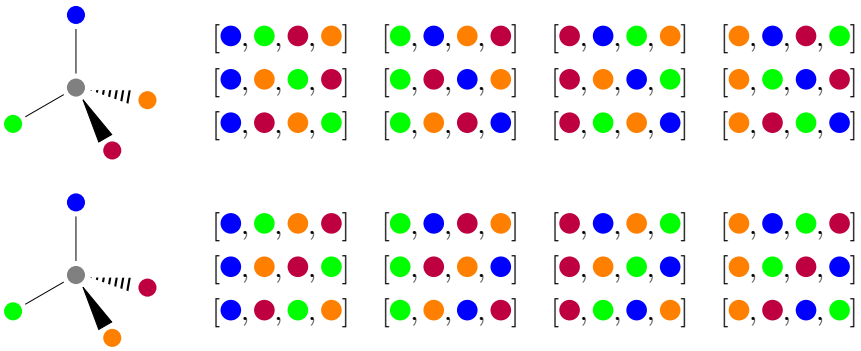


Figure 15.1: Encoding of configurations in the tetrahedral geometry using the ordered list method. Each of the two configurations can be encoded with 12 different permutations of the incident edges.

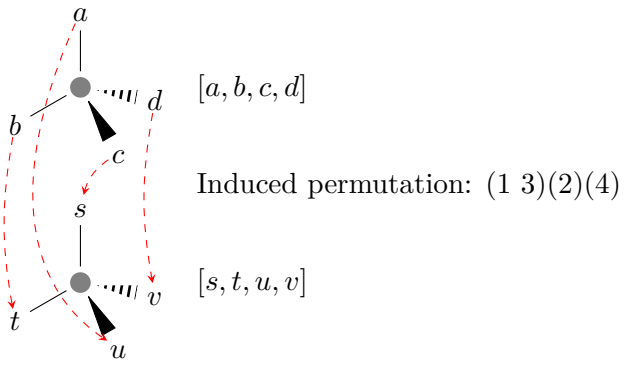


Figure 15.2: Example of isomorphism testing for the tetrahedral geometry, given a morphism which is already an isomorphism in the underlying labelled graph. The given morphism, shown with red arrows, induce a permutation of the neighbours (1 3)(2)(4). This permutation is not in the permutation group for equivalence $G_{\equiv}$, so the candidate morphism is not an isomorphism with respect to stereoisomerism.

stereochemical properties must arise from local data, such that the Double Pushout formalism can be extended to stereochemistry. Another challenge is to enable partial specification of configurations, in a natural manner that works well in a generative setting. On top of the modelling challenges we also need to define how to extend the algorithms for isomorphism detection, monomorphism enumeration, and graph canonicalisation.

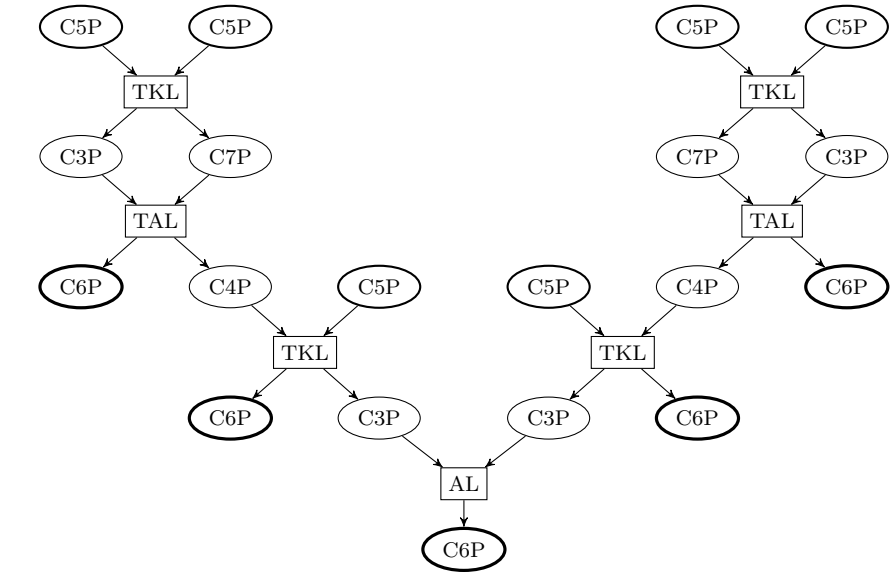
The modelling of local geometry will also lay the foundation for describing the geometry of transition states in reactions, and thereby include adjacency constraints in a formal definition of chemically valid transformation rules.

15.2 Realisable Pathways and Atom Tracing

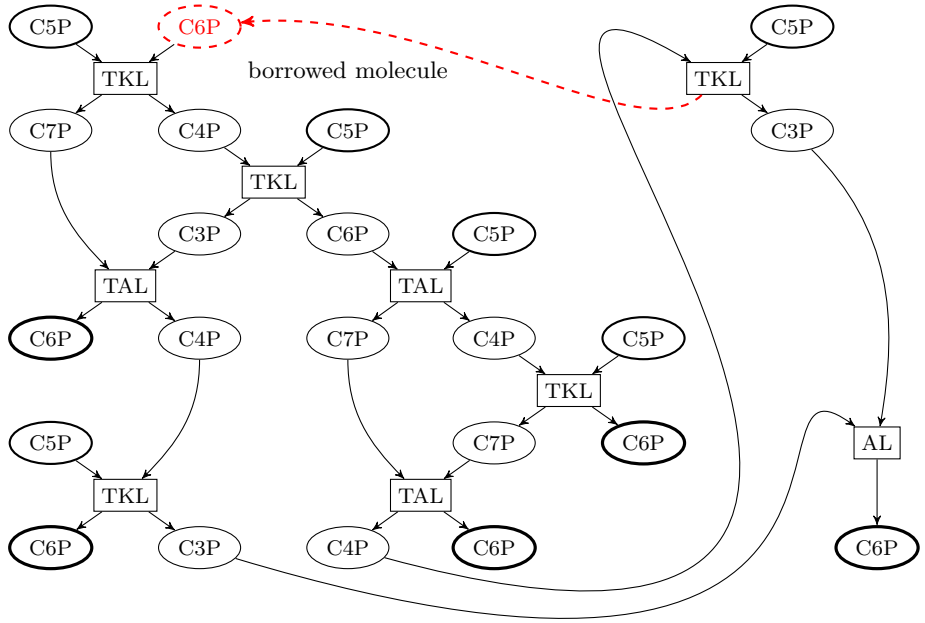
The pathway model described in Chapter 10 is based on integer hyperflows, which is closely related to what is known as *transition invariants* (T-invariants) in Petri net theory, see [Desel 1998] and [Petri 1962]. We can reinterpret an extended hypergraph $\overline{\mathcal{H}} = (V, \overline{E})$ as a Petri net with V as the *places* and $\overline{E}$ as the *transitions*. A flow f on $\overline{\mathcal{H}}$ is fact then exactly equivalent to a T-invariant (cmp. Equation (10.2) and [Desel 1998, Proposition 7]).

Petri net theory has previously been used to analyse reaction networks [Chaouiya 2007, Koch 2010, Koch *et al.* 2010]. Tools such as Pathway Logic [Eker *et al.* 2002, Talcott & Dill 2005] have been developed specifically for working with biological networks and Petri nets. Pathway Logic notably also uses rewriting techniques, though on a higher level of abstraction than the atomic level we have presented in this work. The Petri net view gives rise to a stricter definition of "a pathway" than integer hyperflows provides. Consider a flow f on an extended hypergraph/Petri net $\overline{\mathcal{H}} = (V, \overline{E})$. We can then create an initial marking $m_I \colon V \to \mathbb{N}_0$ representing the input flow: $m_I(v) = f(e_v^-), \forall v \in V$. Similarly we can create a target marking $m_O \colon V \to \mathbb{N}_0$ with $m_O(v) = f(e_v^+), \forall v \in V$, representing the output flow. We can then say that f is a *realisable* pathway if and only if the marking m_O can be reached from m_I using a sequence of transition firings that in total correspond to the flow f. For pathways that are not realisable an interesting question is then how many copies of molecules we need to borrow to make the pathway realisable [Rasmussen 2012]. An example of borrowing is shown in Figure 15.3 with two pathways for the non-oxidative pentose phosphate pathway. Another example was shown already in Figure 12.5 with overall autocatalytic pathways in the formose chemistry, where we informally argued that a "can initialise"-relation can be defined between pathways when one pathway can borrow molecules from another pathway to make itself realisable. This example was in fact found automatically using an experimental implementation for post-processing flow solutions. The post-processing constructs specific reachability problems in Petri nets, and uses the Low Level Petri Net Analyzer (LoLA) [Schmidt 2000] for solving the problems.

In Chapter 11 we defined a basic method for atom tracing that used composition of transformation rules to compute the overall traces. However, this method "blindly" applies the rule sequence to all combinations of molecules that matches, and the sequence of rules must be given in advance. An interesting prospect is to automatically infer the rule sequences by enumerating realisable pathways, and control the rule application to the specific molecules determined by the pathway. With this future approach will thus be possible to automatically enumerate all possible atom traces for all possible pathways conforming to a given specification, in a network automatically generated from a given exploration strategy utilising graph grammars.



(a)



(b)

Figure 15.3: Two pathways implementing the non-oxidative pentose phosphate pathway, with the overall reaction 6 C5P ⟶ 5 C6P. Both pathways are shown with duplication of molecules and reactions, such that each of them represent a single occurrence in the pathway. (a) The "classical" pathway [Meléndez-Hevia & Isidoro 1985], which is realisable. (b) An non-realisable alternative pathway, which is realisable if a copy of C6P is borrowed.

Figure 15.4: Suggestion for structural flow motifs for (a) catalytic pathways and (b) autocatalytic pathways.

15.3 Structural Pathway Constraints

In Chapter 10 we defined an elaborate pathway model that allows for a basic definition of overall (auto)catalytic pathways. However, it is clear that the notion of general (auto)catalysis entails a causal relationship between the input of a compound and its subsequent production. The expansion of reaction networks to enable local routing constraints was introduced, to some extend, for disallowing flows without such relationships.

The challenge is to define, more precisely, which topological constraints are necessary for a hyperflow to (auto)catalytic. A suggestion is to introduce non-local constraints on flows, e.g., requiring that a catalytic flow induces a cycle in the underlying directed graph as depicted in Figure 15.4a. An autocatalytic pathway could then be a catalytic pathway with an additional "ear" of flow for the production of the autocatalytic compound, Figure 15.4b. Such constraints would certainly ensure the causal relationship, but it still leaves the question of whether the concepts of (auto)catalysis entails further topological constraints.

The use of structural constraints may lead to an approach to automatically enumerate chemical hypercycles [Eigen 1971, Eigen & Schuster 1977], Such a hypercycle is, when restricted to a topological definition, a higher-order cycle in the network consisting of autocatalytic cycles connected by catalysis, as illustrated in Figure 15.5. Special care must however be taken to ensure a precise characterisation, to avoid further confusion about the concept [Szathmáry 1988, Szathmáry 2013].

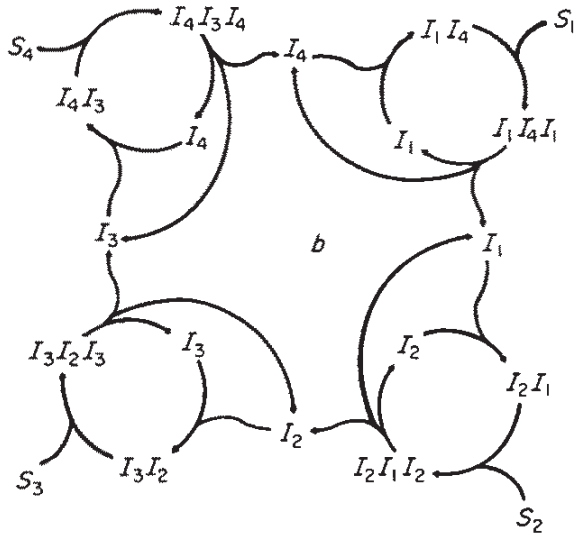


Figure 15.5: [Szathmáry 1988, Fig. 2], schematic of a chemical hypercycle.

Appendices

Appendix A

MedØlDatschgerl

As of March 2016, part of this package has been publicly released as MedØlDatschgerl 0.5 [Andersen 2016, Andersen et al. 2016].

The methods described this work are implemented in a software package called MedØlDatschgerl (MØD), which additionally contains extensive support for automatic generation of figures. MØD consists of the following parts:

libMØD The main library, implemented in C++11.

PyMØD A Python 3 module with bindings to the C++ layer, and additional convenience functionality.

PostMØD A shell script for compiling figures specified by the C++ library.

`mod` A shell script for invocation of a Python interpreter with PyMØD loaded, and subsequent invocation of the post-processor.

`mod.sty` A LaTeX package for direct inclusion of depictions of molecules and transformation rules in documents.

A limited version of MØD can be invoked at `http://cheminf.imada.sdu.dk/mod/playground.html`, where the Python interface is available and automatically generated figures can be downloaded. In the following we provide a very brief overview of the features of MØD.

Graphs and Rules

Graphs for modelling molecules and transformation rules for modelling reaction patterns are available as classes in the library. Molecules can for example be constructed from SMILES strings (with a special interpretation). For example, a graph representing caffeine can be loaded in the Python interface with '`caffeine = smiles("Cn1cnc2c1c(=O)n(c(=O)n2C)C")`'. Transformation rules can loaded with code similar to '`p = ruleGML("ruleFile.gml")`', where '`ruleFile.gml`' is a text file with a specification of the rule. Both graph and rule objects have methods for counting monomorphisms and isomorphisms.

⟨*rcExp*⟩ :: ⟨*rcExp*⟩ '`*`' ⟨*op*⟩ '`*`' ⟨*rcExp*⟩
| '`rcBind(`' ⟨*graphs*⟩ '`)`'
| '`rcUnbind(`' ⟨*graphs*⟩ '`)`'
| '`rcId(`' ⟨*graphs*⟩ '`)`'
| ⟨*rules*⟩

Figure A.1: Grammar for rule composition expressions in PyMØD, where ⟨*graphs*⟩ is a normal Python expression returning either a single graph or a collection of graphs. Similarly is ⟨*rules*⟩ a Python expression returning either a single rule or a collection of rules. See Table A.2 for the description of ⟨*op*⟩. The three functions '`rcBind`', '`rcUnbind`', and '`rcId`' refers to the construction of rules from graphs, described in Section 7.2.

Math Operator	⟨*op*⟩
$\bullet_{\emptyset}$	`*rcParallel*`
$\bullet_{\supseteq}$	`*rcSuper(allowPartial=False)*`
$\bullet^{c}_{\supseteq}$	`*rcSuper*`
$\bullet_{\subseteq}$	`*rcSub(allowPartial=False)*`
$\bullet^{c}_{\subseteq}$	`*rcSub*`
$\bullet_{\cap}$	`*rcCommon*`

Table A.2: List of rule composition operators in PyMØD, usable as ⟨*op*⟩ in the grammar in Figure A.1.

Interface for Rule Composition

In Chapter 7 we defined a series of specialised forms of rule composition, indicated by specific operators, e.g., $\bullet_{\supseteq}$ for full composition and $\bullet_{\cap}$ for the most general type of composition. In the Python interface these operators are directly available and can be used to build expression trees following the grammar in Figure A.1. Table A.2 lists the Python notation for each rule composition operator. They are implemented using overloading on the build-in multiplication operator to obtain infix notation for the expressions.

A rule composition expression can be passed to an evaluator which will carry out the composition and discard duplicate results, as determined by isomorphism between rules. The result of each ⟨*rcExp*⟩ is coerced into a list of rules, and the operators consider all selections of rules from their arguments. That is, if '`P1`' and '`P2`' are two lists of rules, then the evaluation of '`P1 *rcSuper* P2`' results in the list

$$\bigcup_{p_1 \in \mathtt{P1}} \bigcup_{p_2 \in \mathtt{P2}} p_1 \bullet^{c}_{\supseteq} p_2$$

Interface for Exploration Strategies

The exploration strategies for network generation presented in Chapter 9 are implemented in the Python interface, such that strategies can be specified in a similar syntax as the mathematical one. For example, the breadth-first expansion, limited by molecule size, in the formose chemistry (see Chapter 12) can be done with the following code.

```
strat = (
    addUniverse(formaldehyde)
    >> addSubset(glycolaldehyde)
    >> rightPredicate[
        lambda d: all(a.vLabelCount("C") <= 6 for a in d.right)
    ](
        repeat([p1, p2, p3, p4])
    )
)
dg = dgRuleComp([formaldehyde, glycolaldehyde], strat)
dg.calc()
```

The variables '`formaldehyde`' and '`glycolaldehyde`' are graphs representing the starting molecules and '`p1`', '`p2`', '`p3`', and '`p4`' are variables for the transformation rules. Strategies are created as expression trees that are then given to an evaluator which executes the strategy, performs graph isomorphism testing, and creates the derivation graphs representing reaction networks. The derivation graph object, here called '`dg`', can then be queried (e.g., for conversion to external formats) or used for pathway calculations.

Interface for Hyperflow Calculation

The hyperflow model in Chapter 10 is realised using the IBM ILOG CPLEX Optimization Studio for defining and solving integer linear programming models. A model can be specified directly from a given derivation graph object, where extension and expansion is done automatically. Predefine modules for overall catalysis and autocatalysis can be enabled, and using predefined variables one can add a custom linear objective function custom linear constraints. For example, enumeration of the 10 overall autocatalytic solutions using the least amount of unique reactions can be done with the following code.

```
flow = dgFlow(dg)
flow.addSource(formaldehyde)
flow.addSource(glycolaldehyde)
flow.addSink(glycolaldehyde)
flow.overallAutocatalysis.enable()
flow.objectiveFunction = isEdgeUsed
flow.setSolverEnumerateBy(
    enumerationVarSpecifier=isEdgeUsed,
    maxNumSolutions=10
)
flow.calc()
```

Here we additionally specify that solutions are considered equivalent if they use the same set of reactions ('`enumerationVarSpecifier=isEdgeUsed`').

Figure Generation

The software can automatically generate many types of visualisations, e.g., for graphs, rules, derivation graphs, and flow solutions, which all somehow involve the depiction of graphs. The coordinates for graph layouts are generated using Graphviz [Gansner & North 2000] for general graphs and Open Babel [O'Boyle *et al.* 2011] for molecules and transformation rules involving molecules. The final rendering is done using the LaTeX package Tikz [Tantau 2013]. The general procedure for figure generation is:

- In libMØD: Generate a specification of the graph in Tikz format.
- In libMØD: If the graph is a molecule or a transformation rule involving molecules, then generate coordinates for the vertices with Open Babel. Otherwise generate a specification of the graph in Graphviz format.
- In libMØD: Output commands to PostMØD for final figure generation.
- In PostMØD: If it is a general graph, then generate coordinates using the Graphviz specification.
- In PostMØD: Use LaTeX to compile the Tikz specification, using the generated coordinates, into a final PDF file for the figure.

A graph, rule, derivation graph, or flow solution represented by a variable '`a`' can in this manner be visualised by executing '`a.print()`' in the Python interface.

Including Figures in LaTeX

During the writing of this thesis a LaTeX package was developed to make visualisations of molecules and transformation rules more easily available in documents. For example, the depictions in Figure 2.2 are specified with the following LaTeX code.

```
\modset{smiles/.style={none, edges as bonds, raise charges, with colour}}
\smiles{Cn1cnc2c1c(=O)n(c(=O)n2C)C}
\smiles[simple carbons]{Cn1cnc2c1c(=O)n(c(=O)n2C)C}
\smiles[collapse hydrogens]{Cn1cnc2c1c(=O)n(c(=O)n2C)C}
\smiles[collapse hydrogens, simple carbons]{Cn1cnc2c1c(=O)n(c(=O)n2C)C}
```

Each '`\smiles`' macro expands into an '`\includegraphics`' for a specific PDF file, and a PyMØD script is generated which can be executed to compile the needed files. Transformation rules can similarly be visualised, e.g.,

```
\ruleGML{ruleFile.gml}{\dpoRule}
```

which expands into '`\dpoRule{fileL.pdf}{fileK.pdf}{fileR.pdf}`', where the three PDF files depict the left side, context, and right side of the rule. The '`\dpoRule`' macro then constructs the final rule diagram with the PDF filesincluded.

Bibliography

[Ahuja *et al.* 1993] R. K. Ahuja, T. L. Magnanti and J.B. Orlin. Network flows: Theory, algorithms, and applications. Prentice Hall, Englewood Cliffs, NJ, 1993. (Cited on pages 95, 100, 105, and 116)

[Aït-Kaci 1991] Hassan Aït-Kaci. Warren's abstract machine: A tutorial reconstruction. MIT Press, Cambridge, MA, USA, 1991. (Cited on page 6)

[Altman *et al.* 2013] T. Altman, M. Travers, A. Kothari, R. Caspi and P. D. Karp. *A systematic comparison of the MetaCyc and KEGG pathway databases*. BMC Bioinformatics, vol. 14, page 112, 2013. (Cited on page 79)

[Andersen *et al.* 2012] Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Maximizing output and recognizing autocatalysis in chemical reaction networks is NP-complete*. Journal of Systems Chemistry, vol. 3, no. 1, 2012. (Cited on pages viii, 108, 109, 111, and 112)

[Andersen *et al.* 2013a] Jakob L. Andersen, Tommy Andersen, Christoph Flamm, Martin M. Hanczyc, Daniel Merkle and Peter F. Stadler. *Navigating the Chemical Space of HCN Polymerization and Hydrolysis: Guiding Graph Grammars by Mass Spectrometry Data*. Entropy, vol. 15, no. 10, pages 4066–4083, 2013. (Cited on pages viii, 93, and 157)

[Andersen *et al.* 2013b] Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Inferring chemical reaction patterns using rule composition in graph grammars*. Journal of Systems Chemistry, vol. 4, no. 1, page 4, 2013. (Cited on pages viii, 63, 65, and 74)

[Andersen *et al.* 2014a] Jakob L. Andersen, Christoph Flamm, Martin M. Hanczyc and Daniel Merkle. *Towards an Optimal DNA-Templated Molecular Assembler*. In ALIFE 14: The Fourteenth Conference on the Synthesis and Simulation of Living Systems, volume 14, pages 557–564, 2014. (Cited on pages viii, 11, and 157)

[Andersen *et al.* 2014b] Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *50 Shades of Rule Composition: From Chemical Reactions to Higher Levels of Abstraction*. In François Fages and Carla Piazza, editors, Formal Methods in Macro-Biology, volume 8738 of *Lecture Notes in Computer Science*, pages 117–135, Berlin, 2014. Springer International Publishing. (Cited on pages viii, 65, 123, 137, and 141)

[Andersen *et al.* 2014c] Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Generic Strategies for Chemical Space Exploration*. International Journal of Computational Biology and Drug Design, vol. 7, no. 2/3, pages 225 – 258, 2014. (Cited on pages viii, 83, 132, and 153)

[Andersen *et al.* 2015a] Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *Chemical Transformation Motifs — Modelling Pathways as Integer Hyperflows*. 2015. In preparation. (Cited on pages viii, 95, 130, and 144)

[Andersen *et al.* 2015b] Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *In silico Support for Eschenmoser's Glyoxylate Scenario*. Israel Journal of Chemistry, 2015. (Cited on pages viii and 157)

[Andersen *et al.* 2016] Jakob L. Andersen, Christoph Flamm, Daniel Merkle and Peter F. Stadler. *A Software Package for Chemically Inspired Graph Transformation*, 2016. Submitted, TR: http://arxiv.org/abs/1603.02481. (Cited on pages viii and 167)

[Andersen 2016] Jakob L. Andersen. *MedØlDatschgerl (MØD)*. http://mod.imada.sdu.dk, 2016. (Cited on pages viii and 167)

[Andrei *et al.* 2011] O. Andrei, M. Fernández, H. Kirchner, G. Melançon, O. Namet and B. Pinaud. *PORGY: Strategy driven interactive transformation of graphs*. In In Proceedings of the 6th International Workshop on Computing with Terms and Graphs (TERMGRAPH 2011), volume 48 of *Electronic Proceedings in Theoretical Computer Science*, pages 54–68, 2011. (Cited on page 83)

[Apodaca 2007] Richard L. Apodaca. *SMILES and Aromaticity: Broken?*, 2007. http://depth-first.com/articles/2007/11/28/smiles-and-aromaticity-broken/. (Cited on page 38)

[Atanasov *et al.* 2000] B. P. Atanasov, D. Mustafi and Makinen M. W. *Protonation of the beta-lactam nitrogen is the trigger event in the catalytic action of class A beta-lactamases*. Proc. Natl Acad. Sci., vol. 97, no. 7, pages 3160–3165, 2000. (Cited on pages 124 and 126)

[Baader & Snyder 2001] F. Baader and W. Snyder. *Unification Theory*. In Alan JA Robinson and Andrei Voronkov, editors, Handbook of automated reasoning, volume 1, chapter 8, page 447–533. Elsevier, 2001. (Cited on page 5)

[Babai & Luks 1983] László Babai and Eugene M. Luks. *Canonical Labeling of Graphs*. In Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC '83, pages 171–183, New York, NY, USA, 1983. ACM. (Cited on page 23)

[Bang-Jensen & Gutin 2009] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer Monographs in Mathematics, 2009. (Cited on pages 95, 100, 105, and 116)

[Bar-Even *et al.* 2012] Arren Bar-Even, Avi Flamholz, Elad Noor and Ron Milo. *Rethinking glycolysis: on the biochemical logic of metabolic pathways*. Nature Chemical Biology, vol. 8, no. 6, pages 509–517, 2012. (Cited on page 141)

[Behre *et al.* 2012] J. Behre, L. F. de Figueiredo, S. Schuster and C. Kaleta. *Detecting structural invariants in biological reaction networks*. Methods Mol Biol., vol. 804, pages 377–407, 2012. (Cited on page 95)

[Benkö *et al.* 2003] Gil Benkö, Christoph Flamm and Peter F. Stadler. *A graph-based toy model of chemistry*. Journal of Chemical Information and Computer Sciences, vol. 43, no. 4, pages 1085–1093, 2003. (Cited on page 1)

[Benner *et al.* 2010] S. A. Benner, H.-J. Kim, M.-J. Kim and A. Ricardo. *Planetary Organic Chemistry and the Origins of Biomolecules*. Cold Spring Harbor Perspectives in Biology, vol. 2, no. 7, 2010. (Cited on pages 129, 131, 132, 133, and 137)

[Benner *et al.* 2012] Steven A. Benner, Hyo-Joong Kim and Matthew A. Carrigan. *Asphalt, water and the prebiotic synthesis of ribose, ribonucleosides, and RNA*. Acc. Chem. Res., vol. 45, no. 12, pages 2025–2034, 2012. (Cited on page 132)

[Bishop *et al.* 2006] K. J. M. Bishop, R. Klajn and B. A. Grzybowski. *The Core and Most Useful Molecules in Organic Chemistry*. Angew. Chem. Int. Ed., vol. 45, pages 5348–5354, 2006. (Cited on page 79)

[Bissette & Fletcher 2013] Andrew J. Bissette and Stephen P. Fletcher. *Mechanisms of Autocatalysis*. J Angew Chemie Int Ed, vol. 52, no. 49, pages 12800–12826, 2013. (Cited on page 2)

[Bogorad *et al.* 2013] Igor W. Bogorad, Tzu-Shyang Lin and James C. Liao. *Synthetic non-oxidative glycolysis enables complete carbon conservation*. Nature, vol. 502, no. 7473, pages 693–697, 2013. (Cited on pages 141, 145, 147, 150, 151, and 152)

[Borodina *et al.* 2005] Irina Borodina, Charlotte Schöller, Anna Eliasson and Jens Nielsen. *Metabolic Network Analysis of Streptomyces tenebrarius, a Streptomyces Species with an Active Entner-Doudoroff Pathway.* Applied and Environmental Microbiology, vol. 71, no. 5, pages 2294–2302, 2005. (Cited on page 141)

[Braatz *et al.* 2011] Benjamin Braatz, Ulrike Golas and Thomas Soboll. *How to delete categorically — Two pushout complement constructions.* Journal of Symbolic Computation, vol. 46, no. 3, pages 246–271, 2011. Applied and Computational Category Theory. (Cited on page 57)

[Breslow 1959] R. Breslow. *On the Mechanism of the Formose Reaction.* Tetrahedron Letters, vol. 1, no. 21, 1959. (Cited on pages 129 and 137)

[Butlerov 1861] Alexandr Mikhaylovich Butlerov. *Einiges über die chemische Structur der Körper.* Zeitschrift für Chemie, vol. 4, pages 549–560, 1861. (Cited on page 129)

[Cambini *et al.* 1997] Riccardo Cambini, Giorgio Gallo and Maria Grazia Scutellà. *Flows on hypergraphs.* Mathematical Programming, vol. 78, pages 195–217, 1997. (Cited on pages 95 and 108)

[Causey *et al.* 2003] T. B. Causey, S. Zhou, K. T. Shanmugam and L. O. Ingram. *Engineering the metabolism of Escherichia coli W3110 for the conversion of sugar to redox-neutral and oxidized products: Homoacetate production.* Proceedings of the National Academy of Sciences, vol. 100, no. 3, pages 825–832, 2003. (Cited on page 144)

[Centler *et al.* 2008] Florian Centler, Christoph Kaleta, Pietro Speroni di Fenizio and Peter Dittrich. *Computing chemical organizations in biological networks.* Bioinformatics, vol. 24, pages 1611–1618, 2008. (Cited on page 95)

[Chaouiya 2007] Claudine Chaouiya. *Petri net modelling of biological networks.* Briefings in Bioinformatics, vol. 8, no. 4, pages 210–219, 2007. (Cited on page 160)

[Chen *et al.* 2013] William Lingran Chen, David Z. Chen and Keith T. Taylor. *Automatic reaction mapping and reaction center detection.* Wiley Interdisciplinary Reviews: Computational Molecular Science, vol. 3, no. 6, pages 560–593, 2013. (Cited on page 63)

[Conte *et al.* 2004] D. Conte, P. Foggia, C. Sansone and M. Vento. *Thirty Years of Graph Matching in Pattern Recognition.* International Journal of Pattern Recognition and Artificial Intelligence, vol. 18, no. 03, pages 265–298, 2004. (Cited on pages 18 and 20)

[Cook 1971] Stephen A. Cook. *The Complexity of Theorem-proving Procedures.* In Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. (Cited on page 20)

[Cordella *et al.* 2001] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone and Mario Vento. *An improved algorithm for matching large graphs.* In Proc. of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, pages 149–159, 2001. (Cited on page 20)

[Cordella *et al.* 2004] L.P. Cordella, P. Foggia, C. Sansone and M. Vento. *A (Sub) Graph Isomorphism Algorithm for Matching Large Graphs.* IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, page 1367, 2004. (Cited on page 20)

[Cormen *et al.* 2001] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein. Introduction to algorithms. MIT Press and McGraw-Hill, 2 édition, 2001. (Cited on page 45)

[Corradini *et al.* 1997] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe. *Algebraic Approaches to Graph Transformation – Part I: Basic Concepts and Double Pushout Approach.* In Grzegorz Rozenberg, editor, Handbook of Graph Grammars and Computing by Graph Transformation, chapter 3, pages 163–245. World Scientific, 1997. (Cited on pages 1 and 53)

[Darga *et al.* 2008] Paul T. Darga, Karem A. Sakallah and Igor L. Markov. *Faster Symmetry Discovery Using Sparsity of Symmetries.* In Proceedings of the 45$^{\text{th}}$ Annual Design Automation Conference, DAC '08, pages 149–154, New York, NY, USA, 2008. ACM. (Cited on page 23)

[Daylight Chemical Information Systems 2011] Inc. Daylight Chemical Information Systems. *Daylight Theory Manual*, 2011. http://www.daylight.com/dayhtml/doc/theory/index.html. (Cited on page 37)

[Decker *et al.* 1982] Peter Decker, Horst Schweer and Rosmarie Pohlamnn. *Bioids : X. Identification of formose sugars, presumable prebiotic metabolites, using capillary gas chromatography/gas chromatography-mas spectrometry of n-butoxime trifluoroacetates on OV-225.* Journal of Chromatography A, vol. 244, pages 281–291, 1982. (Cited on page 132)

[Delidovich *et al.* 2014] Irina V. Delidovich, Alexandr N. Simonov, Oxana P. Taran and Valentin N. Parmon. *Catalytic Formation of Monosaccharides: From the Formose Reaction towards Selective Synthesis.* ChemSusChem, vol. 7, no. 7, pages 1833–1846, 2014. (Cited on page 129)

[Desel 1998] Jörg Desel. *Basic linear algebraic techniques for place/transition nets.* In Lectures on Petri Nets I: Basic Models, pages 257–308. Springer, 1998. (Cited on page 160)

[Dittrich & Speroni Di Fenizio 2007] Peter Dittrich and Pietro Speroni Di Fenizio. *Chemical Organisation Theory.* In Mohamed Al-Rubeai and Martin Fussenegger, editors, Systems Biology, volume 5 of *Cell Engineering*, pages 361–393. Springer Netherlands, 2007. (Cited on page 96)

[Dittrich *et al.* 2001] Peter Dittrich, J. Ziegler and Wolfgang Banzhaf. *Artificial chemistries — a review.* Artifical Life, vol. 7, no. 3, pages 225–275, 2001. (Cited on pages 1 and 62)

[Ebenhöh *et al.* 2004] O. Ebenhöh, T. Handorf and R. Heinrich. *Structural analysis of expanding metabolic networks.* In Genome informatics. International Conference on Genome Informatics, volume 15, page 35, 2004. (Cited on page 104)

[Ehrig *et al.* 1991] H. Ehrig, A. Habel, H.-J. Kreowski and F. Parisi-Presicce. *Parallelism and Concurrency in High-Level Replacement Systems.* Math. Struct. Comp. Science, vol. 1, pages 361–404, 1991. (Cited on pages 65, 66, and 67)

[Ehrig *et al.* 1997] H Ehrig, R Heckel, M Korff, M. Löwe, L Ribeiro, A Wagner and A Corradini. *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with DoublePushout Approach.* In Grzegorz Rozenberg, editor, Handbook of Graph Grammars and Computing by Graph Transformation, pages 247–312. World Scientific, Singapore, 1997. (Cited on page 64)

[Ehrig *et al.* 2006] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange and Gabriele Taenthzer. Fundamentals of algebraic graph transformation. Springer-Verlag, Berlin, D, 2006. (Cited on pages 53, 54, 55, 56, 58, and 65)

[Ehrig 1979] Hartmut Ehrig. *Introduction to the algebraic theory of graph grammars (a survey).* In Volker Claus, Hartmut Ehrig and Grzegorz Rozenberg, editors, Graph-Grammars and Their Application to Computer Science and Biology, volume 73 of *Lecture Notes in Computer Science*, page 1–69. Springer Berlin Heidelberg, 1979. (Cited on page 53)

[Eigen & Schuster 1977] Manfred Eigen and Peter Schuster. *The hypercycle: A principle of natural self-organization.* Die Naturwissenschaften, 1977. (Cited on pages 2, 81, and 162)

[Eigen 1971] Manfred Eigen. *Selforganization of matter and the evolution of biological macromolecules.* Naturwissenschaften, vol. 58, no. 10, pages 465–523, 1971. (Cited on pages 81 and 162)

[Eker *et al.* 2002] Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, and Carolyn Talcott. *Pathway Logic: Executable Models of Biological Networks.* In Fourth International Workshop on Rewriting Logic and Its Applications (WRLA'2002, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. (Cited on page 160)

[Entner & Doudoroff 1952] Nathan Entner and Michael Doudoroff. *Glucose and Gluconic Acid Oxidation of Pseudomonas Saccharophila.* Journal of Biological Chemistry, vol. 196, pages 853–862, 1952. (Cited on page 141)

[Eppstein 1999] David Eppstein. *Subgraph Isomorphism in Planar Graphs and Related Problems.* Journal of Graph Algorithms and Applications, vol. 3, no. 3, pages 1–27, 1999. (Cited on page 20)

[Eschenmoser 2007a] Albert Eschenmoser. *On a hypothetical generational relationship between HCN and constituents of the reductive citric acid cycle.* Chem. Biodivers., vol. 4, pages 554–573, 2007. (Cited on page 2)

[Eschenmoser 2007b] Albert Eschenmoser. *The search for the chemistry of life's origin.* Tetrahedron, vol. 63, pages 12821–12844, 2007. (Cited on page 157)

[Fagerberg *et al.* 2015] Rolf Fagerberg, Christoph Flamm, Rojin Kianian, Daniel Merkle and Peter F. Stadler. *Finding the K Best Synthesis Plans.* 2015. In preparation. (Cited on pages 81 and 108)

[Fell & Small 1986] D A Fell and J R Small. *Fat synthesis in adipose tissue. An examination of stoichiometric constraints.* Biochemical Journal, pages 781–786, 1986. (Cited on page 95)

[Fernández *et al.* 2012] M. Fernández, H. Kirchner and O. Namet. *A Strategy Language for Graph Rewriting.* In Proceedings of the 21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011), volume 7225 of *Lecture Notes in Computer Science*, pages 173–188, 2012. (Cited on page 83)

[Fialkowski *et al.* 2005] M. Fialkowski, K.J.M. Bishop, V.A. Chubukov, C.J. Campbell and B.A. Grzybowski. *Architecture and Evolution of Organic Chemistry.* Angew. Chem. Int. Ed., vol. 44, pages 7263–7269, 2005. (Cited on page 79)

[Foggia *et al.* 2001] Pasquale Foggia, Carlo Sansone and Mario Vento. *A performance comparison of five algorithms for graph isomorphism.* In Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, pages 188–199, 2001. (Cited on page 20)

[Gallo *et al.* 1993] Giorgio Gallo, Giustino Longo, Stefano Pallottino and Sang Nguyen. *Directed hypergraphs and applications.* Discrete Applied Mathematics, vol. 42, no. 2–3, pages 177–201, 1993. (Cited on pages 81 and 108)

[Gallo *et al.* 1998] G. Gallo, C Gentile, D Pretolani and G. Rago. *Max Horn SAT and the minimum cut problem in directed hypergraphs.* Math. Programming, vol. 80, pages 213–237, 1998. (Cited on pages 95 and 108)

[Gansner & North 2000] Emden R. Gansner and Stephen C. North. *An open graph visualization system and its applications to software engineering.* SOFTWARE - PRACTICE AND EXPERIENCE, vol. 30, no. 11, pages 1203–1233, 2000. (Cited on page 170)

[Garey & Johnson 1975] Michael R. Garey and David S. Johnson. *Complexity results for multiprocessor scheduling under resource constraints.* SIAM J. Comput., vol. 4, pages 397–411, 1975. (Cited on page 109)

[Gillespie 1963] R. J. Gillespie. *The valence-shell electron-pair repulsion (VSEPR) theory of directed valency.* Journal of Chemical Education, vol. 40, no. 6, page 295, 1963. (Cited on page 158)

[Golas 2010] Ulrike Golas. Analysis and correctness of algebraic graph and model transformations. Vieweg+Teubner, Wiesbaden, D, 2010. (Cited on page 65)

[Grzybowski *et al.* 2009] B.A. Grzybowski, K.J.M. Bishop, B. Kowalczyk and C.E. Wilmer. *The 'wired' universe of organic chemistry.* Nature Chemistry, vol. 1, pages 31–36, 2009. (Cited on pages 1, 62, and 79)

[Guptasarma 1995] Purnananda Guptasarma. *Does replication-induced transcription regulate synthesis of the myriad low copy number proteins of Escherichia coli?* Bioessays, vol. 17, no. 11, pages 987–997, 1995. (Cited on page 117)

[Habel *et al.* 2001] Annegret Habel, Jürgen Müller and Detlef Plump. *Double-pushout graph transformation revisited.* Mathematical Structures in Computer Science, vol. 11, pages 637–688, October 2001. (Cited on page 58)

[Handorf *et al.* 2005] Thomas Handorf, Oliver Ebenhöh and Reinhart Heinrich. *Expanding Metabolic Networks: Scopes of Compounds, Robustness, and Evolution.* Journal of Molecular Evolution, vol. 61, pages 498–512, 2005. (Cited on page 104)

[Hartke & Radcliffe 2009] Stephen G Hartke and AJ Radcliffe. *Mckay's canonical graph labeling algorithm.* Communicating mathematics, vol. 479, pages 99–111, 2009. (Cited on pages 23, 26, and 47)

[Heller *et al.* 2015] Stephen R Heller, Alan McNaught, Igor Pletnev, Stephen Stein and Dmitrii Tchekhovskoi. *InChI, the IUPAC International Chemical Identifier.* Journal of Cheminformatics, vol. 7, no. 1, page 23, 2015. (Cited on pages 1, 47, 48, 49, and 50)

[Hendrickson 1977] James B. Hendrickson. *Systematic synthesis design. 6. Yield analysis and convergency.* Journal of the American Chemical Society, vol. 99, no. 16, pages 5439–5450, 1977. (Cited on page 108)

[Holliday *et al.* 2005] Gemma L. Holliday, Gail J. Bartlett, Daniel E. Almonacid, Noel M. O'Boyle, Peter Murray-Rust, Janet M. Thornton and John B. O. Mitchell. *MACiE: a database of enzyme reaction mechanisms.* Bioinformatics, vol. 21, pages 4315–4316, 2005. (Cited on page 123)

[Holliday *et al.* 2012] Gemma L Holliday, Claudia Andreini, Julia D Fischer, Syed Asad Rahman, Daniel E Almonacid, Sophie T Williams and William R Pearson. *MACiE: exploring the diversity of biochemical reactions.* Nucleic Acids Research, vol. 40, pages D783–D789, 2012. (Cited on page 123)

[Hopcroft 1971] John Hopcroft. *An n log n algorithm for minimizing states in a finite automaton.* In Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971), pages 189–196. Academic Press, New York, 1971. (Cited on page 27)

[Hordijk & Steel 2004] W Hordijk and M Steel. *Detecting autocatalytic, self-sustaining sets in chemical reaction systems.* Journal of Theoretical Biology, vol. 227, pages 451–461, 2004. (Cited on page 96)

[Hucka *et al.* 2004] Michael Hucka, ABBJ Finney, Benjamin J Bornstein, Sarah M Keating, Bruce E Shapiro, Joanne Matthews, Ben L Kovitz, Maria J Schilstra, Akira Funahashi, John C Doyle*et al.* *Evolving a lingua franca and associated software*

infrastructure for computational systems biology: the Systems Biology Markup Language (SBML) project. Systems biology, vol. 1, no. 1, pages 41–53, 2004. (Cited on page 79)

[III & Maggio 1981] Howard E. Simmons III and John E. Maggio. *Synthesis of the first topologically non-planar molecule.* Tetrahedron Letters, vol. 22, no. 4, pages 287–290, 1981. (Cited on page 20)

[InChI Trust 2015] InChI Trust, 2015. http://www.inchi-trust.org/. (Cited on page 47)

[InChI 2015] InChI. *The IUPAC International Chemical Identifier (InChI)*, 2015. http://www.iupac.org/home/publications/e-resources/inchi.html. (Cited on page 47)

[increpare games 2011] increpare games. *Catalan*, 2011. http://www.increpare.com/2011/01/catalan/. (Cited on page 153)

[James 2012] Craig A. James. *OpenSMILES specification*, 2012. http://www.opensmiles.org/opensmiles.html. (Cited on pages 37, 38, 39, 40, and 41)

[Jeroslow *et al.* 1992] R. G. Jeroslow, R. K. Martin, R. R. Rardin and J. Wang. *Gainfree Leontief substitution flow problems.* Mathematical Programming, vol. 57, pages 375–414, 1992. (Cited on page 108)

[Junttila & Kaski 2007] Tommi A Junttila and Petteri Kaski. *Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs.* In ALENEX, volume 7, pages 135–149. SIAM, 2007. (Cited on pages 23 and 34)

[Kaleta *et al.* 2006] C Kaleta, F Centler and P Dittrich. *Analyzing molecular reaction networks: from pathways to chemical organizations.* Molecular Biotechnology, vol. 34, pages 117–123, 2006. (Cited on page 95)

[Kaleta *et al.* 2009] Christoph Kaleta, Stephan Richter and Peter Dittrich. *Using chemical organization theory for model checking.* Bioinformatics, vol. 25, pages 1915–1922, 2009. (Cited on page 95)

[Kanehisa *et al.* 2012] M Kanehisa, S Goto, Y Sato, M Furumichi and M Tanabe. *KEGG for integration and interpretation of large-scale molecular data sets.* Nucleic Acids Research, vol. 40, pages D109–D114, 2012. (Cited on pages 1 and 79)

[Karmarkar 1984] Narendra Karmarkar. *A new polynomial-time algorithm for linear programming.* In Proceedings of the sixteenth annual ACM symposium on Theory of computing, pages 302–311. ACM, 1984. (Cited on page 115)

[Karp & Caspi 2011] P.D. Karp and R. Caspi. *A survey of metabolic databases emphasizing the `MetaCyc` family.* Archives of Toxicology, vol. 85, pages 1015–1033, 2011. (Cited on page 79)

[Katebi *et al.* 2010] Hadi Katebi, Karem A Sakallah and Igor L Markov. *Symmetry and satisfiability: An update.* In Theory and Applications of Satisfiability Testing–SAT 2010, pages 113–127. Springer, 2010. (Cited on page 23)

[Kauffman *et al.* 2003] K. J. Kauffman, P. Prakash and J. S. Edwards. *Advances in flux balance analysis.* Current Opinion in Biotechnology, vol. 14, no. 5, pages 491–496, 2003. (Cited on pages 80 and 95)

[Kauffman 1986] S. A. Kauffman. *Autocatalytic sets of proteins.* Journal of theoretical biology, vol. 119, no. 1, pages 1–24, 1986. (Cited on page 96)

[Kauffman 1995] S. A. Kauffman. At home in the universe: The search for laws of self-organization and complexity. Oxford University Press, USA, 1995. (Cited on pages 2 and 96)

[Khachiyan 1980] Leonid G Khachiyan. *Polynomial algorithms in linear programming.* USSR Computational Mathematics and Mathematical Physics, vol. 20, no. 1, pages 53–72, 1980. (Cited on page 115)

[Kim *et al.* 2011] Hyo-Joong Kim, Alonso Ricardo, Heshan I Illangkoon, Jung Kim Kim, Matthew A Carrigan, Fabianne Frye and Steven A Benner. *Synthesis of Carbohydrates in Mineral-Guided Prebiotic Cycles*. J. Am. Chem. Soc., vol. 133, no. 24, pages 9457–9468, 2011. (Cited on pages 131, 132, and 137)

[Klamt & Stelling 2002] S. Klamt and J. Stelling. *Combinatorial complexity of pathway analysis in metabolic networks*. Molecular Biology Reports, vol. 29, pages 233–236, 2002. (Cited on page 95)

[Klamt & Stelling 2003] S. Klamt and J. Stelling. *Two approaches for metabolic pathway analysis?* Trends in Biotechnology, vol. 21, no. 2, pages 64–69, 2003. (Cited on pages 2 and 95)

[Knight 1989] Kevin Knight. *Unification: A Multidisciplinary Survey*. ACM Comput. Surv., vol. 21, no. 1, pages 93–124, March 1989. (Cited on page 5)

[Koch *et al.* 2010] I. Koch, W. Reisig and F. Schreiber. Modeling in systems biology: The petri net approach. Computational Biology. Springer, 2010. (Cited on page 160)

[Koch 2010] Ina Koch. *Petri Nets – A Mathematical Formalism to Analyze Chemical Reaction Networks*. Molecular Informatics, vol. 29, no. 12, pages 838–843, 2010. (Cited on page 160)

[Kun *et al.* 2008] Ádám Kun, Balázs Papp and Eörs Szathmáry. *Computational identification of obligatorily autocatalytic replicators embedded in metabolic networks*. Genome Biology, vol. 9, page R51, 2008. (Cited on pages 96 and 104)

[Lee *et al.* 2000] Sangbum Lee, Chan Phalakornkule, Michael M Domach and Ignacio E Grossmann. *Recursive MILP model for finding all the alternate optima in LP models for metabolic networks*. Computers & Chemical Engineering, vol. 24, no. 2, pages 711–716, 2000. (Cited on page 119)

[Löwe 1993] M. Löwe. *Algebraic approach to single-pushout graph transformation*. Theor. Comp. Sci., vol. 109, pages 181–224, 1993. (Cited on page 64)

[Luks 1982] Eugene M. Luks. *Isomorphism of graphs of bounded valence can be tested in polynomial time*. Journal of Computer and System Sciences, vol. 25, no. 1, pages 42–65, 1982. (Cited on page 20)

[Mann *et al.* 2013a] Martin Mann, Heinz Ekker and Christoph Flamm. *The Graph Grammar Library - A Generic Framework for Chemical Graph Rewrite Systems*. In Keith Duddy and Gerti Kappel, editors, Theory and Practice of Model Transformations, volume 7909 of *Lecture Notes in Computer Science*, pages 52–53. Springer Berlin Heidelberg, 2013. (Cited on pages viii, 1, and 15)

[Mann *et al.* 2013b] Martin Mann, Heinz Ekker and Christoph Flamm. *The Graph Grammar Library - a generic framework for chemical graph rewrite systems*. CoRR, vol. abs/1304.1356, 2013. (Cited on pages viii and 15)

[Mann *et al.* 2013c] Martin Mann, Feras Nahar, Heinz Ekker, Rolf Backofen, Peter F Stadler and Christoph Flamm. *Atom mapping with constraint programming*. CP, vol. 13, pages 805–822, 2013. (Cited on page 63)

[Matheiss & Rubin 1980] TH Matheiss and David S Rubin. *A survey and comparison of methods for finding all vertices of convex polyhedral sets*. Mathematics of operations research, vol. 5, no. 2, pages 167–185, 1980. (Cited on page 119)

[May 2013] John May. *SMILES Implicit Valence of Aromatic Atoms*, 2013. http://efficientbits.blogspot.dk/2013/09/smiles-implicit-valence-of-aromatic.html. (Cited on page 41)

[McKay & Piperno 2014a] Brendan D McKay and Adolfo Piperno. *nauty and Traces*, 2014. http://pallini.di.uniroma1.it/. (Cited on page 23)

[McKay & Piperno 2014b] Brendan D McKay B.ay and Adolfo Piperno. *Practical graph isomorphism, II.* Journal of Symbolic Computation, vol. 60, pages 94–112, 2014. (Cited on pages 21, 23, 24, 26, 27, 28, 29, 34, and 35)

[McKay 1981] Brendan D. McKay. *Practical Graph Isomorphism.* In Congressus Numerantium, volume 30, pages 45–97. Utilitas Mathematica Pub. Incorporated, 1981. http://cs.anu.edu.au/~bdm/papers/pgi.pdf. (Cited on pages 21, 23, 27, 35, 36, and 47)

[Meléndez-Hevia & Isidoro 1985] Enrique Meléndez-Hevia and Angel Isidoro. *The game of the pentose phosphate cycle.* Journal of Theoretical Biology, vol. 117, no. 2, pages 251–263, 1985. (Cited on page 161)

[Neglur *et al.* 2005] Greeshma Neglur, Robert L. Grossman and Bing Liu. *Assigning Unique Keys to Chemical Compounds for Data Integration: Some Interesting Counter Examples.* In Bertram Ludäscher and Louiqa Raschid, editors, Data Integration in the Life Sciences, volume 3615 of *Lect. Notes Comp. Sci.*, pages 145–157. Springer, Berlin, 2005. (Cited on page 46)

[Nilsson & Maluszynski 1995] Ulf Nilsson and Jan Maluszynski. Logic, programming, and prolog. John Wiley & Sons, Inc., New York, NY, USA, 2nd édition, 1995. (Cited on page 5)

[O'Boyle 2012] Noel M O'Boyle. *Towards a Universal SMILES representation-A standard method to generate canonical SMILES based on the InChI.* J. Cheminformatics, vol. 4, page 22, 2012. (Cited on page 37)

[OLBoyle *et al.* 2011] Noel M OLBoyle, Michael Banck, Craig A James, Chris Morley, Tim Vandermeersch and Geoffrey R Hutchison. *Open Babel: An open chemical toolbox.* J Cheminf, vol. 3, page 33, 2011. (Cited on page 170)

[Orth *et al.* 2010] J. D. Orth, I. Thiele and B. Ø. Palsson. *What is flux balance analysis?* Nature Biotech., vol. 28, pages 245–248, 2010. (Cited on pages 2 and 95)

[Özturan 2008] Can Özturan. *On finding hypercycles in chemical reaction networks.* Appl. Math. Letters, vol. 21, pages 881–884, 2008. (Cited on page 81)

[Papin *et al.* 2004] Jason A Papin, Joerg Stelling, Nathan D Price, Steffen Klamt, Stefan Schuster and Bernhard O Palsson. *Comparison of network-based pathway analysis methods.* Trends in biotechnology, vol. 22, no. 8, pages 400–405, 2004. (Cited on pages 81 and 115)

[Papoutsakis 1984] E T Papoutsakis. *Equations and calculations for fermentations of butyric acid bacteria.* Biotech Bioeng, vol. 26, pages 174–187, 1984. (Cited on page 95)

[Petrarca *et al.* 1967] Anthony E. Petrarca, Michael F. Lynch and James E. Rush. *A Method for Generating Unique Computer Structural Representations of Stereoisomers.* Journal of Chemical Documentation, vol. 7, no. 3, pages 154–165, 1967. (Cited on page 158)

[Petri 1962] Carl Adam Petri. *Kommunikation mit automaten.* 1962. (Cited on pages 95 and 160)

[Pinaud *et al.* 2012] Bruno Pinaud, Guy Melançon and Jonathan Dubois. *PORGY: A Visual Graph Rewriting Environment for Complex Systems.* Comput. Graph. Forum, vol. 31, no. 3, 2012. (Cited on page 83)

[Piperno 2008] Adolfo Piperno. *Search space contraction in canonical labeling of graphs (preliminary version).* CoRR, abs/0804.4881, 2008. (Cited on pages 23, 26, 27, and 30)

[Rasmussen 2012] Thomas Glue Rasmussen. Analysing chemical reaction pathways with Petri nets. Master's thesis, University of Southern Denmark, 2012. (Cited on page 160)

[Ricardo *et al.* 2004] A Ricardo, M A Carrigan, A N Olcott and S A Benner. *Borate minerals stabilize ribose.* Science, vol. 303, page 196, 2004. (Cited on pages 132 and 133)

[Ricardo *et al.* 2006] Alonso Ricardo, Fabianne Frye, Matthew A. Carrigan, Jeremiah D. Tipton, David H. Powell and Steven A. Benner. *2-Hydroxymethylboronate as a Reagent To Detect Carbohydrates: Application to the Analysis of the Formose Reaction.* Journal of Organic Chemistry, vol. 71, pages 9503–9505, 2006. (Cited on page 131)

[Roberts *et al.* 1952] John D. Roberts, Andrew Streitwieser and Clare M. Regan. *Small-Ring Compounds. X. Molecular Orbital Calculations of Properties of Some Small-Ring Hydrocarbons and Free Radicals1.* Journal of the American Chemical Society, vol. 74, no. 18, pages 4579–4582, 1952. (Cited on page 15)

[Rosselló & Valiente 2005] F. Rosselló and G. Valiente. *Graph Transformation in Molecular Biology.* Lect. Notes Comp. Sci., vol. 3393, pages 116–133, 2005. (Cited on page 1)

[Ruiz-Mirazo *et al.* 2014] Kepa Ruiz-Mirazo, Carlos Briones and Andrés de la Escosura. *Prebiotic Systems Chemistry: New Perspectives for the Origins of Life.* Chemical Reviews, vol. 14, pages 285–366, 2014. (Cited on page 1)

[Sauer 2006] Uwe Sauer. *Metabolic networks in motion:* ^{13}*C-based flux analysis.* Molecular Systems Biology, vol. 2, page 62, 2006. (Cited on pages 2 and 123)

[Savinell & Palsson 1992] Joanne M. Savinell and Bernhard Ø Palsson. *Network Analysis of Intermediary Metabolism using Linear Optimization. I. Development of Mathematical Formalism.* Journal of Theoretical Biology, vol. 154, pages 421–454, 1992. (Cited on pages 1 and 115)

[Schilling *et al.* 2000] Christophe H Schilling, David Letscher and Bernhard Ø Palsson. *Theory for the systemic definition of metabolic pathways and their use in interpreting metabolic function from a pathway-oriented perspective.* Journal of theoretical biology, vol. 203, no. 3, pages 229–248, 2000. (Cited on page 115)

[Schmidt 2000] Karsten Schmidt. *LoLA A Low Level Analyser.* In Mogens Nielsen and Dan Simpson, editors, Application and Theory of Petri Nets 2000, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer Berlin Heidelberg, 2000. (Cited on page 160)

[Schuster & Hilgetag 1994] Stefan Schuster and Claus Hilgetag. *On elementary flux modes in biochemical reaction systems at steady state.* Journal of Biological Systems, vol. 2, no. 02, pages 165–182, 1994. (Cited on pages 95 and 115)

[Schuster *et al.* 2000] S. Schuster, D. A. Fell and T. Dandekar. *A general definition of metabolic pathways useful for systematic organization and analysis of complex metabolicnetworks.* Nature Biotech., vol. 18, pages 326–332, 2000. (Cited on pages 2 and 95)

[Seress 2003] Á. Seress. Permutation group algorithms. Cambridge Tracts in Mathematics. Cambridge University Press, 2003. (Cited on pages 24 and 36)

[Siek *et al.* 2001] Jeremy G Siek, Lie-Quan Lee and Andrew Lumsdaine. Boost graph library: The user guide and reference manual. Pearson Education, 2001. http://www.boost.org/libs/graph/. (Cited on page 20)

[Solnon 2010] Christine Solnon. *AllDifferent-based filtering for subgraph isomorphism.* Artificial Intelligence, vol. 174, no. 12–13, pages 850–864, 2010. (Cited on page 20)

[Sommerville 2011] I. Sommerville. Software engineering. International Computer Science Series. Pearson, 2011. (Cited on page 47)

[Stanger 2009] Amnon Stanger. *What is... aromaticity: a critique of the concept of aromaticity—can it really be defined?* Chemical Communications, no. 15, pages 1939–1947, 2009. (Cited on page 13)

[Stein *et al.* 2011] S Stein, S Heller, D Tchekhovskoi and I Pletnev. *IUPAC International Chemical Identifier (InChI) - InChI version 1, software version 1.04 (2011) - Technical Manual*, 2011. http://www.inchi-trust.org/download/104/InChI_TechMan.pdf. (Cited on pages 47 and 49)

[Swart 1985] Garret Swart. *Finding the convex hull facet by facet.* Journal of Algorithms, vol. 6, no. 1, pages 17–48, 1985. (Cited on page 119)

[Sylvester 1878] J. J. Sylvester. *On an application of the new atomic theory to the graphical representation of the invari- ants and covariants of binary quantics, with three appendices.* American Journal of Mathematics, vol. 1, no. 1, pages 64–128, 1878. (Cited on page 1)

[Szathmáry 1988] Eörs Szathmáry. *A hypercyclic illusion.* Journal of theoretical biology, vol. 134, no. 4, pages 561–563, 1988. (Cited on pages 162 and 163)

[Szathmáry 2013] Eörs Szathmáry. *On the propagation of a conceptual error concerning hypercycles and cooperation.* J. Syst. Chem., vol. 4, page 1, 2013. (Cited on page 162)

[Talcott & Dill 2005] Carolyn Talcott and David L. Dill. *The pathway logic assistant.* In Third International Workshop on Computational Methods in Systems Biology, pages 228–239, 2005. (Cited on page 160)

[Tantau 2013] Till Tantau. *The TikZ and PGF Packages*, 2013. (Cited on page 170)

[Tao *et al.* 2001] Han Tao, Ramon Gonzalez, Alfredo Martinez, Maria Rodriguez, LO Ingram, JF Preston and KT Shanmugam. *Engineering a homo-ethanol pathway inEscherichia coli: Increased glycolytic flux and levels of expression of glycolytic genes during xylose fermentation.* Journal of bacteriology, vol. 183, no. 10, pages 2979–2988, 2001. (Cited on page 144)

[Ullmann 1976] J. R. Ullmann. *An Algorithm for Subgraph Isomorphism.* J. ACM, vol. 23, no. 1, pages 31–42, January 1976. (Cited on page 20)

[Ullmann 2011] Julian R. Ullmann. *Bit-vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism.* J. Exp. Algorithmics, vol. 15, pages 1.6:1.1–1.6:1.64, February 2011. (Cited on page 20)

[Wagner & Urbanczik 2005] C Wagner and R Urbanczik. *The geometry of the flux cone of a metabolic network.* Biophys J., vol. 89, pages 3837–3845, 2005. (Cited on page 95)

[Warren & Center 1983] David HD Warren and Artificial Intelligence Center. An abstract Prolog instruction set, volume 309. SRI International Menlo Park, California, 1983. (Cited on page 6)

[Watson 1984] M R Watson. *Metabolic maps for the Apple II.* Biochemical Society Transactions, vol. 12, pages 1093–1094, 1984. (Cited on page 95)

[Weininger *et al.* 1989] David Weininger, Arthur Weininger and Joseph L. Weininger. *SMILES. 2. Algorithm for generation of unique SMILES notation.* Journal of Chemical Information and Computer Sciences, vol. 29, no. 2, pages 97–101, 1989. (Cited on pages 21, 37, 42, 43, and 44)

[Weininger 1988] D. Weininger. *SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules.* Journal of Chemical Information and Computer Sciences, vol. 28, no. 1, pages 31–36, 1988. (Cited on pages 1, 21, 37, 38, 39, 41, and 46)

[Wieser *et al.* 2013] Michael E Wieser, Norman Holden, Tyler B Coplen, John K Böhlke, Michael Berglund, Willi A Brand, Paul De Bièvre, Manfred Gröning, Robert D Loss, Juris Meija*et al. Atomic weights of the elements 2011 (IUPAC Technical Report).* Pure and Applied Chemistry, vol. 85, no. 5, pages 1047–1078, 2013. (Cited on page 11)

[Wipke & Dyott 1974] W. Todd Wipke and Thomas M. Dyott. *Stereochemically unique naming algorithm.* Journal of the American Chemical Society, vol. 96, no. 15, pages 4834–4842, 1974. (Cited on page 158)

[Yadav *et al.* 2004] ManeeshK. Yadav, BrianP. Kelley and StevenM. Silverman. *The Potential of a Chemical Graph Transformation System.* In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce and Grzegorz Rozenberg, editors, Graph Transformations, volume 3256 of *Lecture Notes in Computer Science*, pages 83–95. Springer Berlin Heidelberg, 2004. (Cited on page 1)

[Zamboni 2011] Nicola Zamboni. ^{13}C *metabolic flux analysis in complex systems.* Curr Opin Biotech, vol. 22, pages 103–108, 2011. (Cited on pages 2 and 123)

[Zanghellini *et al.* 2013] J Zanghellini, D E Ruckerbauer, M Hanscho and C Jungreuthmayer. *Elementary flux modes in a nutshell: properties, calculation and applications.* Biotechnol J., vol. 8, pages 1009–1016, 2013. (Cited on page 95)

[Zeigarnik 2000] A. V. Zeigarnik. *On Hypercycles and Hypercircuits in Hypergraphs.* In P. Hansen, P. W. Fowler and M. Zheng, editors, Discrete Mathematical Chemistry, volume 51 of *DIMACS series in discrete mathematics and theoretical computer science*, pages 377–383. American Mathematical Society, Providence, RI, 2000. (Cited on pages 79, 80, 81, and 95)