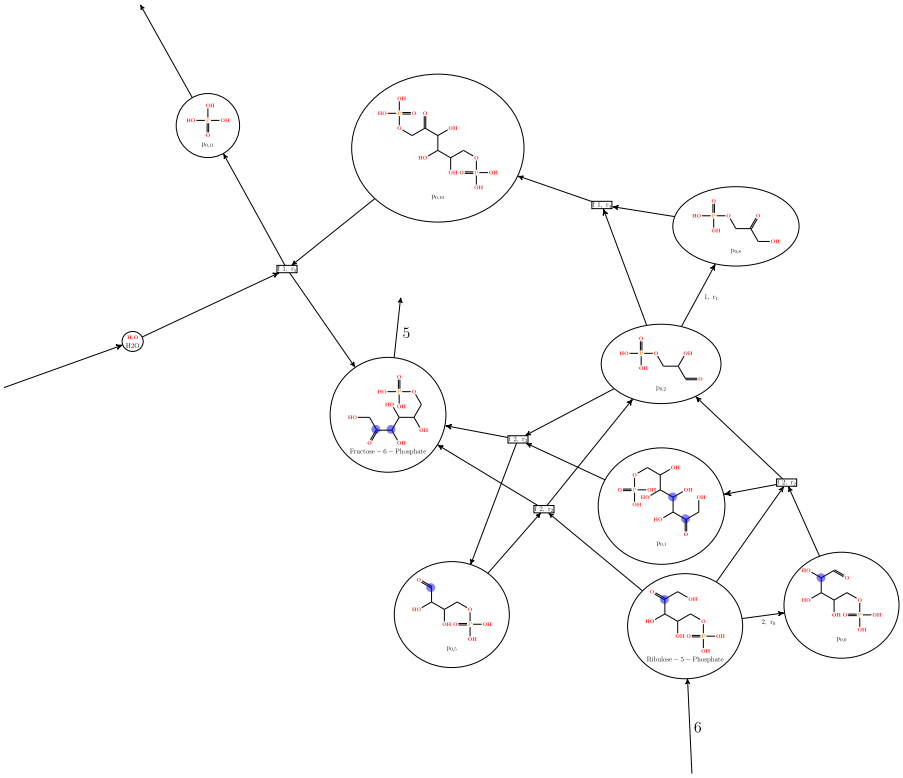
MASTER THESIS

Graph Transformation, Atom Tracing, and Isotope Labelling



Author:
Peter Severin Rasmussen

Supervisor:
Daniel Merkle

SDU
UNIVERSITY OF
SOUTHERN DENMARK

Department of Mathematics
and Computer Science
(IMADA)

June 1, 2018

Abstract

The following thesis revolves around the topic of isotope labelling experiments in cheminformatics and the creation of an automated system to design isotope labelling experiments in order to distinguish chemical pathways which is especially relevant is metabolic networks. The system can however be used in any chemical or non-chemical setting that can be modelled by a generative chemistry framework that uses graph-grammars.

To accomplish this, an approach based on semigroup theory is presented. It uses a series of orbit calculations to see all possible places isotope labels can go. The thesis introduces the Pathway Tables and Pathway Comparison Tables that give the user an overview of what isotope labellings can be useful to try based on orbits and inverted orbits. One of the outcomes of the thesis has been a Python framework that can be used to infer such tables automatically. The Python framework is tested on a couple of chemical examples like Glycolysis, where it is used distinguish the ED and EMP pathways in an automated fashion. It has also been used to distinguish other artificially-created pathways.

This thesis also presents an alternative method to solve the problem of finding vertex maps between educt and product graphs in a chemical network previously created using graph-grammars. It allows e.g. for the ability to optionally ignore certain atoms such as hydrogens which cause a lot of unnecessary combinatorics. This new method is designed and implemented to improve the performance of the semigroup approach on large chemical networks, where a large number of vertex maps needs to be computed. It is demonstrated how this new method can be orders of magnitude faster for certain chemical examples.

The approaches and tools developed in this thesis open up for a wide range of other subsequent analysis approaches for instance using Petri-nets, constraint-based membership testing or creating ordinary differential equations for simulations over time.

Resume

Det følgende speciale omhandler emnet isotope labelling experiments i keminformatik og skabelsen af et automatisk system til isotope labelling experiments for at muliggøre at kunne skelne forskellige kemisks pathways hvilket bl.a. er relevant i metaboliske netværk. Systemet kan dog bruges i et hvert kemiske eller ikke-kemisk scenarie, der kan modelleres af et generative chemistry framework der bruger grafgrammatikker.

For at gøre dette introduceres en metode baseret på semigruppe-teori. Den bruger en række orbit-udregninger for at danne et overblik over hvor alle mulige isotope labels kan ende op. Specialet introducerer Pathway Tables og Pathway Comparison Tables, der giver brugeren et overblik over hvilke isotope labellings som kan være frugtbare at afprøve, baseret på orbits og inverterede orbits. Et produkt af specialet har været et Python framework der kan skabe førnævnte tabeller automatisk. Python frameworket er testet på et par forskellige kemiske eksempler såsom Glycolysis, hvor det kan bruges til at skelne ED og EMP pathways på automatisk vis. Frameworket bruges også til at skelne andre kunstigt-skabte pathways.

Specialet introducerer også en alternativ metode til at løse problemet om at finde vertex maps mellem edukt- og produkt-grafer i et kemisk netværk tidligere skabt af grafgrammatikker. Metoden gør det bl.a. muligt at ignorere atomer såsom hydrogener, der er skyld i en masse unødvendig kombinatorik. Denne nye metode er designet og implementeret for at forberdre køretiden af semigruppemetoden på store kemiske netværk, hvor mange vertex maps skal beregnes. Det demonstreres hvordan denne nye metode kan være størrelsesordener hurtigere for visse kemiske eksempler.

Disse metoder og værktøjer, udviklet i dette speciale, åbner op for en bred vifte af opfølgende analyse-metoder for eksempel ved at bruge Petri-nets, constraint-baseret membership testing eller skabelsen af ordinære differentialligninger til at lave simuleringer over tid.

Acknowledgements

This entire process has been a great experience – both fun and challenging.
I would like to thank my supervisor, Daniel Merkle, for his tremendous enthusiasm and support throughout the run of my thesis. I cannot thank him enough for the number of hours he has dedicated to me, and he has encouraged me to work in directions I found fascinating while always ensuring that it would be useful for the thesis. And in every part of it, his energy and excitement has been very important for my motivation.

A huge thanks should also go to Jakob Lykke Andersen who has invested countless hours sparring and explaining how MØD works in theory and in practice to a point where I was able to implement things directly in MØD. Without him this thesis would not have been possible in its current form.

I would also like to thank my friends and family for their support. An especially big thanks to those that helped me with proof-reading.

Finally, I would like to thank the students of IMADA, especially the residents of the office Balkonen. It has made a huge impact to be surrounded by these amazing people and participating in the social life of IMADA every day.

Contents

1

Introduction & Preliminaries

1.1 Introduction

In chemistry and biology the study of pathways is very important. In this thesis, we use the word *pathway* to refer one or more start-molecules that, via a sequence of reactions, become one or more goal-molecules. For instance a sequences of reactions turn Glucose into fat molecules. Some pathways might be active in some species and inactive in others. It would therefore be interesting to have methods and tools to better understand such pathways. One idea is for instance to keep track of atoms through a chemical or metabolic network.

There has been work by [15] and [24] in the field of flux balance analysis where metabolic networks are extended to include information about how atoms are transported through such networks. These networks are called atom transition networks and they serve as a basis for the analysis of isotope labelling experiments.

One problem is that atom transition networks are directed graphs that do not encode the symmetries of molecules. They also often need to be created by hand. Instead, in this thesis, the aim is to use hypergraphs to model metabolic networks. Such hypergraphs have vertices corresponding to molecular graphs and hyperedges corresponding to chemical reactions based on graph rewriting rules. These hypergraphs can be created automatically and since such hypergraphs model molecules on an atomic level, it is natural to augment them with information of how atoms are transported through the individual reactions and thereby through the whole network.

Using hypergraphs, a group theoretical approach can be employed, where reactions are seen as permutations based on the set of all automorphisms (symmetries) of all molecular graphs and the set of all possible atom-to-atom mappings for each hyperedge in the system.

The remainder of this chapter will build the theoretical foundation that is used throughout the thesis. First, the theory of generative chemistry using graph

grammars is explained. This will include how molecules are modelled as graphs and reactions are modelled as graph transformations. Next, an overview of isotope labelling experiments is presented, and lastly, group theory and semigroup theory is explained as it will be used extensively throughout the thesis. An overview of the rest of the thesis can be found in the last part of this chapter.

1.2 The Molecular Model

In this thesis we will simulate many chemical reactions and molecules. Therefore a suitable molecular model and simulation program is needed. The work of this thesis will use and build on top of MedØlDatschgerl (MØD, `mod`) developed by Jakok Lykke Andersen and others.

Explaining all of MØD and the theory behind it is out of the scope of this thesis, but can be found here [2] [3] [4] [5] [6]. However in order for this thesis to be self-contained, all the theory needed is given here.

MØD operates in the realm of *generative chemistry* which is the study of building complex and interesting chemical systems starting from a small set of molecules and applying rules to produce new molecules. Rules might for instance be well-known chemical reactions. MØD is essentially a graph rewriting machine that finds rule patterns in existing molecules and modify them according to the rules to get new molecules.

We define *molecules*:

Definition 1. In this model, a *molecule* is a simple connected graph $G = (V, E)$.

- Vertices of V are annotated with
  - Atom, e.g. H, C, O etc.
  - Charge, e.g. $-$, $+$ or no charge.
- Edges of E are annotated with
  - Bond type, e.g. single bond ($-$), double bond ($=$) or aromatic bond.

See Figure 1.1 for an example.

Important: The graph corresponding to the molecule is not *labelled* in the sense that each vertex has a unique, distinguishing identifier. *However*, all graphs need to be stored somehow and in MØD each vertex will be assigned a unique but arbitrary *internal id*. These ids can be printed when printing molecules in MØD and will be seen throughout this thesis. They are useful for pointing at specific atoms in a molecule, but can change for every run of MØD.

Informally, *reactions* are transformations on molecule-graphs. This can generally be any desired transformation, but in a chemical setting this is typically the addition and removal of edges (bonds).

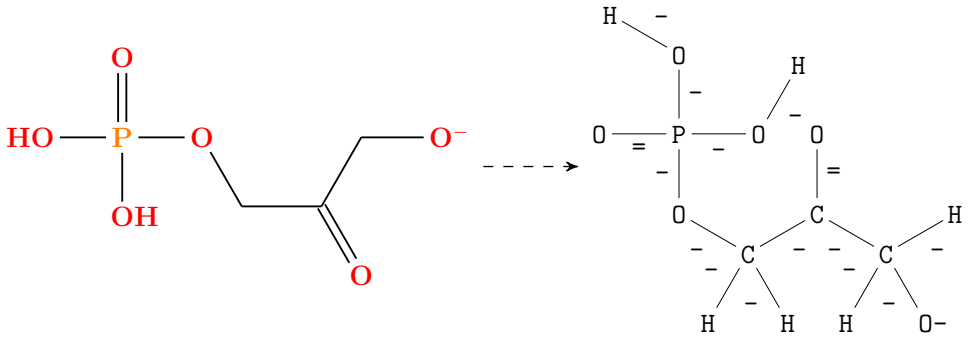


FIGURE 1.1: How a molecule is represented as a graph in this model.

To understand the formalism of how reactions are modelled in MØD, one first needs to know about morphisms and Double-Pushouts.

1.2.1 Morphisms & Double-Pushout

Throughout this thesis, *function*, *mapping*, *map* and *morphism* will be used almost synonymously. They are all functions that take some input and return some output, however depending on the context different words will be used. For instance, when talking about Double-Pushouts and the graph isomorphism problem, the word *morphism* will be used, since this is rooted in category theory. It describes a function that preserves some structure.

Morphisms come in a couple of variants, including *monomorphism* and *isomorphism*. And while one could consider them in a general category theory setting, we will consider the setting of graphs, since this is what they will be used for in this thesis. Specifically, when applied to molecule-graphs, morphisms can be used to model graph transformations.

Definition 2. Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be graphs.
A *graph morphism* $m\colon G \to H$ maps vertices and edges from G to H while preserving the structure in the following way:

$$\text{If } e = (u, v) \in E_G, \text{ then } m(e) = (m(u), m(v)) \in E_H.$$

In other words: If there is an edge between two mapped vertices in G, it must also be there in H.

This definition and the rest of the section is a brief version of [6].

- m is a *monomorphism* if m is injective, i.e. two vertices of G cannot map to the same thing in H.
- m is a *subgraph isomorphism* if m is a monomorphism and

$$(u, v) \in E_G \Leftrightarrow (m(u), m(v)) \in E_H.$$

In other words, we do not only require that an edge in G is preserved in H, but also that if there is *no* edge between two mapped vertices in G, then there should also be no edge between the corresponding vertices in H.

- m is an *isomorphism* if m is a subgraph isomorphism and m is a bijection of the vertices, meaning that $|V_G| = |V_H|$.

  When an isomorphism between G and H exists, we say that they are *isomorphic* and write $G \cong H$.

- m is an *automorphism* if m is an isomorphism from G to itself. The difference between isomorphism and automorphism is mostly semantics, for instance whether we know that G and H are the same graph beforehand.

  Automorphisms are used to describe symmetries of graphs, for instance if vertices can be mapped onto themselves in a way that describes rotation or mirroring of the graph.

(a) A morphism. (b) A monomorphism.

(c) A subgraph isomorphism. (d) An isomorphism.

FIGURE 1.2: The different morphisms visualized.

See Figure 1.2 for an example of the different types of morphisms. How these morphisms are found given two graphs will be discussed in a later section.
It is possible to compose morphisms and this operation is associative.

Notation 1. The composition of morphisms f and g is written $f \circ g$. In this thesis, it is left-to-right, meaning that $f \circ g$ is read as "first f then g" or mathematically $g(f(x))$ for some x.

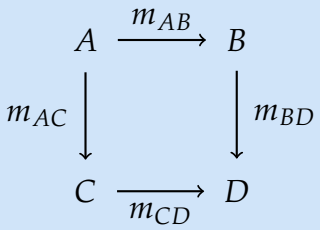
Another concept from category theory is the *pushout*.

Definition 3. Let A, B, C and D be graphs and suppose we have two morphisms, $m_{AB} \colon A \to B$ and $m_{AC} \colon A \to C$.

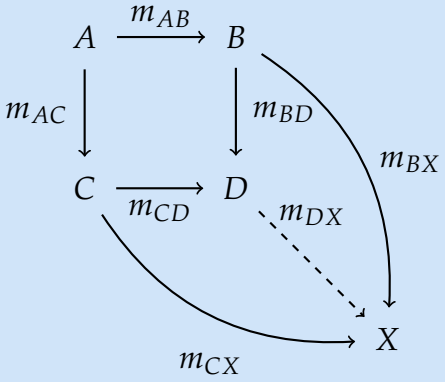
The morphisms $m_{CD} \colon C \to D$ and $m_{BD} \colon B \to D$ form a *pushout* of m_{AB} and m_{AC}, if and only if

(i) $m_{AB} \circ m_{BD} = m_{AC} \circ m_{CD}$.

In other words, if some vertex is mapped from A to B to D (via m_{AB} and m_{BD}), it should give the same vertex as if mapped from A to C to D (via m_{AC} and m_{CD}).



(ii) For all pairs of morphisms $m_{BX}\colon B \to X$ and $m_{CX}\colon C \to X$ that map to some new graph X, where it holds that $m_{AB} \circ m_{BX} = m_{AC} \circ m_{CX}$, there must exist a unique morphism $m_{DX}\colon D \to X$ such that $m_{BX} = m_{BD} \circ m_{DX}$ and $m_{CX} = m_{CD} \circ m_{DX}$.



The use of pushouts will become apparent later. For now, it is just **important** to emphasize that if given A, B and C from the diagram in Definition 3, it is possible to determine D. Generally, given any three of the four, it is possible to determine the last one.

A Double-Pushout (DPO) is the conjunction of two pushout diagrams, one to the left and one to the right, as depicted in Figure 1.3.

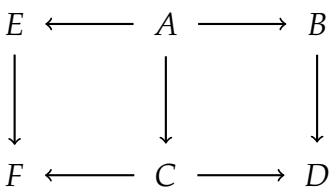


FIGURE 1.3

1.2.2 Reactions & Rule Application

The goal is to model chemical reactions (or any graph transformation) using morphisms and DPOs. Suppose for instance we want to describe the reaction pattern of Figure 1.4. It describes that among all the currently produced molecules, one molecule must contain a pattern of C=COH and another must contain a pattern of C=O (together they are the pattern L). If these patterns are found, edges are removed and added to turn them into one molecule with the pattern R.

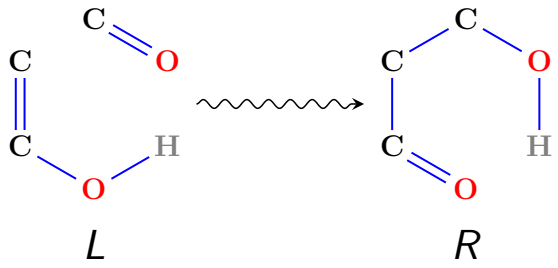


FIGURE 1.4: A desired graph transformation.

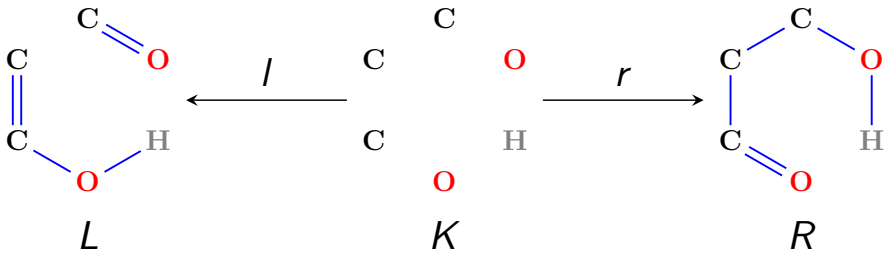


FIGURE 1.5: The corresponding rule in DPO formalism.

The rules of MØD will be written in Double Pushout (DPO) formalism with the format $p = (L \xleftarrow{l} K \xrightarrow{r} R)$. K is the context that does not change during the transformation. L is the left graph that transforms into R through K. Let G be a union graph formed by a multiset of molecules. All connected components in G are individual molecules. A rule is applied to G by finding a monomorphism from L to G (a "pattern match"), then using the morphisms l and r to remove $L \setminus K$ from G and adding $R \setminus K$ in its place, resulting in a transformed graph called H.

We can write the rule from Figure 1.4 in the DPO formalism as depicted in Figure 1.5.

We can now describe the mechanism used by MØD to apply rules, based on the Figure 1.6 and depicted with a chemical example in Figure 1.7.

1. Given a union graph G of a multiset of molecules and a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$.
2. Find L in G, i.e. find a monomorphism m.
3. Determine D based on L, K and G.

4. Determine H based on K, R and D.
5. We now have the modified graph H.

We call this a *direct derivation* from G to H via rule p and morphism m, written $G \overset{p,m}{\Longrightarrow} H$ or $G \overset{p}{\Rightarrow} H$ if the morphism is unimportant.

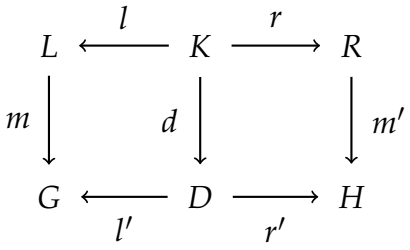


FIGURE 1.6

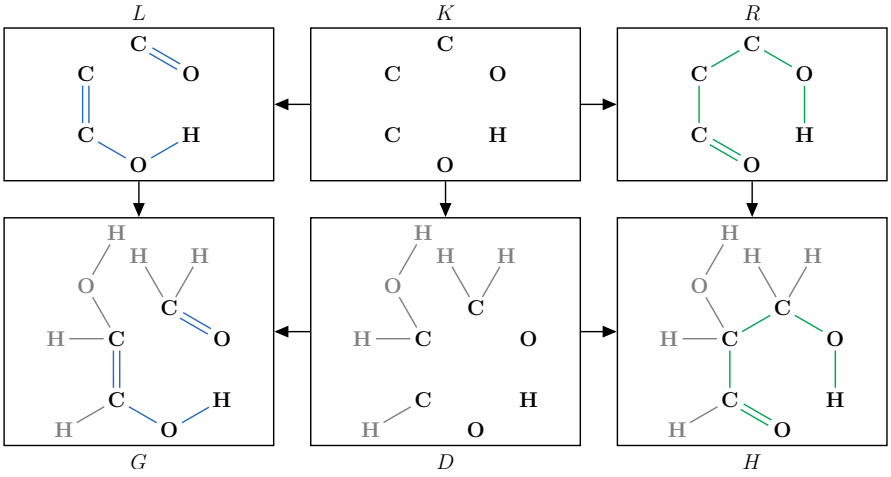


FIGURE 1.7: A chemical example of a DPO diagram being used to apply a rule.

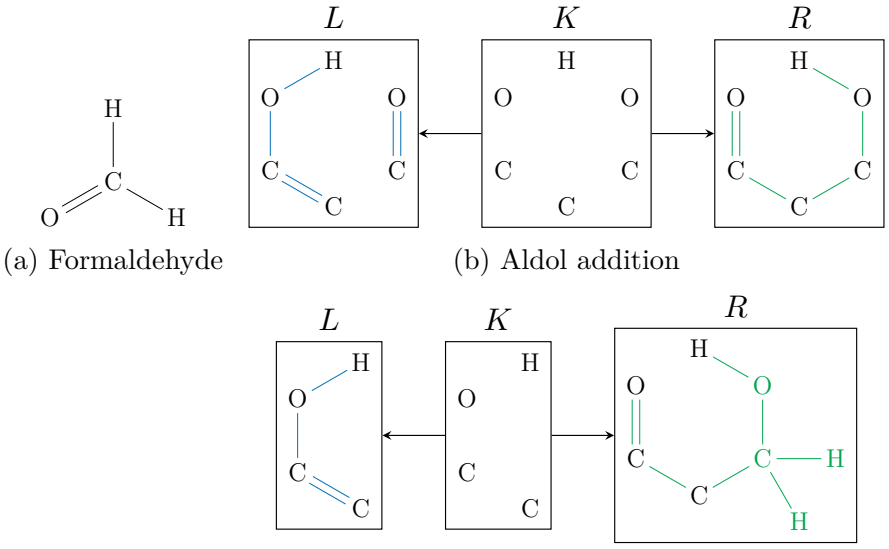
1.2.3 Derivation Graph

A powerful concept is the Derivation Graph (DG) which represents the *chemical space* of all possible molecules that can be produced by various rule applications. The DG will be a central part of our atom tracing analysis.

A DG is a hypergraph $H = (V, E)$ consisting of vertices V and hyperedges E. Vertices are just like ordinary graph vertices and in the case of the DG each vertex contains a molecule-graph. A hyperedge $e = (e^+, e^-)$ is a multiset of tail-vertices e^+ to a multiset of head-vertices e^-. In the case of the DG, hyperedges correspond to rule applications which in turn correspond to chemical reactions.

The DG is computed by starting with an initial set of molecules and then applying rules following some strategy, for instance apply all rules repeatedly, until no new molecules are produced.

When a rule needs to be applied, we consider it as function of multiple arguments where each argument corresponds to a connected component in the left-hand side graph of the rule. So if there are k connected components in the left-hand side, the function will have k arguments. Similar to partial function application, we can then imagine that a rule can be partially applied to a graph, resulting in a new rule, with fewer connected components in its left-hand graph. A DPO derivation can therefore be calculated by iterated binding of graphs to the rule until all arguments have been bound. See Figure 1.8 for an example.



(a) Formaldehyde

(b) Aldol addition

(c) Aldol addition with formaldehyde bound

FIGURE 1.8: A rule with 2 connected components (b). Formaldehyde (a) is applied to one of the connected components resulting in a new rule with 1 connected component (c).

Here is the algorithm for expanding the *chemical space* of the system based on initial molecules, rules and a strategy:

1. Start with a initial set of molecules, S, and a set of rules P.
2. According to some strategy, pick a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ to be applied. Let k be the number of connected components in L.
3. For each of the k connected components in L, look for a subgraph in all currently known molecules S to find an occurrence. The subgraph match is found via a monomorphism.
4. When all k connected components have been bound, transform the affected molecules based on the rule to get new molecules.
5. For each produced molecule, check if it is already present in S, for instance by checking if it is isomorphic to any molecules in S. If not, add the molecule to S.
6. This is repeated until no new molecules are created, or if the strategy specifies otherwise.

To create the actual DG, each initial molecule will have a vertex in the hypergraph. When a rule is applied, a hyperedge is drawn from all affected molecules of the left-hand side to all created molecules of the right-hand side. If a new molecule does not have a vertex in the hypergraph yet, it is created. If a molecule already has a corresponding vertex, it is reused, so each molecule only occurs once in the DG. See an example of a DG in Figure 1.9.

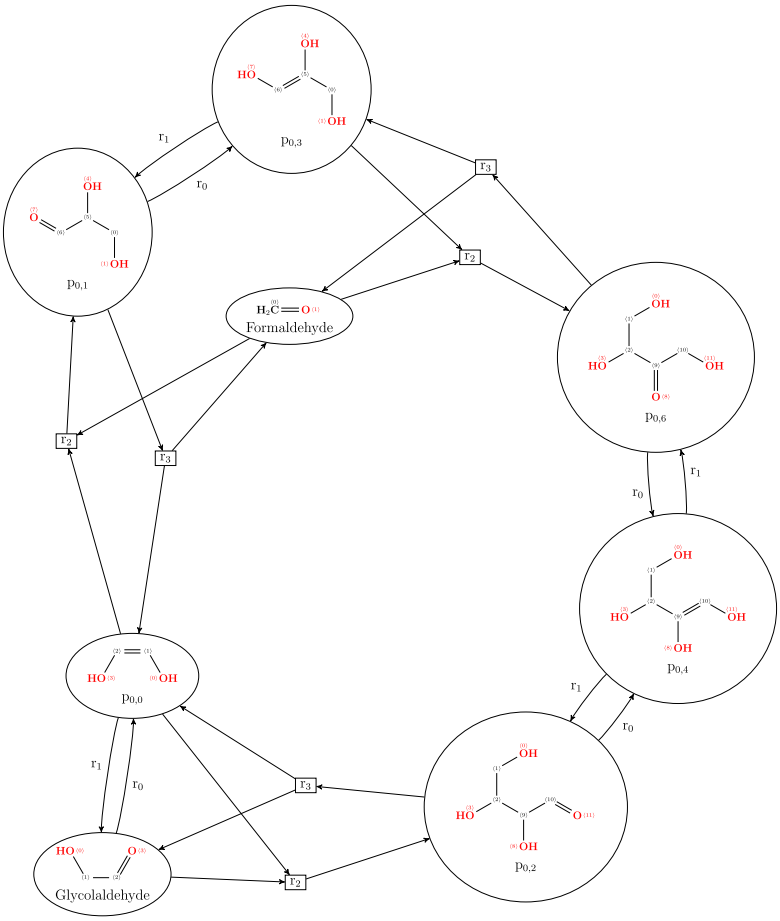


FIGURE 1.9: An example of a DG based on the Formose reaction, created based on two initial molecules, Formaldehyde and Glycolaldehyde, and two rules, Keto-Enol and Aldol (not presented here).

1.2.4 Rule Composition

There are many classes of rule composition used in the theory behind MØD, however relevant for this thesis is the *full composition*. Let $p_1 = (L_1 \leftarrow K_1 \rightarrow R_1)$ and $p_2 = (L_2 \leftarrow K_2 \rightarrow R_2)$ be rules. The full composition is applicable when L_2 is a subgraph of R_1 found via a monomorphism. The full composition $p_1 \bullet_{\supseteq} p_2 = (L \leftarrow K \rightarrow R)$ of p_1 and p_2 is given by applying p_1 to go from L_1 to R_1 and then p_2 is applied to the subgraph of R_1 matching L_2. The final result is a rule from L to R, where $L \cong L_1$ and R is R_1 with p_2 applied to it. See Figure 1.10.

Symmetrically, we define $p_1 \bullet_{\subseteq} p_2$ where R_1 is found as a subgraph in L_2 via a monomorphism.

Both $\bullet_{\supseteq}$ and $\bullet_{\subseteq}$ can refer to an arbitrary full composition or the enumeration of all such compositions depending on the context.

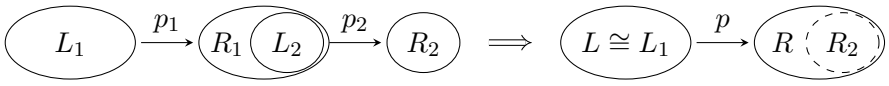


FIGURE 1.10: Abstract depiction of full rule composition $p_1 \bullet_{\supseteq} p_2$.

1.2.5 Vertex Maps

When the DG has been built, one can ask MØD for the set of *vertex maps* for a given hyperedge, e, written VertexMaps(e). A *vertex map* is one concrete mapping from the vertices of G to the vertices of H. Specifically, each internal id of the vertices of G will be mapped to an internal id of the vertices of H.

Note that this mapping is not an isomorphism since the subgraph affected by the rule can change edges (and possibly vertices in a non-chemical example). The part of the graph outside the subgraph is, however, isomorphic.

The set of all vertex maps returned by MØD is not only all possible mappings from G to H, but it also encodes all automophisms of G and of H. In other words, if G has some symmetries, each of them will contribute with a vertex map. The same goes for H.

For instance, suppose that G has a vertex with an internal id of 1 that is mapped to a vertex in H with internal id 7. Suppose further that G has a mirror symmetry such that internal id 1 becomes internal id 5. This will contribute with two vertex maps to the set of vertex maps. One where 1 is mapped 7 and one where 5 is mapped to 7.

The use of vertex maps is crucial for the approach taken in this thesis to distinguish pathways. Combined with semigroup theory it forms the basis of Chapter 2. Due to the importance of vertex maps, being able to compute all relevent vertex maps efficiently will be the topic of Chapter 3.

1.2.6 The (Sub)graph Isomorphism Problem & VF2

As described in section 1.2 it is important for the DPO approach, and therefore MØD, to be able to relatively efficiently find different types of morphisms between two graphs, G and H, in particular monomorphisms and (sub)graph isomorphisms.

The subgraph isomorphism problem as a decision problem can be expressed formally as: Given graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, is there a subgraph $H' = (V_{H'}, E_{H'})$ of H where $G \cong H'$, i.e. is there a bijection $f\colon V_G \to V_{H'}$ such that $(u, v) \in E_G \Leftrightarrow (f(u), f(v)) \in E_{H'}$?

It is known that the subgraph isomorphism problem is NP-complete [19]. The graph isomorphism problem is a special case of the subgraph isomorphism problem where $|V_G| = |V_H|$, and recent work still being peer-reviewed suggests that this problem is sub-exponential [7].

For graphs with vertices of bounded degree – like molecules in a chemical setting – the (sub)graph isomorphism problem can be solved polynomial time [17].

The graph matching algorithm used by MØD to solve graph monomorphism, subgraph isomorphism and graph isomorphism problems is the VF2 algorithm. It is based on a search tree approach that explores a search space, in a similar fashion to the search tree of Ullmann's algorithm [21]. In contrast to Ullmann's algorithm, VF2 maintains a partial mapping that is extended during the execution of the algorthm. VF2 also uses some clever heuristics to prune unprofitable paths. An example of a tree search is shown in Figure 1.11.

The pseudo code for VF2 can be seen below. Specifically, it describes its depth-first search of the state space. The pseudo code does not detail the inner workings of VF2 – it is out of the scope of this thesis. For instance, checking if $S + (n, m)$ is feasible uses some look-ahead techniques. Depending on how the feasibility is checked, VF2 can be used to find monomorphisms, subgraph isomorphisms and graph isomorphisms. Another thing that is not detailed is the Candidate Pairs Set $P(S) \subseteq V_G \times V_H$ of all pairs of vertices that needs to be tested for feasibility. These things are detailed in [10].

The version of VF2 in the pseudo code (which represents more precisely how it is implemented in MØD) takes a CALLBACK-function that is called with any newly found mapping. This callback can be used to store all morphisms found in a list or handle them however the user wants. The callback returns a boolean; true if VF2 should continue searching for matches, false otherwise. This lets the user control how many morphisms should be enumerated. This becomes very relevant for the approach described in Chapter 3.

The VF2 algorithm has a best case time complexity of $\Theta(n^2)$ and a worst case time complexity of $\Theta(n!n)$ [10], and it runs fast in practice especially on bouded-degree graphs like molecules.

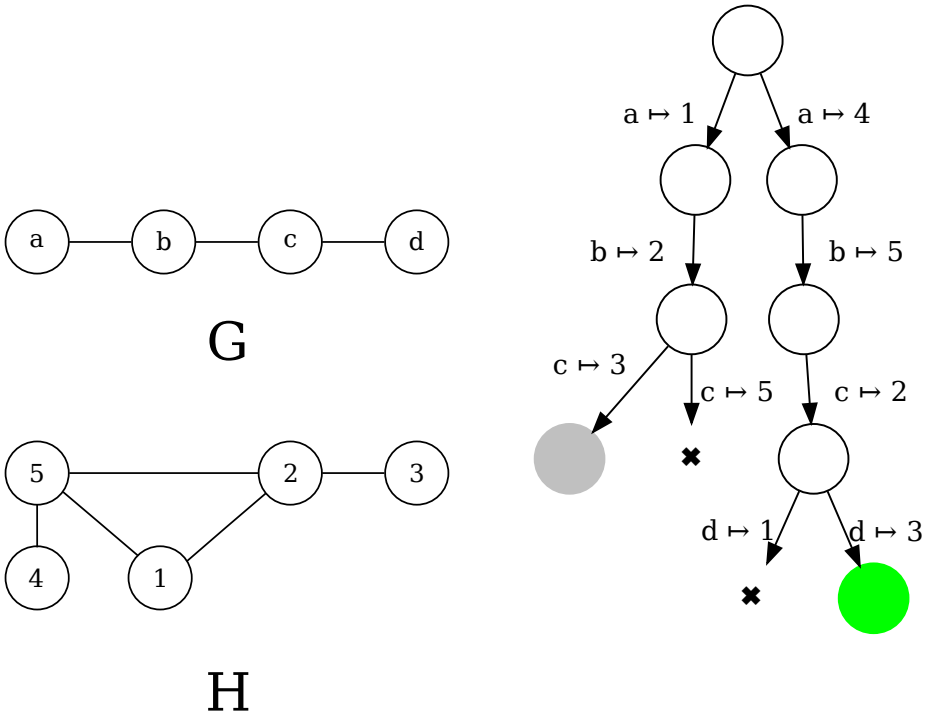


FIGURE 1.11: An example of a tree search (not necessarily VF2) for subgraph isomorphism. Nodes are states in a search tree. Assigning a to 1, b to 2 and c to 3 to c seems promising until a dead-end is reached (gray) and backtracking is done. Trying c to 5 would not be feasible (cross) because of the edge from 1 to 5, but no edge from a to c. Backtracking all the way to the top and trying to assign a to 4 finally yields a valid match (green) after some more searching.

---

function VF2MATCH(G, H, S, CALLBACK)
    // S is a stack of tuples, each representing a match
    // from a vertex of G to a vertex of H.
    if S covers all nodes of G **then**
        // Give the match to CALLBACK
        // and let it decide if the algorithm should keep searching.
        return CALLBACK(S)
    else
        for $(n, m) \in P(S)$ **do**
            if $S + (n, m)$ is feasible **then**
                // Recurse
                continueSearch $\leftarrow$ VF2MATCH(G, H, $S + (n, m)$, CALLBACK)
                if not continueSearch **then**
                    // Backtrack without trying more possibilities.
                    return False
    // No more vertices to try. Backtrack and keep searching.
    return True

function VF2(G, H, CALLBACK)
    VF2MATCH(G, H, $\emptyset$, CALLBACK)

1.3 Isotopic Labelling Experiments

As mentioned in the introduction, the study of pathways is very important in chemistry and biology, and we would like to develop tools to distinguish different pathways.

One tool that is useful in this regard is to buy start-molecules with a specific isotope labelling. Atoms consist of protons and neutrons, and the number of protons determine the chemical element, for instance carbon contains 6 protons. The most commonly found carbon atoms contain 6 neutrons, with a total weight of 12. *Isotopes* are atoms with the same number of protons, but different number of neutrons. Some noticeable examples of carbon isotopes are carbon-12 (the most commonly found isotope of carbon), carbon-13 and carbon-14 (famously used for carbon dating). See Figure 1.12. Some isotopes are unstable, meaning that they decay into other elements, and some are stable. In this thesis, we will only work with stable isotopes or isotopes that decay so slowly that one can carry out an isotope labelling experiment before the atoms decay.

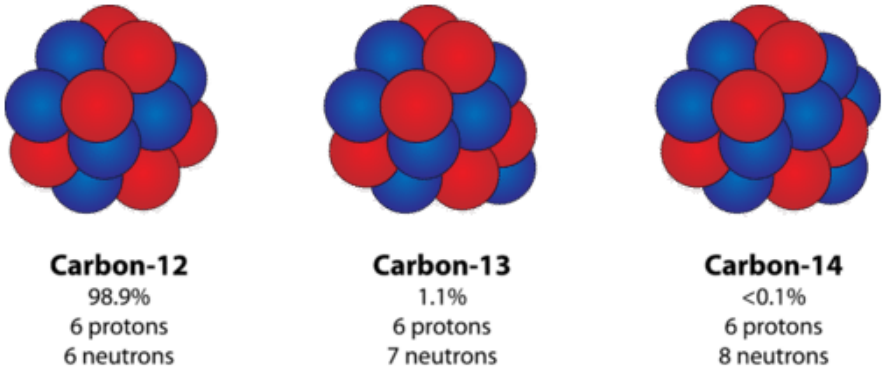


FIGURE 1.12: Three isotopes of carbon. Source: LibreTexts.

Importantly, isotopes have the same chemical properties, but different weights, meaning that tools such as the mass spectrometer can be used to distinguish isotopes based on weight. In practice, different isotopes can have slightly different chemical and physical properties in some cases [22], but we will ignore this in this thesis.

We say that a molecule is *isotope labelled* or simply *labelled* if it has one or more atoms that are isotopes different from the norm. Recall from the previous section that we represent molecules as unlabelled graphs. In this regard, one should not confuse a labelled molecule with a labelled graph. The former is the ability to distinguish some atoms from each other, the latter is the ability to distinguish all vertices uniquely from each other.

In particular, it is possible to distinguish two of the same molecules if they have labels in different positions, when ignoring symmetries. It is also important to point out that such labelled molecules are distinguished using a mass spectrometer and while it is possible based on the position of each label, it is much easier to simply distinguish by weight, since the total number of neutrons of some

molecule will be higher or lower depending on the number and type of labels. See Figure 1.13.

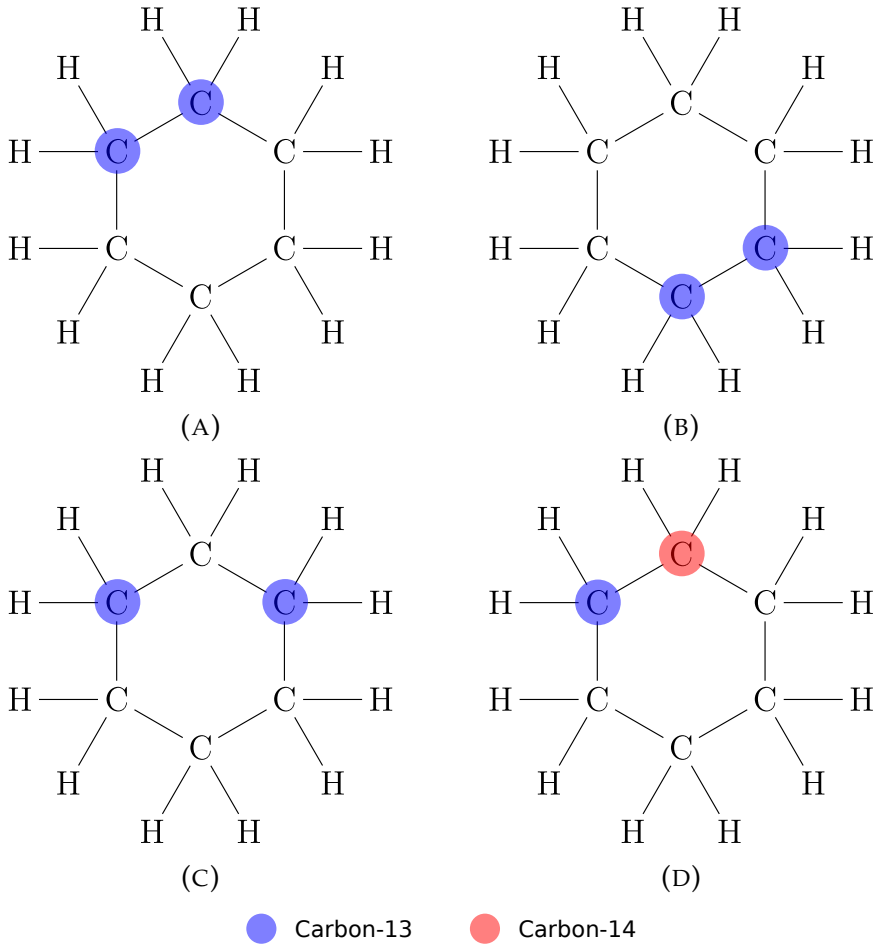


FIGURE 1.13: A Benzene-ring with different labellings. (A) is indistinguishable from (B) due to mirror or rotational symmetry. (A) is distinguishable from (C), but not by weight. (A) is also distinguishable from (D) by weight due to the carbon-14.

The idea behind isotope labelling experiments is to buy or produce isotope labelled start-molecules, for instance Glucose with a carbon-13 isotope in the place of a carbon-12 in a specific position of the molecule. Then these compounds are run through some experiment, for instance the Glucose is ingested or by other means transformed into the goal-molecules. By extracting the goal-molecules and analyzing where in the goal-molecules the isotopes ended up, it would be possible to learn something about the chemical process involved. For instance, if it is known that one of two pathways will be active in some species, and the goal-molecule reveals a large number of isotope labels in a certain position, it might be possible to conclude which of the two pathways was indeed active.

There are a couple of challenges involved with this process:

- The cost of compounds with isotope labels is orders of magnitude higher than their normal counterparts.
- It is also possible to buy compounds (possibly cheaper) with a mix of different isotope labellings, i.e. there is a distribution of various labellings. For instance, it might be possible to buy a solution where 75% of the molecules in the solution have a carbon-13 at some position and 25% of the molecules have the carbon-13 in another position.

  In this thesis, it will however be assumed for simplicity that all labelled start-molecules have the same exact labellings.
- Some labellings might give meaningful information about the pathways while some others might not. The process of buying different labelled start-molecules and trying different labelled goal-molecules in a mass spectrometer is time consuming and costly.
- Given a specific labelling, knowing where the isotopes could end up in the goal-molecules is only half the puzzle. Drawing conclusions on what pathway or pathsways are used based on that information is still challenging.

Having an automated computer system to simulate these possibilities and give an analysis could help to save time and money.

1.4 Groups and Semigroups

Some of the results of this thesis rely on some (semi)group theory, thus we will very briefly recap the basics. It is based on [12] and [16].

1.4.1 Group Basics

Definition 4. A *group* is a set G together with a operation $\bullet$. The operation takes two elements of G and returns an element of G.

The group must satisfy the following requirements:

Closure
If $g, h \in G$, then $g \bullet h$ must be in G.

Associativity
For all $g, h, k \in G$, it must hold that $(g \bullet h) \bullet k = g \bullet (h \bullet k)$.

Identity
There must exists an element $e \in G$ such that for all $g \in G$ it holds that

$$e \bullet g = g = g \bullet e$$

This element is also sometimes written as 1.

Inverse
For all $g \in G$, there must exists an element $g^{-1} \in G$ such that

$$g^{-1} \bullet g = e = g \bullet g^{-1}$$

Example 1. Let $G = \{0, 1, 2, 3\}$ and $\bullet$ is addition modulo 4, i.e. $(\_ + \_) \bmod 4$.

We check a couple of the properties with examples:

Identity
From normal addition, we know that 0 is the identity element that does not change anything when added with other elements. We try with 2.

$$(0 + 2) \bmod 4 = 2 \bmod 4 = 2$$
$$(2 + 0) \bmod 4 = 2 \bmod 4 = 2$$

Inverse
From normal addition, we know that $-x$ is the inverse of x. But negative numbers are not part of the G. Luckily, there exists a corresponding positive number that serve as the inverse. Here is an example with 1 and -1:

$$(1 + (-1)) \bmod 4 = (1 + 3) \bmod 4 = 4 \bmod 4 = 0$$

Closure

We try a couple of different pairs of elements of G to see if they are closed.

$$(1+1) \bmod 4 = 2 \bmod 4 = 2 \in G$$
$$(2+3) \bmod 4 = 5 \bmod 4 = 1 \in G$$

The mod makes elements "wrap around" so it is always the case.

Definition 5. The *order* of a group G, denoted $|G|$, is the number of elements of G.

Example 2. Let $G = \{0,1,2,3\}$ and $\bullet$ is addition modulo 4. The order of G is 4 since $|G| = 4$.

Often we do not store nor specify all elements of the group explicitly.

Definition 6. A *generating set* S is a subset of G such that every element of G can be expressed as combination of elements of S and their inverses. We write $G = \langle S \rangle$. If $S = \{s_1, s_2, \ldots, s_k\}$, it is also common to write $G = \langle s_1, s_2, \ldots, s_k \rangle$.

Example 3. Let $G = \{0,1,2,3\}$ and $\bullet$ is addition modulo 4. G can also be expressed as $\langle 1 \rangle$ since

$$(1+1) \bmod 4 = 2 \bmod 4 = 2$$
$$(2+1) \bmod 4 = 3 \bmod 4 = 3$$
$$(3+1) \bmod 4 = 4 \bmod 4 = 0$$

Thus we were able to generate the entire set. Note that for instance $\langle 2 \rangle \neq G$ since

$$(2+2) \bmod 4 = 4 \bmod 4 = 0$$
$$(0+2) \bmod 4 = 2 \bmod 4 = 2$$

A quite central topic in group theory is the study of subgroups:

Definition 7. H is a subgroup of G, written as $H \leq G$, if H is a subset of G and H forms a group under the same operation.

Example 4. Let G and $\langle 2 \rangle$ be given from Example 3. $\langle 2 \rangle$ is a subgroup of G. It is a subset of G and it is a group on its own. Closure is shown in the Example 3.

What follows is a couple of interesting lemmas and theorems about groups. Proofs are left out as it is outside the scope of this thesis.

Theorem 1 (Lagrange's Theorem)**.** Let H be a subgroup of G. $|H|$ divides $|G|$.

Example 5. Consider again Example 3. We have $|\langle 2 \rangle| = 2$ and $|G| = 4$, and indeed 2 divides 4.

Definition 8. Let H be a subgroup of G. A (right) *coset* of H w.r.t. $g \in G$ is given by

$$Hg = \{h \bullet g \mid h \in H\}$$

Note that cosets are not necessarily subgroups.

Example 6. Consider again Example 3. Let $H = \langle 2 \rangle = \{0, 2\}$ and

$$H1 = \{(0+1) \bmod 4,\ (2+1) \bmod 4\} = \{1, 3\}$$

$H1$ is not a group, since $(1+3) \bmod 4 = 0 \notin H1$. H and $H1$ cover the entirety of G and partition G into equal-sized, non-overlapping sets.

1.4.2 Permutation Groups

Permutation groups are groups where the elements are permutations and the operation is function composition.

The usual notation is that we have a group G of permutations that *act on* a set of *points* Ω. The set of points can be anything, but we usually normalize it to be $\Omega = \{1, 2, \ldots, n\}$.
A permutation in this context is a bijective function σ that given a point from Ω maps it to a point of Ω. One standard notation for a permutation is the two-line notation:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ 3 & 4 & 1 & \cdots & 42 \end{pmatrix}$$

The above is read as "1 goes to 3, 2 goes to 4," and so on. Another more compact notation is the cyclic notation:

$$\sigma = (13)(245)\cdots$$

It is read as "1 goes to 3, 3 goes to 1. 2 goes to 4," and so on. Any elements left out are mapped to themselves. This notation will be preferred in this thesis.

Notation 2. Let $\omega \in \Omega$ and $g \in G$. g acting on ω is written as $g(\omega)$.

Definition 9. The *orbit* of a point $\omega \in \Omega$ is all the points that can be reached via group actions. We write it as

$$\text{Orbit}_G(\omega) = \{g(\omega) \mid g \in G\} \subseteq \Omega$$

Example 7. Let $\Omega = \{1,2,3,4,5\}$ and let

$$G = \left\langle \underbrace{(1\ 2)(3\ 4)}_{g_1}, \underbrace{(2\ 5)}_{g_2} \right\rangle$$

Suppose we want to find the orbit of 1.
All elements of G can be explicitly stated

$$\begin{aligned} G = \{\ &(), (1\ 2)(3\ 4), (2\ 5), (3\ 4), (2\ 5)(3\ 4), (1\ 2), (1\ 2\ 5), \\ &(1\ 2\ 5)(3\ 4), (1\ 5\ 2), (1\ 5\ 2)(3\ 4), (1\ 5), (1\ 5)(3\ 4)\} \end{aligned}$$

Now one can go through each element and see where 1 is mapped for each. It turns out that the orbit of 1 is

$$\text{Orbit}_G(1) = \{1,2,5\}$$

It is also possible find orbits of tuples and sets:

Definition 10. Let $t = (\omega_1, \ldots, \omega_k)$ be a tuple of size k of points from Ω.

$$\text{Orbit}_G(t) = \{(g(\omega_1), \ldots, g(\omega_k)) \mid g \in G\}$$

The orbit of sets is defined similarly with $\{g(\omega_1), \ldots, g(\omega_k)\}$, or equivalently by considering the elements of each tuple to be sorted.

There are some more advanced group theory ideas that will be presented. They have not been used in this thesis, but could be useful for potential future work.

Definition 11. The *stabilizer* of a point $\omega \in \Omega$ is all group elements that do not move ω. We write it as

$$\text{Stab}_G(\omega) = \{g \mid g \in G,\ g(\omega) = \omega\} \leq G$$

While *stabilizers* and *stabilizer chains* are not used in this thesis, they are already used by MØD to do graph canonicalization and might have many more uses.

Lemma 1. Let G be a group that acts on Ω and let $\omega \in \Omega$. Then

$$|\text{Orbit}_G(\omega)| \cdot |\text{Stab}_G(\omega)| = |G|$$

Lemma 2 (Schreier's Lemma)**.** Let H be a subgroup of $G = \langle S \rangle$. Let R be a set of representatives of the cosets of H. Finally, for $g \in G$, denote $\overline{g}$ the representative of the coset Hg. Then H is generated by

$$\{rs(\overline{rs})^{-1} \mid r \in R, s \in S\}$$

There is also an algorithm called the *Schreier-Sims Algorithm* that uses Schreier's Lemma and many of other ideas from group theory to preprocess the group in polynomial time such that membership testing (determining if an object is an element of a group) can be done efficiently.

1.4.3 Transformations & Semigroups

We will use group theory to model reactions and the chaining of reactions. More on this in Chapter 2. It turns out that a reaction resembles a permutation except for one fact: Some reactions do not (or are very unlikely to) go backwards, i.e. they lack an inverse. Therefore reactions will not be represented by permutations, but by transformations.

Definition 12. A *transformation* t is a function that maps a set Ω to itself, i.e. $t \colon \Omega \to \Omega$.

When $\Omega = \{1, \ldots, n\}$ for some n, the following notation is convenient:

Notation 3. A transformation t can be written as a list of length n where the value of the i'th position corresponds to $t(i)$.

Example 8.

$$1 \mapsto 2$$
$$2 \mapsto 1$$
$$3 \mapsto 3$$

can be written as

$$[2, 1, 3]$$

The index of a position in the list can be specified with a number above:

$$[\ldots, \overset{4}{2}, \ldots]$$

The goal is to create a "group of transformations" where the elements of the group are transformations instead of permutations. But since elements of groups are required to have inverses, this is not well-defined. Luckily, such groups without inverses exists under the name *semigroups*:

Definition 13. A *semigroup* is a set G together with a operation $\bullet$. The operation takes two elements of G and returns an element of G.

The semigroup must satisfy the following requirements:

Closure
If $g, h \in G$, then $g \bullet h$ must be in G.

Associativity
For all $g, h, k \in G$, it must hold that $(g \bullet h) \bullet k = g \bullet (h \bullet k)$.

Note that there is no guarantee for an element to have an inverse, and there is no guarantee of the existence of an identity element.

It is now well-defined to think about a *semigroup of transformations* where the elements are transformations and the operation is function composition.

While some properties of groups do not hold for semigroups, others do. For instance, the notion of orbit is exactly the same.

Many problems involving semigroups of a finite set of generators are hard. It can be shown that determining any "sensible" property of such semigroups is undecidable and that membership testing in commutative transformation semigroups is NP-complete. [11] This is however in a general case and in practice, especially with the semigroups we will work with in this thesis, some things like orbits can be computed quite efficiently.

1.5 Overview of Chapters

Chapter 2 introduces an approach for atom tracing called the Hypergraph-Semigroup approach. It uses semigroup theory, specifically orbits, to automatically infer tables that can be used for the design of isotope labelling experiments.

Chapter 3 shows how computing vertex maps – an essential part of the Hypergraph-Semigroup approach – can be made more efficient by using a direct approach instead of rule composition. It takes advantage of the fact that hydrogens can be ignored in many chemical examples to reduce combinatorics.

Chapter 4 gives some concluding remarks, and it presents potential future work as well as some alternative approaches with some initial thoughts and remarks.

2

The Hypergraph-Semigroup Approach

2.1 Introduction

The overarching goal is to be able to distinguish different pathways supplied by the user. To do this, we will use semigroup theory to analyze each pathway and present the results as tables that can assist the scientists when they carry out isotope labelling experiments in laboratory.

The first step is to construct a Hypergraph-Semigroup from a DG. This semigroup can be used to compute orbits. And based on orbits, it is possible to create a Pathway Table where rows are pathways, columns are possible candidates for atom labelling and the entries are orbit calculations. This table can be simplified and can be used to make educated decisions on what isotope labelling experiments to carry out in real-life to distinguish pathways.

If this is not enough, one can create Pathway Comparison Tables that compares two pathways directly to give more fine-grained details.

This approach is discrete and deterministic, thus it will not give probabilities of the various pathways, but rather clear signs when pathways are unambiguously distinguishable.

All of the theory in this chapter has resulted in a Python framework which is presented at the end of the chapter. It can be used in conjunction with the Python interface for MØD (PyMØD) to aid the design of isotope labelling experiments.

2.2 Construction

The idea is this: Consider some DG consisting of multiple molecules and reactions. Each molecule itself consists of multiple atoms. Suppose we collect all

atoms together and give them ids them from 1 to n, where n is the total number of atoms in the system.

We call set of all atom ids $\Omega = \{1, 2, \ldots, n\}$. The semigroup G we construct will act on these points and keep in mind that these points correspond to atoms of the DG.

The elements of G will be transformations corresponding to the reactions of the DG. For instance, if an atom 3 from one molecule maps to 7 from another molecule, then part of the transformation will be

$$3 \mapsto 7$$

We are now ready to define this formally:

Definition 14. An *atomic linearization* of a DG is a list $[1, 2, \ldots, n]$ of ids corresponding to the atoms of molecules one after another. n is the total number of atoms in the system.

Note: The linearization ids are in the range of 1 to n instead of 0 to $n - 1$, since they will be used in a semigroup where the tradition is to use the range 1 to n.

In the rest of the section, we will assume that we, for a given DG, always have access to some atomic linearization, Ω. These will be the points that our semigroup will act on.

Important: Every atom in the DG now has two ids: The *internal id* of the molecule where the atom resides and a *linearization id* just introduced. As mentioned in the Preliminaries, the internal id are the ids that will be printed by MØD and are therefore useful for referring to individual atoms. The internal ids are, however, not unique across molecules since all internal ids are in the range $\{0, \ldots, m - 1\}$, where m is the size of some molecule. Linearization ids are arbitrarily assigned, but unique across the entire DG, which is needed to create a semigroup of transformations.

The type of ids is usually clear from the context, however sometimes it will be specified explicitly.

Next, we create the semigroup based on the reactions of the DG:

Definition 15. The *Hypergraph-Semigroup* of a derivation graph $H = (V, E)$ is a semigroup $G = \langle S \rangle$ acting on Ω, where

$$S = \bigcup_{e \in E} \text{VertexMaps}(e)$$

In other words, every possible vertex map among all reactions become a generator of the semigroup. See Figure 2.1 for an example of a DG and vertex map.

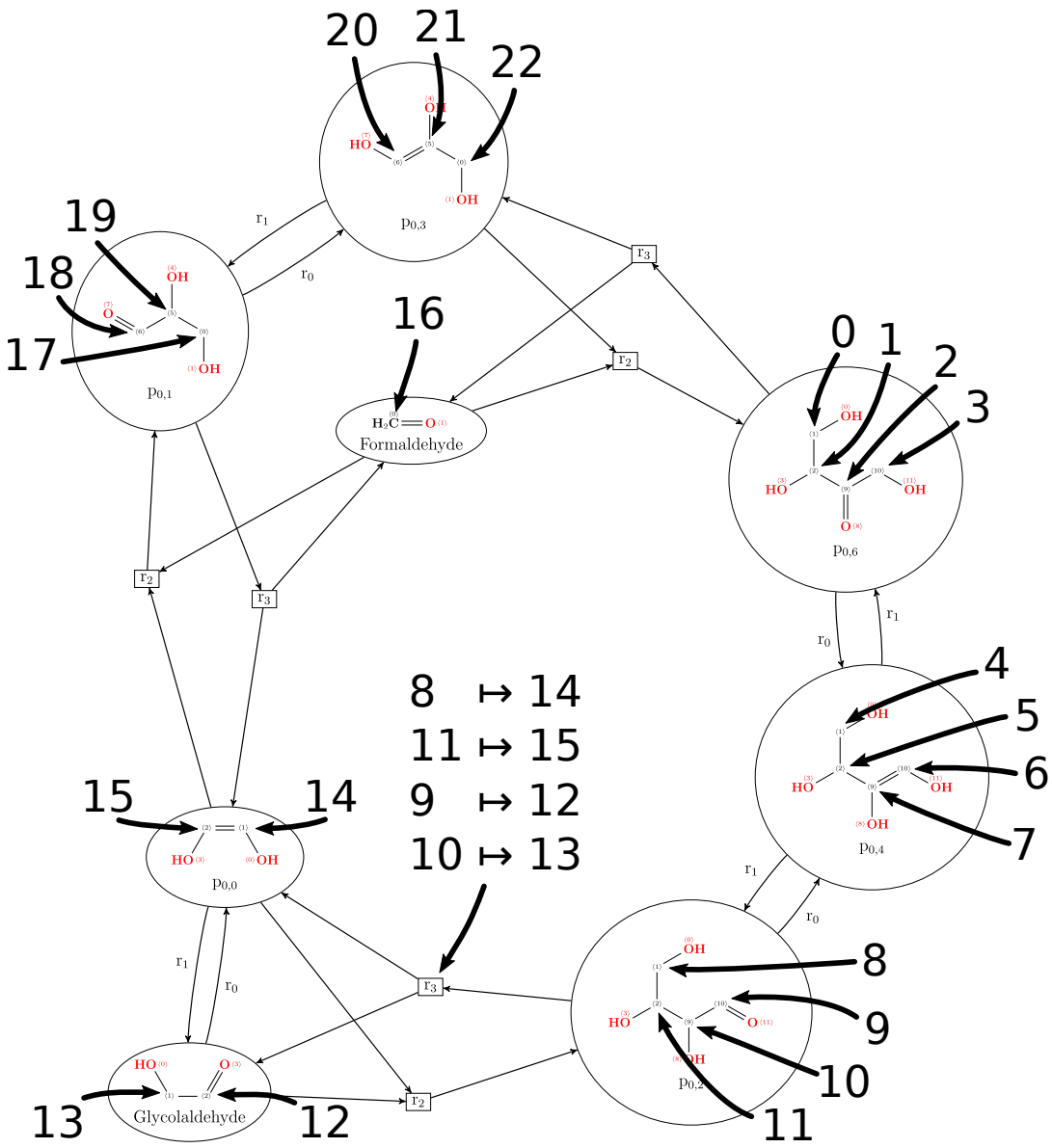


FIGURE 2.1: Part of the Formose reaction. A linearization of the carbons is found. One transformation of VertexMaps(r_3) is shown. This is done for all reactions.

2.3 Orbits

Since we now have a semigroup, we can rely on all the semigroup theory that we know. One tool that will prove useful is the notion of orbit as discussed in the Preliminaries.

Given a Hypergraph-Semigroup G and a single atom a in a start-molecule, we can start by asking the following question: What does $\text{Orbit}_G(a)$ represent? Since the orbit represents all points that some initial point can go to, $\text{Orbit}_G(a)$ represents all the atoms in the DG that a can go to. See Figure 2.2.

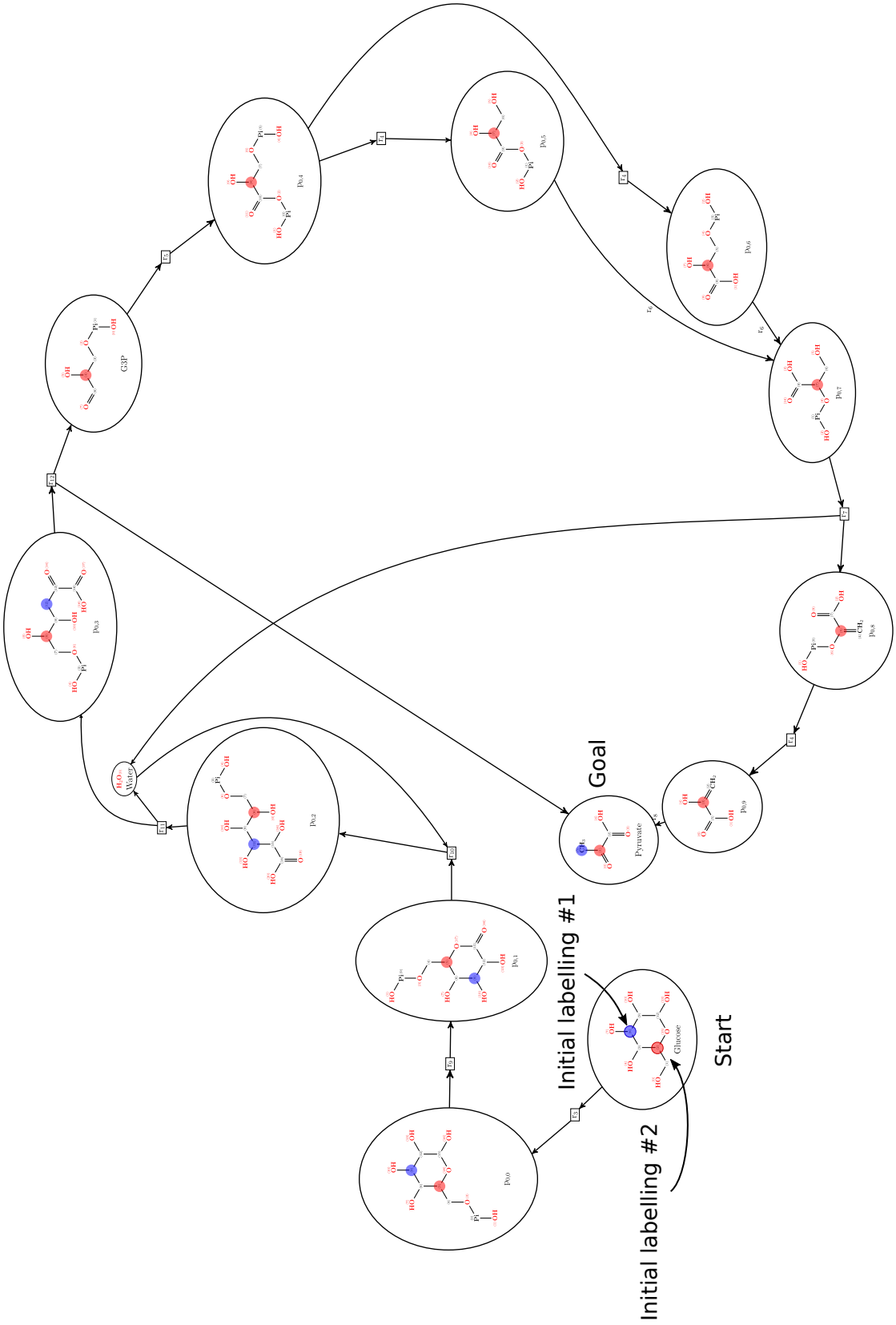


FIGURE 2.2: Glycolysis, ED pathway. Two different orbits. Labelling id 6 (blue) in Glucose (start) yields a label at id 5 in Pyruvate (goal). Labelling id 2 (red) yields a label at id 1.

It does *not* encode time or multiplicities of molecules. Keep in mind that there is no guarantee that any molecule with any combinations of labellings is observed in real-life experiments just because the orbit would suggest it.

The orbit is simply a map of *what-could-happen*, and in the case of very symmetric molecules, for instance the ones in the Formose reaction from Figure 2.1, the orbit would cover most or all the atoms in the DG. Thus, at a glance this approach would not seem very insightful, but utilized in the right way, this approach is very simple to carry out and at the same time very powerful. It is just important to be aware of what orbits can and cannot do.

Suppose for instance that some carbon with linearization id 1 in the start-molecule has the orbit of $\{8, 11, 12\}$ in the goal-molecule. If a carbon is labelled at id 1 and a goal-molecule is ever observed in real-life with a label at id 13, then we can conclude that the model is wrong – the DG does not model all pathways or has some flaw – otherwise the orbit would have included 13. We can generalize this to a hypothesis:

Hypothesis 1. Let G be a Hypergraph-Semigroup of some DG and let a single atom k be labelled. Suppose a molecule with a label at id i is observed in the laboratory. If $i \notin \text{Orbit}_G(k)$, then the DG does not correctly describe the events happening in the laboratory.

We phrase it as a hypothesis, similar to the Church-Turing hypothesis, because it lies on the boundary of theory and reality, and it cannot be proven. One might for instance observe labels that should not be possible due to other chemical or physical factors beyond the model.

In a sense, the Hypergraph-Semigroup includes all possibilities, more than reality, but we also know that we can exclude everything that the Hypergraph-Semigroup does not include.

Orbits of pairs and sets

Another important thing is the fact that orbits can be computed for tuples and sets. Computing the orbit of (a, b) – where a and b are atoms of start-molecule – represents where a and b can go "in sync". If we imagine the orbit computation as each a and b jumping from one atom to another following the atom tracing, then we would see them jump in lock-step. While this is a very good way to intuitively understand orbits of pairs, it can be deceiving at times. Consider Figure 2.3.

Here we have a pair corresponding to two circles connected with a line. Following the hyperedges Glucose $\to p_{0,0} \to p_{0,1} \to p_{0,2} \to p_{0,3}$, we see that the labels move together from molecule to molecule. Then $p_{0,3}$ splits into a Pyruvate and G3P and the labels get separated. One of them gets to its spot at id 5 in the Pyruvate and stays there, while the other goes around from $p_{0,4}$ to $p_{0,9}$ and ends up at id 1 in the Pyruvate (in real-life, this is of course a different Pyruvate-molecule, but using orbits does not reveal this).

One could think that since the first label to reach the Pyruvate has nowhere to go, the second would never reach the Pyruvate because one would expect labels to always "jump" to new molecules. This is however not the case.

This is because when computing the transformations for the reaction G3P $\rightarrow$ $p_{0,4}$, the vertex maps of the hyperedge are turned into transformations. The transformations describe how atoms of G3P are mapped to $p_{0,4}$ while the rest of the atoms are unchanged. Thus the label at id 5 in the Pyruvate will simply be mapped to itself.

When the orbit of a set $\{a, b\}$ is computed it is very similar to the orbit of (a, b), but order of a point does not matter. So $\{a, b\} \rightarrow \{c, d\}$ is the same as $\{a, b\} \rightarrow \{d, c\}$. The result is an orbit set that is either the same as with tuples (which is the case of Figure 2.3) or smaller, because some points end up being equivalent.

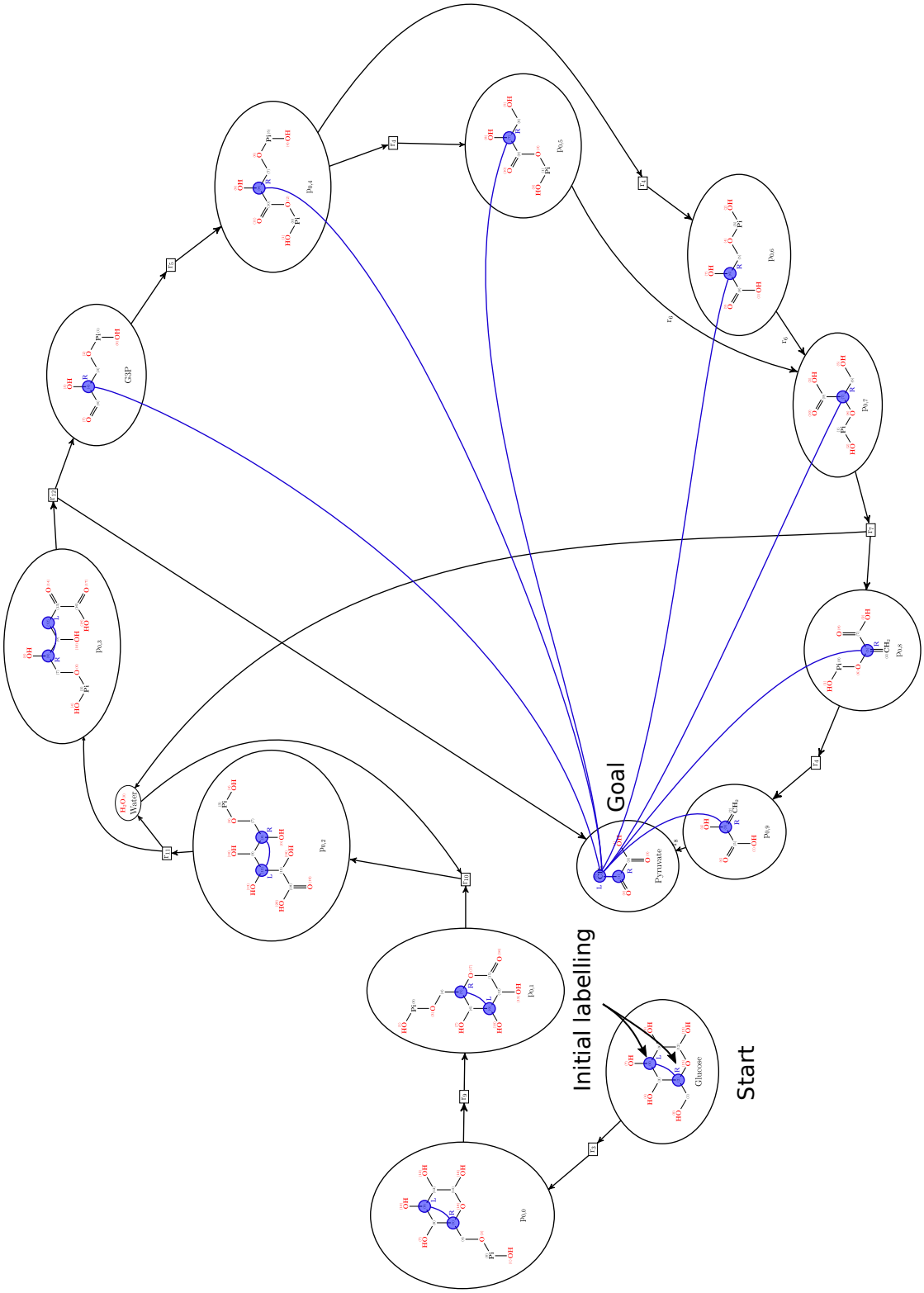


FIGURE 2.3: Glycolysis, ED pathway. Orbit of tuple $(6,2)$ from Glucose (start). Tuple (l,r) in the orbit is represented with two circles connected with a line. l of the tuple is indicated with a little L. Similarly, R for r.

2.4 Inverted Orbits

Another key insight is that we can compute inverted orbits which we can intuitively understand as computing orbits from the goal-molecules backwards to the start-molecules.

Suppose we go back Figure 2.1 and let Glycolaldehyde be the start-molecule and $p_{0,6}$ be the goal-molecule for the sake of the example. Suppose further that an isotope labelling experiment is run with linearization id 13 labelled and $p_{0,6}$ is observed with a label at linearization id 0, 2 and 3. Without even thinking about pathways, it would be desirable to be able to tell if the model is correct – should it even be possible to observe this labelling of $(0, 2, 3)$?

Using the inverted Hypergraph-Semigroup, we can answer in case it is *not*.

Definition 16. Let $G = \langle T \rangle$ be a Hypergraph-Semigroup.
The *inverted Hypergraph-Semigroup* G^{-1} of G is a semigroup $G^{-1} = \langle X \rangle$ where

$$X = \{t^{-1} \mid t \in T\}$$

Each t is a vertex map converted into a transformation.

Intuitively, we can think of the inverted Hypergraph-Semigroup as all generators (transformations) being inverted. Inverting a transformation t means that for each $x \mapsto t(x)$, t^{-1} would have $t(x) \mapsto x$. For this to be well-defined, t must be injective. Luckily, all vertex maps are bijective in a chemical setting.
The inverted orbit of x with respect to some Hypergraph-Semigroup G is therefore $\text{Orbit}_{G^{-1}}(x)$.

We return to the question of whether it should be possible to observe a labelling of $(0, 2, 3)$ starting from the label 13. This can now be phrased as the question: Is $(13, 13, 13)$ in $\text{Orbit}_{G^{-1}}((0, 2, 3))$? This is depicted in Figure 2.4.

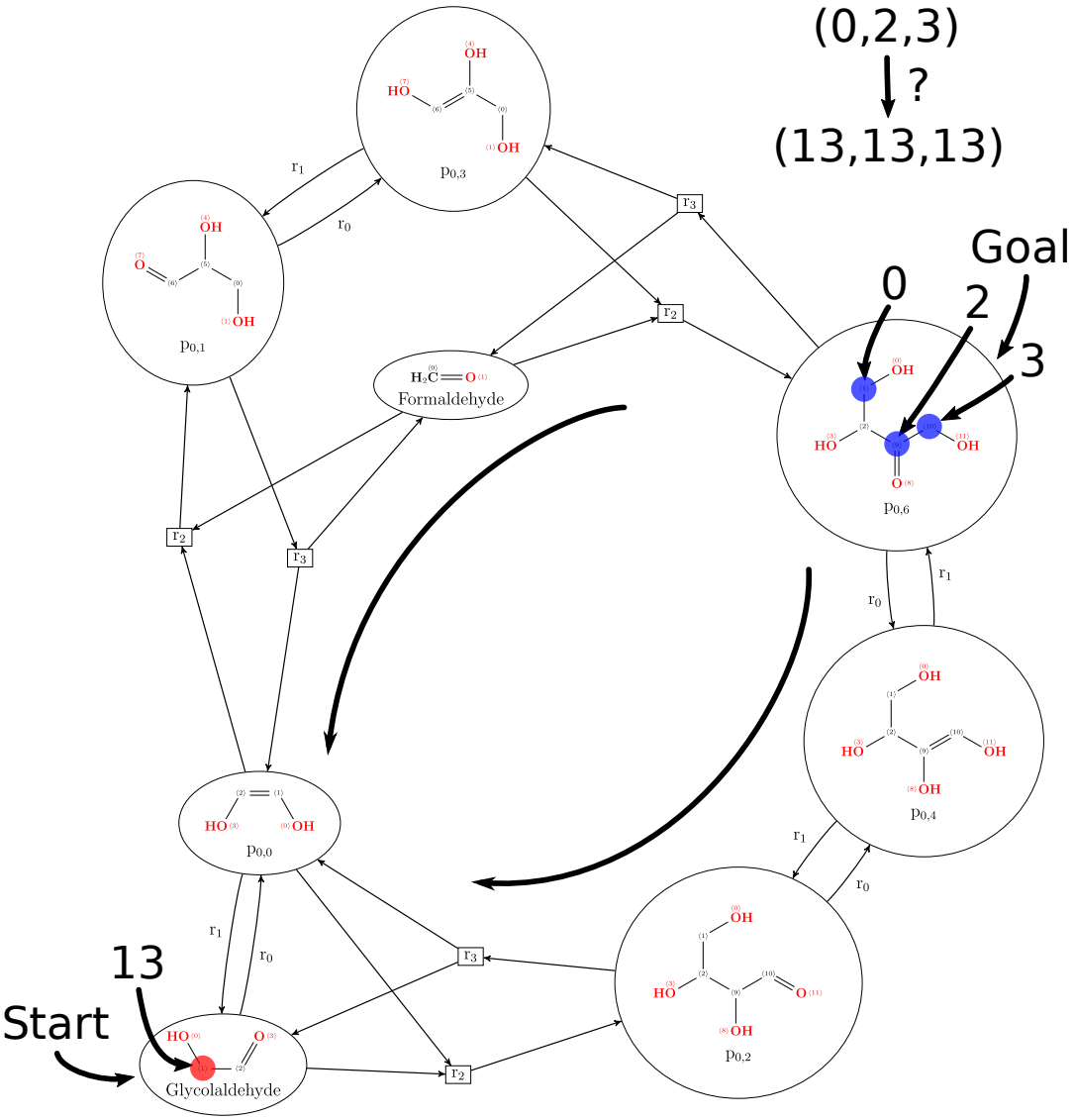


FIGURE 2.4: Compute the orbit backwards for $(0,2,3)$ to see if it could have originated from $(13,13,13)$.

If $(13,13,13)$ is not in the orbit, there is no way someone running the experiment should be able to observe it. If they do, it must be an error in the model. If $(13,13,13)$ is in the orbit, there is no guarantee that it would be observed, it is just a *possibility*.

We can formulate this as a hypothesis:

Hypothesis 2. Let G be a Hypergraph-Semigroup of some DG and let a single atom k be labelled. Suppose the labelling $(i_1,\ldots,i_n)$ is observed in the laboratory. If $(k,\ldots,k) \notin \text{Orbit}_{G^{-1}}((i_1,\ldots,i_n))$, then the DG does not correctly describe the events happening in the laboratory.

It is possible to define a general hypothesis for multiple labellings, but labelling only one is usually sufficient and all we need in the remainder of the chapter.

Hypothesis 2 is the key insight that we can use to distinguish pathways. If some atoms have a similar orbit in two different pathways, using the inverted orbit can sometimes disambiguate the two.

2.5 Pathway Table

Suppose we have a set of pathways, P. Each pathway starts with molecule S and ends with molecule T. Let S_V be the set of atoms in S. We also have a set of atoms $A \subseteq S_V$ that are candidates for isotope labelling. Consider creating the following $P \times A$ table, where each entry $(p, a) \in P \times A$ is the subset of $\text{Orbit}_{G_p}(a)$ that reaches the goal-molecule T, for each Hypergraph-Semigroup G_p of a pathway $p \in P$.

Such a table could look like this:

Pathway \ Atom label	1, C	2, C	3, C
1	0, 2	1	0, 2
2			0, 1, 2
DG	0, 2	1	0, 1, 2

The columns are the internal ids (not linearization ids) of A, namely the carbon atoms of S. The atom type is also shown to ease the reading of the table. The rows are the two pathways of P as well as a union of all pathways called DG. In each entry we see a comma-separated list of internal ids (not linearization ids) that correspond to the T-orbit of the specific column, in other words, the part of the orbit that reaches the goal-molecule T. The reason that the internal ids are used is because it matches the ids shown in the images created by MØD making it much more useful for the user.

We will use the following notation to easier relate the ids to where they are coming from: i_S means the carbon with internal id i in start-molecule S. For example: 1_S corresponds to the carbon with internal id 1 in S, i.e. the column from the table above called "1, C". We use a similar notation for i_T for ids in goal-molecule T.

Remember: The goal is to be able to carry out one or more isotope labelling experiments and then be able to distinguish the different pathways of P. Using a mass spectrometer, one can detect isotope labels based on weight or position. It is much easier to detect whether some molecule has zero or two labels (weight) compared to determining where the labels are in the molecule (position). In particular, it is easy to determine if there are no labels versus if there are some labels. Therefore it is preferred to look for weight-distinguishing factors over positions. In the table above, we see such an example. If someone carries out an isotope labelling experiment where carbon 1_S of S is labelled and any labels at all are observed in T then they know that pathway 2 cannot have been active.

We can also introduce another small feature that is sometimes useful: Consider the orbit of a single labelled atom x_S in S. The orbit represents everywhere x_S potentially could go in the DG. Suppose the DG has dead-ends, i.e. vertices with no outgoing edges; also called sinks. If the orbit of x_S includes atoms of such a dead-end molecule this could mean that in real-life the isotope labellings could leave the system and never go to the goal-molecule(s). Therefore, if the orbit of x_S *does not* include atoms of any dead-end molecules, it gets marked with a star which should give an indication that no matter where the labels go, they stay in the system.

In the Python framework, it is also possible to add custom dead-ends, for instance if helper molecules in reality are dead-ends, but it is not evident from the DG.

A table with stars could look like the following:

Pathway \ Atom label	1, C	2, C	3, C
1	0, 2	1 (*)	0, 2 (*)
2			0, 1, 2 (*)
DG	0, 2	1	0, 1, 2 (*)

This table would tell us that labelling 2_S in pathway 1 would guarantee that we would observe 1_T since we know

- The label could end up at 1_T or leave the system based on the orbit.
- The label cannot leave the system based on the star.

In contrast, labelling 1_S in pathway 1 would result in a molecule T with a label at either 0_T or 2_T, or both 0_T and 2_T, or neither.

Now consider the following table and suppose we want design isotope labelling experiments to determine what pathway is active.

Pathway \ Atom label	0, C	1, C	2, C
1	0, 1, 3, 4	2	1, 3
2	0, 1, 2, 4	3	2, 4
3	1, 2, 3, 4	0	1, 2, 3, 4
4	0, 1, 2, 3, 4		
DG	0, 1, 2, 3, 4	0, 2, 3	0, 1, 2, 3, 4

Firstly, trying a with the label 1_S, we know that if any labels at all are observed, it would exclude pathway 4. Furthermore, if that observation includes something different than 2_T, it excludes pathway 1; different from 3_T, it excludes pathway 2; and different from 0_T, it excludes pathway 3. There are however still a lot of possibilities not handled yet.

Consider labelling 0_S. Each entry has a lot of ids in common with the rest of the entries, but it can still be used. For instance, observing 3_T would exclude pathway 2. Another idea is to consider inverted orbits. Consider a pair of points

that pathway 1 and 3 have in common, for instance $(1_T, 3_T)$. Now compute the inverted orbit for this pair in both pathway 1 and 3:

$$\text{Orbit}_{G_{p_1}^{-1}}((1_T, 3_T)) = \{\ldots, (0_S, 2_S), (2_S, 0_S)\}$$
$$\text{Orbit}_{G_{p_3}^{-1}}((1_T, 3_T)) = \{\ldots, (0_S, 2_S), (2_S, 2_S), (0_S, 0_S), (2_S, 0_S)\}$$

The "..." represents the part of the orbit not in molecule S; it is not relevant for the analysis. We know that we started with a label only at 0_S, thus both elements of the tuple $(1_T, 3_T)$ must have originated from 0_S. The only of the two pathways where this could have happened is pathway 3 because of the $(0_S, 0_S)$. Therefore, if we ever observe $(1_T, 3_T)$, we can exclude pathway 1.

This idea of looking at differences in inverted orbits can be used to create a Pathway Comparison Table that is a more powerful tool when ordinary orbits do not give useful results.

2.6 Simplifying the Pathway Table

Before the idea of considering Pathway Comparison Tables, it is worth pointing out that Pathway Tables can become big, especially horizontally due to the number of atoms in the start-molecule one wants to find the orbit of. To take the first steps towards reducing the size of the table some simple simplification rules are used.

Recall from section 1.3 that different labelled molecules are distinguishable based on the position of the labels using a mass spectrometer, but it is bit challenging. Labelled molecules are also distinguishable simply based on the weight of the molecules which will differ based on the number of labels.

The first rule is to remove columns that are not informative. In this sense a column can either be *position-non-informative* or *weight-non-informative*.

A column is position-non-informative if all its entries are identical. In a sense, this means that the atom corresponding to the column does not tell us anything useful, since it will behave the same in all pathways. See Table 2.1.

A column is weight-non-informative if all its entries have the same size, meaning that there is a possibility for the goal-molecules to have the same weight no matter the labelling. The idea behind this is to allow simplifying the Pathway Table such that it would only be necessary to perform simple weight measurements in contrast to finding the exact positions of the labels.

Pathway \ Atom label	...	i, C	...
1	...	1, 2, 3	...
2	...	1, 2, 3	...
3	...	1, 2, 3	...
⋮	...	...	...
DG	...	1, 2, 3	...

TABLE 2.1: A Pathway Table with an all-equal column that could be removed.

2.7 Pathway Comparison Table

As discussed in Section 2.5, using inverted orbits can also be used to exclude certain pathways. In this section we will introduce a table that shows all these cases where if a certain tuple of labels is observed, it can exclude one of two pathways.

Given a set of pathways, P, a specific column (atom) a from a Pathway Table T, and an integer k, we create a $P \times P$ table where the rows and columns are pathways.

For each entry (p_i, p_j), the algorithm to compute the entry is as follows:

- Initialize entry (p_i, p_j) to be an empty list.
- Let $O_i = T[p_i, a]$ and $O_j = T[p_j, a]$.
  O_i is the subset of $\text{Orbit}_{G_{p_i}}(a)$ that reaches the goal-molecule(s). Similarly for O_j.
- Let $O = O_i \cap O_j$. We only care about what they have in common, i.e. the part that we cannot distinguish by position.
- Let $[O]^k$ be the set of all subsets of O of size k.
- For each $t \in [O]^k$ (seen as a tuple):
  - Compute $d_i = \text{Orbit}_{G_{p_i}^{-1}}(t)$.
  - Compute $d_j = \text{Orbit}_{G_{p_j}^{-1}}(t)$.
  - If d_i contains $(\underbrace{a, \ldots, a}_{k})$ and d_j does not, add t to (p_i, p_j).

This is essentially the same as what was done in Section 2.5, but in an exhaustive manner. Using this algorithm on the table at the end of Section 2.5, $a = 0_S$ and $k = 2$, we get the following table:

Contains \ Not contains	1	2	3	4	DG
1					
2					
3	$(1,3)$	$(2,4)$			
4	$(1,3)$	$(2,4)$			
DG	$(1,3)$	$(2,4)$			

As we already discovered by hand in Section 2.5, observing a pair $(1,3)$ actually let us distinguish pathway 1 from pathway 3. Likewise, we can now also see that $(1,3)$ would also distinguish pathway 1 and 4.

Trying different values of a and k can give more insight about what isotope labelling experiments to perform to distinguish different pathway. And there is still a lot to explore about Pathway Tables and Pathway Comparison Tables. Finally, it is still an open problem to design an algorithm that uses the Pathway Table and different Pathway Comparison Tables to give a full isotope labelling experiment plan that always gives the best way to carry out experiments to distinguish pathways in all cases.

Despite this, it is still possible in many cases to use these tables to learn interesting properties of the pathways and create sufficiently good plans by inspecting the tables manually.

2.8 Examples & Results

TCA

The TriCarboxylic Acid (TCA) cycle is a series of reactions that are very important in the study of metabolism, since it describes how stored energy is released as ATP in all aerobic organisms. [1] The cycle starts with a compound known as Acetyl Coenzyme A (AcCoA) that can be created from a Pyruvate – the end product of the Glycolysis pathway which will be discussed later – as well Oxaloacetic Acid (OAA). The AcCoA and OAA become Citric Acid (Cit) and through a series of reactions a new OAA is obtained, completing the cycle. There is also a "shortcut" called the Glyoxylate cycle that is sometimes included. [8] The TCA cycle (without Glyoxylate) with carbon traces can be seen in Figure 2.5 along with relevant molecules and ids.

Following the different carbons around from Cit to Cit can help understand what is happening in the cycle. Consider the carbon with id 0 in Cit. After one round from Cit to Cit, 0 will end up at 5. 5 will end back up at 0. Due to a symmetry $0 \leftrightarrow 6$ in Cit, 0 could also end up in 6. With another round, 6 becomes 1. 1 in turn becomes 13 and due to a symmetry in Succ, 1 can also end up at 7. Thus, as it turns out, we have just computed the orbit of carbon 0 in Cit by hand, and it showed that it could end up in all possible carbon-positions of Cit.

Using the framework developed, we can see that this is in fact the case. We use the framework to construct the Pathway Table seen in Table 2.2. Note that there

is only one pathway and the table has been split into two to better fit on the page. One interesting point is 1, 7 and 13 cannot go everywhere, but only to a subset of the carbons. Considering Figure 2.5 again shows in fact that this is the case since 1 can become 7 and 13, but both 7 and 13 leave the system via a CO2.

Pathway \ Atom label	0, C	1, C	5, C
TCA	0, 1, 5, 6, 7, 13	1, 7, 13	0, 1, 5, 6, 7, 13

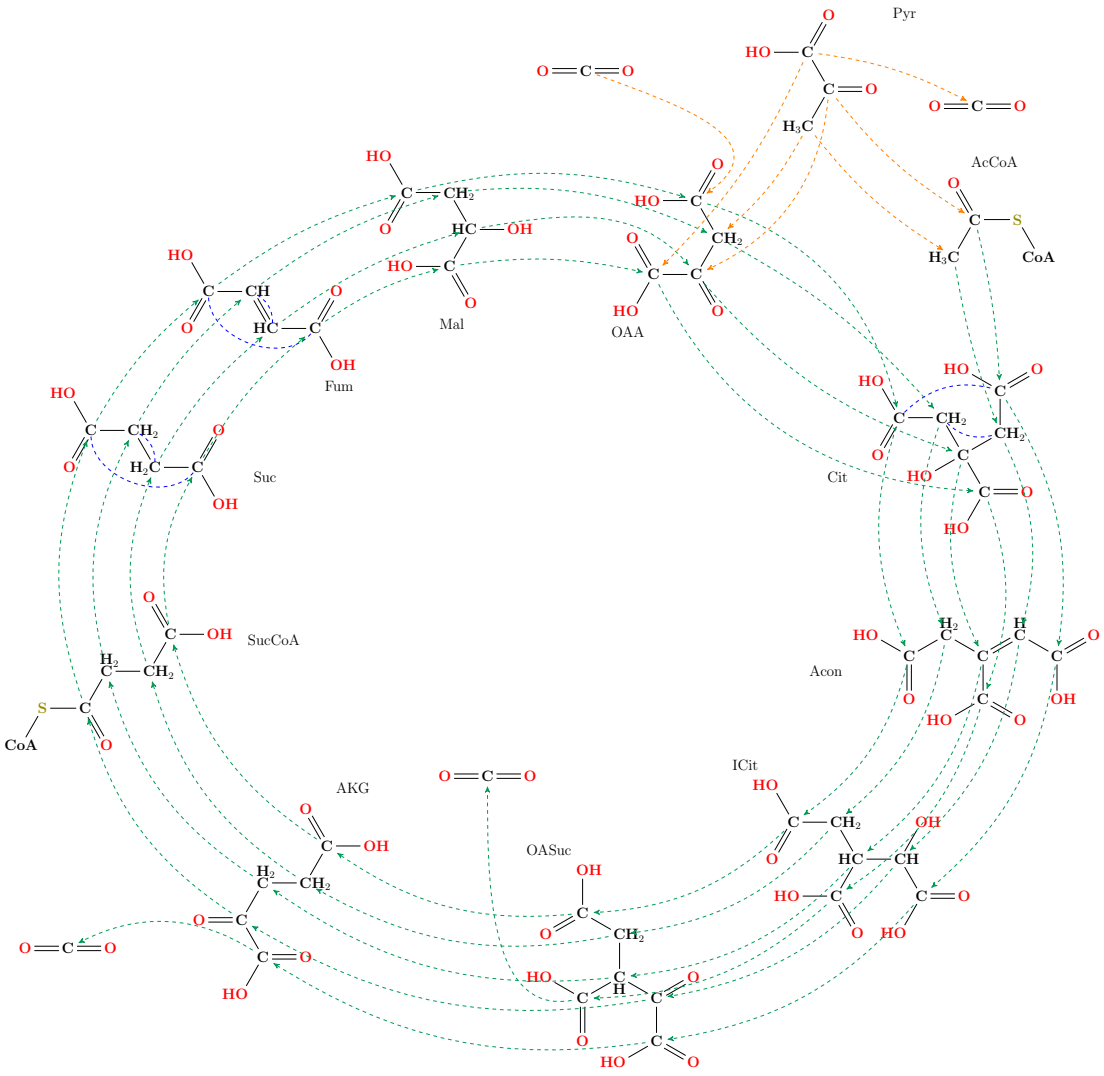
Pathway \ Atom label	6, C	7, C	13, C
TCA	0, 1, 5, 6, 7, 13	1, 7, 13	1, 7, 13

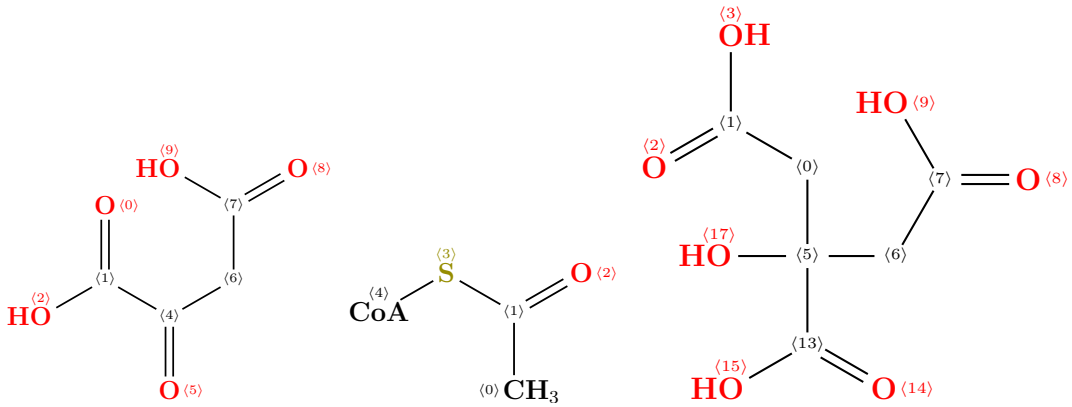
TABLE 2.2: Cit $\rightarrow$ Cit. The table is split in two to fit on the page.

We can also compute orbits from AcCoA to Cit. The AcCoA molecule is asymmetric, so seeing the different orbits might give helpful insight. Labelling carbon 0 in AcCoA becomes id 6 in Cit which can end up everywhere in Cit as we have previously seen. Labelling 1 in AcCoA becomes 7 in Cit which can only end up at 1, 7 and 13. Using the framework to compute the Pathway Table confirms this observation. This is depicted in Table 2.3.

Pathway \ Atom label	0, C	1, C
TCA	0, 1, 5, 6, 7, 13	1, 7, 13

TABLE 2.3: AcCoA $\rightarrow$ Cit.





(A) OAA (B) AcCoA (C) Cit

FIGURE 2.5: The TCA cycle with carbon traces shown (hand-made). Blue lines are symmetries of molecules. Orange lines are the TCA lead-in with Pyruvate. The figure also contains close-ups of molecules with relevant ids.

Glycolysis

Glycolysis is a metabolic pathway that converts one Glucose molecule into two Pyruvates. As a matter of fact there are two pathways that achieve this: The Entner–Doudoroff (ED) and the Embden–Meyerhoff–Parnass (EMP) pathways [14]. Using a MØD-model of Glycolysis and the framework developed for this thesis, it is possible to determine what carbons in the Glucose to label in order to distinguish the ED and EMP pathways.

First, we consider the DG of both pathways together. This is depicted in Figure 2.6. Some atoms have a unique "route" in each of the pathways. Additionally, there are no symmetric molecules, so the orbit of these atoms with a unique route only contain atoms of different molecules. There are no atoms in the orbit that have the same molecule, giving this impression of a unique route for each atom. One does in fact not need to draw these conclusions by hand, as the Pathway Table will reveal this information automatically since the atoms with a unique path will have an entry with only one id as well as a star.

Table 2.4 depicts the simplified Pathway Table of the two pathways as well as the combined DG. It turns out that there are quite a few atoms, especially oxygens, in the glucose that can be used to distinguish the pathways. For instance, if an isotope labelling experiment is carried out where with a carbon-13 in the place of the carbon at position 6 in the Glucose-molecule, and the mass spectrometer shows only Pyruvates with the carbon-13 at positions different from 2, then we can conclude that it is not the EMP pathway, and since there are no other pathways, it must be the ED pathway.

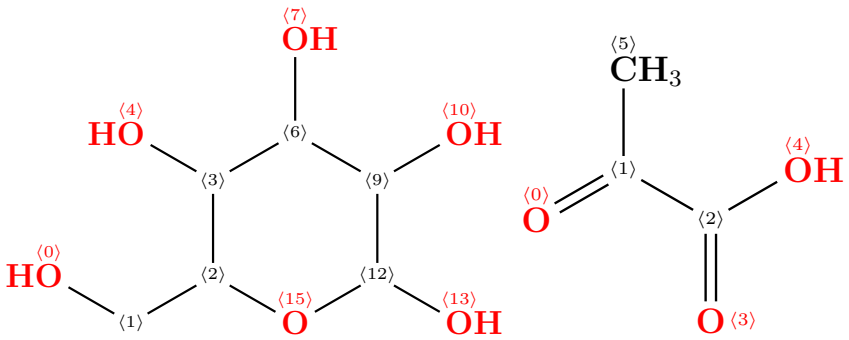


FIGURE 2.7: Close-up of the glucose and pyruvate molecules where the internal ids can be seen.

Pathway \ Atom label	0, O	6, C	7, O	12, C	13, O	15, O
EMP		2	3	5		0
ED	0	5	0	2	3	4
Both	0	2, 5	0, 3	2, 5	0, 3	0, 4

TABLE 2.4

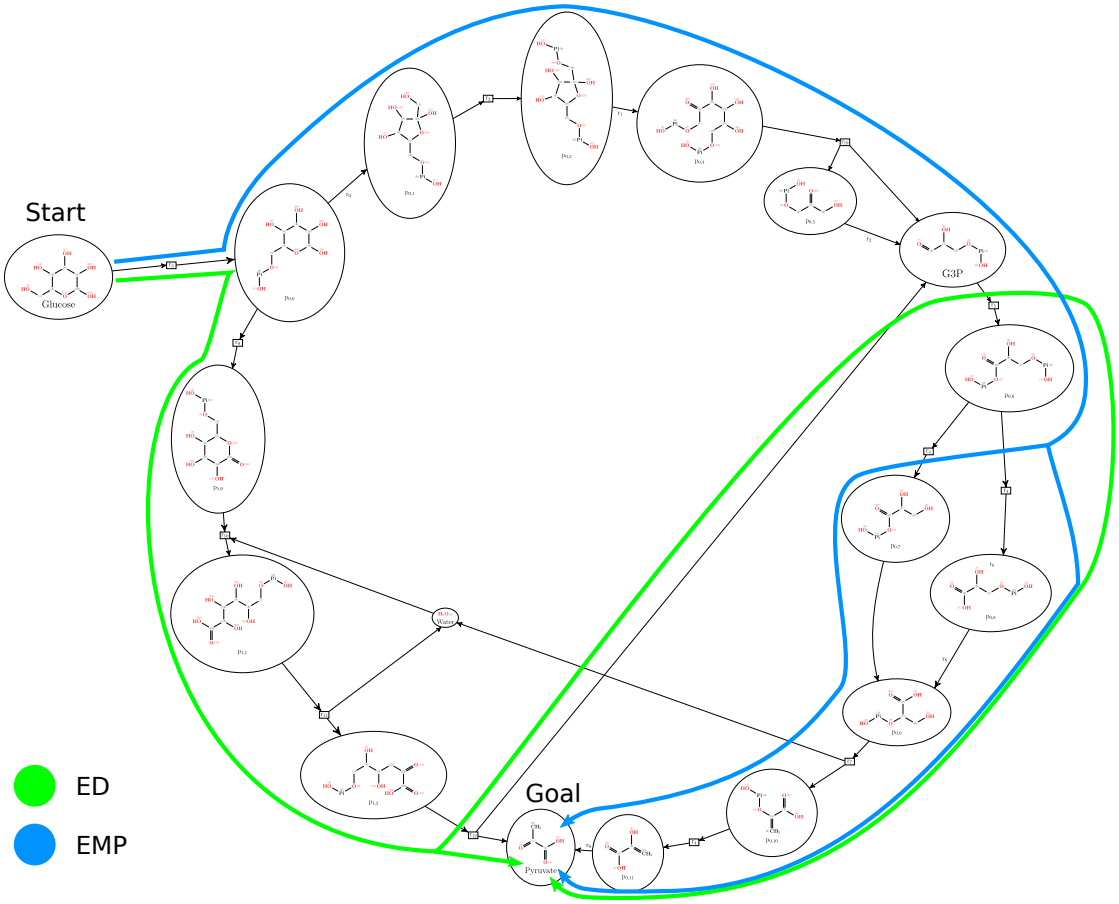


FIGURE 2.6: A combined DG for the ED and EMP pathways.

XY-2-Pathways

Next, we have a non-chemical example that tries to create two pathways that are easily distinguishable using this approach without involving a lot of complexity. In the following we will use SMILES strings.

Start-molecules: CCCY and X.
Goal-molecules: CCCX.
See Figure 2.8 for the internal ids that will be used throughout.

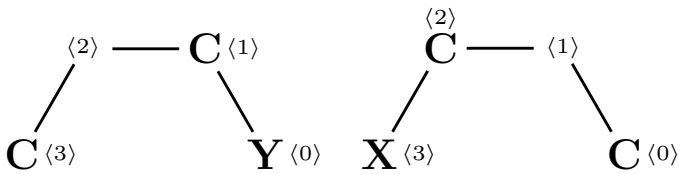


FIGURE 2.8: The start- and goal-molecule of the pathways.

Pathways:

(A) Split into CCC and Y, then merge CCC and X into CCCX.

(B) Split into C and CCY. Attach C to X, then C to CX and then C to CCX.

The pathways are depicted in a combined view in Figure 2.9.

Here we have three choices: Label carbon 1, 2 or 3, where carbon 1 is the carbon closest to the Y.

1. Consider labelling carbon 1.
   (A) In pathway A, 1 would become 0 in molecule CCC. But since this molecule is symmetric, we cannot distinguish whether it is 0 or 2. When the X is attached, the label can therefore be at either 0 or 2.
   (B) In pathway B, 1 would be part of the CCY that is left dangling, thus it would not make it to the goal-molecule.
2. Consider labelling carbon 2.
   (A) In pathway A, 2 would become 1 in molecule CCC and this would carry over to CCCX.
   (B) In pathway B, 2 also would be part of the CCY that is left dangling, thus it would also not make it to the goal-molecule.
3. Consider labelling carbon 3.
   (A) In pathway A, 3 would become 3 in molecule CCC, but again it is symmetric and the label can therefore either be 0 or 2 which carries over to CCCX.
   (B) In pathway B, 3 becomes the lone C. Three of these are then attached to an X one at a time, resulting in CCCX where all three carbons have a label.

Doing this analysis by hand has revealed which carbons would be helpful to label: Labelling any of them would give us different results. If we want to distinguish by weight, we could for instance label carbon 1. Now if we consider the Pathway Table generated by the framework, we see that it reveals exactly the information we obtained by hand.

Pathway \ Atom label	1, C	2, C	3, C
1	0, 2 (*)	1 (*)	0, 2 (*)
2			0, 1, 2 (*)
DG	0, 2	1	0, 1, 2 (*)

Based solely on the table, we could design the following isotope labelling plan: Label 2 and if the final CCCX has any label, we can exclude pathway 2, thus it must be pathway 1. And in the case no label is observed in CCCX, we can exclude pathway 1, since it has a star so eventually some label must go to carbon 1 in CCCX.

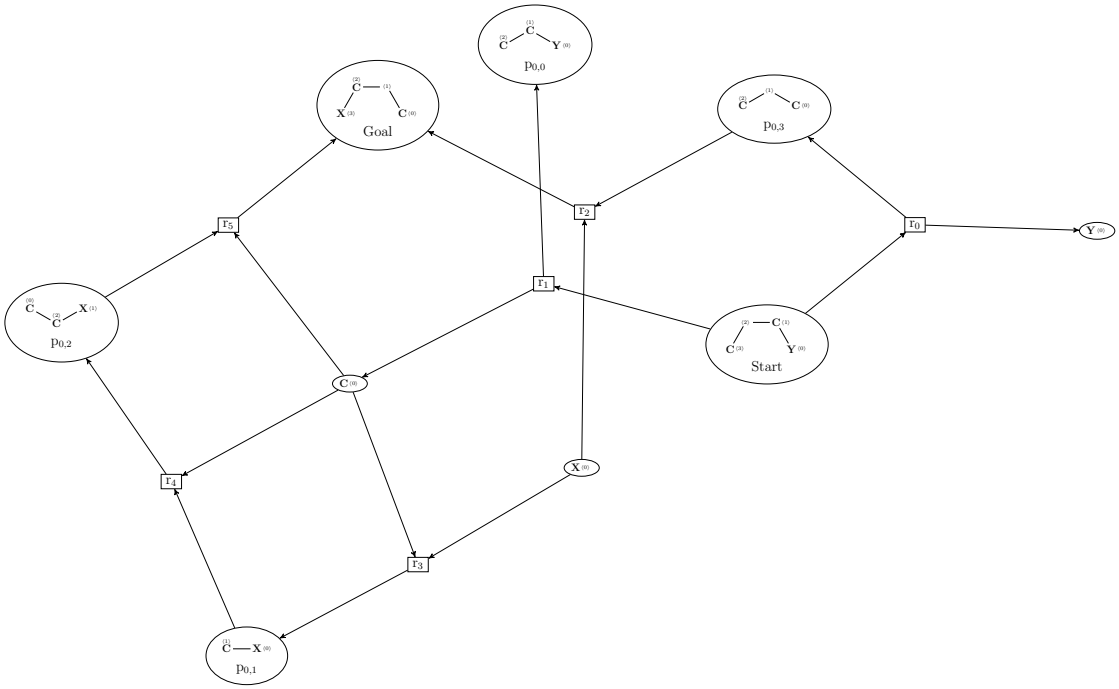


FIGURE 2.9: The combined DG for the two pathways (A) and (B)

XY-4-Pathways

This is a variant of XY-2-Pathways in an attempt to introduce more pathways while still maintaining simple, distinct pathways.
Start-molecules: CCCY and X.
Goal-molecules: CCCCCX.

Pathways:

(A) CCCY splits into CCC, Y or C, CCY. The 2 C and CCC turn into CCCCC which then attaches an X to give CCCCCX.

(B) CCCY again splits into CCC, Y or C, CCY. CCC merge with X into CCCX. Afterwards, the single C attaches to make CCCCX and then another to make CCCCCX.

(C) The usual split into CCC, Y or C, CCY. X is attached to a single C, then the result is attach to another C and another until we have CCCX. CCCX is now attached to CCC to give the too long CCCCCCX. Finally, a single C is cut of the end, leaving CCCCCX.

(D) Here we simply split CCCY into C, CCY. X is attached to C to give CX. Another single C is attached to CX to give CCX. This is continued all the way to CCCCCX.

The DGs for each pathways as well as the combined DG are too big to be meaningfully printed here. They can be found in Appendix A. The manual analysis carried out in the previous example is also out of scope, but we can still consider the simplified Pathway Table computed by the framework:

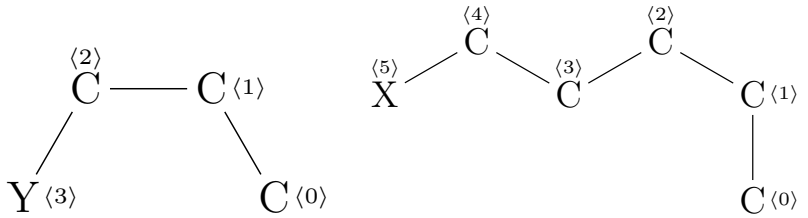


FIGURE 2.10: The start- and goal-molecule of the pathways in XY-4-Pathways.

Pathway \ Atom label	0, C	1, C	2, C
1	0, 1, 3, 4 (*)	2	1, 3
2	0, 1, 2, 4 (*)	3	2, 4
3	1, 2, 3, 4 (*)	0	1, 2, 3, 4
4	0, 1, 2, 3, 4 (*)		
DG	0, 1, 2, 3, 4 (*)	0, 2, 3	0, 1, 2, 3, 4

A variant of this table has already been discussed in Section 2.5. As discussed, we can give the start of an isotope labelling plan: Label 1 in CCCY; if any labels show up in CCCCCX, we can exclude pathway 4; if something different from 2 shows up, we can exclude pathway 1 and so. If for instance 3 is observed in CCCCCX, we can exclude everything except pathway 2.

As also discussed, it can be difficult to distinguish if only looking at carbon 0 in CCCY, thus one can use the Pathway Comparison Table approach to see what observed labels can be used to distinguish. This is discussed in Section 2.7. A Pathway Comparison Table with atom 0 and $k = 2$ can be seen below:

Contains \ Not contains	1	2	3	4	DG
1					
2					
3	$(1,3)$	$(2,4)$			
4	$(1,3)$	$(2,4)$			
DG	$(1,3)$	$(2,4)$			

If ever both carbon 1 and 3 are observed with a label in CCCCCX, this would distinguish pathway 1 from 3 and pathway 1 from 4. If ever both carbon 2 and 4 are observed with a label in CCCCCX, we can distinguish pathway 2 from 3 and 2 from 4.

2.9 Code Overview

2.9.1 Framework

The atom tracing framework handed in with the this thesis is a Python module called `atomtracing` with the following folder structure:

- `grouptheory`
  - `transformation.py`
  - `semigroup.py`
  - `hypergraphsemigroup.py`
- `pathway`
  - `pathway.py`
  - `pathwaytable.py`
  - `pathwaytabulate.py`
- `util`

`transformation` contains a class `Transformation` that represents a semigroup transformation. It is primarily a wrapper around a dictionary with a method to inverse.

`semigroup` contains a class `Semigroup` that represents a semigroup G of transformations. The class contains a list of generators and two methods:

- `invert` returns the inverse semigroup by inverting each generator
- `orbit` which given an α computes the orbit of that point, i.e. $\{g(\alpha)|g \in G\}$. The implementation is based on the group orbit function from the `Sympy` module and allows for multiple actions, for instance as tuples or as sets. This method is the cornerstone of this entire framework.

`hypergraphsemigroup` contains a class `HypergraphSemigroup` that represents a Hypergraph-Semigroup and another class `Linearization` that represents an atomic linearization.
`HypergraphSemigroup` contains many different minor things, but the main points are:

- `hypergraph`: The hypergraph for which the Hypergraph-Semigroup is computed.
- `linearization`: An instance of `Linearization`. Each atom in the hypergraph gets a unique id and mapping back and forth between id and atom is easy.
- `generators`: A list of generators computed as described in the theory section above. In short: Go through each hyperedge, find all vertex maps and turn them into `Transformations`.

- `semigroup` and `inverted_semigroup`: The underlying `Semigroup` based on the `generators` and its inverted counterpart.
- The method `orbit` which is a wrapper around the `semigroup` `orbit` method where instead of giving an α, then an atom (or tuple of atoms) is given instead. It is first converted to an α using the `linearization`, then the orbit is found and the orbit is converted back to atoms using the `linearization`.

`pathway` contains a class `Pathway` that represents a pathway: A set of start-molecules, a set of goal-molecules and an underlying hypergraph containing all start- and goal-molecules. This class has an undelying `HypergraphSemigroup` that is build based on the `hypergraph`.
`Pathway` contains a couple of helper functions:

- `orbit`: A wrapper around the `HypergraphSemigroup` `orbit` method. Without any flags given, it will by default only return the part of the orbit that is in the `goal_molecules`.
- `orbit_backwards`: Like `orbit`, but will use the underlying inverted Hypergraph-Semigroup and by default only return the part of the orbit reaching the `start_molecules`.
- `all_pairs_orbit_backwards`: A helper function that takes an orbit of the goal-molecules, i.e. the result of `orbit`. It will then try all pairs and for each it computes the inverted orbit to `start_molecules` and determines if all elements of the tuple coincide in one single atom, a. The method will return a dictionary that maps from each pair to a list of such a's. The size k can be specified via the optional parameter `tuple_size`. The default is 2.

The class also has two static methods `from_example` and `from_flow_solution` that are useful when creating `Pathways` from an example and a flow solution respectively. Examples come from an example database which is explained later.

Finally, the `pathway` file contains some free-floating functions:

- `examples_to_pathways`: it takes a list of examples and returns a list of `Pathways`. It creates the DG from each example and uses this to create the corresponding pathway. One key feature of this function is that the graph database used for each DG will be the same, so molecules that are isomorphic between DGs will also be guaranteed to have the same internal ids which are used for the labelling of input molecules and output of the various tables. These ids are also the ones that will be shown in the summary file.
- `example_flow_to_pathways`: This is mostly a helper that can be used in most examples when many flow solutions are needed. It takes one example, create a DG and then a flow where each start-molecule is a source and each goal-molecule is a sink. Additionally, each goal-molecule must have an outflow of 1. And vertices with in-degree 0 are marked as sources, as well as vertices with out-degree 0 are marked as targets. By default at

most 5 flow solutions are found, but this can be changed via the optional parameter `maxNumberSolutions`.

The `pathwaytable` file contains a class `PathwayTable` that represents a Pathway Table and a class `PathwayTableSimplified` that is a `PathwayTable` where reduction rules have been applied to simplify it. The `PathwayTable` class takes a list of pathways and a list of atoms from the start-molecules and creates a table where the rows are pathways and the columns are atoms. Each entry (p, a) uses the `orbit` method of p to find the orbit of a that reaches the goal-molecules.
The class has the following methods:

- `__getitem__` gives the `[]`-operator, like `table[p,a]` to get the entry (p, a). There are also a lot of other methods used to access different aspects of the table, for instance `rows` and `columns`.
- `simplify` returns a `PathwayTableSimplified`.
- `pathway_comparison_table` return a Pathway Comparison Table.
- `tabulates` returns a list of `PathwayTabulate`, one for each start- and goal-molecule-pair.
- `to_summary` writes the hypergraph of each pathway to the summary file as well as writes each tabulates to the summary.

The `pathwaytabulate` file contains the `PathwayTabulate` class that represents a cross-section of the `PathwayTable` that focuses only on one start-molecule and one goal-molecule of possibly many. This is primarily for printing purposes to make it more intuitive and less cluttered. When we only have a single start-molecule to worry about, the labels of the column can be referred to by a single number, namely the internal id, which is what is also shown in the DG in the summary file. Similarly, each entry in the tabulate can be a list of internal ids of where that atom from the start-molecule can go in the goal-molecule.
This class uses the pip package `tabulate` to display content with either beautiful ascii-symbols or directly as LaTeX.

Finally, the `util` folder contains a mix of different utilities and helper functions. Some highlights are:

- `FilteredHypergraph`, inspired by Boost `filtered_graph`, creates a view of a subhypergraph based on a hypergraph. It takes two predicates: one for vertices and one for hyperedges. If the predicate returns true, the hyper-vertex/edge is included, otherwise not.

  The `FilteredHypergraph` is used to create the `DGFlowSubgraph` class that given a DG and a flow solution yields a sub-DG, including only the vertices and edges that have a non-zero flow.

  This is useful because it simplifies the atom tracing procedure to think about hypergraphs, start-molecules and goal-molecules. It does not matter if the hypergraphs are different DGs or the subgraphs of the same DG based on different flow solutions.

This `DGFlowSubgraph` is automatically created when using helper functions like `example_flow_to_pathways`.

- `is_carbon`, `is_hydrogen`, `is_not_hydrogen` and other functions from `atomselector` helps to filter out only the atoms that the user is interested in tracing.
- `write_to_summary` takes arbitrary LaTeXcode as a string an writes it to the summary file. This is used by the `PathwayTable` and `PathwayTabulate`.

2.9.2 Example Database

The framework comes with an example database of real-life and non-chemical examples that is useful when testing the different pathway functions.

The database allows for error checking of all examples, loading of a single example and even loading of all examples. [1]

A unifying example format has been designed and it will be described here.

An example database contains example folders. The name of this folder becomes the tag for the example(s) inside. An example folder must contain a file `example.py` which will be included using PyMØD's `include`. The example folder can also contain other helpful files needed to construct the example. The `example.py`-file must contain one or more classes that inherit from `ExampleBase`. Doing this means that the example database framework can find the subclass and the subclass can inherit all the needed properties at the same time. The name of the class that inherits from `ExampleBase` will be used as the name for the example if a name is not explicitly defined. When an example folder is loaded, all `ExampleBase` subclasses found will be loaded, instantiated and returned as a list to the user.

The following properties must be set:

- `start_molecules`: Used to specify a list of molecules that are to be considered start-molecules.
- `start_molecule`: This can be set instead of `start_molecules` if the start-molecules are just one molecule.
- `goal_molecules/goal_molecule`: Similar story for goal-molecules.
- `dg_strat`: A function that returns the strategy used to build the DG of the example.
- `dg`: If `dg_strat` is not supplied, one can override the DG creation directly. By default the `dg` function will call `dg_strat` and give the result to `dgRuleComp`.

[1] A word of warning: Due to the way PyMØD works currently, an example is loaded with side-effects that modifies global variables such as `inputGraphs`. This means that using these variables in DG strategies and other places cause weird results if multiple examples are loaded in the same run.

The following properties are inherited by default, but can be overridden if needed:

- `name`: The name of the specific example class.

  By default it is `<example folder>::<example class name>`

- `label`: The label used by the various tables in the framework to represent the example as a pathway. By default it is `None` which will result in it simply being assigned a number.

- `graph_printer`: Returns a `mod.GraphPrinter`. Override if the molecules of the example should be printed in a special way.

- `dg_printer`: Returns a `mod.DGPrinter`. Override if the DG of the example should be printed in a special way.

- `print_dg`: Takes a DG and prints it. By default it will call `dg_printer` to get a `mod.DGPrinter`.

- `print_graphs`: Prints all graphs of `inputGraphs`. By default it will call `graph_printer` to get a `mod.GraphPrinter`.

- `print_rules`: Prints all rules of `inputRules`. By default it will call `graph_printer` to get a `mod.GraphPrinter`.

When loading an example, it will be sanity checked and warnings might be printed or errors might be thrown if some required fields are not as they should be. Loading an example folder is done like this:

```
include("example_database/common.py")
db = ExampleDatabase("example_database")

exs = db.load_example("example-name")
# exs will be a list of example class instances
# that inherit from ExampleBase.
```

An example of how to find a simplified Pathway Table for the Glycolysis pathways:

```
from atomtracing import *

include("example_database/common.py")
db = ExampleDatabase("example_database")

exs = db.load_example("glycolysis")

pathways = examples_to_pathways(exs)

t = PathwayTable(pathways, is_not_hydrogen)
```

```
t2 = t.simplify(method="position")

# Prints as ascii in the terminal.
t2.print()

# Prints all pathways as well as table to the summary file.
t2.to_summary(ex.dg_printer())
```

An example of how to find a simplified Pathway Table for the XY-2-Pathways example:

An example of how to find a Pathway Comparison Table for the XY-4-Pathways example:

```
from atomtracing import *

include("../example_database/common.py")
db = ExampleDatabase("../example_database")


exs = db.load_example("xy-4-pathways")
ex = exs[0]


pathways = example_flow_to_pathways(ex)

t = PathwayTable(pathways, is_not_hydrogen)
t2 = t.simplify(method="position")

t2.print()

for a in t2.columns():
    # Find some atom to create the Pathway Comparison Table on
    atom = a
    break

t3 = t2.pathway_comparison_table(atom)
tab = t3.pretty().to_tabulate()
print(tab)
```

2.9.3 Exposing Vertex Maps in MØD

To actually be able to access the vertex maps in the Python framework above, they need to be exposed from MØD via Boost Python.

The code discussed is from Jakob Lykke Andersen's MØD repository, branch `peter/develop`, commit `dedbf77e`.

Relevant files:

- `src/mod/DGVertexMap.cpp`
- `src/mod/DGVertexMap.h`
- `src/mod/VertexMapUnionGraph.cpp`
- `src/mod/VertexMapUnionGraph.h`
- `src/mod/Py/VertexMap.cpp`

The `VertexMap` class can be found in `src/mod/lib/DG/VertexMap.h` This class corresponds to VertexMaps(e) for some hyperedge e of the DG, i.e. the set of all vertex maps for that hyperedge. The tail of e is a set of molecules and the union of these is what has been referred to as G in this chapter. Similarly, the union of the molecules of the head of e is H.

The `VertexMap` class resides in the `lib` part of MØD and is not suitable for direct exposure via Boost Python. Therefore a wrapper class `DGVertexMap` is created. Additionally, the union graphs G and H are not exposed via Python, thus the class `VertexMapUnionGraph` is created for this purpose.

The `DGVertexMap` has a couple of important parts:

- `MapProxy`: A class that represents a single vertex map. It has two methods `leftToRight` and `rightToLeft`. `leftToRight` takes a vertex from G to H via the vertex map. `rightToLeft` is its inverse.
- `const_iterator`: An iterator in C++ is a view into a collection. Concretely, it points to one element of the collection and has the ability to continue to the next one. The `lib` `VertexMap` class already has iterator, so the iterator in `DGVertexMap` is simply a wrapper around it, using Boost `iterator_adaptor`. In brief terms, this class describes how given an iterator instance of one type it can be turned into an iterator instance of another type. In this case it is simply wrapping it in a class `MapProxy` that represents a single vertex map.
- `range`: A class that represents the entire span of iterators. It contains the begin- and end-iterator. This is needed to be able to use for-loops in Python, like

  ```
  for m in maps:
      # Do stuff with m
  ```

- `getProxyLeft` (and `getProxyRight`): This method gives the `VertexMapUnionGraph` representation of G (or H) such that its vertices or edges can be traversed and passed through the vertex map.

The `VertexMapUnionGraph`, as mentioned, represents the union of molecule-graphs to one unified graph of the tail or head of e. It is a graph and thus needs a graph interface:

- `Vertex`: A class representing a vertex in the graph.
- `vertices`: A range of all vertices.
- `numVertices`: The number of vertices.
- `Edge`: A class representing a vertex in the graph.
- `edges`: A range of all edges.
- `numEdges`: The number of edges.
- and more.

This is mostly just wrappers around the `union_graph` from `src/jla_boost/-graph/UnionGraph.hpp`. Boost `iterator_adaptor` is again used to convert existing iterators of `union_graph` to iterators of `VertexMapUnionGraph`.

With all this in play, it is possible to create a `DGVertexMap`, `M`, from a hyperedge e and then get a `VertexMapUnionGraph`, `G`, by calling `M.getProxyLeft()`. Now for each veretx map `m` in `M.maps()` and each `v` in `G.vertices()`, one can find what `v` maps to via the vertex map by calling `m.leftToRight(v)`.

The last step is to expose all these classes and methods to the Python interface via Boost Python. This is relatively straight forward by specifying for each class what methods in Python corresponds to some method in C++. An example can be seen here:

```
py::class_<DGVertexMap, std::shared_ptr<DGVertexMap>>("DGVertexMap", py::no_init)
        .def("getLeft", &DGVertexMap::getProxyLeft)
        .def("getRight", &DGVertexMap::getProxyRight)
        .def_readonly("maps", &DGVertexMap::maps)
        .def("__len__", &DGVertexMap::size)
        ;
```

3

Vertex Map Optimization

3.1 Introduction

A core component of the Hypergraph-Semigroup approach is the computation of vertex maps based on hyperedges in the Derivation Graph. The DG might have hundreds of hyperedges and for each, one or more vertex maps has to be found. This can be very time consuming and the entire approach would see a great speed-up if the vertex map computation is sped up. Additionally, from the user's point of view many of the vertex maps currently produced are redundant because only a subset of atoms are relevant when doing isotope labelling experiments, for instance only carbons and oxygens. Two vertex maps might be different only in how they map the hydrogens and further worsening the runtime of Hypergraph-Semigroup approach. See Figure 3.1.

There is, however, nothing special about hydrogens – it is just a vertex in graph like any other, however it is often considered irrelevant for isotope labelling experiments. The approach we will design in this section will be able to ignore any kind of vertex in any graph.

Both the old and the new approach rely on algorithms to find all monomorphisms and isomorphisms between two graphs. The old approach will use such algorithms as black boxes that given two graphs will return a list of all monomorphisms or isomorphisms depending on the need. The new approach will need a black box for monomorphism, but for isomorphisms, it needs some control over how the space of all possibilities is traversed as well as modifying some internal state. The algorithm used in MØD for this task is VF2 which can be modified to give monomorphisms, subgraph isomorphisms or graph isomorphisms. However, any (sub)graph isomorphism finding algorithm can be used if it satisfies these requirements:

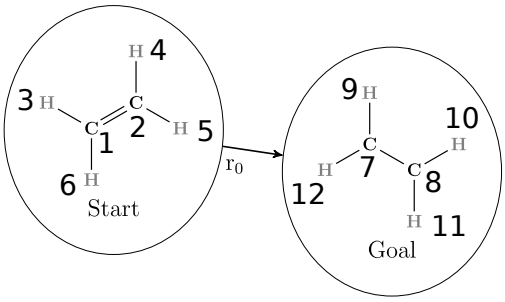


FIGURE 3.1: Isotope labelling experiments often use carbons meaning that only the following partial vertex maps are desired: $(1 \mapsto 7;\ 2 \mapsto 8)$ and $(1 \mapsto 8;\ 2 \mapsto 7)$. For a complete vertex map, we just need *one* isomorphic mapping of the hydrogens, for instance $(3 \mapsto 9;\ 4 \mapsto 10;\ 5 \mapsto 11;\ 6 \mapsto 12)$. The old approach will enumerate them all.

1. Each time the algorithm finds a match, it must report it to the user, in such a way that the user can specify whether to continue the search or not. This is commonly achieved by the algorithm taking a *callback*.
2. The algorithm must have a state of what vertices have already been matched, likely as a stack, in a way that can be modified before starting the search.

The default VF2 implementation in MØD supports requirement 1, but not 2. To implement the new approach, it has therefore been necessary to create a modified version of VF2. This will be described in detail later in this chapter.

Throughout this chapter we will refer to any algorithm that finds all graph **I**somorphisms as $\mathcal{A}_I$ and any algorithm that finds all **M**onomorphisms as $\mathcal{A}_M$. Any **C**ustomizable (sub)graph isomorphism algorithm that satisfy the requirements above will be called $\mathcal{A}_C$.

3.2 The Rule Composition Approach

This is how vertex maps are currently computed in MØD.
A DG has previously been created and $e = (e^+, e^-)$ is the hyperedge for which VertexMaps(e) will be computed. Furthermore, let $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ be the corresponding rule of the hyperedge, and let G be the disjoint union graph of the multiset of tail-vertices e^+. H is defined similarly from e^-.

First, we create identity rules $p_G = (G \leftarrow G \rightarrow G)$ and $p_H = (H \leftarrow H \rightarrow H)$. Next, we compute $P' = p_G \bullet_{\supseteq} p$. Recall that the full composition results in a new rule – or a set of rules in this case, since we want to enumerate all possibilities – from G to G, but where the subgraph L in G is replaced with R.

For each $p' \in P'$, we compose it with p_H as $P'' = p' \bullet_{\subseteq} p_H$. This is again a set of new rules, from G to H. Gather all rules computed in this fashion in a set X.

This means that we have transformed the original rule p from L to R to a set of rules X from G to H. Mathetically, we are done, but in practice the newly created

rules are not proper vertex maps from G to H in the desired way, for instance the ids might not be correct. To correct this, we need to find all isomorphisms from G to the left-hand side of each rule in X, and similarly all isomorphisms from the right-hand side of each rule to H. We can now compute the set of vertex maps from G to H:

$$\text{VertexMaps}(e) = \{m_{GL'} \circ l'^{-1} \circ r' \circ m_{R'H} \mid (L' \xleftarrow{l'} K' \xrightarrow{r'} R') \in X, \\ m_{GL'} \in M_{GL'},\ m_{R'H} \in M_{R'H}\}$$

where $M_{GL'}$ is the set of isomorphisms from G to L' and $M_{R'H}$ is the set of isomorphisms from R' to H. These sets are computed by two calls to $\mathcal{A}_I$.

In Python, the vertex maps on each hyperedge is converted to a semigroup transformation as discussed in the previous chapter and only atoms of interest (i.e. atoms that satisfy some predicate specified by the user) will be included in the transformation. This means that each vertex map needs to be filtered down in size and checked if it is a duplicate.

3.3 The New Approach

The new approach takes the filtering and moves it to C++ while incorporating it into a more direct approach for finding vertex maps using (sub)graph isomorphism algorithm with requirements mentioned in the introduction.

Suppose we have graphs G and H (each of which might be a union of multiple molecule-graphs) and a rule $p = (L \leftarrow K \rightarrow R)$. This gives us the following DPO:

$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ {\scriptstyle M_{LG}}\downarrow & & \downarrow & & \downarrow{\scriptstyle M_{RH}} \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Let M_{LG} be the set of all monomorphisms from L to G. Similarly, M_{RH} from R to H. One monomorphism from M_{LG} and one from M_{RH} together are called the *match* and *co-match* since they describe how L has been matched in G and R in H. These sets can be computed using two calls to $\mathcal{A}_M$.

The goal of this entire approach is to find VertexMaps(e), the set of vertex maps from G to H for some hyperedge e, however an $\mathcal{A}_I$ call will not suffice here since G and H are not isomorphic (assuming that p is not the identity rule). Giving G and H to $\mathcal{A}_I$ would simply yield no mappings since the subgraph induced by the rule p is changed from G to H. But a key observation is that $G \setminus L$ and $H \setminus R$ are isomorphic. Therefore if we can simply pretend that $\mathcal{A}_I$ has already found a match for the vertices of L to vertices of R using l and r, and then continue searching for an isomorphism from the vertices $G \setminus L$ to the vertices of $H \setminus R$,

we can find a complete mapping from G to H. This is where $\mathcal{A}_C$ comes into play.

Using this idea, the entire procedure to compute VertexMaps(e) is therefore as follows:

1. Compute M_{LG} and M_{RH} using $\mathcal{A}_M$.
2. For each $(m_{LG}, m_{RH}) \in M_{LG} \times M_{RH}$, prepare a call to $\mathcal{A}_C$ with G and H as input graphs. Before the matching is done, pre-set a partial mapping based m_{LG}, m_{RH} and the rule p. We call this the fixation.
3. Run $\mathcal{A}_C$ until it has enumerated all full mappings of $G \to H$. It should not be allowed to backtrack beyond the fixation. Report these vertex maps back to the user.

This approach is depicted in Figure 3.2.

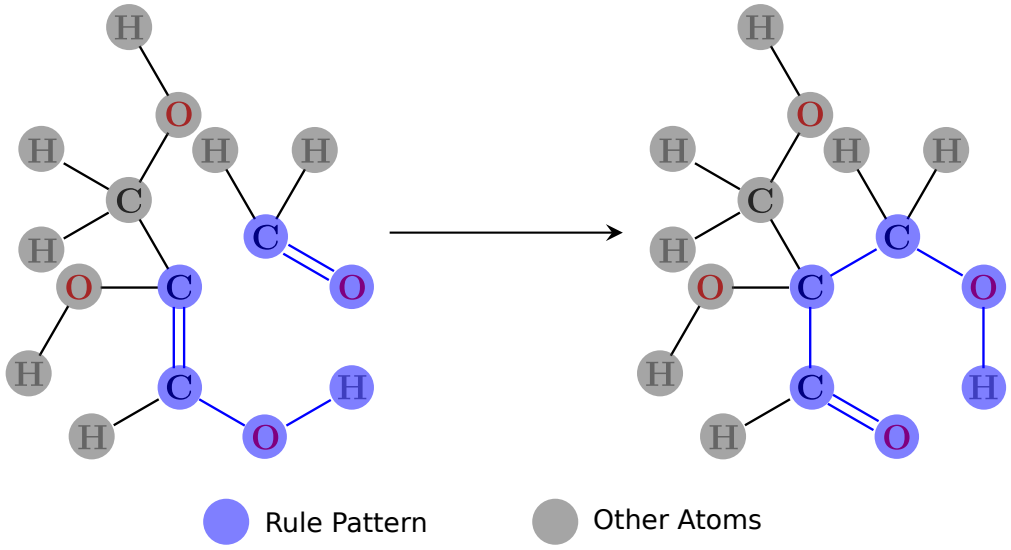


FIGURE 3.2: The subgraph induced by the rule (blue) is fixated for the call to $\mathcal{A}_C$. The algorithm will then find the mapping for the other atoms.

Using the above approach, VertexMaps(e) will include all possible vertex maps that the old method did – including all the redundant wrt. hydrogens – but since we take a much more direct approach with only 2 calls to $\mathcal{A}_M$ and 1 call to $\mathcal{A}_C$, it is more flexible and perhaps even faster.

One nice thing is the ability to get rid of these redundancies using the following idea:

1. Compute M_{LG} and M_{RH} using $\mathcal{A}_M$.
2. First construct a subgraph of G and H where hydrogens are removed (or some other predicate is satisfied). Call these G' and H'.

3. Phase 1: For each $(m_{LG}, m_{RH}) \in M_{LG} \times M_{RH}$, prepare a call to $\mathcal{A}_C$ with G' and H' as input graphs and m_{LG} and m_{RH} fixate a partial mapping. Run the matching.

4. Phase 2: For each mapping m that is found in step 3, create a new $\mathcal{A}_C$ call with G to H as input graphs, where the *entirety* of m is fixated, meaning that we have a partial mapping from G to H of all non-hydrogens. Run $\mathcal{A}_C$ until some match is found, report it and do not look for other matches. Continue the search of the $\mathcal{A}_C$ call from step 3.

This approach is depicted in Figure 3.3.

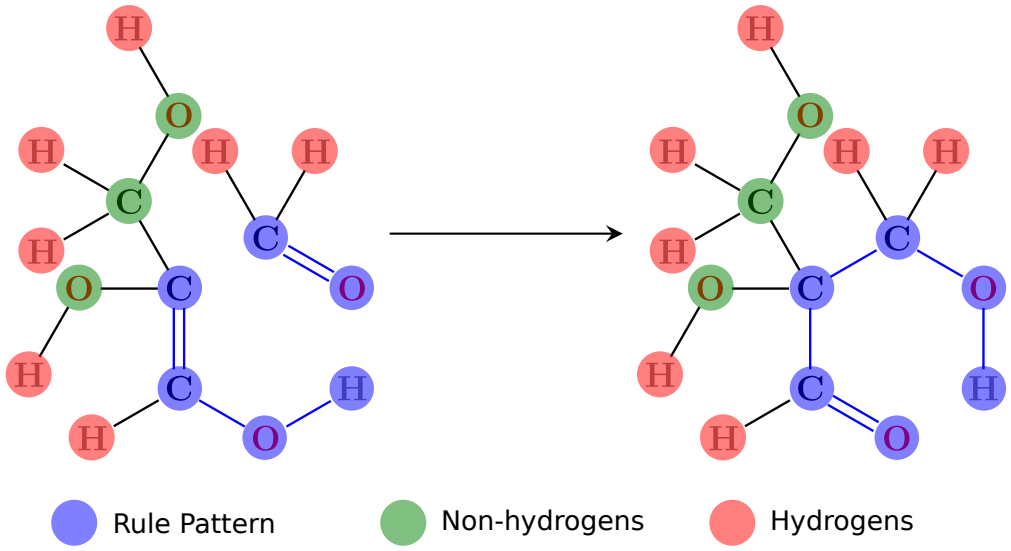


FIGURE 3.3: The subgraph induced by the rule (blue) is fixated for the first call to $\mathcal{A}_C$. The first $\mathcal{A}_C$ call will find a mapping of the non-hydrogens (green) and start a second call to $\mathcal{A}_C$ where the rule subgraph (blue) and non-hydrogens (green) are fixated and only the hydrogens (red) needs to be mapped. If there are multiple possible mappings for non-hydrogens (green), all of them will be found. For each, only one mapping of the hydrogens (red) will be found.

Note: If some hydrogen is part of the subgraph induced by the rule, it needs to be fixated and must remain in G' and H'. If such a hydrogen is ignored, no matches can be found in step 3 because the neighborhood relationships change for that atom.

In total, 2 $\mathcal{A}_M$ calls and 2 $\mathcal{A}_C$ calls are needed for each vertex map reported to the user. Furthermore, this approach has a couple of strengths:

- It is easy to customize and generalize compared to the approach with rule composition.
- It is a direct approach and as such is easier to understand and likely also faster.

- If the pattern L of the rule is large relative to G, this approach is even faster. Suppose that L is the entirety of G or close to the entirety of G, then most of the vertices of G are fixated and only a few needs to be matched by the $\mathcal{A}_C$ call in step 3. Afterwards only one set of hydrogen-matches needs to be found with the $\mathcal{A}_C$ call in step 4.
- The old approach does not support stereo-chemistry and other advanced features of MØD. Neither does the new approach, but since it is a more direct approach, it will be much easier to support such features.
- Sometimes it is also useful to get the match and co-match in addition to each vertex map. This is non-trivial to retrieve using the old approach, but quite trivial with the new approach since the match and co-match are computed directly.

One problem with this approach is that hydrogens in the subgraph induced by the rule cannot be ignored in the first phase. Consider Figure 3.4. The Keto-Enol can match either hydrogen 19 or hydrogen 16, thus this will contribute in M_{LG} with two mappings instead of one, and for each of them a vertex map is found; one of which is redundant. This could potentially be fixed by using the two-stage $\mathcal{A}_C$ approach on each of $L \to G$ and $R \to H$ instead of $G \to H$. However, due to time constraints this is still an open problem. Regardless, it is a minor issue compared to how much faster the new approach is overall, as we will see in the next section.

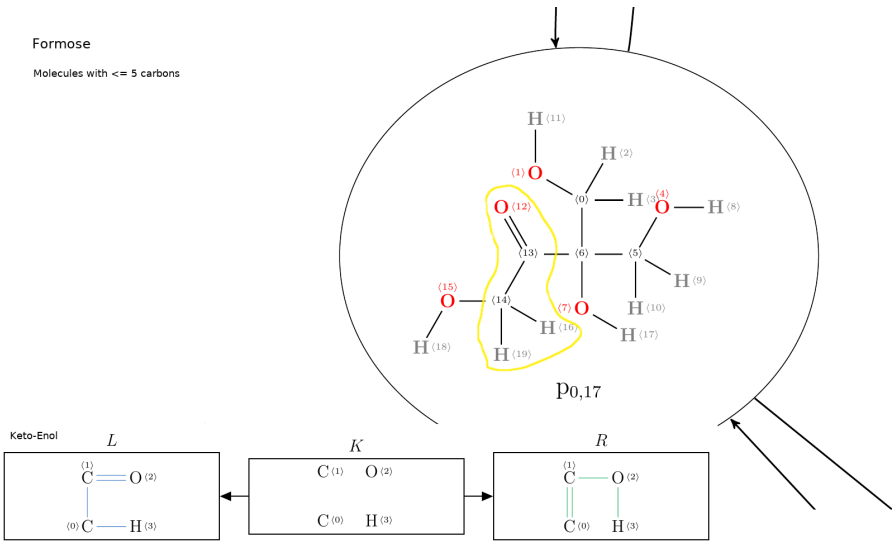


FIGURE 3.4

3.4 Empirical Results

The overall goal of the vertex map optimization has been to improve the runtime of finding all vertex maps of a hyperedge in a DG as well as cut down on the number of redundant vertex maps due to hydrogen symmetries; all with the purpose of reducing the *total* time it takes to compute a Hypergraph-Semigroup.

In the following a couple of chemical and non-chemical examples and experiments will be presented and discussed.

3.4.1 Improvement in classical chemical examples

Many of the bigger examples that one would come across when first learning about MØD, for instance Formose and Pentose Phosphate Pathway (PPP), as well as other known pathways like Glycolysis (ED and EMP pathways) has been tested.

All the tests presented in this section have been carried out in the same way:

1. Compute the DG for the example.
2. Compute the Hypergraph-Semigroup using the old method. Specifically: For each hyperedge, find all vertex maps using the old approach. Remove redundant vertex maps when ignoring hydrogens.
3. Compute the Hypergraph-Semigroup using the new method. Specifically: For each hyperedge, find all vertex maps using the new approach.
4. 2 and 3 are repeated 5 times and the average total time taken to compute the Hypergraph-Semigroup is stored. The number of vertex maps found for each hyperedge is also stored for 2 and 3.

Note: All of this is done in Python with calls to the C++ backend of MØD. The idea behind the testing procedure is how the user would experience the speed-up.

First, we will consider the **Formose** example with molecules of at most 5 carbons depicted in Figure 3.5. The image is too small to see details, but the important thing is this: There are 46 hyperedges and many of the molecules have quite a few symmetries and places where hydrogens create redundant vertex maps.

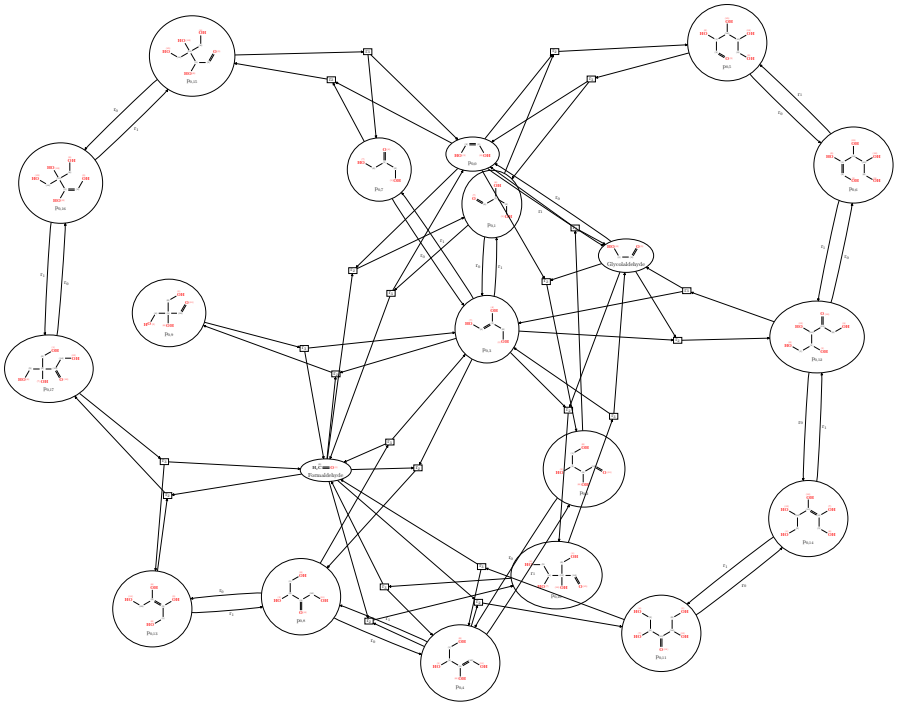


FIGURE 3.5: Formose DG with 46 hyperedges and many symmetries.

When creating the Hypergraph-Semigroup with the old approach it took roughly 0.69 seconds and with the new approach it only took 0.19 seconds, so in this case we have an approximate speed-up of 3.5 times. If we look at the number of vertex maps created on each hyperedge, we see a pretty clear picture of the improvement: The amount of vertex maps is greatly reduced for each hyperedge meaning that additional processing of the vertex maps into semigroup transformations is reduced.

This is depicted in Figure 3.6 as a side-by-side histogram where the bins are the number of vertex maps created per hyperedge. Each bin has two bars, one for the old and one for the new approach. The height of each bar is determined by the number of hyperedges in the DG that yield the number of vertex maps corresponding to the bin. For instance, in the case of Figure 3.6, roughly 22 hyperedges gave 4 vertex maps each with the old approach, but only around 10 gave 4 vertex maps with the new approach. If there is no bar, no hyperedges resulted in that number of vertex maps.

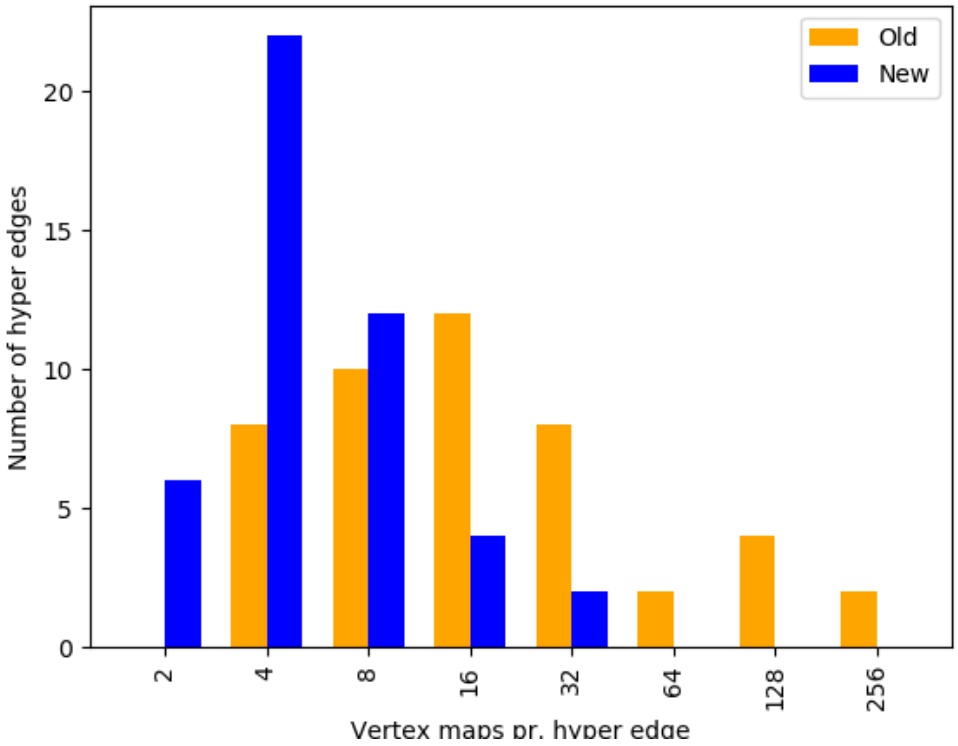


FIGURE 3.6: Formose: A side-by-side histogram of the number of vertex maps per hyperedge. Higher bars to the left is better.

Another example is **Glycolysis**, specifically the EMP pathway (the ED pathway showed similar results). Here we have 19 hyperedges. The old approach took around 0.10 seconds while the new approach took around 0.05 seconds to compute the entire Hypergraph-Semigroup. Here the speed-up is 2-fold, showing that the speed-up can heavily depend on the concrete example. In the case of the ED and EMP pathways there are no symmetric molecules and the effect of hydrogens is much smaller which also becomes apparent from the large bars of 2 and 4 in Figure 3.7 of the old method. If we consider the old and the new approach side-by-side, it is still an improvement, but not a considerable one.

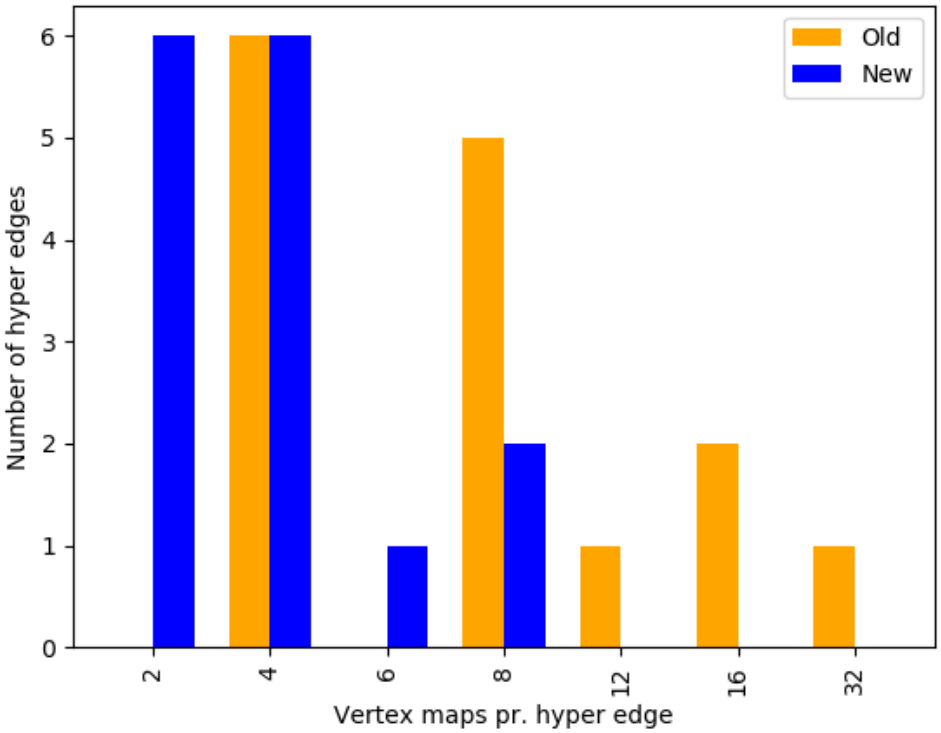


FIGURE 3.7: Glycolysis, EMP: A side-by-side histogram of the number of vertex maps per hyperedge. Higher bars to the left is better.

Finally, we can look at **PPP**. It has two variants: Normal and strict. The distinction comes in how the DG is computed. The strategy for the normal DG follows a very generative chemistry approach: Based on some known chemical rules, what can be created? This is usually a superset that contains much more than what is usually sought after. The strict strategy, on the other hand, applies only the specific rules in a specific order such that exactly the desired pathway(s) are created. To illustrate this further, see Figure 3.8. The normal has 333 hyperedges while the strict has 24.

The timing results can be seen in the table below:

	Normal [s]	Strict [s]
Old	135.34	30.14
New	8.08	1.01

Here we see a 16-fold speed-up in the normal DG and a speed-up of 30 times for the strict DG. Both of these DGs contains a lot of molecules with a lot of hydrogens and being able to not having to enumerate them all greatly speeds up the process. Consider Figures 3.9 and 3.10. Here it becomes very apparent how much of an improvement it is, especially for the normal DG, where the majority of edges would previously produce vertex maps in the triple-digits and now it is in the single- and double-digits.

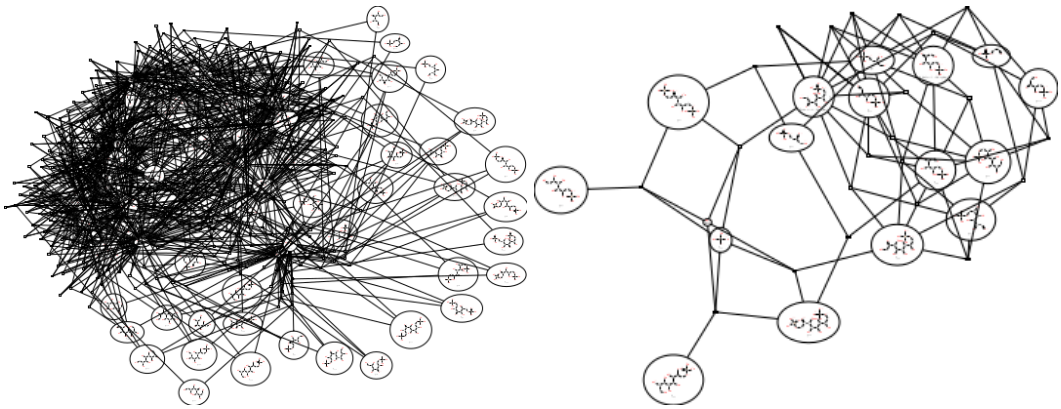


FIGURE 3.8: The PPP normal DG (left) compared to the PPP strict DG (right). The normal has 333 hyperedges while the strict has 24.

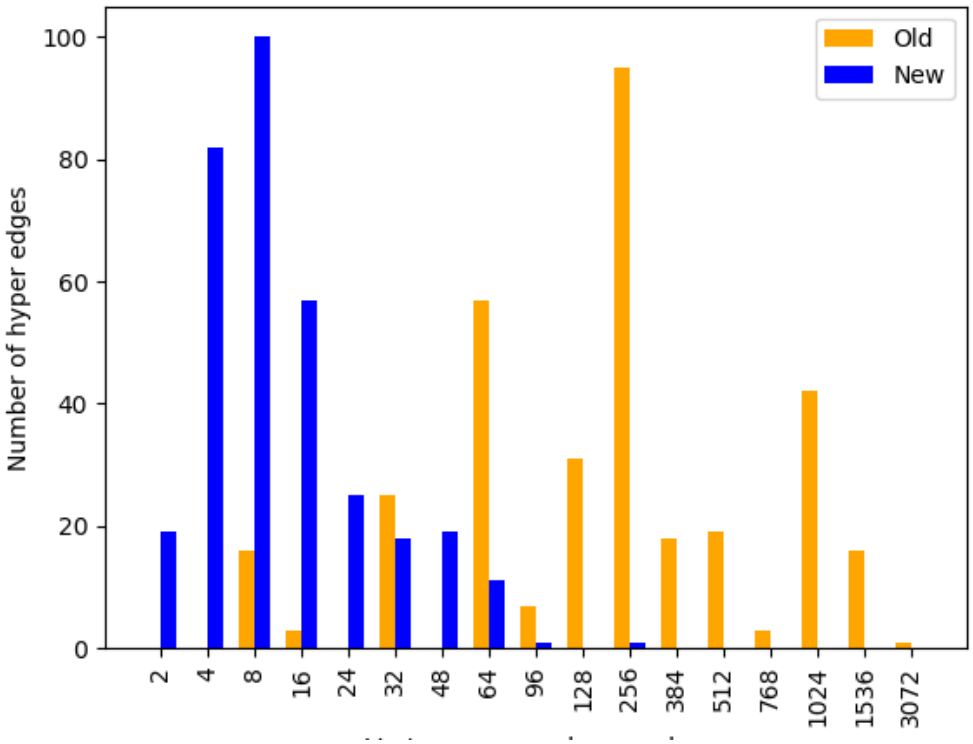


FIGURE 3.9: PPP, normal: A side-by-side histogram of the number of vertex maps per hyperedge. Higher bars to the left is better.

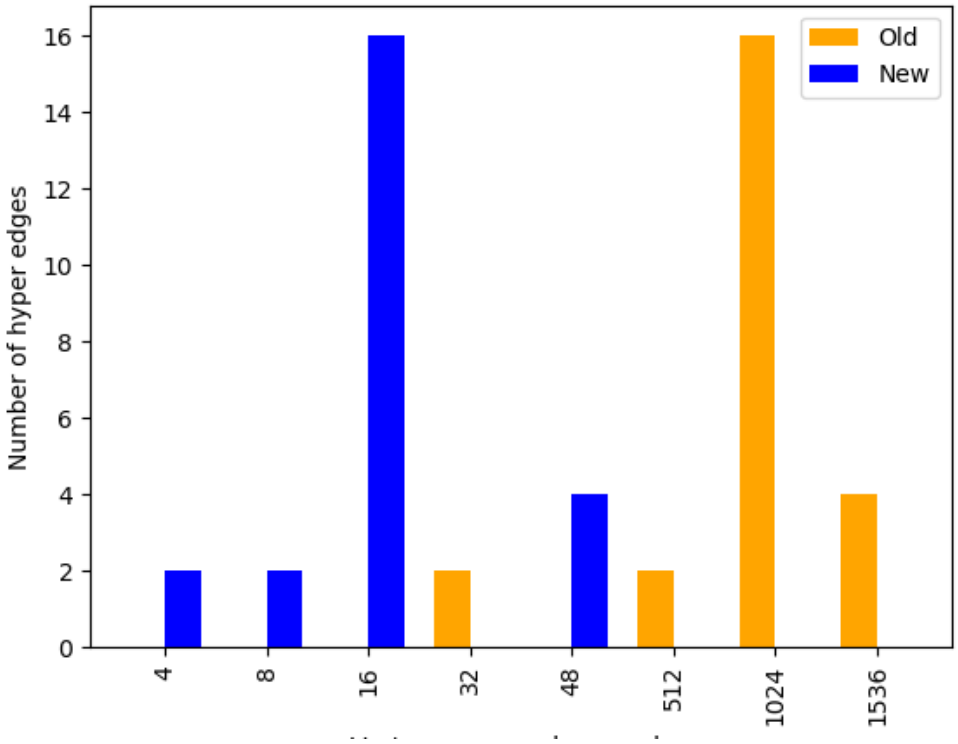


FIGURE 3.10: PPP, strict: A side-by-side histogram of the number of vertex maps per hyperedge. Higher bars to the left is better.

3.4.2 Linear molecules of growing size

Linear molecules, in this context, will refer to the molecules with SMILES strings on the form

$$\underbrace{C\cdots C}_{N}$$

Such a general linear molecule is depicted in its flesh-out form in figure 3.11 below.

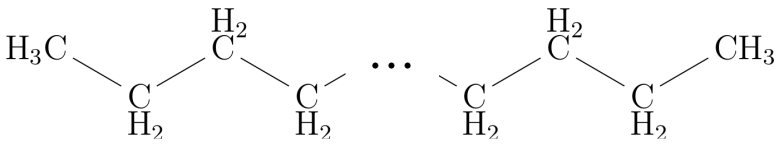


FIGURE 3.11

The advantage of testing with these types of molecules is that one can test for increasing N and that all linear molecules have an excessive amount of hydrogens that cause redundant vertex maps.

To test the two approaches, we will devise four different pattern families of the type

$$\underbrace{C-C-\cdots-C}_{k} \rightarrow \underbrace{C=C=\cdots=C}_{k}$$

Note: This is non-chemical since it introduces bonds while still maintaining the same number of hydrogen-neighbors for each carbons. However this is sufficient to gain insight into the strengths and weaknesses of the old and the new methods.
The four patterns are

- SMALL: A constant-sized pattern where $k = 2$.
- MEDIUM: A pattern where $k = \lfloor N/2 \rfloor$.
- LARGE: A pattern where $k = N - 1$. It is constant in the size of what is not matched by the pattern.
- FULL: A pattern where $k = N$ that covers the entire linear molecule.

The test is carried out as follows:

- For $N \in \{2, \ldots, 5\}$ and for each of the four patterns:
- Compute the Hypergraph-Semigroup using the old approach. Repeat 5 times. Store the average time.
- For $N \in \{2, \ldots, 100\}$ and for each of the four patterns:
- Compute the Hypergraph-Semigroup using the new approach. Repeat 5 times. Store the average time.

(We stop the old approach at $N = 5$ due to the incredibly slow running time.)

The hypothesis is this:
The old approach produces many more vertex maps than needed in these situations. It will suffer in all different pattern sizes, more for larger patterns like LARGE and FULL because there are even more symmetries with the hydrogens.

One could expect the new approach to be much faster in general because the possibilities of the hydrogens will be collapsed down to one and the only thing giving the complexity is finding all possible places for the pattern. Therefore one could expect the SMALL pattern is the most demanding for the new approach since there are more possibilities to try out for the monomorphism algorithm.

The next couple of pages will include figures for results of tests conducted on the SMALL, MEDIUM, LARGE and FULL patterns. Each will have a combined plot with old and new method together as well as them separately to better see the grow of each approach individually.

One thing that is apparent right away is the fact that in this family of molecules the old approach is painfully slow – even for very small molecules of size $N = 4$ or $N = 5$ where it takes tens of seconds.
If one looks at the figures 3.12–3.15, the hypotheses above seems to match up very well with reality. The old method does indeed suffer a tremendous amount more than the new approach across the board. If one were to choose a pattern

family that is better than the rest, it would be the LARGE patterns since the running time for $N = 5$ is halved compared to the other pattern families. The reason for this is however not clear. As predicted, the best cases for the new approach is for bigger patterns like LARGE and FULL.
For $N = 5$ (the biggest N tested in the old approach), the speed-up is quite substantial as one can see from the table below.

Pattern	Speed-up factor
SMALL	18439
MEDIUM	18062
LARGE	18690
FULL	39417

3.4.3 Direct Comparison

It is pretty fair to conclude that the new approach hugely improves runtime across a wide range of examples. It might therefore be interesting to compare the two approaches more directly since there is a lot of overhead calling back and forth between C++ and Python. To make a level playing field for the two implementations the hydrogen-filtering is removed in the new approach. Concretely, this is done by modifying the predicate determining whether something is a non-hydrogen to always return true. With this setting, the sets of vertex maps computed by the two approaches are identical. Furthermore, the timing is done in C++ in contrast to Python and the timer only includes when a new VertexMaps(e) is computed; not counting the iteration over all e or what happens with the vertex map set after it is computed. This serves as a fair comparison since the implementations now solve the exact same problem and are timed equally.

See Figure 3.16 for a couple of examples. The x-axis corresponds to each edge e in the DG (in no particular order) and the y-axis is the time it took to compute VertexMaps(e) for that edge. For each edge the computation of VertexMaps(e) is repeated 20 times to even out running time. The line is the average and error bars show the lowest and highest timings among the 20 samples.

In general, it can be concluded that the new approach is only marginally faster when it comes to a direct comparison, and sometimes it even runs slower than the old approach. For instance in the case of PPP-Strict there is a strange spike for some edges. Investigating the edges in question does not show any interesting differences between the graphs involved in the pattern matching. The pattern size for the edges in the spike does seem smaller than the rest, but not significantly. Another cause of slowdown might be due to lack of optimization when writing the new approach in C++.

Overall, the difference in timing of the direct comparison is very insignificant compared to the fact that the new approach is much more flexible and allows for many customizations such as ignoring hydrogens which we have seen makes a huge difference.

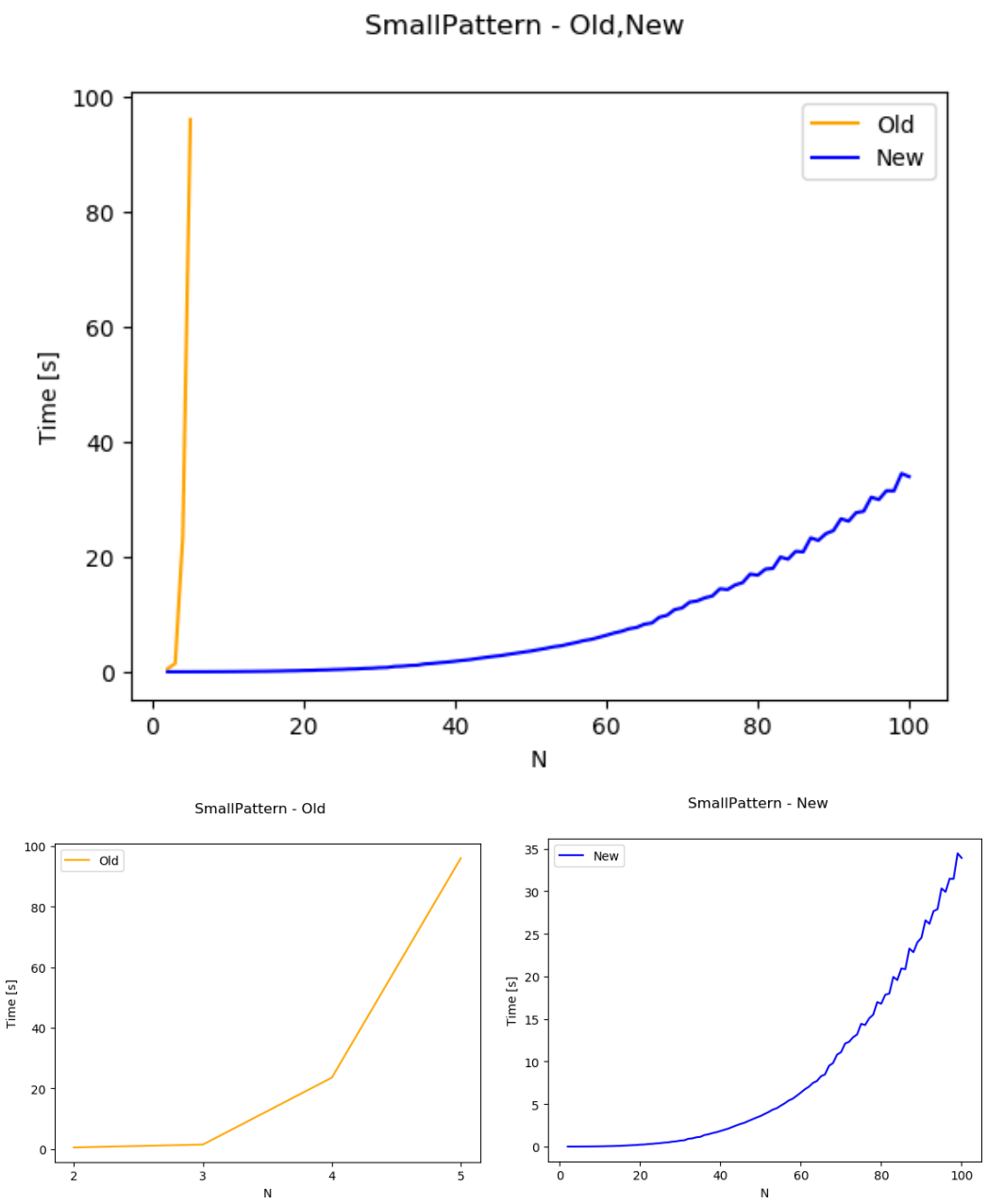

SmallPattern - Old,New
Old
New
Time [s]
N
SmallPattern - Old
SmallPattern - New

FIGURE 3.12

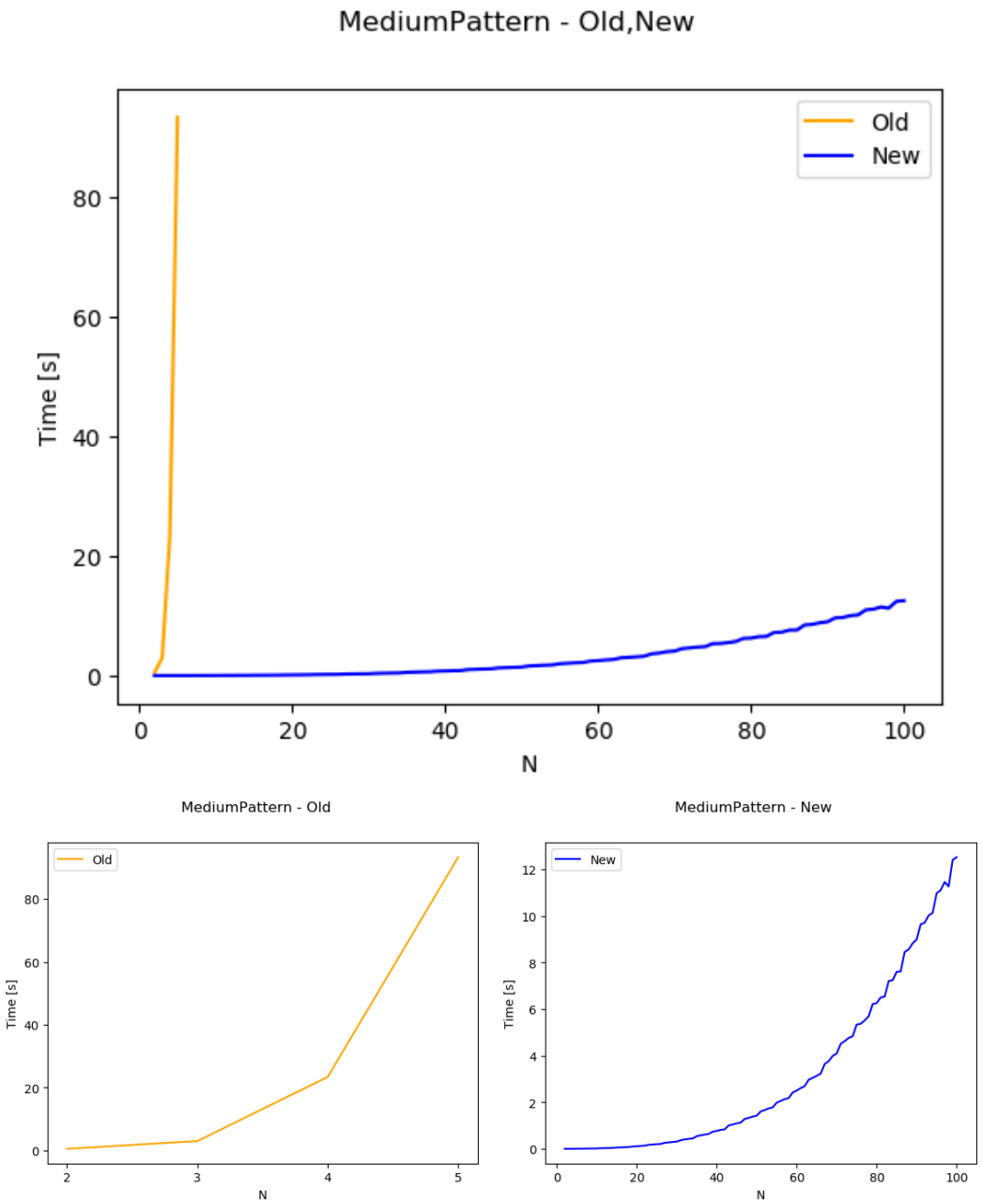

MediumPattern - Old,New
Old
New
Time [s]
N
MediumPattern - Old
MediumPattern - New

FIGURE 3.13

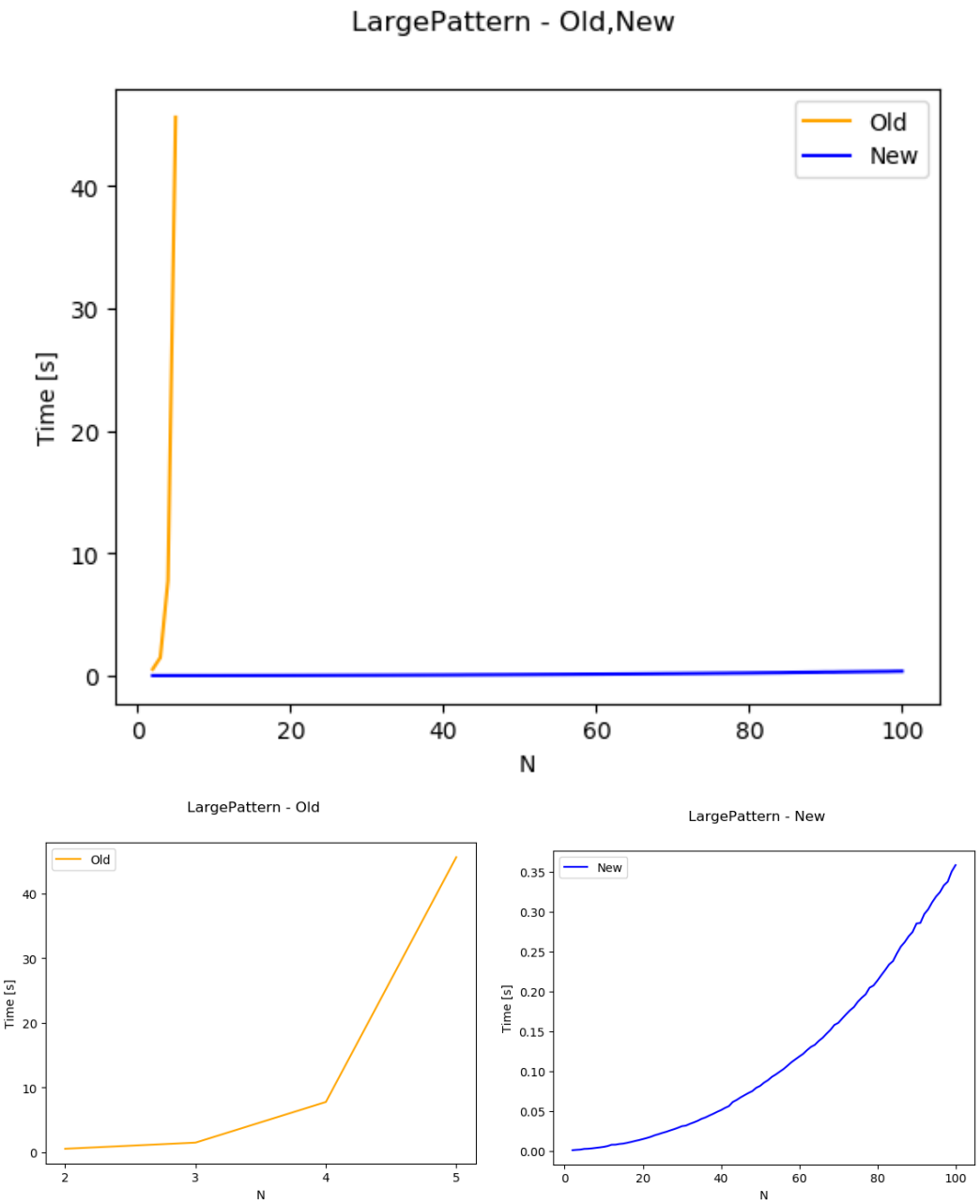

LargePattern - Old,New
Old
New
Time [s]
N
LargePattern - Old
LargePattern - New

FIGURE 3.14

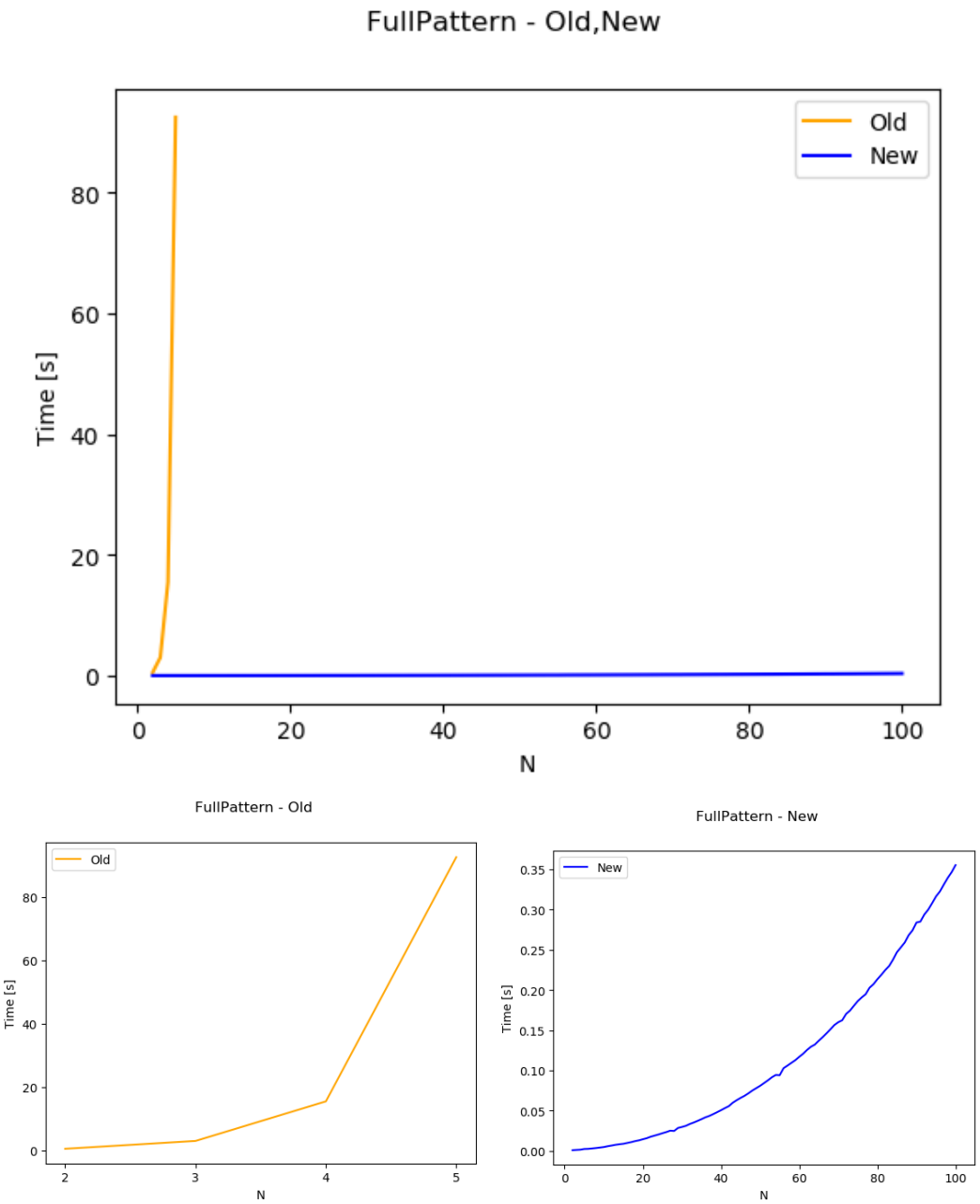

FullPattern - Old,New
Old
New
Time [s]
N
FullPattern - Old
FullPattern - New

FIGURE 3.15

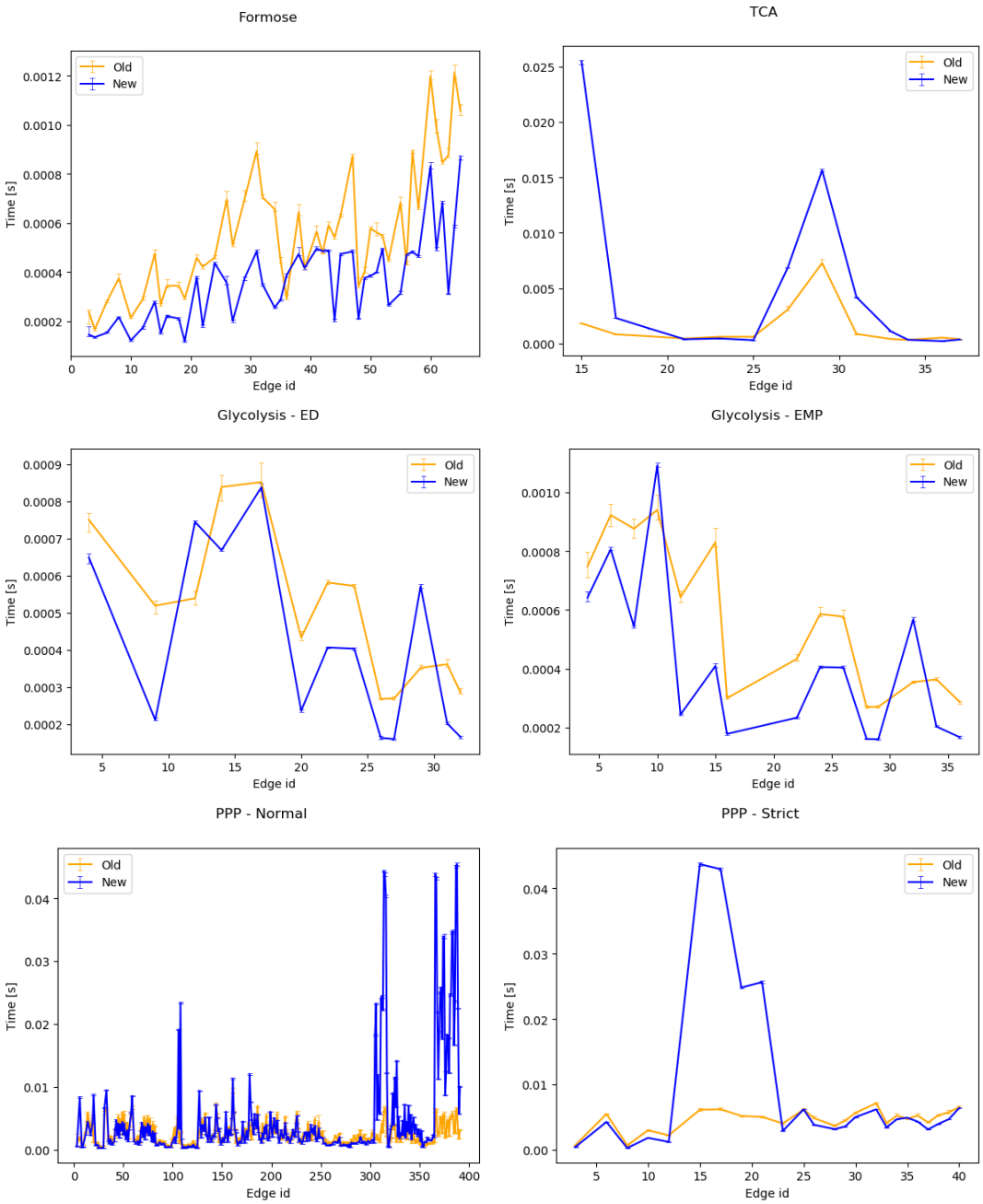


FIGURE 3.16: Examples of old and new approach timed directly against each other. The x-axis corresponds to each edge e in the DG (in no particular order) and the y-axis is the time it took to compute VertexMaps(e) for that edge.

3.5 Code Overview

The code discussed is from Jakob Lykke Andersen's MØD repository, branch `peter/develop`, commit `dedbf77e`.

3 files are important for changes to the vertex map approach:

- `src/mod/lib/DG/VertexMap.cpp`
- `src/mod/lib/DG/VertexMap.h`
- `src/jla_boost/graph/morphism/vf2_class.hpp`

VF2 is the algorithm of choice to compute (sub)graph ismorphisms between graphs in MØD. The implementation works on any graph-like structure with vertices and edges. In our case, we will use it on `union_graphs` representing G and H. A `union_graph` represents a union of multiple connected graphs into a bigger graphs with disjoint connected components. This structure is used because it is desirable to store each molecule as its own graph structure, and then when multiple molecules are needed in the same graph a `union_graph` is used.

In MØD, VF2 is implemented in `src/jla_boost/graph/morphism/vf2.hpp`. This file includes structs for the internal state of VF2 as well as functions that given graphs like G and H as well as a callback (to store the produced vertex maps, as described in the Preliminaries-chapter) will create an internal state, do a depth-first search tree traversal and modify the state as the search progresses. The state is however not accessible outside the functions hindering the fixation of a predetermined partial map.

Therefore a VF2 wrapper class has been created in `vf2_class.hpp`. The class has a public method `force_push` that takes a pair (g, h) of vertices (one from G and one from H) and modifies the internal state by adding the mapping $g \mapsto h$. There is also a `try_push` method that takes a (g, h)-pair, but before modifying the state, it checks if the mapping is actually feasible in accordance to the feasibility rules of VF2 (not discussed here). Finally, the class has the method `match` that carries out the depth-first search tree traversal. The search tree traversal is implemented as a recursive backtracking algorithm in contrast to the original `match`-function from `vf2.hpp` that is iterative with `gotos`. In the VF2 paper a "Candidate Pair Set" is mentioned when trying different potential (g, h)-pairs. This is made more explicit in the recursive implementation by using iterator class `CandPairSet_iterator` instead of raw calls to `state.feasible_candidate1` and `state.feasible_candidate2`.

A snippet from the `match`-function is shown below.

```
const auto cands = cand_pair_set();
for (auto cand : asRange(cands)) {
    boost::tie(v_dom, v_codom) = cand;

    bool was_pushed = try_push(v_dom, v_codom);
```

```
    if (!was_pushed) {
        continue; // Not valid push, try another
    }

    if(state.valid()) {
        // Recurse if valid
        bool continue_search = match(depth + 1);
        if (!continue_search) {
            return false;
        }
        // Then proceed to backtrack
    }

    pop(); // Backtrack
} // end for
```

`vf2_class.hpp` also includes the important `vf2_custom_match`-function which is the entrypoint for the two-stage VF2 approach. Here the subgraphs G' and H' are created, a VF2 class is initialized, fixations are made and the match is started. The subgraphs are created using Boost filtered graphs.

The class `CallbackPartial` represents second stage VF2 call that fixates every non-hydrogen based on the vertex map that the callback received.
The class `CallbackInner` hands the final vertex map to the user using the callback that the user originally specified. If the user callback asks for more maps, this will be passed back to `CallbackPartial`.

The last thing worth mentioning from `vf2_class.hpp` is `VF2Settings` which is a class that gathers all the parameters of needed to create a `VF2` instance. This is handy when having to pass all these parameters through various function calls and creating two different VF2 class with similar settings.

Snippets below shows the majority of `vf2_custom_match`. G' is called `filt_dom` and H' is called `filt_codom`. For a technical reason, a `FilteredWrapper` is wrapped around `filt_dom` and `filt_codom`.

```
// Put the input graphs in wrapped filtered graphs
const auto filt_dom = boost::make_filtered_graph(
    graph_dom,
    settings.edge_incl_dom_pred,
    settings.vertex_incl_dom_pred
);

const auto filt_codom = boost::make_filtered_graph(
    graph_codom,
    settings.edge_incl_codom_pred,
    settings.vertex_incl_codom_pred
);
```

```
const auto wrapped_dom   = jla_boost::makeFilteredWrapper(filt_dom);
const auto wrapped_codom = jla_boost::makeFilteredWrapper(filt_codom);

const auto callback_partial = detail::make_callback_partial(
    graph_dom, graph_codom,
    user_callback, settings
);

const auto wrapped_settings = make_vf2_settings(wrapped_dom, wrapped_codom,
    settings.edge_cmp_pred,
    settings.vertex_cmp_pred,

    settings.edge_incl_dom_pred,
    settings.vertex_incl_dom_pred,

    settings.edge_incl_codom_pred,
    settings.vertex_incl_codom_pred
);

auto vf2 = make_VF2<vf2_problem_selector::isomorphism>(
    wrapped_dom,
    wrapped_codom,
    callback_partial,
    wrapped_settings
);

for (auto p : fixations) {
    vf2.force_push(p.first, p.second);
}

vf2.match();
```

`VertexMap.cpp` previously contained the old vertex map approach with rule composition. This has now been replaced with the new VF2 approach as outlined above. The approach with a few technical details is roughly as follows

- Let $e = (e^+, e^-)$ be the hyperedge for which the vertex maps will be computed. e^+ is the tail of the edge e and is a set of vertices corresponding to molecules. Similar for e^- which is the head.
- Turn e^+ into G (a `union_graph` of molecule-vertices) and e^- into H.
- Irrelevant for the theory, but important for the implementation is the creation of `LabelledUnionGraphs` based on G and H. These wrappers contain atom and bond information for each vertex and edge in the underlying `union_graph`. The `union_graph` in itself is just the topology of the molecules.

- For each rule $r = (L \leftarrow K \rightarrow R)$:

  (Conceptually, we think of one hyperedge of the DG as one rule application, but in theory there could be multiple)

  - Retrieve graphs L and R.
  - Retrieve corresponding labelled graphs of L and R.
  - Compute $M_{LG} : L \rightarrow G$ and $M_{RH} : R \rightarrow H$ using normal monomorphism VF2 calls. These sets are called `left_maps` and `right_maps` in the code.
  - For each $(m, m') \in M_{LG} \times M_{RH}$:
    * Create an empty list of fixations.
    * For each vertex in K, determine the corresponding vertex in L (R) via identity function and then in G (H) via m (m'). This is added to the list of fixations.
    * Create predicates for filtered graphs of G and H that excludes hydrogens as well as predicates that test the equality of vertices and edges based on atom types and bond types. Also create `VF2Settings`.
    * Call `vf2_custom_match` with all the parameters computed above.

`VertexMap.h` contains the `VertexMap` class as well as the default predicates to exclude hydrogens. In the code this vertex-predicate looks like this:

```
struct VertexPredicate {
    ...
    bool operator()(Vertex v) const {
        // If the atom was fixed in a mapping,
        // we should still include it in the filtered graph.
        if (fixed->find(v) != fixed->end()) {
            return true;
        }

        const auto& g = get_graph(*lg);
        const auto& atom_data = get_molecule(*lg)[v];

        // Ignore everything that is hydrogens of degree 1.
        return !(atom_data.getAtomId() == AtomIds::H && out_degree(v, g) == 1);
    };
    ...
};
```

4

Conclusion & Future Work

4.1 Concluding Remarks

In this thesis an approach for assisting in the design of isotope labelling experiments has been presented. It is called the Hypergraph-Semigroup approach and it uses semigroup theory, in particular orbit computations, to create a Pathway Table that gives an overview of where different potential labelling might end up, starting from some start-molecules to some goal-molecules. It can be used to gain insight into what atoms in the start-molecules should be labelled to better understand some chemical network or to distinguish different pathways. A Pathway Comparison Table can also be created which compares only two pathways using inverted orbits to gain even more insight.

One outcome of the thesis has been a Python framework that allows for all the theory presented to be automated based on a model in MØD. It supports importing examples from an example database and creating pathways where orbits can be computed. Additionally, it allows for the automatic creation of Pathway Tables and Pathway Comparison tables.

Another contribution has been optimizing how vertex maps are computed. Vertex maps play a huge role in the Hypergraph-Semigroup approach since it needs to be computed for each hyperedge in a derivation graph. Currently in MØD, the approach has been to use rule composition. A new approach has been presented which uses a two-stage process to ignore unwanted atoms such as hydrogens to reduce the combinatorial explosion, then later add them back. It has been shown how drastically faster this is for chemical examples with a lot of hydrogens. Additionally, it is a more direct approach which makes it more flexible in terms of making other customizations to the algorithm.

The rest of this chapter presents a brief overview of some of the possible other topics that one could delve into if one wanted to pick up where this thesis left.

4.2 Building on this Thesis

The theory and Python framework presented in chapter 2 is very helpful when designing isotope labelling experiment, but it is not fully automatic yet. The user still needs to look at the Pathway Table and decide how to proceed based on the content. The first next steps would therefore be to develop an algorithm that based on a Pathway Table or one or more Pathway Comparison Tables can construct a plan for what isotope labellings to try; preferably with as few different labellings that need to be tried as possible, to save time and money.

Concretely, there are some changes to the Python framework that would be useful:

- Union of pathways: Being able to union two hypergraphs together could be helpful for certain metabolic networks where it is perhaps known that pathways are active in pairs.
- Create a proper class for the Pathway Comparison Table to easier work with it.
- Find a way to automatically create visualizations of orbits and atom traces superimposed on top of the DG, similar to the TCA illustration in Figure 2.5 from Chapter 2.
- Support splitting up long Pathway Tables into smaller tables to save horizontal space.
- Currently it is possible to consider pairs of labels for the `PathwayTable`, but it is not as easy as simply specifying for instance `is_not_hydrogen` as the `mark_candidates` parameter to automatically get all singleton-atoms that are not hydrogens. Having an easy way to specify pairs of atoms would be nice.

Another direction would be to delve into other group and semigroup theory and apply that to the Hypergraph-Semigroup to gain new insight.

4.3 Petri-Nets

The use of Petri-nets in MØD is part of a 3-step analysis that can be carried out:

1. Create a DG of the chemical network.
2. Find a flow solution using integer-linear programming. This describes how to balance input and output in such a way that some number of start-molecules can eventually become some number of goal-molecules.
3. Convert the flow solution into a "recipe" of what reactions to apply in what order to go from the start-molecules to the goal-molecules. It gives a temporal understanding of the pathway.

The last step is done via Petri-net theory. The entire field of Petri-net will not be repeated here, but the essence is given here (based on [18]):

A Petri-net is a hypergraph of places (vertices) and transitions (hyperedges). Some places will initially hold some tokens. Transitions can fire to move tokens from one place to another. See Figure 4.1. t_2 can fire which takes one token from p_1 and puts it in p_3. Then t_2 can no longer fire since there are no tokens left in p_1. We can represent the entire state of the Petri-net by a *marking* which is a tuple with each entry corresponding to the number of tokens in a given place. The corresponding marking for the Petri-net in Figure 4.1 would be $(\overset{p_1}{1}, \overset{p_2}{0}, \overset{p_3}{0}, \overset{p_4}{0})$.

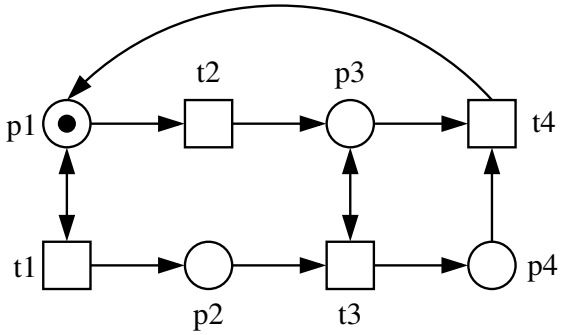


FIGURE 4.1: A example of a Petri-net.

Many interesting questions about Petri-nets are generally really hard, ranging from undecidable to PSPACE-complete and NP-complete. [13]

For step 3 above, a Petri-net is created based on the DG. The initial marking is based on the previously created flow solution, namely the amount of input molecules needed. It is asked if some desired marking is possible to obtain, namely a marking based on the amount of output molecules needed in the flow solution. Essentially, it is a reachability question where the answer is a concrete sequence of transitions that needs to fire to achieve the marking. If Figure 4.1 was a Petri-net created based on a DG, where a flow solution says that one p_1 is needed to create one p_4, then we can ask if it is possible to go from $(\overset{p_1}{1}, \overset{p_2}{0}, \overset{p_3}{0}, \overset{p_4}{0})$ to $(\overset{p_1}{0}, \overset{p_2}{0}, \overset{p_3}{0}, \overset{p_4}{1})$.

The solution to such a reachability question induces a hypergraph similar to the DG, but where the same molecule can occur multiple times as a vertex in the hypergraph and where each vertex has at most one in-edge and out-edge. Vertices with no in-edges are input-molecules and vertices with no out-edges are output-molecules. An example of such a hypergraph can be seen in Appendix A, Figure A.6.

Since it is very similar to a DG, it is possible to directly use the entire Hypergraph-Semigroup approach developed in this thesis on such a Petri-net induced hypergraph. This could lead to direct atom traces where an atom has a unique path through the hypergraph – except for symmetries of molecules – and would give a very good idea of what is going on in the different pathways.

There are however still a lot of challenges:

- Although the Petri-net hypergraph is printable in MØD, it is not easily accessible with a coherent hypergraph interface, so this would have to be implemented in MØD.
- Even with a hypergraph interface for the Petri-net, the tracing would still have to be implemented either in C++ or the entire Petri-net hypergraph would have to be exposed to the Python interface, similarly to how the DG is exposed to Python, to use the Python framework developed in this thesis.
- The Petri-net hypergraph might only be one of many since there can be many ways to reach the desired marking. Therefore to get all possible atom traces one should enumerate all possible Petri-net reachability solutions. MØD simply uses an external tool to find it. This tool does not support enumerating all of them, thus a Petri-net framework would need to be created from scratch in MØD to be able to enumerate all solutions. in such a way that it is possible to distinguish different pathways.

There are a lot of interesting theoretical results that can perhaps also prove useful when working with Petri-nets, especially if one has to implement a Petri-net solving framework, for instance the notion of *stubborn sets* used for state space reduction. [23]

4.4 Constraint-based Membership Testing

In this thesis the primary use of (semi)group theory has been orbits, in particular using orbit calculations to determine where labels could end up and based on that inferred what pathways could have caused that. There are however other aspects of (semi)group theory that could prove useful.

As mentioned in the Preliminary chapter, there exists an algorithm called the Schreier-Sims algorithm that preprocesses a group in polynomial time to be able to answer membership testing efficiently. Membership testing means answering the question "Is the element x in the group G?" However, we work with semigroups so one would first have to investigate possible variants of the Schreier-Sims algorithm for semigroups. Alternatively, it could be assumed for simplicity that all reactions are reversible and use the normal Schreier-Sims algorithm for groups.

Regardless, suppose that it is possible to determine if a given transformation or permutation is in the (semi)group or not. For instance, suppose molecule A has some atom x. It might be interesting to know if x could end up in atom y of molecule B. In other words, one would like to ask the question:

$$[\ldots, \overset{x}{y}, \ldots] \overset{?}{\in} G$$

Not only would the Schreier-Sims algorithm give us an answer, it would also be possible to reconstruct the individual transformations that composed together

gives the above transformation. This would give a very clear picture of how isotope labels would move around in the chemical system.

There is however a problem: Even though we don't care about the other positions than position x, we need to specify the transformation fully. There is no direct way to specify wildcards in a transformation. There are constraints on what transformations would be valid, namely if we want y to be a fixed value, it poses constraints on the hyperedges leading to molecule B etc. Based on the DG and vertex maps, it is possible to infer such constraints and enumerate feasible values for the wildcards.

A constraint-programming library such as Gecode [20] can then be used to enumerate all valid transformations with y at position x and use the Schreier-Sims algorithm to determine if any of them are in the group.

It is unknown if this would give any new insights or if the approach would yield too many valid transformations or be infeasible due to other combinatorics. Regardless, it could probably be used to investigate different atom traces for different pathways and perhaps be used to distinguish them.

4.5 Simulation over Time

The Hypergraph-Semigroup approach presented in this thesis does not give any insight in the passing of time. It is merely a static image of what could happen.

An entirely different approach to atom tracing and isotope labelling would be to do a stochastic simulation of kinetics. For instance, one could run a Gillespie simulation by having x copies of start-molecules with a certain labelling, then randomly apply reactions to existing molecules and keep track of where the labels are. This can be repeated indefinitely and after an adequate amount of time, one would expect it to reach an equilibrium. Looking at all goal-molecules can then give a distribution of where the labels would end up. To do this kind of simulation properly, the rate at which the reactions happens must be taken into account, according to the law of mass action.

A related approach is the creation of ordinary differential equations (ODEs). Usually this is done on a molecular level. Suppose for instance that we have molecules A, B, C and D and reaction between them as

$$aA + bB \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} cC + dD$$

where a, b, c and d are multiplicities of the molecules. The rate at which the reaction happens forward is given by

$$k_1[A]^a[B]^b$$

where k_1 is some reaction constant and $[X]$ is the concentration of molecule X. The rate at which the reaction happens backwards is given by

$$k_{-1}[C]^c[D]^d$$

This can be translated into a system of differential equations

$$\begin{aligned}
\frac{d[A]}{dt} &= -k_1[A]^a[B]^b + k_{-1}[C]^c[D]^d \\
\frac{d[B]}{dt} &= -k_1[A]^a[B]^b + k_{-1}[C]^c[D]^d \\
\frac{d[C]}{dt} &= \ \ k_1[A]^a[B]^b - k_{-1}[C]^c[D]^d \\
\frac{d[D]}{dt} &= \ \ k_1[A]^a[B]^b - k_{-1}[C]^c[D]^d
\end{aligned}$$

This can be done for all reactions in a chemical system and then the ODEs can be solved numerically to get a distribution over time of the different molecules.

To be useful for atom tracing and isotope labelling, the ODEs need to be on an *atomic level*, not molecular level, meaning that each atom would get an ODE. The construction is very similar to the above and can automatically be inferred using the vertex maps. When done on an atomic level it will result in a tremendous amount of ODEs. Solving such a system is likely not feasible. Luckily, there exists tools such as ERODE [9] that reduces the ODE system.

A

Appendix: Large Hypergraphs

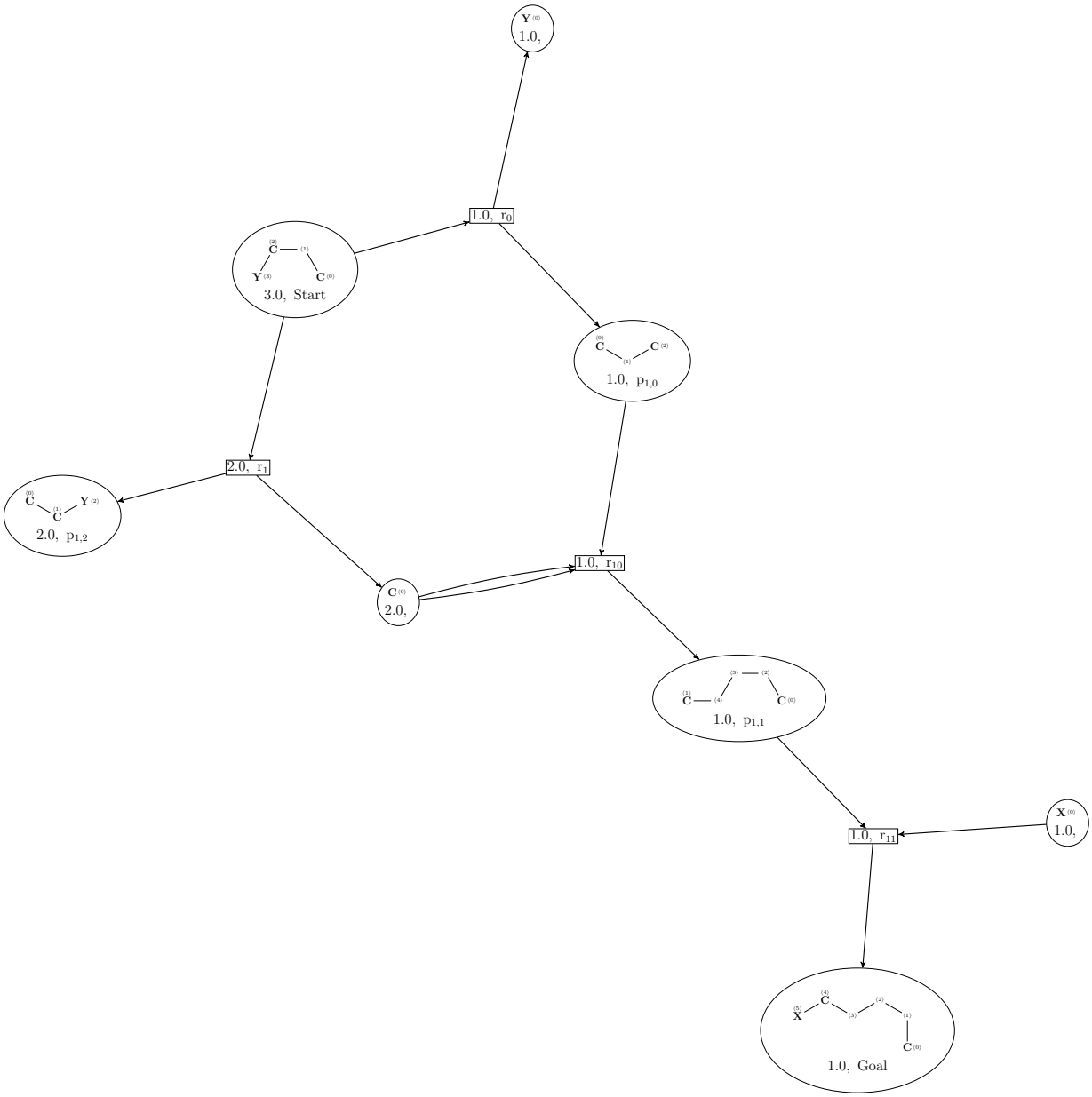


FIGURE A.1: XY-4-Pathways – Pathway 1

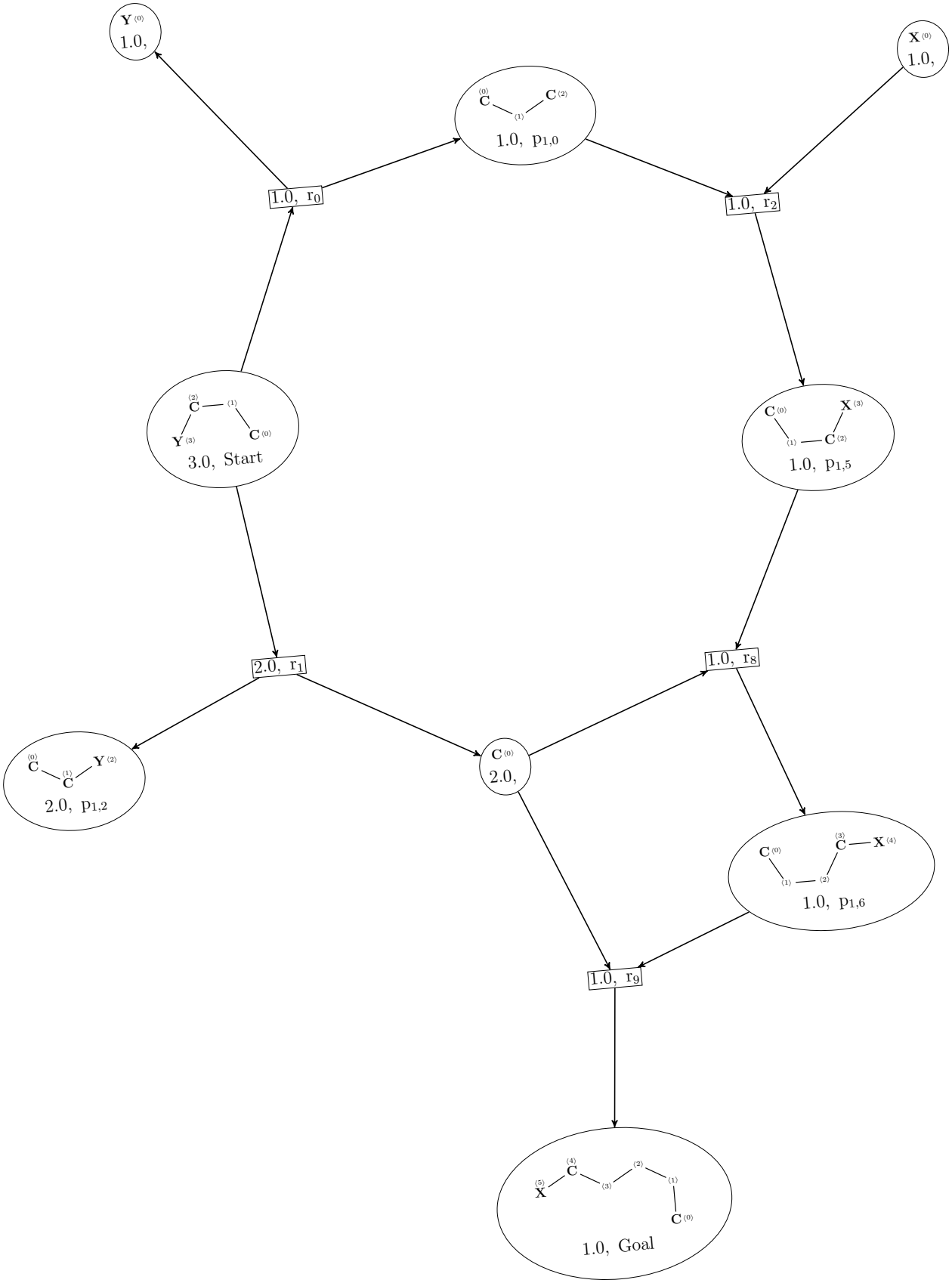


FIGURE A.2: XY-4-Pathways – Pathway 2

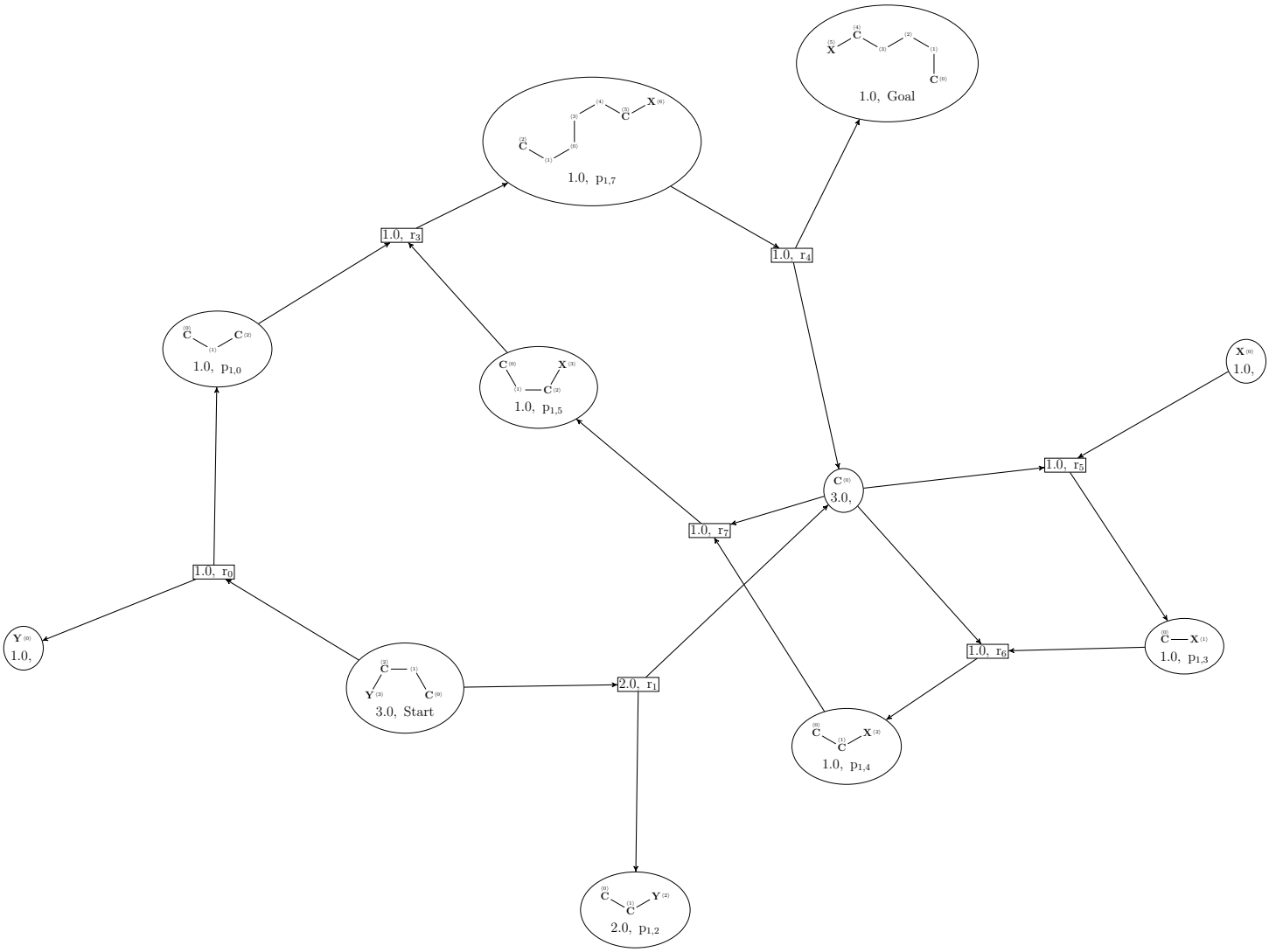


FIGURE A.3: XY-4-Pathways – Pathway 3

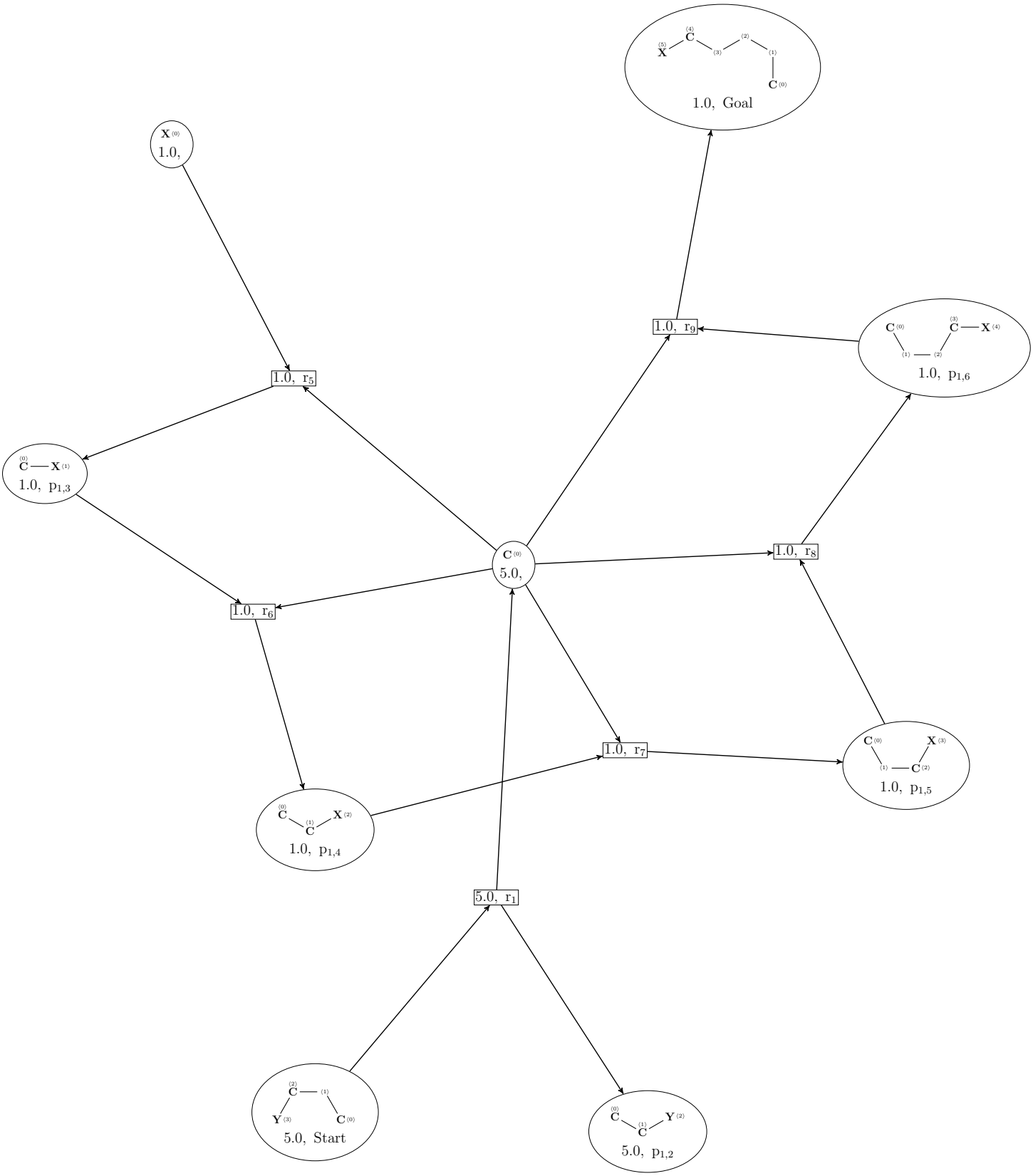


FIGURE A.4: XY-4-Pathways – Pathway 4

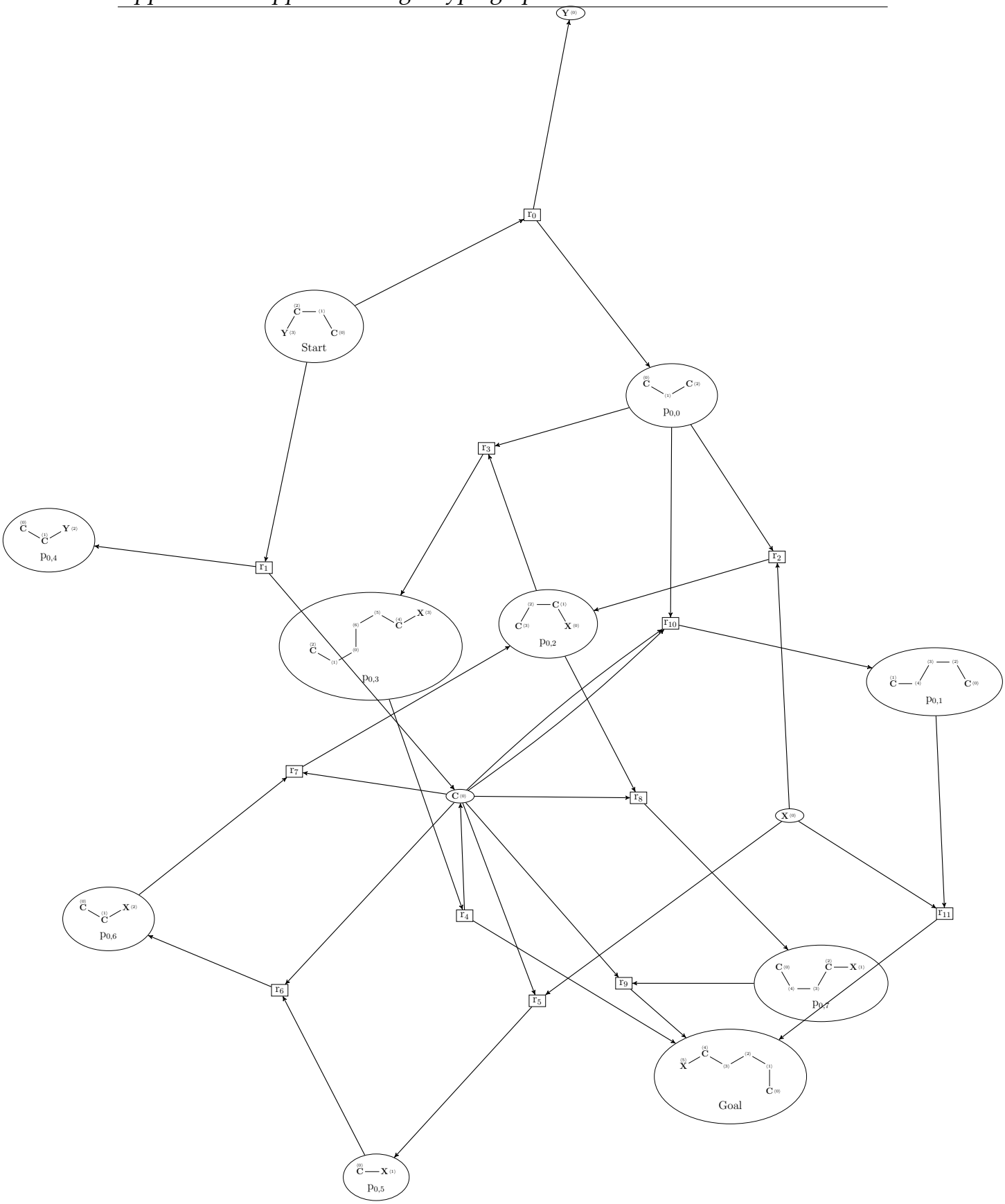


FIGURE A.5: XY-4-Pathways – Combined DG

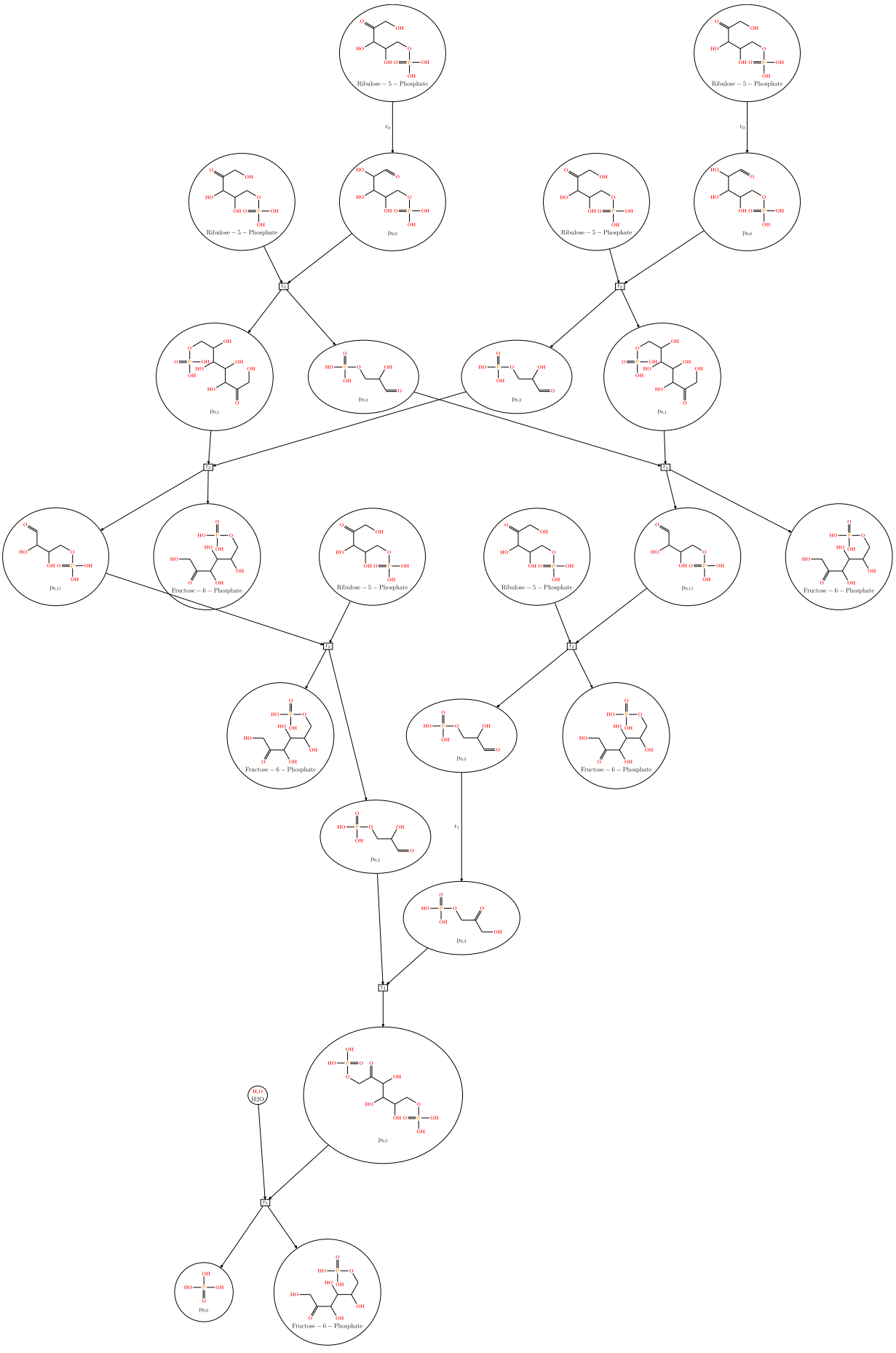


FIGURE A.6: A Petri-net solution for PPP, read top to bottom. Converts six Ribulose-5-Phosphate into five Fructose-6-Phosphate.

B

Appendix: Software Package

The software package that has been handed in can also be found here:

`http://cheminf.imada.sdu.dk/isotope`

It contains a PDF-version of this thesis as well as three folders:

`atomtracing`
: The Python framework developed in Chapter 2.

`example_database`
: The example database also explained in Chapter 2.

`vf2_test`
: Some Bash- and Python-scripts used to gather and analyze data about the performance of the old and new vertex map approach.

As mentioned in Chapter 2 and Chapter 3, the changes to MØD can be found on Jakob Lykke Andersen's MØD repository, branch `peter/develop`, commit `dedbf77e`.

Bibliography

[1] Muhammad Akram. "Citric Acid Cycle and Role of its Intermediates in Metabolism". In: *Cell Biochemistry and Biophysics* vol. 68, no. 3 (2014), pp. 475–478.

[2] Jakob L. Andersen; Christoph Flamm; Daniel Merkle, and Peter F. Stadler. "50 Shades of Rule Composition". In: *Formal Methods in Macro-Biology*. Vol. 8738. Springer, 2014, pp. 117–135.

[3] Jakob L. Andersen; Christoph Flamm; Daniel Merkle, and Peter F. Stadler. "A Software Package for Chemically Inspired Graph Transformation". In: *Graph Transformation: 9th International Conference, ICGT 2016, Proceedings*. Springer, 2016, pp. 73–88.

[4] Jakob L. Andersen; Christoph Flamm; Daniel Merkle, and Peter F. Stadler. "An intermediate level of abstraction for computational systems chemistry". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* vol. 375, no. 2109 (2017).

[5] Jakob L. Andersen; Christoph Flamm; Daniel Merkle, and Peter F. Stadler. "Chemical Transformation Motifs — Modelling Pathways as Integer Hyperflows". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2017).

[6] Jakob Lykke Andersen; Christoph Flamm; Daniel Merkle, and Peter F. Stadler. "Rule Composition in Graph Transformation Models of Chemical Reactions". In: *MATCH Commun. Math. Comput. Chem.* Vol. 80, no. 3 (2018).

[7] László Babai. "Graph Isomorphism in Quasipolynomial Time". In: *CoRR* vol. abs/1512.03547 (2015). arXiv: 1512.03547. URL: http://arxiv.org/abs/1512.03547.

[8] George M. Bodner. "Metabolism Part II: The tricarboxylic acid (TCA), citric acid, or Krebs cycle". In: *Journal of Chemical Education* vol. 63, no. 8 (1986), p. 673.

[9] Luca Cardelli; Mirco Tribastone; Max Tschaikowski, and Andrea Vandin. "ERODE: A Tool for the Evaluation and Reduction of Ordinary Differential Equations". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 310–328.

[10] L. P. Cordella; P. Foggia; C. Sansone, and M. Vento. "A (sub)graph isomorphism algorithm for matching large graphs". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. 26, no. 10 (2004), pp. 1367–1372.

[11] James East; Attila Egri-Nagy; James D. Mitchell, and Yann Péresse. "Computing finite semigroups". In: *Journal of Symbolic Computation* (2018).

[12] Minking Eie and Shou-Te Chang. *A Course on Abstract Algebra*. World Scientific, 2010. ISBN: 9789814271905.

[13] Javier Esparza. "Decidability and complexity of Petri net problems — An introduction". In: *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. Springer, 1998, pp. 374–428.

[14] Avi Flamholz; Elad Noor; Arren Bar-Even; Wolfram Liebermeister, and Ron Milo. "Glycolytic strategy as a tradeoff between energy yield and protein cost". In: *Proceedings of the National Academy of Sciences* vol. 110, no. 24 (2013), pp. 10039–10044.

[15] Allison P. Heath; George N. Bennett, and Lydia E. Kavraki. "Finding metabolic pathways using atom tracking". In: *Bioinformatics* vol. 26, no. 12 (2010), pp. 1548–1555.

[16] Alexander Hulpke. "Notes on Computational Group Theory". In: (2010).

[17] E. M. Luks. "Isomorphism of Graphs of Bounded Valence can be Tested in Polynomial Time". In: *J. Computer System Science* (1982), pp. 42–65.

[18] Tadao Murata. "Petri Nets: Properties, Analysis and Applications". In: *Proceedings of the IEEE*. Vol. 77. 4. IEEE, 1989, pp. 541–580.

[19] R. Pruim and I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005.

[20] Christian Schulte. *Programming Constraint Services*. Springer, 2002.

[21] J. R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: *An Algorithm for Subgraph Isomorphism* vol. 23, no. 10 (1976), pp. 31–42.

[22] Harold C. Urey. "Chemical Properties of the Hydrogen Isotopes". In: *Review of Scientific Instruments* vol. 4, no. 8 (1933), pp. 423–425.

[23] Antti Valmari and Henri Hansen. "Stubborn Set Intuition Explained". In: *Transactions on Petri Nets and Other Models of Concurrency XII*. Springer, 2017, pp. 140–165.

[24] Michael Weitzel; Wolfgang Wiechert, and Katharina Nöh. "The topology of metabolic isotope labeling networks". In: *BMC Bioinformatics* vol. 8, no. 1 (2007), pp. 315–342.