# Parsing Commutative Diagrams

*Christopher Wilson*


4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2024

# Abstract

This project converts between commutative diagrams and sets of composed morphism equations. We design text-representations for sets of morphism equations and commutative diagrams. We then design and implement methods of converting from one text-representation to the other via a graph. Most conversions are fairly direct, aside from converting to a set of equations from a graph, where we consider multiple methods and settle on modifying a depth-first search. This method both converts to a set of equations and mostly avoids redundant information to our set of equations.

# Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Christopher Wilson*)

# Acknowledgements

I would like to thank Dr. Chris Heunen for supervising this project.

# Contents

# Chapter 1

# Introduction

## 1.1 Project Goals

Suppose we have some functions and some equations showing which compositions of these functions are equal, say, for example, the set of equations:

$$\begin{aligned} S = \{&g \circ f = i \circ h, \\ &k \circ j = h, \\ &m \circ l = i, \\ &l \circ k \circ n = p\}. \end{aligned}$$

While representing this information as a set of equations does tell us a lot about how these functions are structured in relation to each other, much of this information is obscured or hard to figure out. For example, is $i \circ k \circ n = m \circ p$? In fact, are $i \circ k \circ n$ and $m \circ p$ even valid function compositions?

To answer this, we could notice that $m \circ l$ and $l \circ k \circ n$ are valid compositions, so $m \circ l \circ k \circ n$ must be a valid combination. Furthermore, $m \circ l = i$ and $l \circ k \circ n = p$, so

$$\begin{aligned} i \circ k \circ n &= m \circ l \circ k \circ n = m \circ p \\ i \circ k \circ n &= m \circ p. \end{aligned}$$

So, the answer to our questions is yes. However, figuring this out was not a trivial task, and the task becomes much more challenging for larger sets with more composed functions equal to each other.

To make this structure more obvious, we can represent the information in $S$ with a *commutative diagram*, which for our purposes is a directed graph (a more rigorous definition is given in Section 2.3). Each function would be an edge, and since we do not know the type of any function we represent the domain and codomain with $\bullet$. Then we use the fact that for $f_1 \circ f_2$ to be a valid composition, the domain of $f_1$ must be the codomain of $f_2$, so the edge $f_2$ will go into the vertex $f_1$ comes out of. Finally, if $f_1 = f_2$, then they must share a domain and codomain, so every composed function in each equation will go between the two vertices that every other composed

function in that equation goes between. For example $m \circ l = i$ will be represented by $\bullet \xrightarrow[l]{} \bullet \xrightarrow[m]{} \bullet$ with $k$ as the arrow from the first to the last vertex. Figure 1.1 shows $S$ as a commutative diagram.

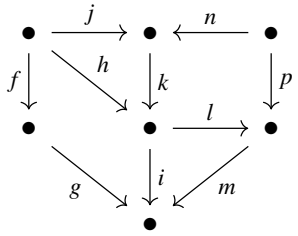



Figure 1.1: A commutative diagram representing set of equations $S$ from Section 1.1.

Now, to answer if $i \circ k \circ n = m \circ p$ with Figure 1.1 we just need to check if the sequence of edges $\bullet \xrightarrow{n} \bullet \xrightarrow{k} \bullet \xrightarrow{i} \bullet$ and $\bullet \xrightarrow{p} \bullet \xrightarrow{m} \bullet$ start at the same vertex and end at the same vertex. If they do, which is the case in Figure 1.1, then $i \circ k \circ n = m \circ p$. If either sequence of edges doesn't appear in the commutative diagram, then the corresponding composed function isn't a valid composition.

There are two main goals in this project:

- Given a set of equations showing which composed functions are equal to others, produce a text-representation of a commutative diagram.
- Given a text-representation of a commutative diagram, produce a set of equations showing which composed functions are equal, ideally with as little redundant information as possible.

Additionally, we want to be able to produce a visual representation of a commutative diagram, ideally with as few edge crossings as possible and with well positioned vertices.

## 1.2 Achievements

We manage to:

- Produce a text representation of a commutative diagram from a set of equations.
- Produce a set of equations given a text representation of a commutative diagram.

Additionally, we largely remove the redundant information in our equations, and use already implemented libraries to produce a visual representation of a commutative diagram with Ti*k*Z.

## 1.3 Report Overview

Chapter 2 introduces category theory to give additional context for commutative diagrams, and explores existing methods of drawing commutative diagrams.

Chapter 3 explains the design of our text-representations of morphisms and commutative diagrams, and explains the algorithms used to convert between them.

Chapter 4 gives details on how the algorithms in Chapter 3 are implemented, what language and libraries where used, and what data-structures and existing code is used to complete certain tasks.

Chapter 5 evaluates the designs and implementations discussed in chapters 3 and 4.

Finally, Chapter 6 summarises the report, and suggests future work.

# Chapter 2

# Background

## 2.1 Directed Graphs

For clarity, we define some graph theory terms that appear often in this report.

**Definition 1.** A directed graph (or digraph) $G$ contains two collections:

1. A non-empty finite set $V(G)$ of vertices, called the vertex set of $G$.
2. A finite set $E(G)$ of ordered pairs of vertices $(v,w)$, where $v,w \in V(G)$, called edges. We say $(v,w)$ connects vertex $v$ to vertex $w$, but *does not* connect vertex $w$ to vertex $v$.

Note that $E(G)$ is a *set*, meaning we are not considering graphs with more than one edge from any vertex to any other.

**Definition 2.** A path is a sequence of consecutive edges such that no edge appears twice in the sequence.

A path consisting of a single edge will be called a "trivial path".

**Definition 3.** A cycle is a path that starts and ends at the same vertex.

## 2.2 Category Theory

Commutative diagrams are heavily used by mathematicians studying category theory, so this section offers a brief overview of category theory to contextualise commutative diagrams.

When studying mathematics, we encounter many different mathematical objects, such as groups, graphs, vector spaces, sets, natural numbers, and rings, all with their own rules and structure. There are often similarities and relationships between these objects, however, when working directly with these objects, these similarities and relationships can sometimes be hard to see through the different notations and ways of describing them, especially if the objects come from different areas of mathematics. Category theory aims to provide a framework for expressing every mathematical object and

its structure, abstracting away the details so the structures and relationships of these objects become more apparent. This is done by defining a new mathematical object: the category.

**Definition 4** ([22, Def 1.1.1])**.** A category $\mathscr{A}$ consists of:

- a collection $\mathrm{ob}(\mathscr{A})$ of objects;
- for each $A, B \in \mathrm{ob}(\mathscr{A})$, a collection $\mathscr{A}(A,B)$ of maps or arrows or morphisms from $A$ to $B$;
- for each $A, B, C \in \mathrm{ob}(\mathscr{A})$, a function

$$\begin{aligned} \mathscr{A}(B,C) \times \mathscr{A}(A,B) &\to \mathscr{A}(A,C) \\ (g,f) &\mapsto g \circ f, \end{aligned}$$

  called composition;
- for each $A \in \mathrm{ob}(\mathscr{A})$, an element $1_A$ of $\mathscr{A}(A,A)$, called the identity on $A$.

satisfying the following axioms:

- associativity: for each $f \in \mathscr{A}(A,B)$, $g \in \mathscr{A}(B,C)$, and $h \in \mathscr{A}(C,D)$, we have $(h \circ g) \circ f = h \circ (g \circ f)$;
- identity laws: for each $f \in \mathscr{A}(A,B)$, we have $f \circ 1_A = f = 1_B \circ f$.

An example of a category is **Set**, where the objects are sets, and the morphisms are set functions between the sets.

Categories do not have to represent mathematical objects such as groups, sets or numbers. For example, the category **Hask** [19] represents programs written in the language Haskell, with the objects being types and the morphisms being functions between those types.

## 2.3 Commutative Diagrams

Suppose we have an arbitrary category $\mathscr{B}$, and we want to talk about some of the morphisms in the category. We can list them using the notation in Definition 4: $f \in \mathscr{B}(A,B)$, $g \in \mathscr{B}(A,C)$, $h \in \mathscr{B}(B,C)$, $i \in \mathscr{B}(D,B)$, and $j \in \mathscr{B}(D,C)$. However, this representation suffers similar problems as the set of equations from the introduction (Section 1.1), namely the structure of the morphisms is hard to intuit. Therefore, we can use the same solution, and represent our morphisms as a directed graph, where vertices are objects and morphisms edges.

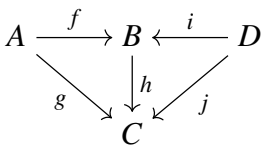



Figure 2.1: A diagram representing the example morphisms from Section 2.3.

The directed graph representation is called a *diagram*. From it, we can easily tell which morphisms share a domain or codomain by just checking an object's incoming/outgoing edges. Additionally, any path in the graph corresponds to a morphism formed by composing the morphisms that lie on that path.

Figure 2.1 contains two non-trivial paths: the path corresponding to the morphism $h \circ f : A \to C$, and the path corresponding to $h \circ i : D \to C$. This means we have two morphisms going from $A \to C$: $h \circ f$ and $g$; and two going from $C \to D$: $h \circ i$ and $j$. If it is the case that $h \circ f = g$ *and* $h \circ i = j$ we call Figure 2.1 a commutative diagram. More generally:

**Definition 5.** A diagram is a commutative diagram, or commutes, if, for every pair of objects ($A$, $B$) in the diagram, every path from $A$ to $B$ is equal as a morphism to every other path from $A$ to $B$.

## 2.4 Existing Commutative Diagram Drawing Methods

There are many existing methods of drawing commutative diagrams, from hand-drawn to dedicated software. However, a method for generating commutative diagrams from a set of equations does not appear to exist. This section discusses some of these methods and their benefits and drawbacks.

### 2.4.1 Hand-drawn

Drawing commutative diagrams by hand is fairly easy and quick, but drawing a good-looking diagram can be much harder and time-intensive. Arrows, for instance, take effort to draw uniformly, and curved arrows can be hard to draw smoothly. It can also be hard to know where to initially position the objects in the diagram to avoid creating a tangled web of intersecting arrows, especially in large and interconnected categories.

### 2.4.2 LaTeX packages

LaTeX is a typesetting system frequently used by mathematicians to typeset mathematical documents. LaTeX is recommended by the American Mathematical Society for authors who want to publish with them [27], and a study in 2009 found that 96.9% of submissions to 4 randomly selected mathematical journals were typeset in LaTeX [4]. Therefore, to look at how to create professional-looking commutative diagrams, we should look at what LaTeX has to offer.

LaTeX packages are used to add extra functionality to LaTeX. There are general-purpose packages such as xy-pic[25] and PGF/Ti*k*Z [28] for drawing many kinds of diagrams, but there are also packages that specialise in commutative diagrams. These specialised packages often build off the more general packages, adding shortcuts and macros to make commutative diagram creation easier.

The Comprehensive TeX Archive, or CTAN, is the main repository of LaTeX packages. Browsing the "Commutative Diagrams" topic [9] gives us 16 packets, some of which build on other packets. All the packets ask us to position the objects ourselves, either

using cartesian coordinates or by using a grid structure. In general, we have to define the starting position and direction of the arrows ourselves, but some packages such as CoDi [3] and DCpic [24] just ask us to define which objects the arrows go between.

The benefits and drawbacks of drawing commutative diagrams using these packages are almost inverse of hand-drawing; they produce professional-looking diagrams, but typing out the diagrams is much slower and less intuitive compared to the speed and simplicity of hand-drawing. As with hand-drawing, these packages also have the problem of a bad initial placement of objects leading to a messy diagram. The ease of fixing a bad placement relative to hand-drawing depends on the packet being used, ones that require us to define the position and direction of arrows will need large complicated re-writes, potentially from scratch, whereas packages that just require object positions will be fairly easy to re-organise. In conclusion, LaTeX packages are good at producing professional looking commutative diagrams, but take much more effort to produce than hand-drawing.

### 2.4.3 Graphical Editors

Dedicated graphical editors for creating commutative diagrams, such as quiver [1] and tikzcd-editor, [26] mitigate the drawbacks of the LaTeX packages by making the creation of the diagrams much easier. The graphical interface is much more intuitive to use than typing out the diagram manually, and the ability to drag objects helps mitigate the object placement problem by making modifications to the diagram much easier.

# Chapter 3

# Design

The overall design approach can be summarised by Figure 3.1.

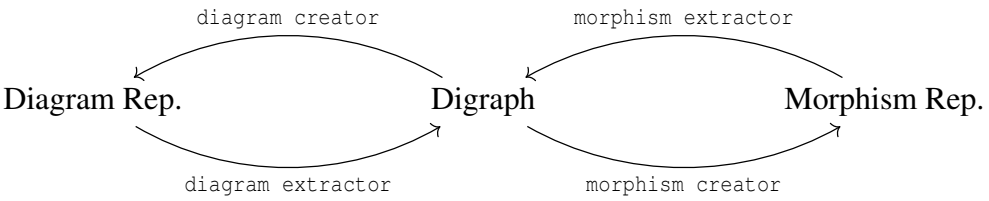



Figure 3.1: Design approach represented as a diagram.

In words, we define a text-based representation for commutative diagrams and morphism equations. We then design methods to convert each representation into a graph, and methods to convert the graph into each representation. Each text representation is stored in a file.

## 3.1 Representations

Along with converting to text representations, we will want to output a visual form of the diagram. To do this, we output Ti*k*Z code that will render the diagram. Therefore, we want our text representations to be similar to LaTeXso that we do not need to do much work to make the conversion to Ti*k*Z. As such, we delimit objects and morphisms in the text representations by wrapping them in braces: {}.

### 3.1.1 Commutative Diagrams

The text representation of a commutative diagram is split in two. First we have the optional labelling section, followed by the actual diagram section. The labelling section comes before the diagram section in the file, but makes more sense once we have seen the diagram section.

In the diagram section, each line represents an edge of the diagram, and takes the form:

```
{morphism}{domain}{codomain}
```

For example, the edge $A \xrightarrow{f} B$ will be represented as `{f}{A}{B}`. This is meant to mimic common notation for expressing the type of a morphism, such as $f : A \to B$, or $f \in \mathscr{A}(A,B)$.

There is a problem with this, what do we do if we want the same object to appear twice? This is where the labelling section comes in. At the top of the document we add a line of the form:

```
L{object}{label}
```

where `object` is the object that appears as a `domain` or `codomain` in the diagram section, and `label` is what we want that object to be displayed as. `L` stands for label and denotes that this line belongs to the label section.

If we go back to our previous example, suppose we want the edge $A \xrightarrow{f} B$ to be $A \xrightarrow{f} A$ instead. Then we can keep the line `{f}{A}{B}`, and at the top of the file add the line `L{B}{A}`. The actual vertex in our graph will still be called `B`, but if we display the graph it will appear as `A`.

Labelling is also useful if we have objects with long names, such as $\mathscr{A}^{\mathrm{op}} \times \mathscr{B}$, which in latex is written as `\mathscr{A}^{\mathrm{op}}\times\mathscr{B}`. Labelling lets us assign a shorthand so we do not need to type this out every time we want to add an edge involving $\mathscr{A}^{\mathrm{op}} \times \mathscr{B}$.

The labelling system is not extended to morphisms because it is not as important that multiple morphisms can appear. When we convert from the morphism representation to a diagram we do not know what objects are involved, only the morphisms names. Therefore, we use a bullet $\bullet$ as a placeholder to represent any possible object, and we do this for all objects so we need to record that every object is a bullet somehow, hence the labels.

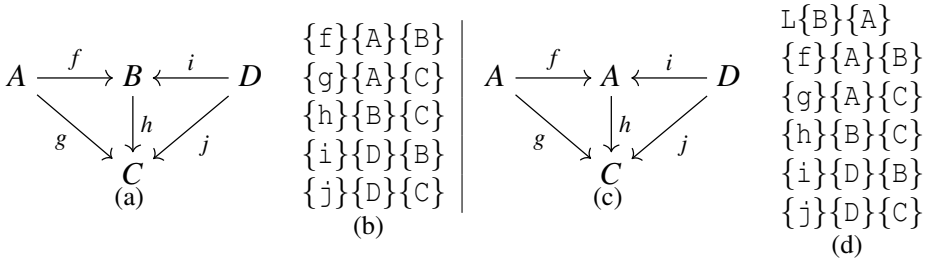



Figure 3.2: Examples of text representation. (a) is Figure 2.1; (b) is the text representation of Figure 2.1; (c) is Figure 2.1 but $B$ is replaced with another $A$; (d) is the text representation of (c).

### 3.1.2 Morphism Equations

Equations are already text representations of the fact that two morphisms are equal, so we can let each line be an equation. However, there are two tweaks/notation choices

that should be highlighted.

First, we allow multiple equalities in an equation, e.g. $f_1 = f_2 = f_3$. We also allow *no* equalities in a line, which is useful for telling us two morphisms can be composed even if there are no equations that can tell us this.

Additionally, consider the composition operator $\circ$. Writing $\circ$ in an equation is useful for two reasons: telling us the morphisms to the left and right are being composed and not operated on in some other way, like being summed, and delineating where two morphisms end and begin. However, in our representation the only operation we can apply to two morphisms is composition, and we wrap each morphism in braces so they are clearly delineated. Therefore, we do not include the composition operator in the representation.

| | |
|---|---|
| $S = \{g \circ f = i \circ h,$ | `{g}{f} = {i}{h}` |
| $k \circ j = h,$ | `{k}{j} = {h}` |
| $m \circ l = i,$ | `{m}{l} = {i}` |
| $l \circ k \circ n = p\}.$ | `{l}{k}{n} = {p}` |
| (a) equations | (b) text representation |

Figure 3.3: Example of a text representation of a set of equations.

## 3.2 Converting

### 3.2.1 To and From Diagram Representation

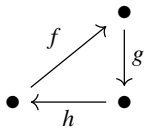
To go from our representation to the graph, we first check the first character of the current line. If it is `L`, we know that we have labels, so extract the `object`, add it as a vertex to the graph, and then extract and attach the `label` to that vertex. W repeat this until we reach a line where `L` is not the first character.

From here we are in the diagram section, so we extract the `morphism`, `domain`, and `codomain`, then add an edge from `domain` to `codomain` with label `morphism` to our graph, also adding any vertices when needed. Doing this for the remaining lines will create our desired graph.

Converting from a graph is slightly more involved. We need to create two strings, one for the label section, and one for the diagram section. We will also need a set $S$ to keep track of all the vertices we have already seen. Then, for each edge from $u$ to $v$ with label $f$ of the graph, we append "$\{f\}\{u\}\{v\}$`\n`" to the end of our diagram string. Next, we check our set $S$ to see if $u$ or $v$ have been seen before. If they have not, then we check to see if they have special labels, and if they are labelled then we add the relevant line to the label string. We then add $u$ and $v$ to $S$. Checking the set $S$ stops us from declaring a label multiple times.

Once we have done this for all edges, we append the diagram string to the label string, giving us our text representation.

To represent a directed cycle, such as the following,




we can store a composed morphism of every edge in the cycle, with the last morphism and the first morphism the same. This means our example would be stored as:

$$h \circ g \circ f \circ h.$$

### 3.2.2 From Morphism Representation

Recall that a composed morphism corresponds to a path in a commutative diagram and that the order morphisms appear in the composition will be the reverse of the order in which the relevant edges appear in the path. For example, $f \circ g \circ h$ corresponds to $\bullet \xrightarrow{h} \bullet \xrightarrow{g} \bullet \xrightarrow{f} \bullet$. Additionally, if two composed morphisms are on the same line they share a domain and codomain, which means the corresponding paths will go between the same pair of vertices.

This means we can view each line of our representation as a list of reversed paths that all go between the same pair of vertices. With this framing, we can then use Algorithm 1 to create our directed graph. The basic idea is to assign a distinct start vertex $u'$ and a distinct end vertex $v'$ to each line, then break each line into a list of composed morphisms then further break each composed morphism into a list of morphisms that make up the composition. We will iterate through each morphism and if the morphism doesn't have an edge in the graph, we give it one. If the morphism does have an edge $(u,v)$ and would have gone into $w$ if it didn't already have an edge, we merge $v$ and $w$ into a single vertex $vw$. Similarly, if the morphism already has an edge and the morphism is the final one in the composition, we merge $u$ and $u'$.

The algorithm assumes a few functions:

- $G$.add_edge($(u,v)$, morphism), which takes a pair of vertices $(u,v)$, and a morphism. The function adds the edge $(u,v)$ to the graph $G$, and records that it represents the morphism;
- $G$.get_edge(morphism), which takes a morphism and returns the edge $(u,v)$ that represents that morphism; and
- $G$.contract_vertices($u,v$), which takes vertices $u$, $v$. This function takes all the edges that go into/out of $u$ and redirects them to go into/out of $v$, leaving $u$ as a floating vertex with no edges attached to it. $u$ is then removed.

In Algorithm 1 we use $G$.contract_vertices in such a way that $v_{start}$ and $v_{end}$ will not change. This keeps the algorithm simpler, and so makes the main ideas clearer, but this isn't optimal. Since $G$.contract_vertices($u$, $v$) needs to update all the edges that go into or out of $u$, when possible the sum of the in-degree and the out-degree of $u$ should be less than or equal to $v$. If we do this, we need to make sure we update $v_{start}$ or $v_{end}$ if we contract it into a different vertex.

**Algorithm 1** Converting morphism text representation to a graph

```
1:  procedure ParseMorphisms(representation)
2:      i ← 0
3:      G ← empty directed graph
4:      for line in representation do
5:          v_start ← i                ▷ Assigning the start and end vertex of the line
6:          v_end ← i + 1
7:          i ← i + 2
8:          for composed morphism in line do
9:              ▷ The first morphism in a composition will correspond to the last in a
                path ◁
10:             v_prev ← v_end
11:             for morphism in composed morphism do
12:                 if morphism has corresponding edge ∈ G then
13:                     (u, v_curr) ← G.get_edge(morphism)
14:                     ▷ We know the current morphism must go into v_prev, but if it
                        already has an edge (v,u) in the graph it must also go from
                        v to u. Therefore, if u ≠ v_prev we contract them into a single
                        vertex. ◁
15:                     G.contract_vertices(u, v_prev)
16:                 else
17:                     v_curr ← i
18:                     i ← i+1
19:                     G.add_edge((v_curr, v_prev), morphism)
20:                 v_prev ← v_curr
21:             G.contract_vertices(v_prev, v_start)
```

## 3.3 Converting to the Morphism Representation

Just as in Section 3.2.2 we re-frame this problem from a graph-theory point of view. Recall that we can view an equation as a set of paths, where each path in a list goes from the same vertex and ends at the same vertex, and our morphism representation is a set of equations. Then the problem becomes: find a set of sets of paths, where each edge in the path is only represented by it's label, such that if we know only that each path in a set has the same start and end-point, we can recreate the graph.

### 3.3.1 A Redundant Solution

A good start for solving this would be to find some possible start and endpoints. To do this, we define the in-degree of a vertex to be the number of edges that end at that vertex, and the out-degree to be the number of edges that start at the vertex.

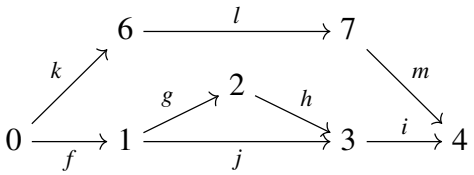
If we encounter a vertex $v$ with an out-degree of two or more, we can build at least two paths with distinct edges from $v$ - the path that starts with the first edge out and the path that starts with the second edge out. If any two of these paths end at the same vertex $u$ those paths form an equation. Similarly, every vertex $u$ with an in-degree of two or more will be the end-vertex of at least two paths that contain edges distinct from one another.

To find all the paths between two vertices we can use a depth-first search, with the modification that we only treat a vertex as visited if it is part of the path we are currently constructing. This avoids creating an infinite loop while letting the same vertex be part of multiple paths.

To find all equations we can build two sets, *Domains* and *Codomains*, by iterating through all the vertices and adding any vertex with out-degree $\geq 2$ to *Domains*, and any vertex with in-degree $\geq 2$ to *Codomains*. Next, for each pair $(u,v)$, where $u \in Domains$ and $v \in Codomains$, we find every path between $(u,v)$. This will generate all our equations, but there may be edges that are not part of these paths, for example, any edge with in-degree zero and out-degree one will not be included.

To account for these, we create a set $UnrecordedEdges$ containing all the edges of the graph at the beginning of the algorithm. Whenever we use an edge in an equation we will remove it from $UnrecordedEdges$. This will identify all these edges, meaning we just need to pick one and build a path that either starts or ends with an edge not in $UnrecordedEdges$, and remove all the edges from the path from $UnrecoredededEdges$. We keep doing this until $UnrecoredededEdges$ is empty.

However, this method produces a lot of redundant information. For example, consider the following commutative diagram.



The algorithm described would output

$$S = \{h \circ g \circ f = j \circ f,$$
$$i \circ h \circ g \circ f = i \circ j \circ f = m \circ l \circ k,$$
$$h \circ g = j,$$
$$i \circ h \circ g = i \circ j\},$$

however we only need to know that $h \circ g = j$, and $i \circ j \circ f = m \circ l \circ k$,. Everything else can be derived from these equations. A set without redundancy would be $S = \{h \circ g = j, i \circ j \circ f = m \circ l \circ k\}$.

Ideally we want to output a representation with no redundant information. There are two main approaches considered: trying to find paths; and trying to use a minimal-cycle basis of the underlying un-directed graph.

### 3.3.2 Graph Traversal

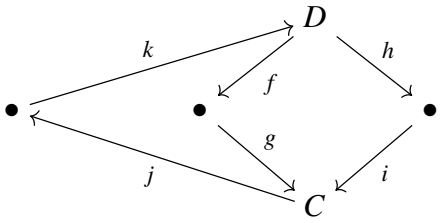
Finding all the paths between every domain and codomain is overkill. This approach aims to modify a depth-first search to reduce the number of redundant paths found as much as possible.

Before we start modifying our depth-first search, we should try and reduce the number of times we need to run the it. The depth-first search will consider every vertex reachable from the one it starts at, so we do not need to find codomains. We can just check each vertex we encounter to see if it is a codomain, and if it is store the current path.

We want to start with a vertex that can reach as many vertices as possible. A vertex with in-degree 0 will not have any vertices that can reach it, so if our goal is to visit every vertex we will need to call a depth-first search from each vertex with in-degree 0 at some point. Therefore, for this approach, we expand the definition of a domain in our graph to be any vertex with out-degree greater than or equal to 2 or in-degree equal to 0. Similarly we will extend the definition of a codomain to be any vertex with in-degree greater than or equal to 2 or out-degree equal to 0.

With these expanded definitions, we gather a list of domains and sort the list in ascending order by the in-degree. When we run a depth-first search we will mark every vertex we visit. Then we iterate through our list, if the vertex hasn't been visited we run our depth-first search from it, if it has we skip to the next domain in the list. Once we have iterated through the list we should have visited every vertex in the graph, unless the graph is a cycle. If the graph is a cycle every vertex has an in-degree and out-degree of one, so a domain won't be detected.

We do not only go for the vertices with in-degree 0 as a start vertex in case we get a graph like the following.




No vertex in this graph has an out-degree or in-degree of 0, but we still need to find the equation $i \circ h = g \circ f$, with domain $D$ and codomain $C$.

Algorithm 2 is the pseudocode for our modified recursive depth-first search. MODIFIED_DFS takes five parameters as input:

- `graph`: whatever graph we are running the depth-first search on;
- `current_vertex`: the vertex of the depth-first search we are currently on;
- `path`: list of vertices showing the path from the vertex MODIFIED_DFS was originally called on to the current vertex;
- `prev_domains`: the domains in `path`, in the order they appear in `path` (so the first item will be the first domain in `path`, the second item the second domain in `path`, e.c.t.); and
- `prev_codomains`: the codomains that appear in `path`.

Note that `path` is not a normal path, instead of a sequence of edges we store the sequence of vertices those edges go between. So for example the path $A \to B \to C$ would be stored as `[A, B, C]`. When we say "path from A" we mean the portion of the path that starts at A. In our previous example saying the "path from B" would mean `[B,C]`.

When we have something of the form `path + vertex`, such as on lines 17 and 22, we mean the list produced by adding `vertex` to the end of `path`, so `[..., A] + B = [..., A, B]`. Similarly, when we have `path1 + path2` we mean the path produced by adding `path2` to the end of `path1`, for example `[..., A] + [B, ...] = [..., A, B, ...]`.

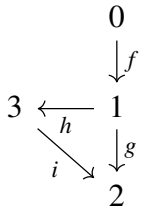
In a recursive depth-first search, if our function is called on a vertex it is the first time we will have seen the vertex, so we mark it as visited. We then check to see if it is a domain or codomain, it is we add it to the relevant collection of previous vertices. Additionally, if the vertex is a codomain we store the path from each previous domain in our path to this vertex. We do this since the vertex is a previously unseen codomain and we do not yet know what domains, if any, form an equation at this codomain. We do the same for all the codomains. This will be explained later.

Lines 12 to 14 continue like any normal recursive depth-first search. If the adjacent vertex is un-visited visit it. Where we start differing is in the following lines that deal with adjacent vertices we have already visited. If we have already visited a vertex it must have two incoming edges, so it is a codomain. Therefore, we loop through the domains in our path in reverse order and store the current path from the domain to the adjacent vertex via the current vertex. If we encounter a domain that already has

**Algorithm 2** Finding equal composed morphisms with minimal redundancy.

```
1: function MODIFIED_DFS(graph, current_vertex, path, prev_domains,
   prev_codomains)
2:     append current_vertex to path
3:     mark current_vertex as visited from path[0]
4:     if current_vertex is a codomain then
5:         for domain in prev_domains do
6:             store path from domain
7:         for codomain in prev_codomains do
8:             store path from codomain
9:         add current_vertex to prev_codomains
10:    if current_vertex is a domain then
11:        add current_vertex to prev_domains
12:    for adjacent_vertex to current_vertex do
13:        if adjacent_vertex hasn't been visited then
14:            MODIFIED_DFS(graph, adjacent_vertex, path, prev_domains,
               prev_codomains)
15:        else
16:            for domain in REVERSE(prev_domains) do
17:                store path from domain + adjacent_vertex
18:                if a different path from domain to adjacent_vertex is stored then
19:                    terminate loop
20:            for codomain in prev_codomains do
21:                if there isn't a path from codomain to adjacent_vertex then
22:                    store path from codomain + adjacent_vertex
23:            for future_path from adjacent_vertex to future codomain do
24:                for prev_codomain in prev_codomains do
25:                    if there isn't a path from prev_codomain to future codomain
                       then
26:                        store path from prev_codomain + future_path
27:                if adjacent vertex not visited from path[0] then
28:                    for domain in prev_domains do
29:                        store path from domain + future_path
30:                        if there exists path from domain to future codomain then
31:                            terminate loop
32:                mark future codomain as visited from path[0]
33:            mark adjacent vertex as visited from path[0]
```

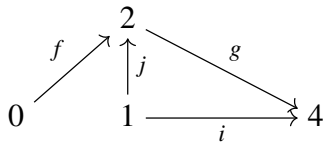
a path we can form an equation from the two paths, so we store the path, then stop. If we continued, every path added would be redundant. As an example, consider the following graph.




We will run the modified depth-first search from 0, and assume when given a choice the depth-first search will travel to the smallest vertex first. We go from 0 to 1, then from 1 to 2. When we reach 2 the `path` will be `[0, 1, 2]`, and `prev_domains` will be `[0, 1]`. Therefore, we will store that there is a path from 1 to 2: `[1, 2]`, and that there is a path from 0 to 2: `[0, 1, 2]`. We then go back up to 1, then down to 3. From 3 we see the only adjacent vertex is 2, but 2 has been visited, so we start working through `prev_domains` (which has not changed from when we were in 2) in reverse. The first domain will be 1, so we store the new path from 1 to 2: `[1, 3, 2]`. We already have a path from 1 to 2: `[1, 2]`, so storing `[1, 3, 2]` can be thought of as saying `[1, 3, 2]` = `[1, 2]`. Now, consider what would happen if we did not stop, and added the path from 0 to 2, which would be `[0, 1, 3, 2]`. We already have the path `[0, 1, 2]`, and we know that `[1, 2]` = `[1, 3, 2]`, which implies that `[0, 1, 2]` = `[0, 1, 3, 2]`. Therefore storing `[0, 1, 3, 2]` is redundant. We know it exists from the already stored paths.

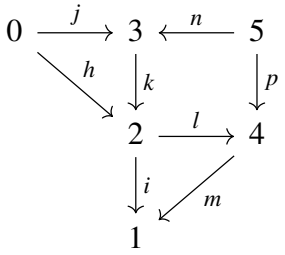
This works because we keep a vertex marked as visited between runs of our modified depth-first search, and do not visit vertices twice. We only add vertices to `prev_domains` if we visit that vertex, so we know that if a vertex is in `prev_domains` it was visited this run. This means that if there is already a path between a domain from `prev_domains` and an already visited adjacent vertex, that path must be equivalent to our path for a certain number of vertices and then splits into a different path at some domain vertex. Since `prev_domains` stores the domains in the order they are encountered in and we work through `prev_domains` in reverse, the first time we see a path from a previous domain to our adjacent vertex this must be the vertex at which the split in the path happens. Therefore, for every domain in the reversed `prev_domains` after this "split domain" we will have a path to the adjacent vertex that we found the first time we visited the adjacent vertex. Furthermore, the path we could store if we did not stop iterating at the split domain would be implied by the existing path and the paths from the split domain to the adjacent vertex, so they would be redundant and thus we do not store them.

This is a similar reason to why we only call our modified depth-first search from unvisited vertices, since it is a depth-first search if we call it from a vertex we have seen before it will just visit the exact same vertices it did the previous time. However, if it is a new run of the modified depth-first search the path and previous domains/codomains will be different. We don't want to find paths we have already found before, but if we don't save any information about the vertices reachable from the codomains we will miss out on some paths. For example, consider the following graph.




If we start our modified depth-first search at 0, and don't store codomains, when we go for our next run starting at 1 we will try to go to 2, see that it is visited, and only store the path `[1, 2]` as we will have no way of knowing that we could have reached 4 via 2. This means we will not find the equation $g \circ j = i$. The loops on lines 7 and 20 store which codomains are reachable from other codomains, and a single path to each. The loop on line 23 uses these paths to simulate our modified depth-first search.

To understand these lines, we will go through a slightly more complicated example. The following is the example commutative diagram from Figure 1.1, with a vertex removed.

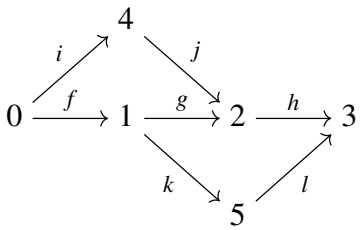



Like in our last example, we will start from 0 and assume whenever given a choice we go for the lower number. From 0 we go to 2, which is both a domain and a codomain, so we store the path from 0. Then we go down to 1, which is just a codomain, so we store the path from 2, then from 0. There is nowhere to go from 1, so we go back up to 2 then along to 4. As 4 is also just a codomain, we store the path to it from 2 and 0. Then we can only go to 1, which has been visited, so we store the path `[2, 4, 1]` from previous domain 2 to 1. We already have a path from 2 to 1: `[2, 1]`, so we stop cycling through our previous domains. For previous codomains, we store the path from 4 to 1, but again we have an existing path from 2 to 1 so do not bother storing a new one. Now we go all the way back up our graph to 0, then along to 3. As 3 is a codomain, we store the path from 0, then the only place we can go is 2, which is already visited, so we store the path `[0, 3, 2]`. Now, we also need to update our codomains, the only one is 3, and it can directly reach 2 along $k$, so we store that path. Then, via 2 it can also reach 4 along $l$ and 1 along $i$, so we store both those paths as well. Now, we go back to 0 and we are done with this run of the modified depth-first search. Throughout this whole run we have been marking each vertex visited as visited from `path[0]`, which was always 0. More generally, `path[0]` is the first vertex in the path, which will always be the vertex the modified depth-first search was called from, so we can use it as a unique id for each run since we will not call a depth-first search from that vertex again.

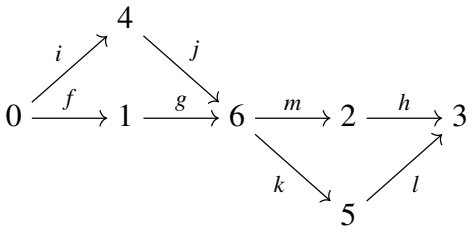
Back to our example, the domain 5 has not been visited yet, so we call a second depth-first search from it. It's first move will be to try and go to 3, but 3 has been visited. As 3 is a codomain we store the path from 5 to 3. However, 3 hasn't been visited from 5 before, so we store a path from our only previous domain, 5, to every codomain

reachable from 3. Namely 2: with path `[5, 3, 2]`; 4: `[5, 3, 2, 4]`; and 1: `[5, 3, 2, 1]`. We also mark 2, 4, 3, and 1 as visited from 5. Now we move to the next adjacent vertex, 4, which is marked as visited from 5. Therefore we store the path from 5 to 4: `[5, 4]`. This successfully finds the equation $l \circ k \circ n = p$.

After we finish our modified depth-first search runs, we will be left with pairs of (domain, codomain)s with either one or multiple paths between them, and single paths between codomains. We can discard any path solely between codomains. For any pair of domain and codomain with more than one path between them, we convert the paths into equations of composed morphisms and add them to our text representation. For (domain, codomain) pairs with only a single path between them we need to be careful. In graphs like the following:




the only (domain, codomain) pair with a single path between them is (0, 3), with possible paths corresponding to $h \circ g \circ f$, $h \circ j \circ i$, and $l \circ k \circ f$. The equations stored will have been $j \circ i = g \circ f$ and $l \circ k = h \circ g$. Note that we have $g \circ f$ and $h \circ g$ in the equations, so we can combine them to get $h \circ g \circ f$, one of our paths. Once we have this path we can then use the equations to create the other paths, so including this single path would be redundant. However, if we slide the bottom triangle along a vertex, so it no longer has an edge in common with the top triangle, we get the following graph:




From this we would create the equations $j \circ i = g \circ f$, and $l \circ k = h \circ m$. From these equations there is no way of knowing we can compose $j$ and $m$ or $k$, or similarly if we can compose $g$ with $m$ or $k$. However, we will still have a (domain, codomain) pair with a single path: (0, 3), which corresponds to either $h \circ m \circ h \circ f$, $l \circ k \circ g \circ f$, $h \circ m \circ j \circ i$, or $l \circ k \circ j \circ i$. Each of these possible paths would tells us that we could compose $j$ or $g$ with $m$ or $k$, so it *is not* redundant.

To deal with this, after generating our paths with Algorithm 2 we go through each pair of (domain, codomain) which have more than 2 paths between them, convert them into equations, and store them, with the added step that whenever we query an edge for the morphism it represents we add (domain, codomain) to a set associated with that edge. As each (domain, codomain) pair will correspond to a line of our representation, this works as a unique id for each line and as an id for each composed morphism in a line.

Then for each pair (domain, codomain) with only one path, we go through each pair of adjacent edges $(edge_1, edge_2)$ in the path. We are check the intersection of the two sets

of line ids for each edge. If the intersection is non-empty that composition is already recorded in our representation, but if it is empty the fact the two morphisms these edges represent can be composed is not recorded. If this was not true for the previous pair of edges, i.e. the pair of edges where $edge_1$ was $edge_2$ had a non-empty set intersection, then we create a new composed morphism from the two morphisms represented by our edges, and add the current (domain, codomain) to the sets for each edge. Doing this stops duplication. If the previous pair of edges also had an empty intersection, we just add the morphism for $edge_2$ to the composed morphism for that pair of edges, and update the relevant set. We keep doing this until we encounter a pair of morphisms with a non-empty set intersection or the end of the path, at which point we will store our composed morphism as it's own line in the text representation.

This will store all the compositions lost between equations, but because we expanded the definition of domain and codomain in our graph, every edge should now be part of a path (unless the diagram is just a series of disconnected cycles). Any edge that does not appear in an equation will have an empty set associated with it, so trivially the intersection of the sets in any pair involving that edge will be empty, so this also deals with the stragglers not involved in any equation.

### 3.3.3 Minimal-Cycle Basis

This approach attempts to construct a minimal set of equations in the graph using a minimal-cycle basis of the underlying graph.

The cycle space of a non-directed graph is the set of all cycles in the non-directed graph. If we take two non-directed cycles which share an edge, we can create a new cycle by joining the cycles together and removing the shared edges. This is known as taking the symmetric difference of the cycles.

**Definition 6.** A cycle basis of an un-directed graph is a minimal set of cycles such that any cycle in the graph's cycle space can be created by taking the symmetric difference of a number of cycles from the cycle basis.

A graph can have many different cycle bases, so in order to reason about whatever basis our algorithm produces, we need a way to somewhat control what basis gets produced.

If we have a weighted non-directed graph, where each edge contains some numeric value, then we can assign the weight of a cycle to be the sum of all the weights of the edges in the cycle. The minimal-cycle basis is then the cycle basis where the sum of the weights of each cycle in the basis is minimised.

Any directed graph will have an underlying non-directed graph, formed by ignoring the fact the directed graph is directed and treating each edge as two-way. Looking at a graph representing a single equation, say $f \circ g = h$, we see that the underlying graph is a cycle.

This means every equation is a cycle in the underlying graph, so if we can store a cycle-basis of the underlying graph we will have all our equations represented with no redundancy.


(a) directed (b) underlying graph

Figure 3.4: The directed graph representing $f \circ g = h$ and its underlying graph.

The minimal cycle basis of a weighted un-directed graph with $m$ edges, $n$ vertices, and integer weights, can be calculated in $O(m^2 n)$ time [21]. Therefore, we just need to ensure that we can encode whatever cycle basis we find.

Adding the edge directions back into a cycle from the underlying graph and counting the out-degree of the vertices based *solely on the edges in the cycle* will let us classify each cycle in the cycle basis:

1. cycles with no domains (vertices with out-degree strictly greater than one) are *directed-cycles*;
2. cycles with one domain are *equation-cycles*; and
3. cycles with multiple domains are *unrepresentable-cycles*.

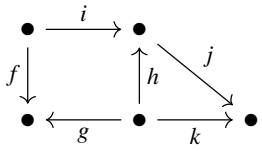
We can represent directed-cycles and equation-cycles as a line of the morphism representation, but we cannot do the same for unrepresentable-cycles. Therefore, to encode our cycle basis, we need every edge of the cycle-space to be contained in a directed-cycle or an equation-cycle. With this in mind, we can now try and find such a cycle-space.

As a starting point we can assign each edge of our underlying graph weight one, which will produce a cycle basis with cycles as small as possible. The major problem here is that some of these cycles may be unrepresentable-cycles. If we could reverse the direction of some edges in our graph, this would not be a problem, as we could convert all of the unrepresentable-cycles into equation-cycles. However, swapping edge direction corresponds to inverting a morphism, and we cannot assume that a morphism has an inverse.

One way to overcome this problem could be to find all the equation-cycles that can be created by the symmetric difference of two or more unrepresentable-cycles. However, this requires checking every possible combination, and in some cases the number of possible combinations could be massive.

Another approach could be to discourage a cycle basis from using an unrepresentable-cycle by increasing the weights along each edge. We could find a cycle basis, and then classify each cycle in that basis. If the cycle is an unrepresentable-cycle then we increase each edge weight in the cycle by one. If the cycle is not representable then we record each edge in the cycle. If every edge in the cycle basis is contained in some directed-cycle or equation-cycle we can safely halt. However, there are directed graphs where every edge in the cycle basis can not be represented by a directed or

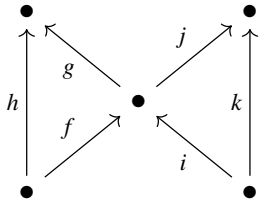
equation-cycle, such as the following.




To deal with these cases, we could either give up on finding a cycle basis after a number of iterations or restrict the cycle space to the edges that are part of a directed or equation-cycle.

To restrict the cycle space, we could use Johnson's algorithm [20] to find all the elementary cycles (cycles where no vertex appears twice) in a graph with $m$ edges, $n$ vertices, and $c$ elementary cycles in $O((n+e)(c+1))$ time. We then classify each cycle, and make a subgraph of all the edges that appear in some directed or equation-cycle. However, for a graph that can be drawn on a piece of paper with no edges crossing (known as a planar graph) with $n$ vertices, the number of elementary cycles is bound above by $2.8927^n$ [5]. This number will be higher for non-planar graphs. This means the time taken to find and iterate through all elementary cycles will be exponential in the worst case.

Additionally, even if we have a good way of finding a cycle basis, the cycle basis won't contain all the information we need. Consider the following diagram:




The minimal cycle basis would correspond to the equations $g \circ f = h$ and $j \circ i = k$, but from this we have no way of knowing that $f$ and $j$, or similarly $g$ and $i$, can be composed. We would have to check every vertex in our cycle after recording it to verify that we are not missing composition information.

# Chapter 4

# Implementation

## 4.1 Language and Libraries

This project is implemented in Python 3.10, for no reason other than familiarity. The only non-standard library directly used in this project is NetworkX [17], which implements the graph data structure. Specifically, we use the **DiGraph** class, which implements a directed graph with only a single edge allowed for each pair of vertices. NetworkX also depends on NumPy [18] for some graph layout algorithms, but NumPy is not directly used in this project. We use version 3.2.1 of NetworkX and 1.26.4 of NumPy.

Another Python graph library considered was igraph [8]. Both libraries are open source and well documented; the most significant difference is that igraph is an interface for a C library with the same name, whereas NetworkX is implemented in Python. The main advantage igraph has over NetworkX is that igraph is significantly faster on larger graphs than NetworkX [23]. On the other hand, since NetworkX is Python-based, which is more familiar, the source code will be easier for me to understand. Also, NetworkX has subjectively nicer documentation.

NetworkX graphs allow us to store both strings and integers as vertices[1]. We can also store information inside of vertices and edges, essentially treating them as dictionaries. These bits of stored information are called attributes, and we make heavy use of them in implementation. The only attribute used across all classes is the "label" attribute, which in vertices stores the label associated with the object represented by the vertex, and in edges stores the morphism that the edge is representing.

## 4.2 Classes

The project has three classes: **Converter**, **MorphismParser**, and **DiagramParser**.

**DiagramParser** and **MorphismParser** are sub-classes of **Converter**, and parse the text representations of commutative diagrams and morphism equations, as described

[1] NetworkX uses "node" instead of "vertex".

in Chapter 3, respectively. They do this by taking a file-path pointing to a text file containing the representation as input on initialisation.

**DiagramParser** and **MorphismParser** both take file-paths as input instead of the raw string mainly because of how much both representations, but especially the diagram representation, rely on having multiple lines. If we want to write something with multiple lines we are going to want to use a text-editor, which will give us a text file to read from anyway, so we might as well use it.

**Converter** mostly contains methods that convert the directed graph into our text representations, or Ti*k*Z code used to generate a graph.

Structuring our classes in this way lets both representation parsers convert into both kinds of representation. Being able to convert back into the representation that was just parsed is helpful for testing and evaluation, as we can get a second parser to parse the output and check if the graph produced is equivalent in terms of edges and edge labels.

### 4.2.1 DiagramParser and MorphismParser

Both classes will raise a file not found error if the file-path does not point to a file.

#### 4.2.1.1 Text Parsing

Both **DiagramParser** and **MorphismParser** do text-parsing in largely the same way, only really differing in what happens to the text afterwards and what form the text takes. The parsers will go through the text-representation line by line, character by character. In general, any character other than "{" or "%" will be ignored, except for "=" in **MorphismParser** and an "L" at the beginning of a line in **DiagramParser**. Both parsers treat "%" as a new-line token, allowing us to add comments to a line. This notation is borrowed from LaTeX. In the test files comments are mostly used to store links to visualisations of the diagram that file is storing.

Ignoring characters does mean errors in the representations will be hard to debug. Error messages or warnings for unexpected characters were considered, but given low priority for implementation.

Whenever "{" is encountered in a line, we use the method `extract_label(line: str, start_pos: int)`, which is implemented in **Converter**, to extract objects/morphisms from the text. `line` is a line of text, `start_pos` is the position of the first character in `line` after an open brace. The method will return all the text between and including the open brace and its *matching* closed brace, and the position of said closed brace. This method is used to extract objects and morphisms from our text representations. Returning the position lets whichever parser is using it skip to the end of the closed brace.

The closed brace being the matching pair and not just the first closed brace encountered is important. We will want the option of displaying whatever is inside the braces in a LaTeX environment, which means we should expect the string inside the braces to contain multiple open and closed brace pairs.

Additionally, this method returns the entire string with no whitespace removal or any other modifications, meaning each occurrence of an object/morphism needs to be exactly equal to every other occurrence in the representation for the parser to realise they are the same. This is inconvenient when writing a large representation with complex morphism names. However, it does allow for the same morphism/object to appear twice in a Ti*k*Z diagram by adding whitespace to differentiate between the two occurrences. Doing so will cause the same symbols to render in Ti*k*Z, but the morphisms/objects will be treated as distinct by the code.

#### 4.2.1.2 DiagramParser

**DiagramParser** largely follows the design described in Section 3.2.1. The graph produced will have string vertices containing the parsed string for each object. If an object has a label assigned this will be stored in the label attribute, otherwise the object string will also be stored in the label attribute. For example, the text

```
L{B}{A}
{f}{A}{B}
```

will produce a **DiGraph** with two vertices: a vertex "`{A}`" with label attribute "`{A}`", and a vertex "`{B}`" with label attribute "`{A}`"; and an edge (`{A}`, `{B}`) with label attribute "`{f}`".

The parsing of label lines differs from normal. Label lines must be written exactly in the form

L{*object*}{*label*}

with no extra characters in between "L" and "{", or "}" and "{". Once *label* has been parsed **DiagramParser** will assume the line is over, and directly move onto the next line. **DiagramParser** will raise an Exception if it detects an unexpected character.

The labels section is parsed differently as an experiment in more strict parsing and how Exceptions could be used. However, there did not seem to be much benefit in changing either method of text-parsing, so they were left different.

**DiagramParser** will also output exceptions if it detects empty braces, i.e. "{}", and if it doesn't find a *morphism*, *domain*, and *codomain* in a line of the diagram section.

#### 4.2.1.3 MorphismParser

**MorphismParser** mostly follows the design described in Section 3.2.2. The main difference is that Section 3.2.2 says we break each line into a list of composed morphisms, then break each composed morphism down into a list of morphisms. It says this because thinking about each line that way makes the algorithm easier to describe, but we do not actually need to do this step in practice. We can just scan through the line, whenever we encounter a "{" we have just found a new morphism and whenever we encounter an "=" we have found the end of the current composed morphism, or the end of the "list of morphisms".

**MorphismParser** will produce a graph where each vertex is an integer with label attribute `{$\\bullet$}`, which renders as • in LaTeX. Edges, as usual, store whatever morphism they represent in the label attribute.

To keep track of what edge is representing which morphism, **MorphismParser** has three dictionaries: `morphs`, which takes a morphism as key and gives the representing edge as a value; `morphs_by_domain`, which takes a vertex as a key and returns a list of the morphisms that the edges that exit the vertex represent; similarly, `morphs_by_codomain` takes a vertex as key and returns the list of morphisms with that vertex as a codomain (i.e. whose representing edges go into the vertex). `morphs` is used in place of $G$.get_edge(morphism) from Algorithm 1. `morphs_by_domain` and `morphs_by_codomain` are used to update `morph` when contracting vertices. When contracting vertices we use the more optimal approach to contraction mentioned at the end of Section 3.2.2.

### 4.2.2 Converter

As mentioned, this class mostly contains methods to convert the directed graph into a text-representation, but also contains a method to produce Ti*k*Z code to display the graph.

**Converter** has two fields,

- `graph`: a NetworkX **DiGraph** representing a commutative diagram;
- `comp_morph_paths`: paths from a graph domain to a graph codomain. It is a dictionary that takes a domain vertex as a key, and returns another dictionary that takes a codomain vertex as key, which finally returns the list of paths found between the original domain and the codomain.

The field `comp_morph_paths` being a dictionary of dictionaries is a hold-over from a method where I would search through all the paths from a domain, so it was beneficial to have only the domain as a key. It could be re-worked to take the tuple (domain, codomain) as a key, but there is very little cost in readability for this method, only space taken, therefore it was decided the refactor was not worth the time it would take. For the use of `comp_morph_paths` see Section 4.2.2.3.

#### 4.2.2.1 To the Diagram Representation

Implemented as described in 3.2.1. We start with an empty list for the label section, an empty list for the diagram section, and an empty set of vertices. Then we iterate through each edge $(u, v)$ in the graph. We extract the label from the label attribute, and the vertices from the edge, and add the relevant string to the diagram list. Then we check both vertices to see if they are in our set, and if a vertex is not we check the vertex's label. If the label does not match the vertex, we add the relevant string to the label array. The vertices are then added to the vertex set, and we continue.

Once all edges are iterated through, add the diagram list to the label list and use Python's built-in `join` function to convert the new list into a string, where each element is separated by a `\n`. This string is our diagram representation, so we return it.

#### 4.2.2.2 To Ti*k*Z

This is done by the `to_tikz_diagram` function, that takes a single parameter `scale`. NetworkX has implemented the function `to_latex_raw` [13] that creates a representation of our graph in Ti*k*Z. For parameters, we use defaults except for `edge_label`, `node_label`, `default_edge_options`, and `pos`. We assign to parameters `edge_label` and `node_label` the string `"label"`, which tells the function to use the label attribute of our vertices and edges to name the vertices and edges of the Ti*k*Z graph. The parameter `default_edge_options` is set to `"[->, auto]"`, and tells Ti*k*Z to draw edges as arrows, and to automatically try and position edge labels so that they are readable. Finally, `pos` sets the position of each vertex in the resulting diagram. For this parameter, we pass in a dictionary generated by another NetworkX function, `spring_layout` [12].

The function `spring_layout` implements the Fruchterman-Reingold force-directed algorithm [15]. The basic idea behind this algorithm is that each vertex emits a force that repels every other vertex, but each edge also applies a force attracting the two vertices it connects together. This spreads the vertices out on the graph, but also means connected vertices should stay closer together than unconnected vertices. By default the algorithm positions all the nodes randomly, then simulates these forces until the node positions are close to stable. However, we can supply initial start positions, and this algorithm does not guarantee edges crossings will not occur.

Therefore, we use NetworkX's `is_planar` [10] function, which is based off Brandes' Left-Right planarity test [2]. If the algorithm is planar, we make use of NetworkX's `planar_layout` [11] function to create a planar layout of the graph using Chrobak and Paynes linear-time algorithm for drawing a planar graph on a grid [6]. We feed the resulting positions in as the start positions for `spring_layout` to try an ensure no edge-crossings. If the graph is not planar, we just use the spring algorithm. This is where the `scale` parameter that `to_tikz_diagram` takes comes into play. `spring_layout` also has a `scale` parameter, which we feed `to_tikz_diagram`'s `scale` parameter. We do this because by default `spring_layout` clusters the vertices too close together, the `scale` parameter spreads them out.

#### 4.2.2.3 To Morphism Representation

We implement the graph traversal method described in Section 3.3.2 in the function `to_morphism_representation`, which takes no parameters.

A step not mentioned in Section 3.3.2 is that we reverse the graph, so that all the edges in the path will be in the same order as the relevant morphisms in the corresponding composed morphism. This was done entirely to avoid having to deal with reversing paths. The morphism representation is a set of strings called `rep`.

The function `__search_for_comp_morphs` is our implementation of Algorithm 2.[2] It has parameters

[2]The double underscore at the beginning of a function name is the closest thing Python has to a private method keyword.

- `graph`: a NetworX **DiGraph**, assumed to the reversed of the graph representing the commutative diagram we are converting;
- `curr_node`: the vertex of the graph we are calling the search from;
- `path`: the list of vertex we visited on the way to `curr_node` from the vertex we initially called the function from, in order of appearance;
- `prev_domains`: a list of tuples of the form (domain vertex, domain position), that containing all the graph domains we have seen in `path`, and their position in `path`, in the order that they appear in `path`;
- `prev_codomains`: similar to `prev_codomains`, a set of tuples of the form (codomain vertex, codomain position) that contains all the graph codomains seen in `path` and their position.

We store paths from domains to codomains in the field `comp_morph_paths`. We store paths from codomains to codomains by creating an attribute containing a dictionary with key (codomain 1, codomain 2) which returns a path from codomain 1 to codomain 2 in each codomain vertex. The attribute is called `codomain_children`. To check if a path exists between two objects, we can check if an entry for it exists in the relevant dictionary.

To mark each vertex as visited we create another attribute called `visited` that contains a set of vertices. We create this attribute the first time we visit a vertex, and to store that the vertex is visited from `path[0]` we just add `path[0]` to the set. This way we can tell if a vertex has ever been visited by if it has the `visited` attribute, then we can tell where the vertex is visited from by checking the set.

As previous domains and codomains are always stored with their position in the path, we can use these and Python list slicing to get the path from a domain/codomain. We can iterate through lists in reverse with the inbuilt function `reverse(list)`.

# Chapter 5

# Evaluation and Discussions

## 5.1 Going To and From the Diagram Representations

Since the text representation is so close to the graph already, there isn't much to evaluate. When converting to a graph we only have to iterate through each line once. At each line we either add a vertex or an edge to the graph. NetworkX uses dictionaries to store nodes and adjacency information [7][1] which have on average constant time to add elements, so the time complexity of our algorithm should be $O(l)$, where $l$ is the number of lines.

Similarly, the worst case for converting to the diagram text representation will be a graph where every vertex has a label, since we need to create a line for every vertex and for every label. Thus the worst case for algorithm will have time complexity $O(v+e)$, where $v$ is the number of vertices and $e$ is the number of edges in the graph.

However, while the diagram text representation is easy for the computer to parse, there is a reason the test files contain links to the visualisations of the graphs: the text-representation is not easy for humans to parse. This is why we have the Ti*k*Z converter, but even that can suffer from a poor object placement. Consider for example the diagram from the introduction:

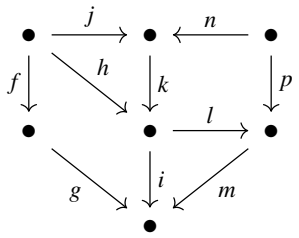



Figure 5.1 shows the output of `to_tikz_diagram`. It's not clear which edge many of the labels belong to, and it is oriented at an awkward angle. A better solution for visualising could have been to make an application that directly output a visualisation

[1] Also from inspecting source code.

of the diagram that the user could interact with from the morphisms, removing the need for a text-representation at all.

If implementing the current project goals took less time implementing something like this would have been considered, but designing and implementing the graph to morphism conversion took much longer than expected. Building a visual interface from the start was not considered do to a lack of experience in building anything remotely similar to what would be needed. Therefore, the more interesting and novel challenge of figuring out the conversions were prioritised. There are also existing tools to draw commutative diagrams, there are none to extract morphism equations from them.

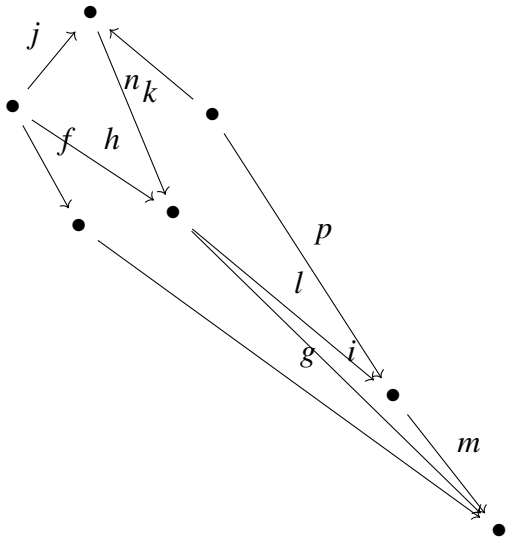



Figure 5.1: The generated Ti*k*Z representation of Figure 1.1.

Originally the plan was to experiment with Graphviz [14] layout algorithms, especially dot's algorithm [16]. However, due to a problem with Big Sur's command line tools the required libraries could not be installed. Therefore it was decided to priorities working on implementing new code rather than getting third-party code to function.

## 5.2 To and From Morphism Representation

### 5.2.1 From Morphism Representation

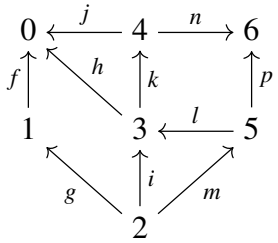
Similar to the approach for parsing the diagram representation, the morphism representation only scans through the representation once. While iterating through the morphisms the most costly procedure we do is contraction, but we minimise the cost of this procedure as much as possible. Loosely, the worst cost would be updating half of the existing morphisms, so the algorithm will be bounded by $O(m^2)$ where $m$ is the number of morphisms in the representation. In practice it should be lower however, since we shouldn't ever update $m$ edges.

### 5.2.2 To Morphism Representation

In hindsight, `prev_codomains` and `prev_domains` should have been one list. Once we reach a domain with a path to a codomain $C$, every domain *and* codomain further back in the path will already have a path to $C$, so we can stop checking both domains and codomains there. Additionally, while we make use of recursion to pass information into a function, we don't make use of the fact we can also pass information out of our recursive call. Instead we iterate over the vertices in our path when the algorithm is already going to revisit those vertices, we should be passing information along to be used then. Both these flaws are relics of previous iterations of the design, which due to a lack of time couldn't be reworked once the ideas of the design were fully formed.

Evaluating if an unseen graph contains redundancy automatically is hard, as if method to detect redundancy exists it can be used to remove it. Therefore, to evaluate redundancy the implementation was tested on both the graphs in Appendix A and the reverse of the graphs in Appendix A. These tests work by converting each graph to its morphism representation, then feeding that representation to a **MorphismParser**. If the graphs had the same shape, then the test passed. Redundancy was then manually checked for. This method had the double benefit of also further testing **MorphismParser**.

A way redundancy can slip in has been identified in the reverse of Figure 1.1. For reference:



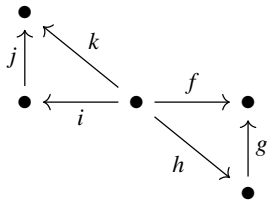

The equations found are:

$$f \circ g = h \circ i,$$
$$h = j \circ k,$$
$$i = l \circ m,$$
$$n \circ k \circ l = p,$$
$$k \circ i,$$
$$h \circ l.$$

The two single composed morphisms, $k \circ i$ and $h \circ l$, are redundant, we could figure out $h \circ l$ from $h = j \circ k$, which tells us $j \circ k$ is possible, and $n \circ k \circ l$, which tells us $k$ can be composed with $l$, so we can form $k \circ l$, then $j \circ k \circ l = h \circ l$. We can use similar logic to figure out $k \circ i$ from $n \circ k \circ l$ and $i = l \circ m$.

A possible way to prune this redundancy is: when we get an empty intersection to then form a second set that is the union of all the incoming edge's line-id sets. If the intersection of this set and the second edge's set is still empty, *then* we create our composed morphism. Although a graph that produces a redundant equation may exist, one has not yet been found.

The method of converting to morphism equations then parsing those equations and comparing graphs doesn't work for unseen graphs, as not all diagrams can be re-created solely using equations. Consider for example:




The only equations we can extract from this diagram are: $j \circ i = k$ and $g \circ h = f$, neither of which tell us that $k$, $i$, $h$, or $f$ share a domain. We would need typing information to be able to reconstruct this commutative diagram.

# Chapter 6

# Conclusions

We have managed to design and implement an algorithm that can take a set of composed morphism equations and produce a commutative diagram. We have also managed to design and implement an algorithm that takes a commutative diagram and produces a set of equations, and additionally the algorithm largely manages to avoid including redundant information in the produced set of equations.

We also can produce a visualisation of our commutative diagram, although the layout could still be improved.

The most challenging section of this project was designing and implementing our graph to composed morphism equations algorithm, as there was little existing work found to build off. A variety of approaches were researched and considered, and we settled on modifying a depth-first search algorithm.

For future work, a front-end that directly builds a graphical commutative diagram would greatly improve the usability of this project. Additionally, further work can be done to refine the graph to morphism equation algorithm by taking full advantage of its recursive nature, further removing the redundant information that still gets output, and showing it's time complexity.

# Bibliography


[1] Nathanael Arkor. *quiver*. Version 1.3.0. Aug. 2023. URL: https://q.uiver.app.

[2] Ulrik Brandes. 'The left-right planarity test'. In: *Manuscript submitted for publication* 3 (2009).

[3] Paolo Brasolin. *Commutative-diagrams – CODI: Commutative diagrams for tex*. Aug. 2023. URL: https://ctan.org/pkg/commutative-diagrams (Accessed: 18/10/2023).

[4] François Brischoux and Pierre Legagneux. 'Don't format manuscripts'. In: *The Scientist* 23.7 (2009), p. 24.

[5] Kevin Buchin et al. 'On the Number of Cycles in Planar Graphs'. In: *Computing and Combinatorics*. Ed. by Guohui Lin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 97–107. ISBN: 978-3-540-73545-8.

[6] Marek Chrobak and Thomas H Payne. 'A linear-time algorithm for drawing a planar graph on a grid'. In: *Information Processing Letters* 54.4 (1995), pp. 241–246.

[7] conradlee. *What scalability issues are associated with NetworkX?* 2015. URL: https://stackoverflow.com/questions/7978840/what-scalability-issues-are-associated-with-networkx (Accessed: 1/4/2024).

[8] Gábor Csárdi and Tamás Nepusz. 'The igraph software package for complex network research'. In: *InterJournal, Complex Systems* (2006), p. 1695.

[9] *CTAN: Commutative Diagrams*. URL: https://www.ctan.org/topic/diagram-comm (Accessed: 16/10/2023).

[10] NetworkX Developers. *is_planar - NetworkX 3.2.1 documentation*. 2023. URL: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.planarity.is_planar.html (Accessed: 31/3/2024).

[11] NetworkX Developers. *planar_layout - NetworkX 3.2.1 documentation*. 2023. URL: https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.planar_layout.html (Accessed: 31/3/2024).

[12] NetworkX Developers. *spring_layout - NetworkX 3.2.1 documentation*. 2023. URL: https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.spring_layout.html (Accessed: 31/3/2024).

[13] NetworkX Developers. *to_latex_raw - NetworkX 3.2.1 documentation*. 2023. URL: https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_latex.to_latex_raw.html (Accessed: 31/3/2024).

[14] John Ellson et al. 'Graphviz and dynagraph—static and dynamic graph drawing tools'. In: *Graph drawing software* (2004), pp. 127–148.

[15] Thomas M. J. Fruchterman and Edward M. Reingold. 'Graph drawing by force-directed placement'. In: *Software: Practice and Experience* 21.11 (1991), pp. 1129–1164. DOI: https://doi.org/10.1002/spe.4380211102. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211102. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102.

[16] Emden R Gansner et al. 'A technique for drawing directed graphs'. In: *IEEE Transactions on Software Engineering* 19.3 (1993), pp. 214–230.

[17] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart. 'Exploring Network Structure, Dynamics, and Function using NetworkX'. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[18] Charles R. Harris et al. 'Array programming with NumPy'. In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[19] HaskellWiki. *Hask — HaskellWiki*. 2023. URL: https://wiki.haskell.org/index.php?title=Hask&oldid=65566 (Accessed: 15/10/2023).

[20] Donald B. Johnson. 'Finding All the Elementary Circuits of a Directed Graph'. In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84. DOI: 10.1137/0204007. eprint: https://doi.org/10.1137/0204007. URL: https://doi.org/10.1137/0204007.

[21] Telikepalli Kavitha et al. 'An $\tilde{O}(m^2n)$ Algorithm for Minimum Cycle Basis of Graphs'. In: *Algorithmica* 52.3 (Nov. 2008), pp. 333–349. ISSN: 1432-0541. DOI: 10.1007/s00453-007-9064-z. URL: https://doi.org/10.1007/s00453-007-9064-z.

[22] Tom Leinster. *Basic Category Theory*. 2016. arXiv: 1612.09375 [math.CT].

[23] Timothy Lin. *Benchmark of popular graph/network packages V2*. May 2020. URL: https://www.timlrx.com/blog/benchmark-of-popular-graph-network-packages-v2 (Accessed: 30/3/2024).

[24] Pedro Quaresma. *DCpic – Commutative diagrams in a LaTeXand TeXdocuments*. 2013. URL: https://ctan.org/pkg/dcpic (Accessed: 18/10/2023).

[25] Kristoffer H. Rose and Ross Moore. *xypic – Flexible diagramming macros*. 2013. URL: https://ctan.org/pkg/xypic (Accessed: 26/3/2024).

[26] Yichuan Shen. *tikzcd-editor*. 2020. URL: https://tikzcd.yichuanshen.de (Accessed: 18/10/2023).

[27] American Mathematical Society. *AMS :: Why Do We Recommend Latex?* URL: https://www.ams.org/publications/authors/tex/latexbenefits (Accessed: 16/10/2023).

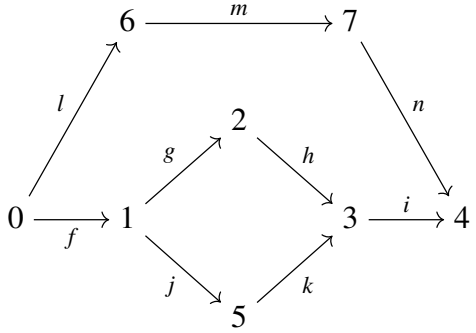
[28] The PGF/TikZ Team. *pgf – Create PostScript and PDF graphics in TEX*. 2023. URL: https://ctan.org/pkg/pgf (Accessed: 26/3/2024).

# Appendix A

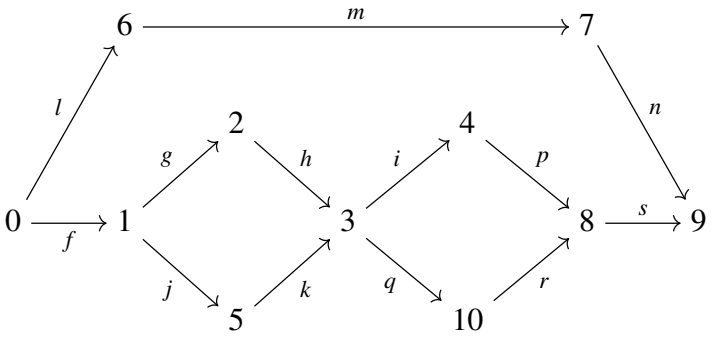
# Testing Graphs

The following graphs and the reverse of the graphs were used to evaluate the conversion to morphism-equations and back.

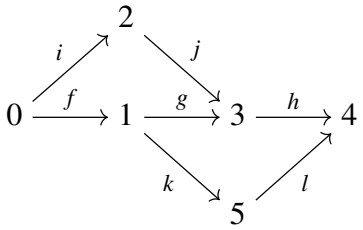
**Bridge:**



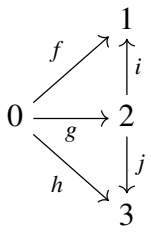

**Goggles:**



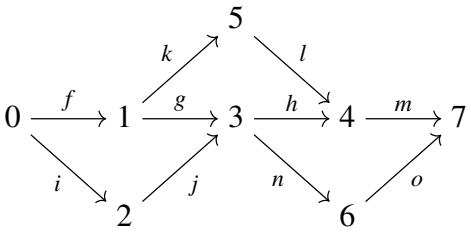

**Staggered:**



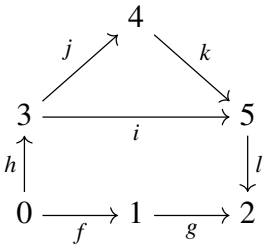
**Limit:**



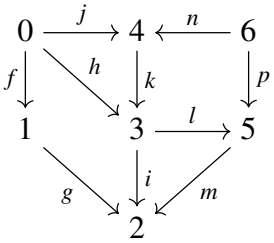

**Doubly Staggered:**



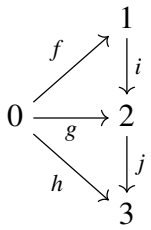

**House:**




**Introduction Figure:** Needs to have two runs of the modified depth first search.

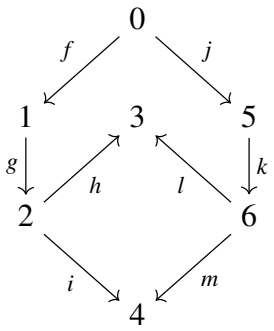



**Big Cycle Triangles:** The goal with this graph is to try and get the depth-first search to go to deep.

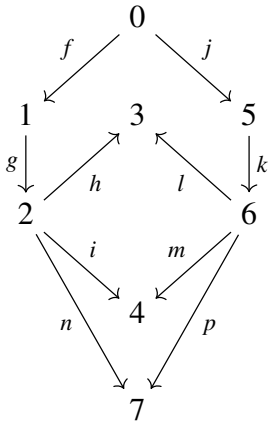



**Bulky Diamond:** Largely a trap for the cycle-basis approach, but also reveals the need

for codomain → codomain paths:




**Bulkier Diamond:** Seeing if making it bigger causes problems.




**Three Branches:** Most previous graphs only have two out-edges that cause an equation to arise, so testing that our methods work when vertices have more edges.

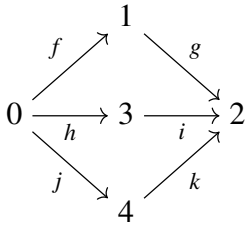
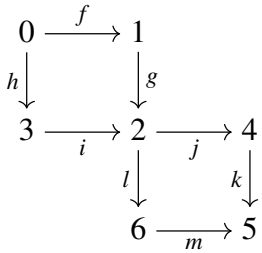



**Figure of Eight:** Needs the single-branch path to tell us about composition.




**Wedge:** testing graphs with cycles