The Category of Directed Labelled Multigraphs - Implementation of Some Universal Constructions Report

Marta Zheplinska, Department of Systems Design Engineering,
University of Waterloo

September 2023

Introduction

The report describes in detail the implementation of a program that calculates such universal constructions in category theory as pullback, pushout, product, and coproduct, particularly in the context of the category of *directed labelled multigraphs*. General definitions (without proofs) about the category theory and, in particular, categories of directed labeled multigraphs will be given below. The features of the application, the selected project architecture, and, accordingly, the implementation of the previously specified categorical objects with examples will be described in the following sections. A discussion of potential improvements and expansion of the application can be found at the end of the report.

Category Theory

"Mathematics of mathematics" - that's what category theorist Eugenia Cheng calls it in her book "The Joy of Abstraction"[1]. Category theory serves as a conceptual framework for the entire field of mathematics. Its fundamental principle involves studying objects within a specific context, rather than in isolation. Here, "context" refers to the relationships between objects, and it's important to note that a single object can take on different "roles" when examined within different contexts. While is set theory, focus is on objects themselves, category theory focuses on relations (morphisms) between objects. A category is formed by a collection of objects and their relations. That's it. Now we can apply it to any collections and study them! It can be the category of sets and functions between them, the category of groups and their homomorphisms, or you might want to study your family tree, because you still cannot get who is your cousin's husband for you. Category theory works everywhere.

Terminology

Definition. A category C is a 5-tuple (O, M, s, t, c), where

O is a set of objects;

M is a set of morphism;

$s : M \to O$ is a source of morphisms;

$t : M \to O$ is a target of morphisms;

$c : D \to M$ is a composition of morphisms, where
$D = \{(a, b) : s(b) = t(a); a, b \in M\}$;

A category C satisfies the associativity law for composition and the existence of an identity arrow for each object [3].

Definition. For any category C the dual (or opposite) category C^{op} is given as follows:

Objects: $obC^{op} = obC$;

Morphisms: given any objects a, b. Then $C^{op}(a, b) = C(b, a)$;

Identities are the same;

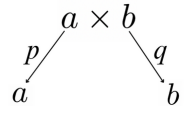
Composition is reversed: $f \circ g = g \odot f$, where $\circ$ is a composition in category C, $\odot$ is a composition in a category C^{op}.

An important concept in category theory is **universal properties**, which are extreme cases of certain constructions in the categories. Looking ahead, each universal construction is an initial object in some auxiliary category. In various branches of mathematics, we often encounter these universal constructions and work with them without mentioning that they are universal. In this report, we consider such universal properties as product, coproduct, pullback, and pushout. Also we will mention initial and terminal objects, the simplest examples of universal properties.

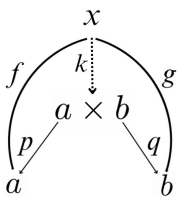
Definition. An object I in category C is *initial* if for every object $X \in C$ there is a unique morphism $I \to X$.

Definition. An object T in category C is *terminal* if for every object $X \in C$ there is a unique morphism $X \to T$.

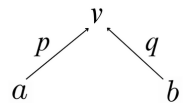
Definition. Given objects a, b in the category C. A *product* of them is an object $a \times b$ equipped with morphisms p, q,



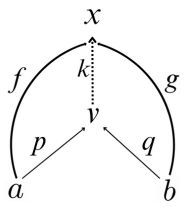
such that $\forall x \in C \;\; \exists! k : x \to a \times b$ making the following diagram commute:



Definition. Given objects a, b in the category C. A *coproduct* of them is an object v together with morphisms p, q

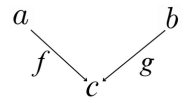


such that $\forall x \in C \;\; \exists! k : v \to x$ making the following diagram commute:

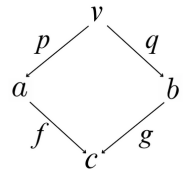


Definition. A limit over a diagram is a *universal cone* over it.

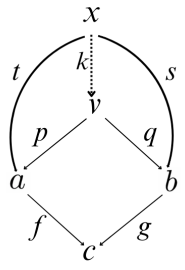
Definition. Given the diagram of the following shape in the category C.



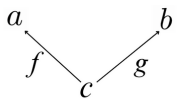
A *pullback* of this diagram is an object v equipped with morphisms p, q



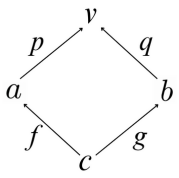
such that $\forall x \in C \;\; \exists! k : x \to v$ making the following diagram commute:



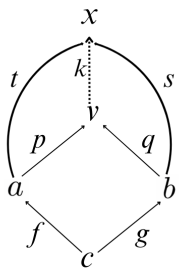
Definition. Given the diagram of the following shape in the category C.



A *pushout* of this diagram is an object v equipped with morphisms p, q



such that $\forall x \in C \;\; \exists! k : v \to x$ making the following diagram commute:



Category theory allows you to go to a higher level of abstraction to cover more areas of mathematics and see their similarities without going into details. However, to study a specific category, we must zoom out and get into the details, which we do in the next section.

The DLMG (Directed Labelled Multigraphs) category

The specific category that we consider and implement programmatically is the category of *directed labelled multigraphs* (*DLMG*)[2]. The objects of this category are obviously directed labelled multigraphs, and the arrows represent morphisms (structure preserving mappings) between them.

Definition A directed labelled multigraph is an 8-tuple $\Gamma = (V, E, \Lambda_V, \Lambda_E, \sigma : E \to V, \tau : E \to V, \lambda_V : V \to \Lambda_V, \lambda_E : E \to \Lambda_E)$, where

V is the set of vertices,

E is the set of edges,

Λ_V is a finite alphabet of vertex labels,

Λ_E is a finite alphabet of edge labels,

$\sigma : E \to V$ is the function assigning a tail to the edge,

$\tau : E \to V$ is the function assigning a head to the edge,

$\lambda_V : V \to \Lambda_V$ is the function giving labels to vertices,

$\lambda_E : E \to \Lambda_E$ is the function giving labels to edges.

Definition A morphism $\phi : \Gamma \to \Gamma^{'}$ in the category *DLMG* is a 4-tuple $\phi = (\phi^{Vertex} : V \to V^{'}, \phi^{Edge} : E \to E^{'}, \phi^{Label,V} : \Lambda_V \to \Lambda_{V'}, \phi^{Label,E} : \Lambda_E \to \Lambda_{E'})$ such that always

$\sigma(\phi^{Edge}(e)) = \phi^{Vertex}(\sigma(e))$,

$\tau(\phi^{Edge}(e)) = \phi^{Vertex}(\tau(e))$,

$\lambda_V(\phi^{Vertex}(v)) = \phi^{Label,V}(\lambda_V(v))$,

$\lambda_E(\phi^{Edge}(e)) = \phi^{Label,E}(\lambda_E(e))$.

It should be noted that some resources indicate that only edges have labels, but we also label graph vertices. Each vertex has the unique label, i.e.

$$\forall v, v' \in V \lambda_V(v) = \lambda_V(v') \Rightarrow v = v'.$$

On the one hand, we can allow two different vertices to have the same labels. This will add additional restrictions when defining morphisms between graphs. However, we further say that two different vertices must have different labels. Hence, there will be no need to check that two vertices with the same labels will map to vertices with the same labels or to the same vertex. Additionally, we consider that all edges are distinct in the *DLMG* category, i.e

$$\forall e, e' \in E : \sigma(e) = \sigma(e') \ \& \ \tau(e) = \tau(e') \ \& \ \lambda_E(e) = \lambda_E(e') \Rightarrow e = e'.$$

Universal constructions in *DLMG* category

Initial and terminal objects

The *initial* object in the *DLMG* category is the *empty* graph with no vertices, no edges, and empty labels' alphabets. The *terminal* object is the labelled multigraph with exactly one vertex and exactly one loop. The alphabets of edge and vertex labelles contain exactly one label each.

Product

Let $A = (V_A, E_A, \sigma_A : E_A \to V_A, \tau_A : E_A \to V_A, \lambda_{V_A} : V_A \to \Lambda_{V_A}, \lambda_{E_A} : E_A \to \Lambda_{E_A})$ and $B = (V_B, E_B, \sigma_B : E_B \to V_B, \tau_B : E_B \to V_B, \lambda_{V_B} : V_B \to \Lambda_{V_B}, \lambda_{E_B} : E_B \to \Lambda_{E_B})$ be two directed labelled multigraphs. The *product* of them is a labelled directed multigraph $\Pi = (V_\Pi, E_\Pi, \sigma_\Pi : E_\Pi \to V_\Pi, \tau_\Pi : E_\Pi \to V_\Pi, \lambda_{V_\Pi} : V_\Pi \to \Lambda_{V_\Pi}, \lambda_{E_\Pi} : E_\Pi \to \Lambda_{E_\Pi})$ where

$V_\Pi = V_A \times V_B$,

$E_\Pi = E_A \times E_B$,

$\Lambda_{V_\Pi} = \Lambda_{V_A} \times \Lambda_{V_B}$,

$\Lambda_{E_\Pi} = \Lambda_{E_A} \times \Lambda_{E_B}$,

$$\sigma_\Pi : E_\Pi \to V_\Pi,\ e = (e_a, e_b) \mapsto (\sigma_A(e_a), \sigma_B(e_b)),$$

$$\tau_\Pi : E_\Pi \to V_\Pi,\ e = (e_a, e_b) \mapsto (\tau_A(e_a), \tau_B(e_b)),$$

$$\lambda_{E_\Pi} : E_\Pi \to \Lambda_{E_\Pi},\ e = (e_a, e_b) \mapsto (\lambda_{E_A}(e_a), \lambda_{E_B}(e_b)),$$

$$\lambda_{V_\Pi} : V_\Pi \to \Lambda_{V_\Pi},\ v = (v_a, v_b) \mapsto (\lambda_{V_A}(v_a), \lambda_{V_B}(v_b)).$$

together with two directed labelled multigraph morphisms $\pi_A : \Pi \to A$ by

$$\pi_A^{Vertex} : V_\Pi \to V_A \text{ with } (v_a, v_b) \mapsto v_a,$$

$$\pi_A^{Edge} : E_\Pi \to E_A \text{ with } (e_a, e_b) \mapsto e_a,$$

$$\pi_A^{Vertex\ Label} : \Lambda_{V_\Pi} \to \Lambda_{V_A} \text{ with } (l_{v_a}, l_{v_b}) \mapsto l_{v_a},$$

$$\pi_A^{Edge\ Label} : \Lambda_{E_\Pi} \to \Lambda_{E_A} \text{ with } (l_{e_a}, l_{e_b}) \mapsto l_{e_a}.$$

and $\pi_B : \Pi \to B$ by

$$\pi_B^{Vertex} : V_\Pi \to V_B \text{ with } (v_a, v_b) \mapsto v_b,$$

$$\pi_B^{Edge} : E_\Pi \to E_B \text{ with } (e_a, e_b) \mapsto e_b,$$

$$\pi_B^{Vertex\ Label} : \Lambda_{V_\Pi} \to \Lambda_{V_B} \text{ with } (l_{v_a}, l_{v_b}) \mapsto l_{v_b},$$

$$\pi_B^{Edge\ Label} : \Lambda_{E_\Pi} \to \Lambda_{E_B} \text{ with } (l_{e_a}, l_{e_b}) \mapsto l_{e_b}.$$

The *product* of two directed multigraphs in the *DLMG* category is a **Tensor product** of them together with two morphism as defined above.

Coproduct

Let $A = (V_A, E_A, \sigma_A : E_A \to V_A, \tau_A : E_A \to V_A, \lambda_{V_A} : V_A \to \Lambda_{V_A}, \lambda_{E_A} : E_A \to \Lambda_{E_A})$ and $B = (V_B, E_B, \sigma_B : E_B \to V_B, \tau_B : E_B \to V_B, \lambda_{V_B} : V_B \to \Lambda_{V_B}, \lambda_{E_B} : E_B \to \Lambda_{E_B})$ be two directed labelled multigraphs. The *coproduct* of them is a labelled directed multigraph $\Delta = (V_\Delta, E_\Delta, \sigma_\Delta : E_\Delta \to V_\Delta, \tau_\Delta : E_\Delta \to V_\Delta, \lambda_{V_\Delta} : V_\Delta \to \Lambda_{V_\Delta}, \lambda_{E_\Delta} : E_\Delta \to \Lambda_{E_\Delta})$ where

$$V_\Pi = V_A \sqcup V_B,$$

$$E_\Pi = E_A \sqcup E_B,$$

$$\Lambda_{V_\Pi} = \Lambda_{V_A} \sqcup \Lambda_{V_B},$$

$$\Lambda_{E_\Pi} = \Lambda_{E_A} \sqcup \Lambda_{E_B},$$

$$\sigma_\Pi : E_\Pi \to V_\Pi,\ e = (e', k) \mapsto \begin{cases} \sigma_A(e') & \text{if } k = 0, \\ \sigma_B(e') & \text{if } k = 1, \end{cases}$$

$$\tau_\Pi : E_\Pi \to V_\Pi,\ e = (e', k) \mapsto \begin{cases} \tau_A(e') & \text{if } k = 0, \\ \tau_B(e') & \text{if } k = 1, \end{cases}$$

$$\lambda_{E_\Pi} : E_\Pi \to \Lambda_{E_\Pi},\ e = (e', k) \mapsto \begin{cases} \lambda_{E_A}(e') & \text{if } k = 0, \\ \lambda_{E_A}(e') & \text{if } k = 1, \end{cases}$$

$$\lambda_{V_\Pi} : V_\Pi \to \Lambda_{V_\Pi},\ v = (v', k) \mapsto \begin{cases} \lambda_{V_A}(e') & \text{if } k = 0, \\ \lambda_{V_A}(e') & \text{if } k = 1. \end{cases}$$

together with two directed labelled multigraph morphisms $\delta_K : K \to \Delta$ where $K \in \{A, B\}$ by

$$\delta_K^{Vertex} : V_K \to V_\Delta \text{ with } v \mapsto \begin{cases} (v, 0) & \text{if } K = A, \\ (v, 1) & \text{if } K = B. \end{cases}$$

$$\delta_K^{Edge} : E_K \to E_\Delta \text{ with } e \mapsto \begin{cases} (e, 0) & \text{if } K = A, \\ (e, 1) & \text{if } K = B. \end{cases}$$

$$\delta_K^{Vertex\ Label} : \Lambda_{V_K} \to \Lambda_{V_\Delta} \text{ with } l_v \mapsto \begin{cases} (l_v, 0) & \text{if } K = A, \\ (l_v, 1) & \text{if } K = B. \end{cases}$$

$$\delta_K^{Edge\ Label} : \Lambda_{E_K} \to \Lambda_{E_\Delta} \text{ with } l_e \mapsto \begin{cases} (l_e, 0) & \text{if } K = A, \\ (l_e, 1) & \text{if } K = B. \end{cases}$$

The *coproduct* of two directed multigraphs in the *DLMG* category is a **disjoint union** of them together with two morphism as defined above.

Pullback

Let $A = (V_A, E_A, \sigma_A : E_A \to V_A, \tau_A : E_A \to V_A, \lambda_{V_A} : V_A \to \Lambda_{V_A}, \lambda_{E_A} : E_A \to \Lambda_{E_A})$, $B = (V_B, E_B, \sigma_B : E_B \to V_B, \tau_B : E_B \to V_B, \lambda_{V_B} : V_B \to \Lambda_{V_B}, \lambda_{E_B} : E_B \to \Lambda_{E_B})$, $C = (V_C, E_C, \sigma_C : E_C \to V_C, \tau_C : E_C \to V_C, \lambda_{V_C} : V_C \to \Lambda_{V_C}, \lambda_{E_C} : E_C \to \Lambda_{E_C})$ be directed labelled multigraphs, $\alpha : A \to C$ and $\beta : B \to C$ be two graph morphisms. The *pullback* of A and B is the directed labelled multigraph $H = (V_H, E_H, \sigma_H : E_H \to V_H, \tau_H : E_H \to V_H, \lambda_{V_H} : V_H \to \Lambda_{V_H}, \lambda_{E_H} : E_H \to \Lambda_{E_H})$ where

$$V_H = V_A \times_C V_B = \{(v_a, v_b) : f(v_a) = g(v_b),\ v_a \in V_A \ \&\ v_b \in V_B\},$$

$$E_H = E_A \times_C E_B = \{(e_a, e_b) : f(\sigma_A(e_a)) = g(\sigma_B(v_b)) \ \&\ f(\tau_A(e_a)) = g(\tau_B(v_b)) \ \&\ f(\lambda_{E_A}(e_a)) = g(\lambda_{E_B}(v_b)),\ v_a \in V_A \ \&\ v_b \in V_B\}\ ,$$

$$\Lambda_{V_H} = \Lambda_{V_A} \times_C \Lambda_{V_B} = \{(l_{v_a}, l_{v_b}) = (\lambda_{V_A}(v_a), \lambda_{V_B}(v_b)) \text{ where } (v_a, v_b) \in V_H\},$$

$$\Lambda_{E_H} = \Lambda_{E_A} \times_C \Lambda_{E_B} = \{(l_{e_a}, l_{e_b}) = (\lambda_{E_A}(e_a), \lambda_{E_B}(e_b)) \text{ where } (e_a, e_b) \in E_H\},$$

$$\sigma_H : E_H \to V_H,\ e = (e_a, e_b) \mapsto (\sigma_A(e_a), \sigma_B(e_b)),$$

$$\tau_H : E_H \to V_H,\ e = (e_a, e_b) \mapsto (\tau_A(e_a), \tau_B(e_b)),$$

$$\lambda_{E_H} : E_H \to \Lambda_{E_H},\ e = (e_a, e_b) \mapsto (\lambda_{E_A}(e_a), \lambda_{E_B}(e_b)),$$

$$\lambda_{V_H} : V_H \to \Lambda_{V_H},\ v = (v_a, v_b) \mapsto (\lambda_{V_A}(v_a), \lambda_{V_B}(v_b)).$$

together with morphisms that are defined the same way as for *product* of the graphs.

The *pullback* of two graphs is a subgraph of the *product* of these two graphs.

Pushout

Let $A = (V_A, E_A, \sigma_A : E_A \to V_A, \tau_A : E_A \to V_A, \lambda_{V_A} : V_A \to \Lambda_{V_A}, \lambda_{E_A} : E_A \to \Lambda_{E_A})$, $B = (V_B, E_B, \sigma_B : E_B \to V_B, \tau_B : E_B \to V_B, \lambda_{V_B} : V_B \to \Lambda_{V_B}, \lambda_{E_B} : E_B \to \Lambda_{E_B})$, $C = (V_C, E_C, \sigma_C : E_C \to V_C, \tau_C : E_C \to V_C, \lambda_{V_C} : V_C \to \Lambda_{V_C}, \lambda_{E_C} : E_C \to \Lambda_{E_C})$ be directed labelled multigraphs, $\alpha : C \to A$ and $\beta : C \to B$ be two graph morphisms. The *pushout* of A and B is the directed labelled multigraph $H = (V_H, E_H, \sigma_H : E_H \to V_H, \tau_H : E_H \to V_H, \lambda_{V_H} : V_H \to \Lambda_{V_H}, \lambda_{E_H} : E_H \to \Lambda_{E_H})$ where

$V_H = V_A \sqcup V_B / \equiv$, where $\equiv$ is the equivalence relation generated by the relation R on $V_A \sqcup V_B$, $v_A R v_B \Leftrightarrow \exists v_C$ such that $\alpha^{vertex}(v_C) = v_A$ & $\beta^{vertex}(v_C) = v_B$,

$E_H = \{([v], [w]) \mid \exists (v', w') \in V_A \sqcup V_B, [v] = [v'] \text{ and } [w] = [w']\}$,

Λ_{V_H} is predefined by V_H,

$\Lambda_{E_H} = \Lambda_{E_A} \sqcup \Lambda_{E_B} / \equiv$, where $\equiv$ is the equivalence relation generated by the relation R on $\Lambda_{E_A} \sqcup \Lambda_{E_B}$, $\gamma R \delta \Leftrightarrow \exists \phi$ such that $\alpha^{edge\ label}(\phi) = \gamma$ & $\beta^{edge\ label}(\phi) = \delta$,

$\sigma_H : E_H \to V_H, e = ([v], [w]) \mapsto [v]$,

$\tau_H : E_H \to V_H, e = ([v], [w]) \mapsto [w]$,

$\lambda_{E_H} : E_H \to \Lambda_{E_H}$, $e = ([v], [w]) \mapsto \lambda_k(v', w')$, where $\exists (v', w') \in E_k$ such that $[v] = [v']$ and $[w] = [w']$ and $k \in \{A, B\}$,

$\lambda_{V_H} : V_H \to \Lambda_{V_H}$, $[v] \mapsto \lambda_{V_k}(v')$, where $\exists v' \in V_k$ such that $[v] = [v']$ and $k \in \{A, B\}$.

together with two morhpisms $i_K : K \to H$, where $K \in \{A, B\}$ by

$i_K^{vertex} : V_K \to V_H, v \mapsto [v]$,

$i_K^{edge} : E_K \to E_H, e = (v_1, v_2) \mapsto (i_K^{vertex}(v_1), i_K^{vertex}(v_2))$,

$i_K^{vertex\ label} : \Lambda_{V_K} \to \Lambda_{V_H}, \delta \mapsto [\delta]$,

$i_K^{edge\ label} : \Lambda_{E_K} \to \Lambda_{E_H}, \gamma \mapsto [\gamma]$.

When we consider a *pushout* in the *DLMG* category, a concept of gluing some specific parts of the graphs arises. The parts to be glued are predefined by morphisms f, g from graph C to graphs A, B respectively. The vertices and edges from graphs A and B that originally "came from" C belong to the same equivalence class in the *pushout*.

Implementation

For the computational implementation of universal constructions in the category of directed labeled multigraphs, we chose the .Net platform and, specifically, the C# programming language. C# is an object-oriented programming language, which gives us the flexibility to build a convenient project architecture and potentially expand it in the future. Note that this method of creating a program is declarative, which is a "slow" way for the user to interact with the designed software, but this is not our priority at this stage of development.

As mentioned earlier, our goal was to computationally implement universal constructs in DLMG categories. We divided our classes into abstractions and concrete examples.

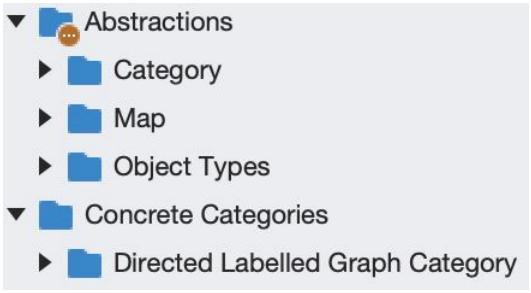


Figure 1: Directories that define the project architecture.

Abstractions

Let's define three levels of details: objects, maps between objects, and categories of objects.

Objects

The objects of our study are graphs, but graphs themselves are also an abstraction, but a more concrete one. There are a considerable number of different types of graphs - simple, complete, directed, labeled, multigraphs, etc. The characteristics of morphisms between them changes from one type to another, and accordingly, the different categories will be formed.

Graphs are made up of edges that connect graph vertices. So we'll start with vertices. A vertex is an object that has a label.

```
public class Vertex
{
    private readonly string label_;
    /*...*/
}
```

Two vertices are considered equal if they have the same label (we can think of labels as unique identifiers of vertices in a graph). There cannot be more than one vertex with a particular label in the graph.

Edges connect the vertices of a graph. An edge may or may not have a direction. The type of graph depends on this. Depending on whether an edge is directed or not, the definition of edge equality also changes. Therefore, we created an abstract class Edge, and a inherited one DEde, a class of directed edges, which contains an implementation for checking for equality in the context of directed edges.

If we wanted to create undirected graphs, we would need to create a new class called UEdge to represent undirected edges.

```
public abstract class Edge
{
    private readonly Vertex tail_;
    private readonly Vertex head_;
    private string label_;
    public bool IsALoop => Tail == Head;
    /*...*/
}
```

```
public class DEdge : Edge
{
    public override bool Equals(object obj)
    {
        return Label == ((Edge)obj).Label &&
        Tail == ((Edge)obj).Tail &&
        Head == ((Edge)obj).Head;
    }
    /*...*/
}
```

A graph contains vertices and edges. The type of a graph is partially determined by the type of edges. In other words, concrete graph type will have a concrete edge type. However, at an abstract level, we say that a graph is a collection of edges and vertices.

```
public abstract class Graph<T> where T : Edge
{
    protected List<T> edges_;
    protected List<Vertex> vertices_;
    /*...*/
}
```

Maps

Maps define relations between objects. They indicate how one object can be mapped to another. Examples include functions from one set to another, homomorphisms of graphs, groups, semigroups, etc. In general, all maps, regardless of the details, have a source and a target.

```
public abstract class Map<T>
{
    protected T source_;
    protected T target_;
    /*...*/
}
```

Categories

The category itself is a higher-level abstraction that contains objects of a certain type and the maps between them. Therefore, the category is already of a certain type when we define the relationships (class inherited from class Map). Having implemented classes of objects and morphisms, we can calculate universal constructions in the category. The skeleton of any category is represented below.

```
public interface ICategory<T>
{
    public T GetInitialObject();
    public T GetTerminalObject();
    public Coproduct<T> GetCoproduct(T t1, T t2);
    public Product<T> GetProduct(T t1, T t2);
    public Pushout<T> GetPushout(Map<T> h1, Map<T> h2);
```

```
    public Pullback<T> GetPullback(Map<T> h1, Map<T> h2);
}
```

The product, coproduct, pushout, and pullback are objects of universal constructions, and consist of an object of type T and two morphisms. For example

```
public class Product<T>
{
    private readonly T product_;
    private readonly Map<T> hom1_;
    private readonly Map<T> hom2_;
    /*...*/
}
```

The ICategory<T> interface could potentially have more methods to compute universal constructions of more complex diagrams or add initial and terminal objects.

In summary, all the above classes are intended to represent the structure needed to define a certain category, to be a skeleton. They are all abstract, so we can't use them to create class instances. They are definitions, not examples. For concrete examples, we'll create child classes and add all the necessary details to them.

Concrete Category

As we mentioned earlier, we created a specific DEdge class for creating directed edges. The concrete object we are interested in is a directed labelled multigraph containing collection of DEdge objects and Vertex objects.

```
public class DLMGraph : Graph<DEdge> //directed labeled multigraph
{
    public DLMGraph(List<DEdge> edges, List<Vertex> vertices) :
       base(edges, vertices) { }
    /*...*/
}
```

An example of defining a directed labelled multigraph and its graphical representation can be seen below.

```
Vertex v1 = new Vertex("v1");
Vertex v2 = new Vertex("v2");
Vertex v3 = new Vertex("v3");
Vertex v4 = new Vertex("v4");
List<DEdge> edges1 = new List<DEdge>() {
    new DEdge(v1, v3, "a"),
    new DEdge(v1, v4, "b"),
    new DEdge(v2, v4, "c")};
DLMGraph g1 = new DLGraph(edges1);
```

We can ask for the graph structure in dot format using method ToString() as shown below. We can then visualize it with any available tool.

```
digraph G {
0 [label ="v1"];
```

```
1 [label ="v2"];
2 [label ="v3"];
3 [label ="v4"];
0 -> 2 [label="a"] ;
0 -> 3 [label="b"] ;
1 -> 3 [label="c"] ;
}
```

The DLMGHomomorphism class is a child class of the Map<DLMGraph> class, showing that this class implements a specific type of mapping between directed labeled multigraphs. We can divide their homomorphism into three parts: vertices, edges, and edges' labels. When initiating a class object, all three parts are checked, and the object will be created only if the homomorphism is correct. The pseudocode for a function that checks whether input data represents a homomorphism, can be found in the **Algorithm 1**. We have not implemented any algorithm for finding a homomorphism from one graph to another, so this is to be done manually.

Algorithm 1: Algorithm for checking a graph homomorphism.

```
1  function CheckHomomophism (source, target, v_map, l_map);
   Input  : source and target are directed labelled multigraphs, v_map and l_map are vertex
            and label maps respectively, represented as lists of tuples.
   Output: boolean value.
2  foreach vertex in source do
3      if v_map doesn't contain any tuple (vertex,w), where w is in the target then
4          return false;
5      else if v_map contains more than one tuple (vertex,w), where w is in the target then
6          return false;
7      else if v_map contains a tuple (vertex,w), where w is NOT the in target then
8          return false;
9  end
10 foreach edge label in source do
11     if l_map doesn't contain any tuple (label,w), where w is in the target then
12         return false;
13     else if l_map contains more than one tuple (label,w), where w is in the target then
14         return false;
15     else if l_map contains a tuple (label,w), where w is NOT the in target then
16         return false;
17 end
18 foreach edge in source do
19     newTail ← v_map(edge.tail) ;
20     newHead ← v_map(edge.head) ;
21     newLabel ← l_map(edge.label) ;
22     newEdge ← new  edge with newTail, newHead, newLabel;
23     if target does not contain newEdge then
24         return false;
25     end
26 end
27 return true;
```

The class that represent a directed labelled graph homomorphism can be shown below.

```
 public class DLMGHomomorphism : Map<DLMGraph>
{
    private readonly Dictionary<Vertex, Vertex> vertex_map_;
    private readonly Dictionary<string, string> edge_labeles_map_;
    private Dictionary<DEdge, DEdge> edge_map_;
    /*...*/
}
```

An example of defining a homomorphism between two graphs is represented in the listing below.

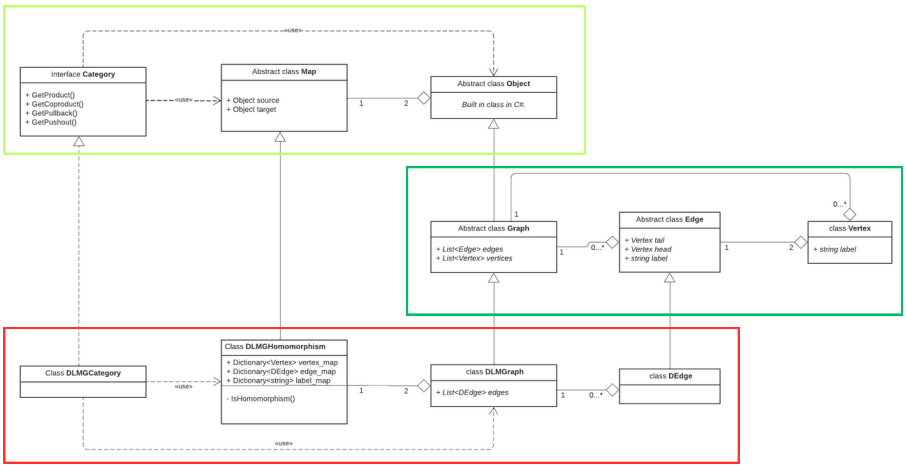
```
Vertex a = new Vertex("a");
Vertex b = new Vertex("b");
Vertex c = new Vertex("c");
List<DEdge> edges1 = new List<DEdge>()
{ new DEdge(a, b, "a"), new DEdge(b, a, "b")};
DLMGraph g_1 = new DLMGraph(edges1);
List<DEdge> edgesc = new List<DEdge>()
{ new DEdge(a, b, "a"),  new DEdge(c, b, "b") };
DLMGraph g_c = new DLMGraph(edgesc);
Dictionary<Vertex, Vertex> v_c1 = new Dictionary<Vertex, Vertex>()
{ {a,a },{c,a},{b,b}};
Dictionary<string, string> l_c1 = new Dictionary<string, string>()
{{"a","a" },{"b","a"},{"c","a"} };
DLMGHomomorphism h1 = new DLMGHomomorphism(g_c, g_1, v_c1, l_c1);
```

The DLMGCategory class is the peak of our iceberg, covering everything we have previously implemented. In the class, we already directly implement 4 methods that relate specifically to directed labeled multigraps: GetProduct(), GetCoproduct(), GetPullback(), GetPushout(), whose algorithms will be described in the next section.

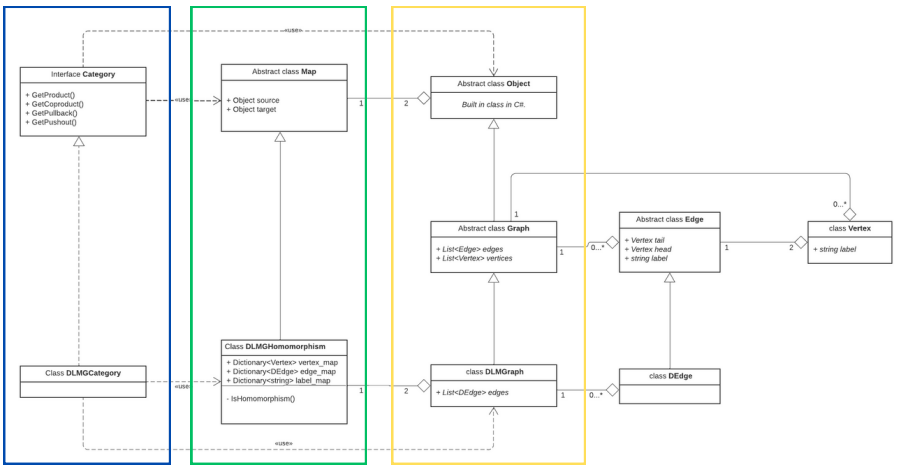
```
public class DLMGCategory : ICategory<DLMGraph>  //directed labeled
    graph category
{
    public Coproduct<DLMGraph> GetCoproduct(DLMGraph t1, DLMGraph
        t2)[...]
    public Product<DLMGraph> GetProduct(DLMGraph t1, DLMGraph t2)[...]
    public Pullback<DLMGraph> GetPullback(Map<DLMGraph> h1,
        Map<DLMGraph> h2)[...]
    public Pushout<DLMGraph> GetPushout(Map<DLMGraph> h1, Map<DLMGraph>
        h2)[...]
    /*...*/
}
```

Below is a UML diagram of classes and the relationships between them. We've highlited the abstract classes - object, map, and category - in light green. Dark green is used for intermediate classes - graph, edge, and vertex - which are still abstract but contain more details. The lowest red level shows a concrete implementation of the category of directed labeled multigraphs. Instead of (or in addition to) these classes, we can create other categories implemetation, such as the Set class,

the Function class, and the SetCategory class, where the objects are sets and the arrows are the functions between them.



We can also divide the diagram into three other logical levels (there were 2 main levels in the previous one), where the highest level (blue) is the categories. And if we look inside the category, we'll come to the level of maps (green), and then zoom out more to the lowest level of objects (yellow). Overall, this project architecture is potentially flexible for expanding and implementing new categories.



DLMG Category - implementation of universal constructions

Coproduct

Coproduct in the category of directed labeled multigraphs is a disjoint union of two graphs. In our implementation, graphs are collections of edges and vertices so that a coproduct graph will contain the edges and vertices of two graphs. The labels will be updated as shown in the definition of a coproduct. Below is the pseudocode for implementing the function that finds the coproduct of the directed label multigraphs and the homomorphisms f and g.

Algorithm 2: Algorithm for finding a coproduct of two directed labelled multigraphs.

```
1  function Coproduct (graph_a, graph_b);
   Input  : Two directed labelled multigraphs graph_a and graph_b
   Output: triple (coproduct, f, g)
2  Let coproduct be a directed labelled multigraph;
3  Let f be a homomorphism from graph_a to graph coproduct;
4  Let g be a homomorphism from graph_b to graph coproduct;
5  foreach vertex in graph_a do
6  |  newLabel ← vertex label +" − 0";
7  |  newVertex ← vertex with newLabel;
8  |  add tuple (vertex, newVertex) to f;
9  |  add newVertex to coproduct.
10 end
11 foreach vertex in graph_b do
12 |  newLabel ← vertex label +" − 1";
13 |  newVertex ← vertex with newLabel;
14 |  add tuple (vertex, newVertex) to g;
15 |  add newVertex to coproduct.
16 end
17 foreach edge in graph_a do
18 |  newLabel ← edge label +" − 0";
19 |  newTail ← f(edge tail);
20 |  newHead ← f(edge head);
21 |  newEdge ← edge with newLabel, newTail, newHead;
22 |  add tuple (edge label, newLabel) to f;
23 |  add newEdge to coproduct.
24 end
25 foreach edge in graph_b do
26 |  newLabel ← edge label +" − 1";
27 |  newTail ← f(edge tail);
28 |  newHead ← f(edge head);
29 |  newEdge ← edge with newLabel, newTail, newHead;
30 |  add tuple (edge label, newLabel) to g;
31 |  add newEdge to coproduct.
32 end
```

Example

Below is a code example of how to compute the coproduct of two graphs. In our case, the graphs are identical. We also give their graphical representation and details of the homomorphisms f and g.

```
DLMGCategory category = new DLMGCategory();
Vertex a = new Vertex("a");
Vertex b = new Vertex("b");
List<DEdge> edges1 = new List<DEdge>() {
new DEdge(a, b, "a"),
new DEdge(b, a, "b"),
};
DLMGraph g1 = new DLMGraph(edges1);
System.Console.WriteLine(g1);
var coproduct = category.GetCoproduct(g1, g1); ;
System.Console.WriteLine(coproduct);
```

Figure 2: (1) graph g, (2) the coproduct of g with itself.

The description of the homomorphism f from the firt graph to the coproduct:

```
on vertices:
a => a - 0
b => b - 0
on labeles:
a => a - 0
b => b - 0
on edges:
a: a -> b => a - 0: a - 0 -> b - 0
b: b -> a => b - 0: b - 0 -> a - 0
```

The description of the homomorphism g from the second graph to the coproduct:

```
on vertices:
a => a - 1
b => b - 1
```

```
on labeles:
a => a - 1
b => b - 1
on edges:
a: a -> b => a - 1: a - 1 -> b - 1
b: b -> a => b - 1: b - 1 -> a - 1
```

Product

Product in the $DLMG$ category is a Tensor product of two directed labelled multigraphs. An edge $e = ((u, v), (u', v'))$ will exist in the product graph only if the exist an edge $e_1 = (u, u')$ in the first graph, and there exist an edge $e_2 = (v, v')$ in the second graph. This type of graphs product is different from the Cartesian product, which is not a categoretical construction. Below there is a pseudocode of the function implementation that finds a product of two directed labelled multigraphs together with two morphisms to the graphs.

Algorithm 3: Algorithm for finding a product of two directed labelled multigraphs.

```
1  function Product (graph_a, graph_b);
   Input  : Two directed labelled multigraphs graph_a and graph_b
   Output: triple (coproduct, f, g)
2  Let product be a directed labelled multigraph;
3  Let f be a homomorphism from product to graph_a;
4  Let g be a homomorphism from product to graph_b;
5  foreach vertex a in graph_a do
6  |  foreach vertex b in graph_b do
7  |  |  newVertex ← (vertex a,vertex b);
8  |  |  add tuple (newVertex, a) to f;
9  |  |  add tuple (newVertex, b) to g;
10 |  |  add newVertex to product.
11 |  end
12 end
13 foreach edge a in graph_a do
14 |  foreach edge b in graph_b do
15 |  |  newLabel ← (edge a label , edge b label);
16 |  |  newTail ← (edge a tail , edge b tail);
17 |  |  newHead ← (edge a head , edge b head);
18 |  |  newEdge ← edge with newLabel, newTail, newHead;
19 |  |  add tuple (newEdge, a) to f;
20 |  |  add tuple (newEdge, b) to g;
21 |  |  add newEdge to product.
22 |  end
23 end
```

Example

Below is a code example of how to compute the product of two graphs. In our case, the graphs are identical. We also give a graphical representation of graph g and a coproduct together with details of the homomorphisms f and g.

```
DLMGCategory category = new DLMGCategory();
Vertex a = new Vertex("a");
Vertex b = new Vertex("b");
List<DEdge> edges1 = new List<DEdge>() {
new DEdge(a, b, "a"),
new DEdge(b, a, "b"),
};
DLMGraph g1 = new DLMGraph(edges1);
System.Console.WriteLine(g1);
var product = category.GetProduct(g1, g1); ;
System.Console.WriteLine(product);
```

Figure 3: (1) graph g, (2) a product of g with itself.

The **graph** *product* itself is not a universal construction in the category. But it is with two homomorphisms. The description of the homomorphism f from the product to the first graph:

```
on vertices:
a - a => a
a - b => a
b - a => b
b - b => b
on labeles:
a - a => a
a - b => a
b - a => b
b - b => b
on edges:
a - a: a - a -> b - b => a: a -> b
a - b: a - b -> b - a => a: a -> b
b - a: b - a -> a - b => b: b -> a
b - b: b - b -> a - a => b: b -> a
```

The description of the homomorphism g from the product to the second graph:

```
on vertices:
a - a => a
a - b => b
b - a => a
b - b => b
on labeles:
a - a => a
a - b => b
b - a => a
b - b => b
on edges:
a - a: a - a -> b - b => a: a -> b
a - b: a - b -> b - a => b: b -> a
b - a: b - a -> a - b => a: a -> b
b - b: b - b -> a - a => b: b -> a
```

The result turned out to be isomorphic to the coproduct of the same graph with itself.

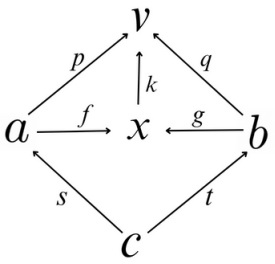
Pushout

Below is the pseudocode of function that finds a pushout of two directed labelled multigraphs. To

Algorithm 4: Algorithm for finding a pushout of two directed labelled multigraphs.

```
1  function Pushout (s, t);
   Input  : Two homomorphisms s and t of directed labelled multigraphs
   Output: triple (pushout, f, g)
2  if if source of s and t are different then
3  |  return null;
4  end
5  graph_a ← target of s;
6  graph_b ← target of t;
7  Let pushout be a directed labelled multigraph;
8  Let f be a homomorphism from graph_a to pushout;
9  Let g be a homomorphism from graph_b to pushout;
10 (*) Compute (coproduct, p, q) of graphs graph_a and graph_b;
11 k ← p · s;
12 e ← q · t;
13 (1) Generate vertex equivalence classes v_classes from k and e;
14 (2) Generate label equivalence classes l_classes from k and e;
15 (3) (pushout, r) ← glued coproduct with v_classes and l_classes, where r is a
     homomorphism from coproduct to pushout;
16 f ← r · p;
17 g ← r · q;
```

compute pushout we will use the definition of a coproduct. Coproduct x is a universal cocone over diagram containing graphs a and b, such that there exist unique homomorphism k from coproduct to any other graph , particularly to a pushout v, such that the diagram commute (see figure below).



We say that pushout is a glued coproduct, where some vertices of the coproduct will be mapped to the same place, as homomorphisms s and t from graph c prescribe. Hence, we will compute a pushout in a few steps. First, we will generate equivalence classes using homomorphisms $g \cdot t$ and $f \cdot s$. We can do that because f and g map graphs a and b to some isomorphic subgraphs in the coproduct that are disjoint with each other. A pseudocode describing function that generates equivalence classes can be found in the **Algorithm 6**. When equivalence classes are generated we will change the structure of the coproduct in order to make a pushout. All vertices and edge labeles will be replaced with class representatives they belong to. Every edge will be automatically replaced. Homomorphism k can be easily defined from equivalence classes. Every vertex and edge label is mapped to the representative of the class it belongs to. The detailed pseudocode is represented in the **Algorithm 5**. Homomorphisms $p = k \cdot f$ and $q = k \cdot g$ are the homomorphisms we were to find from graphs a and b to the pushout. The full proposed instructions for finding a pushout can be found in the **Algorithm 4**.

Algorithm 5: Algorithm for glueing a coproduct with equivalence classes in order to create pushout and a homomorphism from coproduct to pushout.

```
1  function GlueCoproduct (coproduct, v_classes, l_classes);
   Input  : A directed labelled multigraph coproduct, the vertex equivalence classes v_classes,
            and the label equivalence classes l_classes.
   Output: tuple (pushout, r)
2  foreach vertex in coproduct do
3  |  Replace vertex with representative of equivalence class it belongs to;
4  |  Add tuple (vertex, representative) to homomorphism r;
5  end
6  foreach edge in coproduct do
7  |  Replace its tail with representative of equivalence class it belongs to;
8  |  Replace its head with representative of equivalence class it belongs to;
9  |  Replace its label with representative of equivalence class it belongs to;
10 |  Add tuple (label, representative) to homomorphism r;
11 |  modified coproduct is a pushout, return it with homomorphism r.
12 end
```

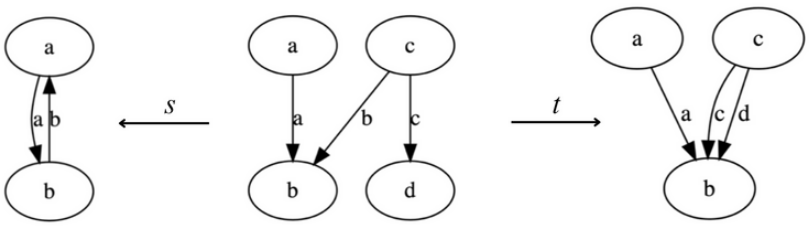
The algorithm as below is used for finding vertex equivalence classes. With it, we can also find label equivalence classes, operating with edge labels instead of vertices.

Algorithm 6: Algorithm for finding a vertex equivalence classes *v_classes*.

```
1  function GenerateVclasses (k, e);
   Input  : Two homomorphisms k and e of directed labelled multigraphs, both from c to
            coproduct.
   Output: v_classes
2  if if source or target of k and e are different then
3  |  return null;
4  end
5  graph_c ← source of k(or e);
6  coproduct ← target of k(or e);
7  foreach vertex in graph_c do
8  |  a ← k(vertex);
9  |  b ← e(vertex);
10 |  if a already exist in some equivalence class [a'] then
11 |  |  add b to [a'];
12 |  |  go to the next vertex;
13 |  end
14 |  if b already exist in some equivalence class [b'] then
15 |  |  add a to [b'];
16 |  |  go to the next vertex;
17 |  end
18 end
19 create a new equivalence class [x];
20 add a and b to the equivalence class [x] and make the vertex as representative;
21 foreach vertex in coproduct do
22 |  if vertex already exist in some equivalence class then
23 |  |  go to the next vertex;
24 |  else
25 |  |  create a new equivalence class [vertex];
26 |  |  add vertex to the equivalence class [vertex] and make it a representative;
27 |  end
28 end
```

Example

Given the diagram:



where homomorphism s is defined as follows:

```
on vertices:
a => a
c => a
b => b
d => b
on labeles:
a => a
b => a
c => a
on edges:
a: a -> b => a: a -> b
b: c -> b => a: a -> b
c: c -> d => a: a -> b
```

and the description of a homomorphism t is defined as follows:

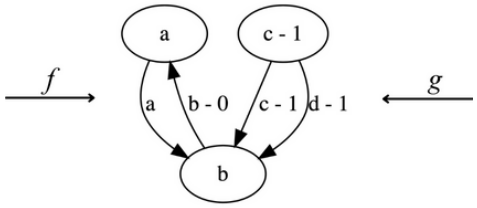
```
on vertices:
a => a
c => a
b => b
d => b
on labeles:
a => a
b => a
c => a
on edges:
a: a -> b => a: a -> b
b: c -> b => a: a -> b
c: c -> d => a: a -> b
```

We are set to demonstrate the code example of finding a pushout over the diagram (shown above) that is shown on the listing below.

```
DLMGCategory category = new DLMGCategory();
Vertex a = new Vertex("a");
Vertex b = new Vertex("b");
Vertex c = new Vertex("c");
Vertex d = new Vertex("d");
List<DEdge> edges1 = new List<DEdge>() {
new DEdge(a, b, "a"),
new DEdge(b, a, "b"),};
DLMGraph g_1 = new DLMGraph(edges1);
List<DEdge> edgesc = new List<DEdge>() {
new DEdge(a, b, "a"),
new DEdge(c, b, "b"),
new DEdge(c, d, "c")};
DLMGraph g_c = new DLMGraph(edgesc);
List<DEdge> edges2 = new List<DEdge>() {
new DEdge(a, b, "a"),
```

```
new DEdge(c, b, "c"),
new DEdge(c, b, "d")};
DLMGraph g_2 = new DLMGraph(edges2);
Dictionary<Vertex, Vertex> v_c1 = new Dictionary<Vertex, Vertex>()
{ {a,a },{c,a},{b,b},{d,b }};
Dictionary<string, string> l_c1 = new Dictionary<string, string>()
{{"a","a" },{"b","a"},{"c","a"} };
Dictionary<Vertex, Vertex> v_c2 = new Dictionary<Vertex, Vertex>()
{{a,a },{c,a},{b,b},{d,b }};
Dictionary<string, string> l_c2 = new Dictionary<string, string>()
{{"a","a" },{"b","a"},{"c","a"}};
DLMGHomomorphism h1 = new DLMGHomomorphism(g_c,g_1,v_c1, l_c1);
DLMGHomomorphism h2 = new DLMGHomomorphism(g_c, g_2, v_c2, l_c2);
var pushout = category.GetPushout(h1,h2); ;
System.Console.WriteLine(pushout);
```

The computed pushout over the diagram is:



Together with homomorphisms f and g. Description of a homomorphism f:

```
on vertices:
a => a
b => b
on labeles:
a => a
b => b - 0
on edges:
a: a -> b => a: a -> b
b: b -> a => b - 0: b -> a
```

Description of a homomorphism g:

```
on vertices:
a => a
c => c - 1
b => b
on labeles:
a => a
c => c - 1
d => d - 1
on edges:
a: a -> b => a: a -> b
```

```
c: c -> b => c - 1: c - 1 -> b
d: c -> b => d - 1: c - 1 -> b
```

Pullback

Pullback can be computed easier than pushout. Pullback over the diagram $a \xrightarrow{f} c \xleftarrow{g} b$ is a subgraph of a product of graphs a and b. The only vertices (v, w) and edge labeles (l, i) will be a part of the pullback if and only if $f(v) = g(w)$ and $f(l) = g(i)$. Everything else is going to be the same as for product. The pseudocode for function that finds a pullback over the defined diagram can be found in the **Algorithm 7**.

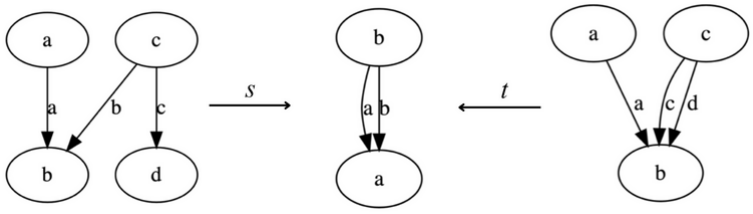
Algorithm 7: Algorithm for finding a pullback of two directed labelled multigraphs.

```
1  function Pullback (s, t);
   Input  : Two homomorphisms s and t of directed labelled multigraphs
   Output: triple (pullback, f, g)
2  if if target of s and t are different then
3  |  return null;
4  end
5  graph_a ← source of s;
6  graph_b ← source of t;
7  Let pullback be a directed labelled multigraph;
8  Let f be a homomorphism from pullback to graph_a;
9  Let g be a homomorphism from pullback to graph_b;
10 foreach vertex a in graph_a do
11 |  foreach vertex b in graph_b do
12 |  |  if s(a) and t(b) are the same then
13 |  |  |  newVertex ← (vertex a,vertex b);
14 |  |  |  add tuple (newVertex, a) to f;
15 |  |  |  add tuple (newVertex, b) to g;
16 |  |  |  add newVertex to product.
17 |  |  end
18 |  end
19 end
20 foreach edge a in graph_a do
21 |  foreach edge b in graph_b do
22 |  |  if s(a) and t(b) are the same then
23 |  |  |  newLabel ← (edge a label , edge b label);
24 |  |  |  newTail ← (edge a tail , edge b tail);
25 |  |  |  newHead ← (edge a head , edge b head);
26 |  |  |  newEdge ← edge with newLabel, newTail, newHead;
27 |  |  |  add tuple (newEdge, a) to f;
28 |  |  |  add tuple (newEdge, b) to g;
29 |  |  |  add newEdge to product.
30 |  |  end
31 |  end
32 end
```

The only difference with Product pseudocode are lines 12 and 22, where we added the if conditions whether vertices and labeles are mapped to the same place in the graph c.

Example

Given the diagram:



Description of a homomorphism s:

```
on vertices:
a => b
c => b
b => a
d => a
on labeles:
a => a
b => b
c => a
on edges:
a: a -> b => a: b -> a
b: c -> b => b: b -> a
c: c -> d => a: b -> a
```

Description of a homomorphism t:

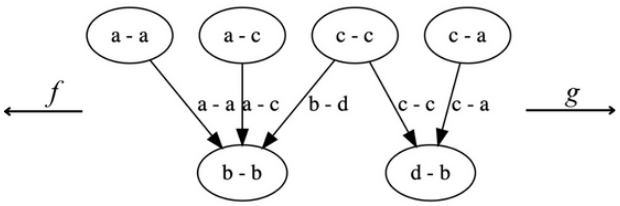
```
on vertices:
a => b
c => b
b => a
on labeles:
a => a
d => b
c => a
on edges:
a: a -> b => a: b -> a
c: c -> b => a: b -> a
d: c -> b => b: b -> a
```

We are set to demonstrate the code example of finding a pullback over the diagram (shown above) that is shown on the listing below.

```
DLMGCategory category = new DLMGCategory();
```

```
Vertex a = new Vertex("a");
Vertex b = new Vertex("b");
Vertex c = new Vertex("c");
Vertex d = new Vertex("d");
List<DEdge> edges1 = new List<DEdge>() {
new DEdge(b, a, "a"),
new DEdge(b, a, "b"),
};
DLMGraph g_1 = new DLMGraph(edges1);
List<DEdge> edgesc = new List<DEdge>() {
new DEdge(a, b, "a"),
new DEdge(c, b, "b"),
new DEdge(c, d, "c")
};
DLMGraph g_c = new DLMGraph(edgesc);
List<DEdge> edges2 = new List<DEdge>() {
new DEdge(a, b, "a"),
new DEdge(c, b, "c"),
new DEdge(c, b, "d")
};
DLMGraph g_2 = new DLMGraph(edges2);
Dictionary<Vertex, Vertex> v_c1 = new Dictionary<Vertex, Vertex>()
{ {a,b },{c,b},{b,a},{d,a }};
Dictionary<string, string> l_c1 = new Dictionary<string, string>()
{{"a","a" },{"b","b"},{"c","a"} };
Dictionary<Vertex, Vertex> v_21 = new Dictionary<Vertex, Vertex>()
{{a,b },{c,b},{b,a}};
Dictionary<string, string> l_21 = new Dictionary<string, string>()
{{"a","a" },{"d","b"},{"c","a"} };
DLMGHomomorphism h1 = new DLMGHomomorphism(g_c, g_1, v_c1, l_c1);
DLMGHomomorphism h2 = new DLMGHomomorphism(g_2, g_1, v_21, l_21);
var pullback = category.GetPullback(h1,h2); ;
System.Console.WriteLine(pullback);
```

The computed pullback over the diagram is:



Together with homomorphisms f and g. Description of a homomorphism f:

```
a - a => a
a - c => a
c - a => c
c - c => c
b - b => b
d - b => d
on labeles:
a - a => a
a - c => a
b - d => b
c - a => c
c - c => c
on edges:
a - a: a - a -> b - b => a: a -> b
a - c: a - c -> b - b => a: a -> b
b - d: c - c -> b - b => b: c -> b
c - a: c - a -> d - b => c: c -> d
c - c: c - c -> d - b => c: c -> d
```

Description of a homomorphism g:

```
on vertices:
a - a => a
a - c => c
c - a => a
c - c => c
b - b => b
d - b => b
on labeles:
a - a => a
a - c => c
b - d => d
c - a => a
c - c => c
on edges:
a - a: a - a -> b - b => a: a -> b
a - c: a - c -> b - b => c: c -> b
b - d: c - c -> b - b => d: c -> b
c - a: c - a -> d - b => a: a -> b
c - c: c - c -> d - b => c: c -> b
```

Further Development

In the future, we plan to use implemented software to analyze and create user interfaces using affordances. Also, this software is open to future improvements and optimizations.

You can find the source code at github.

References

[1] Eugenia Cheng. *The joy of abstraction.* 9 2022.

[2] Chrystopher Nehaniv, Fariba Karimi, Daniel Schreckling, Nicoline Den Breems, Agnes Bonivart, Maria Schilstra, Alastair Munro, and Attila Egri-Nagy. *BIOMICS Deliverable D2.1 Category Theoretic Framework: Functors and Adjoints for Discrete and Continuous Dynamical Cellular Systems Symmetries.* 03 2014.

[3] John R. Stallings. *Category Language.* 2000.