Computerpraktikum

Block I

Gerichtete Graphen

Matthias Künzer

Universität Stuttgart

21. März 2022

Inhalt

Vorwort

Ein (gerichteter) Graph H besteht aus einer Menge von Punkten V_H (engl. vertices), einer Menge von Kanten E_H (engl. edges) und zwei Abbildungen, Start $\mathrm{s}_H : \mathrm{E}_H \to \mathrm{V}_H$ (engl. source) und Ziel $\mathrm{t}_H : \mathrm{E}_H \to \mathrm{V}_H$ (engl. target).

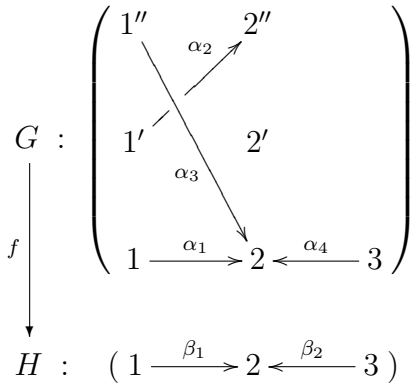
Z.B. hat im Graphen

$$H \ : \ (\ 1 \xrightarrow{\beta_1} 2 \xleftarrow{\beta_2} 3 \ )$$

die Kante β_2 den Startpunkt $\beta_2 \mathrm{s}_H = 3$ und den Zielpunkt $\beta_2 \mathrm{t}_H = 2$.

Ein Morphismus von Graphen $G \xrightarrow{f} H$ besteht aus einer Abbildung $\mathrm{V}_f : \mathrm{V}_G \to \mathrm{V}_H$ und einer Abbildung $\mathrm{E}_f : \mathrm{E}_G \to \mathrm{E}_H$ derart, daß der Startpunkt der Bildkante gleich dem Bild des Startpunktes ist, dito für den Zielpunkt.

Z.B. ist ein Graphmorphismus $G \xrightarrow{f} H$ dadurch gegeben, daß in



alle Kanten und alle Punkte von G senkrecht nach unten abgebildet werden. Auf den Punkten bildet f also wie folgt ab: $1\mathrm{V}_f = 1$, $1'\mathrm{V}_f = 1$, $1''\mathrm{V}_f = 1$, $2\mathrm{V}_f = 2$, $2'\mathrm{V}_f = 2$, $2''\mathrm{V}_f = 2$, $3\mathrm{V}_f = 3$. Auf den Kanten bildet f wie folgt ab: $\alpha_1\mathrm{E}_f = \beta_1$, $\alpha_2\mathrm{E}_f = \beta_1$, $\alpha_3\mathrm{E}_f = \beta_1$, $\alpha_4\mathrm{E}_f = \beta_2$.

Die Graphen und ihre Objekte bilden die Kategorie Gph. Darin können Graphmorphismen komponiert werden.

Gemäß Bisson und Tsemo [**1**] kann man in Gph spezielle Sorten von Morphismen in sinnvoller Weise definieren: Faserungen, Cofaserungen und Quasiisomorphismen.

Wir wollen das Verhalten dieser Morphismen im Falle endlicher Graphen mittels Magma studieren. Insbesondere werden wir versuchen, mit Rechnerunterstützung Gegenbeispiele zu naheliegenden, aber falschen Aussagen zu finden.

Dank geht an Jannik Hess für die Ausarbeitung der meisten Magma-Programme [**2**], [**3**]. Dank geht an Bernhard Speck, Melanie Roos, Danning Liu, Carmen Heidrich, Anna Dannecker, Savvas Aslanidis für Hinweise auf Fehler.

Für weitere Hinweise auf Fehler und Unklarheiten bin ich dankbar.

Stuttgart, Herbst 2021

Matthias Künzer

1 Konventionen

- Sind X und Y Mengen, ist $x \in X$ ein Element und $X \xrightarrow{f} Y$ eine Abbildung, dann schreiben wir $xf \in Y$ für das Bild von x unter f.
- Das Kompositum der Abbildungen $X \xrightarrow{f} Y \xrightarrow{g} Z$ wird $X \xrightarrow{f \cdot g} Z$ geschrieben.
- Ist X eine endliche Menge, so bezeichnet $|X|$ die Anzahl ihrer Elemente.
- Für $a, b \in \mathbb{Z}$ sei $[a, b] = \{\, i \in \mathbb{Z} :\ a \leqslant i \leqslant b \,\}$ das ganzzahlige Intervall.
- Sei $n \geqslant 1$. Wir schreiben Elemente aus

$$\mathbb{Z}/n\mathbb{Z} \ = \ \{\, i + n\mathbb{Z} :\ i \in \mathbb{Z} \,\} \ = \ \{\, i + n\mathbb{Z} :\ i \in [0, n-1] \,\}$$

  auch als $i := i + n\mathbb{Z}$, wenn aus dem Kontext hervorgeht, daß Elemente von $\mathbb{Z}/n\mathbb{Z}$ gemeint sind.

2 Magma

Wir verwenden das Computeralgebra-System Magma [**2**].

Das Handbuch findet sich unter `magma.maths.usyd.edu.au/magma/`.

Auf einen stud-Rechner kommt man unter Verwendung des eigenen stud-Accounts mittels

```
ssh -CX st[Nummer]@stud.uni-stuttgart.de@stud105.mathematik.uni-stuttgart.de
```

Dies funktioniert so mittels Linux und mittels Windows 10.

Verfügbar sind dabei: `stud105` bis `stud116`.

Es kann dort Magma aufgerufen werden durch Eingabe von `magma` in einer Shell auf einem `stud`-Rechner.

Man teste einmal `1+1;` und dann `Enter`.

Als Notbehelf kann auch `magma.maths.usyd.edu.au/calc/` dienen.

Die Konventionen in §1, was die Schreibung von Abbildungen angeht, sind Magma angepaßt.

3 Graphen

Ein *Graph* G besteht aus

- einer Menge von Punkten V_G (engl. vertices),
- einer Menge von Kanten E_G (engl. edges)

und zwei Abbildungen,

- Start $\mathrm{s}_G : \mathrm{E}_G \to \mathrm{V}_G$ (engl. source),
- Ziel $\mathrm{t}_G : \mathrm{E}_G \to \mathrm{V}_G$ (engl. target).

Ein Graph G heißt *endlich*, wenn V_G und E_G endliche Mengen sind.

Hierbei schreiben wir graphisch $x \xrightarrow{\alpha} y$ für eine Kante $\alpha \in \mathrm{E}_H$ mit Start $\alpha \mathrm{s}_G = x$ und Ziel $\alpha \mathrm{t}_G = y$.

Alle Punkte und Kanten zusammengenommen ergeben so das Bild eines Graphen, bei dem jede Kante noch eine Richtung hat, dargestellt durch einen Pfeil.

Also zum Beispiel wie folgt.

$$G \;=\; \Big(1 \underset{\alpha_2}{\overset{\alpha_1}{\rightrightarrows}} 2 \xleftarrow{\alpha_3} 3 \qquad 4 \Big)$$

In Magma stellen wir dies wie folgt dar.

```
G := <[1,2,3,4],[<1,1,2>,<1,2,2>,<3,3,2>]>;
```

In der ersten Liste stehen die Punkte, in der zweiten Liste stehen die Kanten.

Hierbei haben wir uns von den Kanten nur die Nummern gemerkt.

Eine Kante bekommt immer als mittleren Eintrag diese Nummer, als ersten Eintrag ihren Startpunkt und als dritten Eintrag ihren Zielpunkt. So z.B. ist die Kante α_2 durch `<1,2,2>` aufgelistet. Damit sind die Abbildungen s_G und t_G implizit in den Daten enthalten.

Aufgabe. Geben Sie einen Graphen mit wenigstens zwei Kanten, bei denen Start- und Zielpunkt übereinstimmen, in Magma ein. Zeichnen Sie diesen Graphen.

Sei $n \geqslant 1$. Sei C_n der *zyklische* Graph mit

$$\begin{aligned} \mathrm{V}_{\mathrm{C}_n} &:= \{\, \mathrm{v}_i : \; i \in \mathbb{Z}/n\mathbb{Z} \,\}, \\ \mathrm{E}_{\mathrm{C}_n} &:= \{\, \mathrm{e}_i : \; i \in \mathbb{Z}/n\mathbb{Z} \,\} \end{aligned}$$

und mit $i\mathrm{s}_{\mathrm{C}_n} := i$ und $i\mathrm{t}_{\mathrm{C}_n} := i + 1$ für $i \in \mathbb{Z}/n\mathbb{Z}$.

Zum Beispiel wird

$$\mathrm{C}_4 \;=\; \left(\begin{array}{ccc} & \overset{\mathrm{e}_1}{\longrightarrow} & \\ \mathrm{v}_1 & & \mathrm{v}_2 \\ {\scriptstyle \mathrm{e}_4}\uparrow & & \downarrow{\scriptstyle \mathrm{e}_2} \\ \mathrm{v}_4 & & \mathrm{v}_3 \\ & \underset{\mathrm{e}_3}{\longleftarrow} & \end{array} \right) .$$

Dabei ist $\mathrm{v}_4 = \mathrm{v}_0$, $\mathrm{v}_5 = \mathrm{v}_1$, $\mathrm{e}_4 = \mathrm{e}_0$, usf.

Via Magma:

```
CyclicGraph := function(n) // returns cyclic graph with n edges
 return <[i : i in [1..n]],[<i,i,i+1> : i in [1..n-1]] cat [<n,n,1>]>;
end function;
```

Ein Graph G heißt *dünn*, falls für $v, w \in \mathrm{V}_G$ stets $|\{ e \in \mathrm{E}_G : e\mathrm{s}_G = v, e_{\mathtt{G}} = \mathtt{w} \}| \leqslant \mathtt{1}$ ist.

4 Graphmorphismen

4.1 Begriff

Seien G und H Graphen.

Ein *Graphmorphismus* $f : G \to H$ besteht aus zwei Abbildungen

- $\mathrm{V}_f : \mathrm{V}_G \to \mathrm{V}_H$,
- $\mathrm{E}_f : \mathrm{E}_G \to \mathrm{E}_H$

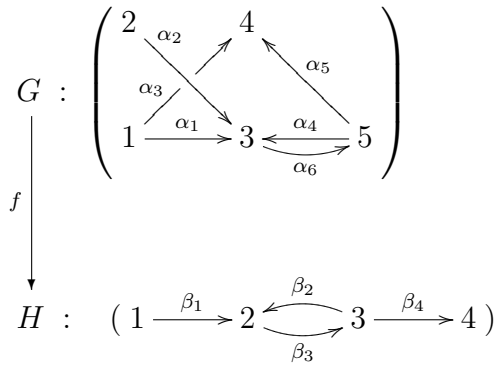
derart, daß

- $\mathrm{E}_f \cdot \mathrm{s}_H = \mathrm{s}_G \cdot \mathrm{V}_f$,
- $\mathrm{E}_f \cdot \mathrm{t}_H = \mathrm{t}_G \cdot \mathrm{V}_f$

gelten.

Kurz, das Bild einer Kante läuft zwischen den Bildern ihrer Endpunkte, unter Beibehaltung ihrer Richtung.

Also zum Beispiel wie folgt.



Hierbei soll alles senkrecht nach unten abgebildet werden. Das ist oft eine bequeme Methode, um nicht alle Kanten und Punkte einzeln abbilden zu müssen, insbesondere dann, wenn der Zielgraph zwischen je zwei Punkten in eine vorgegebene Richtung höchstens eine Kante hat.

Einzeln wird hierbei $1\mathrm{V}_f = 1$, $2\mathrm{V}_f = 1$, $3\mathrm{V}_f = 2$, $4\mathrm{V}_f = 2$, $5\mathrm{V}_f = 3$, sowie $\alpha_1\mathrm{E}_f = \beta_1$, $\alpha_1\mathrm{E}_f = \beta_1$, $\alpha_2\mathrm{E}_f = \beta_1$, $\alpha_3\mathrm{E}_f = \beta_1$, $\alpha_4\mathrm{E}_f = \beta_2$, $\alpha_5\mathrm{E}_f = \beta_2$, $\alpha_6\mathrm{E}_f = \beta_3$.

Es wird in der Tat z.B. für $\alpha_3 \in \mathrm{E}_G$

$$\begin{array}{rclclclcl} \alpha_3\mathrm{s}_G\mathrm{V}_f & = & 1\mathrm{V}_f & = & 1 & = & \beta_1\mathrm{s}_H & = & \alpha_3\mathrm{E}_f\mathrm{s}_H \\ \alpha_3\mathrm{t}_G\mathrm{V}_f & = & 4\mathrm{V}_f & = & 2 & = & \beta_1\mathrm{t}_H & = & \alpha_3\mathrm{E}_f\mathrm{t}_H\,. \end{array}$$

In Magma schreiben wir

```
G :=
<
[1,2,3,4,5],
[<1,1,3>,<2,2,3>,<1,3,4>,<5,4,3>,<5,5,4>,<3,6,5>]
>;
```

und

```
H :=
<
[1,2,3,4],
[<1,1,2>,<3,2,2>,<2,3,3>,<3,4,4>]
>;
```

und

```
f :=
<
[<1,1>,<2,1>,<3,2>,<4,2>,<5,3>],
[<<1,1,3>,<1,1,2>>,
 <<2,2,3>,<1,1,2>>,
 <<1,3,4>,<1,1,2>>,
 <<5,4,3>,<3,2,2>>,
 <<5,5,4>,<3,2,2>>,
 <<3,6,5>,<2,3,3>>]
>;
```

Aufgabe. Wählen Sie sich n, $m \geqslant 1$ und geben Sie alle Graphmorphismen von C_n nach C_m an.

Aufgabe. Sei G ein Graph. Wieviele Graphmorphismen gibt es von G nach C_1?

Die Liste aller Graphmorphismen von `G` nach `H` erhält man mittels der Funktion `ListGraphMorphisms`, die noch ein paar Vorbereitungen braucht.

```
IsRightUnique := function(u)
 // u: relation, e.g. u := [<1,3>,<1,4>,<2,3>], or u := {<1,3>,<1,4>,<2,3>}
 right_unique := true;
```

```
 left_elements := {x[1] : x in u};
 for y in left_elements do
  if #{x[2] : x in u | x[1] eq y} ge 2 then
   right_unique := false;
   break y;
  end if;
 end for;
 return right_unique;
end function;

RelationOnNodesFromPartialMapOnEdges := function(x);
 // x: partial map on edges from graph G to graph H (G, H not required as data)
 return {<z[1][1],z[2][1]> : z in x} join {<z[1][3],z[2][3]> : z in x};
end function;

CompletionsToMaps := function(D,C,u)
 // D/C: list of elements in the domain/codomain,
 // u: right unique relation, e.g.
 // D := [1,2,3,4]; C := [1,2,3,4,5]; u := {<1,3>, <3,5>};
 to_be_mapped := [x : x in D | not x in {y[1] : y in u}];
 list := [u];
 for i in to_be_mapped do
  list_new := [];
  for v in list do
   for j in C do
    list_new cat:= [v join {<i,j>}];
   end for;
  end for;
  list := list_new;
 end for;
 return list;
end function;

ListGraphMorphisms := function(G,H)
 list := [[]];
 for z in G[2] do
  list_new := [];
  for w in H[2] do
   for x in list do
    x_test := x cat [<z,w>];
    if IsRightUnique(RelationOnNodesFromPartialMapOnEdges(x_test)) then
     list_new cat:= [x_test];
    end if;
   end for;
  end for;
  list := list_new;
 end for;
```

```
 list_mor := []; // da sollen die kompletten Morphismen rein
 for y in list do
  list_completions_to_maps_on_nodes :=
   [Sort(SetToSequence(x)) :
    x in CompletionsToMaps(G[1],H[1],RelationOnNodesFromPartialMapOnEdges(y))];
  list_mor cat:= [<x,y> : x in list_completions_to_maps_on_nodes];
 end for;
 return list_mor;
end function;
```

Sei G ein Graph. Es gibt als Graphmorphismus die *Identität*

$$\mathrm{id}_G : G \to G \,,$$

für welche $\mathrm{V}_{\mathrm{id}_G} := \mathrm{id}_{\mathrm{V}_G}$ und $\mathrm{E}_{\mathrm{id}_G} := \mathrm{id}_{\mathrm{E}_G}$ ist.

Via Magma:

```
Identity := function(G);
 return <[<x,x> : x in G[1]], [<x,x> : x in G[2]]>;
end function;
```

Seien G, H und K Graphen. Seien $f : G \to H$ und $\tilde{f} : H \to K$ Graphmorphismen. Es gibt als Graphmorphismus das *Kompositum*

$$f \cdot \tilde{f} : G \to K \,,$$

für welches $\mathrm{V}_{f.\tilde{f}} := \mathrm{V}_f \cdot \mathrm{V}_{\tilde{f}}$ und $\mathrm{E}_{f.\tilde{f}} := \mathrm{E}_f \cdot \mathrm{E}_{\tilde{f}}$ ist.

Via Magma:

```
ComposeGraphMorphisms := function(p,q)
 u := []; // u for nodes
 for x in p[1] do
  u cat:= [<x[1],y[2]> : y in q[1] | x[2] eq y[1]];
 end for;
 v := []; // v for edges
 for x in p[2] do
  v cat:= [<x[1],y[2]> : y in q[2] | x[2] eq y[1]];
 end for;
 return <u,v>;
end function;
```

Ein Graphmorphismus $f : G \to H$, für welchen V_f und E_f bijektiv sind, heißt *Graphisomorphismus* oder kurz *Isomorphismus*. Ein Graphisomorphismus $f : G \to G$ heißt Graphautomorphismus.

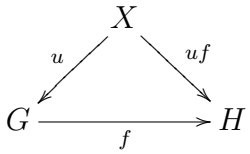
> *Projekt.* Die Automorphismen eines Graphen bilden eine Gruppe. Man finde Graphen mit einer Automorphismengruppe, die weder isomorph zu einer symmetrischen Gruppe noch isomorph zu einer zyklischen Gruppe ist. Man untersuche jeweils Stabilisatoren von Punkten und Kanten in der Automorphismengruppe.

Für Graphen X und G schreiben wir

$$(X,G)_{\mathrm{Gph}} \;:=\; \{\, X \xrightarrow{u} G :\; u \text{ ist ein Graphmorphismus} \,\}$$

Für einen Graphen X und einen Graphmorphismus $f : G \to H$ schreiben wir

$$\begin{array}{ccc} (X,G)_{\mathrm{Gph}} & \xrightarrow{(X,f)_{\mathrm{Gph}}} & (X,H)_{\mathrm{Gph}} \\ u & \mapsto & u \cdot f \end{array}$$

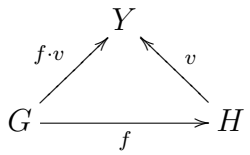


Für einen Graphen G ist $(X, \mathrm{id}_G)_{\mathrm{Gph}} = \mathrm{id}_{(X,G)_{\mathrm{Gph}}}$. Für Graphmorphismen $G \xrightarrow{f} H \xrightarrow{\tilde{f}} K$ ist $(X, f \cdot \tilde{f})_{\mathrm{Gph}} = (X,f)_{\mathrm{Gph}} \cdot (X,\tilde{f})_{\mathrm{Gph}}$.

Aufgabe. Man finde ein $n \in \mathbb{Z}_{\geqslant 1}$ und einen Graphmorphismus $G \xrightarrow{f} H$, für welchen die Abbildung $(\mathrm{C}_n, f)_{\mathrm{Gph}}$ nicht injektiv und nicht surjektiv ist.

Für einen Graphen Y und einen Graphmorphismus $f : G \to H$ schreiben wir

$$\begin{array}{ccc} (G,Y)_{\mathrm{Gph}} & \xleftarrow{(f,Y)_{\mathrm{Gph}}} & (H,Y)_{\mathrm{Gph}} \\ f \cdot v & \mapsfrom & v \end{array}$$



Für einen Graphen G ist $(\mathrm{id}_G, Y)_{\mathrm{Gph}} = \mathrm{id}_{(G,Y)_{\mathrm{Gph}}}$. Für Graphmorphismen $G \xrightarrow{f} H \xrightarrow{\tilde{f}} K$ ist $(f \cdot \tilde{f}, Y)_{\mathrm{Gph}} = (\tilde{f}, Y)_{\mathrm{Gph}} \cdot (f,Y)_{\mathrm{Gph}}$.

Projekt.

(a) Man finde Graphen X, für die die Abbildung $(X,G)_{\mathrm{Gph}} \xrightarrow{(X,f)_{\mathrm{Gph}}} (X,H)_{\mathrm{Gph}}$ surjektiv ist für alle Graphmorphismen $G \xrightarrow{f} H$ mit V_f und E_f surjektiv. Man finde auch Beispiele für Graphen X, die dies nicht erfüllen.

(b) Man finde Graphen Y, für die die Abbildung $(G,Y)_{\mathrm{Gph}} \xleftarrow{(f,Y)_{\mathrm{Gph}}} (H,Y)_{\mathrm{Gph}}$ surjektiv ist für alle Graphmorphismen $G \xrightarrow{f} H$ mit V_f und E_f injektiv. Man finde auch Beispiele für Graphen Y, die dies nicht erfüllen.

Nützlich kann auch noch sein:

```
IsEqual := function(f,ff) // returns true if graph morphisms f, ff: G -> H are equal
 return (SequenceToSet(f[1]) eq SequenceToSet(ff[1]))
    and (SequenceToSet(f[2]) eq SequenceToSet(ff[2]));
end function;
```

Projekt. Man verallgemeinere Graphmorphismen zu Graphrelationen. Man schreibe ein Magma-Programm für die Komposition solcher Relationen.

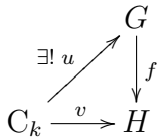
4.2 Quasiisomorphismen

Ein Graphmorphismus $G \xrightarrow{f} H$ heißt *Quasiisomorphismus*, falls für $k \geqslant 1$ die Abbildung

$$\begin{array}{ccc} (\mathrm{C}_k, G)_{\mathrm{Gph}} & \xrightarrow{(\mathrm{C}_k, f)_{\mathrm{Gph}}} & (\mathrm{C}_k, H)_{\mathrm{Gph}} \\ u & \mapsto & u \cdot f \end{array}$$

bijektiv ist.

Mit anderen Worten, f ist genau dann ein Quasiisomorphismus, wenn es für jedes $k \geqslant 1$ und jeden Graphmorphismus $\mathrm{C}_k \xrightarrow{v} H$ genau einen Graphmorphismus $\mathrm{C}_k \xrightarrow{u} G$ gibt mit $u \cdot f = v$.



Aufgabe. Finden Sie einen Quasiisomorphismus, der kein Isomorphismus ist.

Aufgabe. Finden Sie einen Graphmorphismus $f : G \to H$, der kein Quasiisomorphismus ist, für welchen aber $(\mathrm{C}_k, f)_{\mathrm{Gph}}$ bijektiv ist für $k \in [1, 4]$.

Frage (Jannik Hess). Sei $G \xrightarrow{f} H$ ein Graphmorphismus. Sei $m := \max\{|\mathrm{E}_G|, |\mathrm{E}_H|\}$. Sei $(\mathrm{C}_k, f)_{\mathrm{Gph}}$ bijektiv für $k \in [1, m]$. Ist f ein Quasiisomorphismus?

4.3 Faserungen

Für einen Graphen G und $v \in \mathrm{V}_G$ schreiben wir

$$G(v, *) \ := \ \{\, e \in \mathrm{E}_G : \ e\mathrm{s}_G = v \,\} \ .$$

Sei $G \xrightarrow{f} H$ ein Graphmorphismus. Wir schreiben

$$\mathrm{E}_{v,f} \ := \ \mathrm{E}_f|^{H(v\mathrm{V}_f,*)}_{G(v,*)} \ : \ G(v, *) \ \to \ H(v\mathrm{V}_f, *) \ : \ e \mapsto e\mathrm{E}_{v,f} := e\mathrm{E}_f \ .$$

Es heißt $G \xrightarrow{f} H$ eine *Faserung*, falls $\mathrm{E}_{v,f}$ surjektiv ist für alle $v \in \mathrm{V}_G$.

Es heißt $G \xrightarrow{f} H$ *Etalfaserung*, falls $\mathrm{E}_{v,f}$ bijektiv ist für alle $v \in \mathrm{V}_G$.

Aufgabe. Falls $G \xrightarrow{f} H$ eine Faserung ist, ist dann E_f surjektiv? Falls $G \xrightarrow{f} H$ eine Faserung ist mit V_f surjektiv, ist dann E_f surjektiv?

Es heißt $e \in \mathrm{E}_H$ *einzielig* bezüglich f, falls $|\{\, \hat{e}\mathrm{t}_G : \hat{e} \in \mathrm{E}_G, \hat{e}\mathrm{E}_f = e \,\}| = 1$ ist.

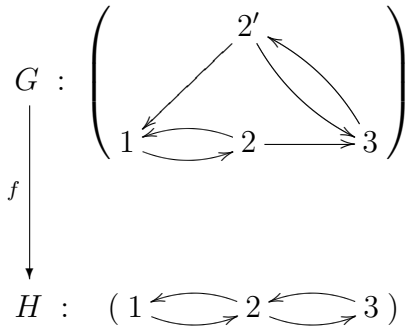
Aufgabe. Man gebe einen Graphmorphismus an, bei welchem eine Kante im Zielgraphen nicht einzielig ist.

Satz (Jannik Hess). Seien G und H dünne Graphen. Sei $G \xrightarrow{f} H$ eine Etalfaserung.

Gebe es für jedes $n \geqslant 1$ und jeden Graphmorphismus $u : \mathrm{C}_n \to H$ ein $i \in \mathbb{Z}/n\mathbb{Z}$ mit $\mathrm{e}_i\, \mathrm{E}_u$ einzielig bezüglich f.

Dann ist $G \xrightarrow{f} H$ ein Quasiisomorphismus.

Beispiel. Folgender Graphmorphismus ist dank Satz ein Quasiisomorphismus.



Aufgabe. Welche Kanten im Graphen H des Beispiels sind einzielig bezüglich f?

Aufgabe. Man finde eine Etalfaserung zwischen dünnen Graphen, die kein Quasiisomorphismus ist. Man überlege sich, daß für diese die Bedingung des Satzes bezüglich Einzieligkeit verletzt ist.

> *Projekt.* Man finde weitere Beispiele für Quasiisomorphismen $G \xrightarrow{f} H$ unter Verwendung des genannten Satzes. Man beschreibe dabei für geeignete $n \geqslant 1$ die Abbildung $(\mathrm{C}_n, f)_{\mathrm{Gph}}$.

```
IsFibration := function(G,H,f) // G,H: graphs, f: G -> H graph morphism
 for x in G[1] do
  y := [a[2] : a in f[1] | a[1] eq x][1]; // <x,y> in f[1] Punktabbildung
  for b in [h : h in H[2] | h[1] eq y] do
   if #[0 : a in G[2] | <a,b> in f[2]] eq 0 then
    return false;
   end if;
  end for;
 end for;
 return true;
end function;
```

```
IsEtaleFibration := function(f,G,H)
 for x in G[1] do
  y := [a[2] : a in f[1] | a[1] eq x][1]; // <x,y> in f[1] Punktabbildung
  if not #[e : e in G[2] | e[1] eq x] eq #[h : h in H[2] | h[1] eq y] then
   return false;
  elif not Sort([a[2] : a in f[2] | a[1][1] eq x])
        eq Sort([h : h in H[2] | h[1] eq y]) then
    return false;
  end if;
 end for;
 return true;
end function;
```

4.4 Azyklische Cofaserungen

Ein Graphmorphismus $G \xrightarrow{f} H$ heißt *azyklische Cofaserung*, falls die folgenden Bedingungen (AcC 1–5) erfüllt sind.

(AcC 1) Es ist V_f injektiv.

(AcC 2) Es ist E_f injektiv.

(AcC 3) Für $v \in \mathrm{V}_H$ mit $v \notin \mathrm{V}_G\mathrm{V}_f$ ist $|\{\, e \in \mathrm{E}_H : \, e\mathrm{t}_H = v \,\}| = 1$.

(AcC 4) Für $e \in \mathrm{E}_H$ mit $e \notin \mathrm{E}_G\mathrm{E}_f$ ist $e\mathrm{t}_H \notin \mathrm{V}_G\mathrm{V}_f$.

(AcC 5) Für $v \in \mathrm{V}_H$ mit $v \notin \mathrm{V}_G\mathrm{V}_f$ gibt es $n \geqslant 1$ und $e_i \in \mathrm{E}_H$ für $i \in [1, n-1]$ so, daß $e_1\mathrm{s}_H \in \mathrm{V}_G\mathrm{V}_f$, $e_i\mathrm{t}_H = e_{i+1}\mathrm{s}_H$ und $e_n\mathrm{t}_H = v$ ist.

Illustration zu (AcC 5) im Falle $n = 3$:

$$e_1\mathrm{s}_H \xrightarrow{e_1} \bullet \xrightarrow{e_2} \bullet \xrightarrow{e_3} v$$

Sei D_0 der Graph mit $\mathrm{V}_{\mathrm{D}_0} := \{0\}$ und $\mathrm{E}_{\mathrm{D}_0} := \emptyset$.

Ein Graph G heißt *Baum* mit *Wurzel* $r \in \mathrm{V}_G$, falls der Graphmorphismus $u : \mathrm{D}_0 \to G$, für welchen $0\mathrm{V}_u = r$ ist, eine azyklische Cofaserung ist.

Aufgabe. Man finde einen Baum G mit $|\mathrm{E}_G| = 3$. Man finde noch einen anderen solchen.

Anschaulich ist $G \xrightarrow{f} H$ eine azyklische Cofaserung, wenn dabei "H aus G entsteht durch Ankleben von Bäumen".

Jede azyklische Cofaserung ist ein Quasiisomorphismus.

Magma-Werkzeuge (brauchen auch `ListGraphMorphisms`):

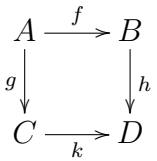
```
D := function(n)
 if n eq 0 then
  return <[0],[]>;
 end if;
 return <[i : i in [0..n]],[<i,i,i+1> : i in [0..n-1]]>;
end function;

AcCofib1to4 := function(G,H,f) // G,H: graphs, f: graph morphism
 if #[0 : x in H[1] | #[0 : a in G[1] | <a,x> in f[1]] ge 2] ge 1 then
  // asks: is f[1] not inj?
  return false;
 end if;
 if #[0 : x in H[2] | #[0 : a in G[2] | <a,x> in f[2]] ge 2] ge 1 then
  // asks: is f[2] not inj?
  return false;
 end if;
 HH1 := [x : x in H[1] | #[0 : a in G[1] | <a,x> in f[1]] eq 0];
 HH2 := [x : x in H[2] | #[0 : a in G[2] | <a,x> in f[2]] eq 0];
 if #[0 : x in HH1 | not #[a : a in H[2] | a[3] eq x] eq 1] ge 1 then
  // asks: does every vertex x not in the image of f have exactly one edge with target x
  return false;
 end if;
 if #[0 : x in HH2 | not x[3] in HH1] ge 1 then
  // asks: does each edge of HH have a target vertex in HH ?
  return false;
 end if;
 return true;
end function;


AcCofib5 := function(G,H,f)
 HH1 := [x : x in H[1] | #[0 : a in G[1] | <a,x> in f[1]] eq 0];
 max := #H[2];
 for v in HH1 do
  list := [];
  for i in [1..max] do
   L := ListGraphMorphisms(D(i),H);
   list cat:= [l : l in L | not l[1][1][2] in HH1 and l[1][#l[1]][2] eq v];
  end for;
  if #list eq 0 then
   return false;
  end if;
 end for;
 return true;
end function;
```

```
IsAcCofib := function(G,H,f)
 return AcCofib1to4(G,H,f) and AcCofib5(G,H,f);
end function;
```

5 Pullbacks und Pushouts

5.1 Pullbacks

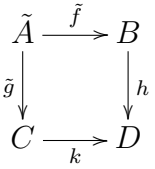
Sei ein kommutatives Viereck von Graphen und Graphmorphismen



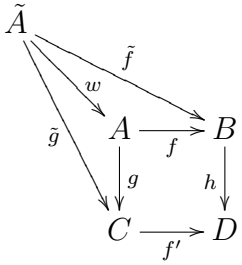
gegeben, sei darin also $f \cdot h = g \cdot k$.

Dieses Viereck, kurz (A, B, C, D) geschrieben, heißt *Pullback*, falls folgende universelle Eigenschaft (UE) gilt.

(UE) Für jedes kommutative Viereck von Graphen und Graphmorphismen

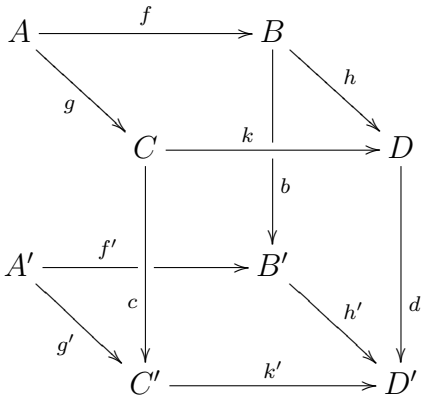


gibt es genau einen Graphmorphismus $w : \tilde{A} \to A$ mit $w \cdot g = \tilde{g}$ und $w \cdot f = \tilde{f}$.



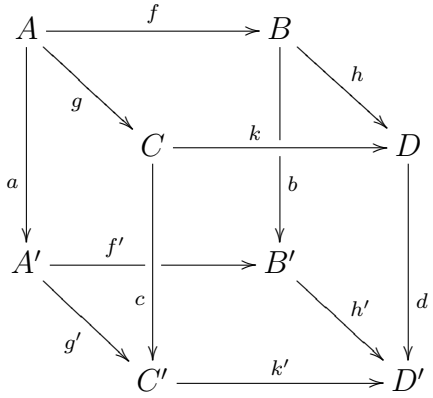
Aufgabe. Man konstruiere in einer solchen Situation einen Morphismus w, der kein Isomorphismus ist.

Sei nun ein kommutatives Diagramm wie folgt gegeben.



Seien darin die Vierecke (A, B, C, D) und (A', B', C', D') Pullbacks.

Dank der universellen Eigenschaft des Pullbacks (A', B', C', D') gibt es genau einen Graphmorphismus $A \xrightarrow{a} A'$ mit $a \cdot f' = f \cdot b$ und $a \cdot g' = g \cdot c$.



> *Projekt.* Man finde Beispiele für solche Diagramme, in welchen die vertikal eingetragenen Morphismen b, c, d noch Eigenschaften haben, wie Faserung, azyklische Cofaserung, Quasiisomorphismus etc. zu sein. Man untersuche in diesen Beispielen, welche Eigenschaften a dann hat oder eben nicht hat.
>
> Spezialfall: Die Pullbacks (A, B, C, D) und (A', B', C', D') können aus Faserungen bestehen. Oder teilweise aus Faserungen bestehen. Wie ist die Lage mit den genannten Beispielen dann?

Magma-Werkzeuge:

```
PullbackSets := function(X,Y,Y2,f,g) // f: X -> Y, g: Y2 -> Y, maps between sets
 // (set als Liste umgesetzt), gegeben in der Form
 // f = [<x_1,y_1>,...,<x_n,y_n>], was x_i |-> y_i bedeuten soll
 P := Sort([<x,y2> : x in X, y2 in Y2 |
      #[<s,t> : s in f, t in g | x eq s[1] and y2 eq t[1] and s[2] eq t[2]] eq 1]);
 return P;
end function;
```

```
PullbackGraphs := function(X,Y,Y2,f,g)
 // f: X -> Y, g: Y2 -> Y graph morphisms
 nodes := PullbackSets(X[1],Y[1],Y2[1],f[1],g[1]);
 edges := PullbackSets(X[2],Y[2],Y2[2],f[2],g[2]);
 N := [i : i in [1..#nodes]];
 E := [i : i in [1..#edges]]; // Edges noch ohne Start und Ziel, numeriert
 EE := [<Index(nodes,<edges[e][1][1],edges[e][2][1]>),
         e,
         Index(nodes,<edges[e][1][3],edges[e][2][3]>)> : e in E];
  // Edges, numeriert, mit Start und Ziel, auch numeriert
 PP := <N, EE>;
 vE := [<ee,edges[ee[2]][1]> : ee in EE]; // <p,p@v>
 uE := [<ee,edges[ee[2]][2]> : ee in EE]; // <p,p@u>
 vN := [<n,nodes[n][1]> : n in N]; // <p,p@v>
 uN := [<n,nodes[n][2]> : n in N]; // <p,p@u>
 v := <vN,vE>;
 u := <uN,uE>;
 numnodes := [<i,nodes[i]> : i in [1..#nodes]];
 numedges := [<i,edges[i]> : i in [1..#edges]];
 num := <numnodes,numedges>;
 return <PP,u,v,num>;
end function;



InducedMorphismGraphsPB := function(X,X2,Y,Y2,u,u2,v,v2)
 // u: X->X2, u2: X2->Y2, v: X->Y, v2: Y->Y2
 T := <X,v,u>;
 P := PullbackGraphs(Y,Y2,X2,v2,u2);
 c_uncode_nodes := [<t,<[r[2] : r in T[2][1] | r[1] eq t][1],
                       [r[2] : r in T[3][1] | r[1] eq t][1]>> : t in T[1][1]];
 c_uncode_edges := [<t,<[r[2] : r in T[2][2] | r[1] eq t][1],
                       [r[2] : r in T[3][2] | r[1] eq t][1]>> : t in T[1][2]];
 c_nodes := [<r[1],Index([p[2] : p in P[4][1]],r[2])> : r in c_uncode_nodes];
 c_edges := [<r[1],Index([p[2] : p in P[4][2]],r[2])> : r in c_uncode_edges];
 c := <c_nodes,c_edges>;
 return c;
end function;
```

5.2 Pushouts

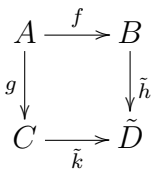
Sei ein kommutatives Viereck von Graphen und Graphmorphismen

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle h} \\ C & \xrightarrow[k]{} & D \end{array}$$

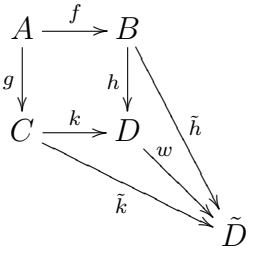
gegeben, sei darin also $f \cdot h = g \cdot k$.

Dieses Viereck, kurz (A, B, C, D) geschrieben, heißt *Pushout*, falls folgende universelle Eigenschaft (UE) gilt.

(UE) Für jedes kommutative Viereck von Graphen und Graphmorphismen

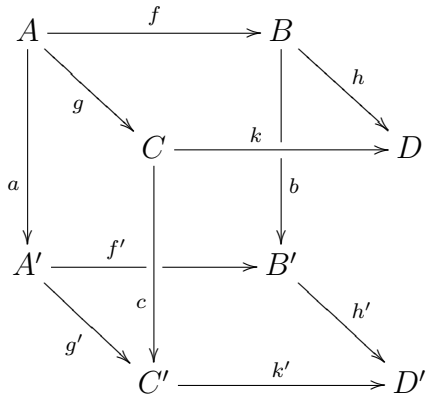


gibt es genau einen Graphmorphismus $w : D \to \tilde{D}$ mit $k \cdot w = \tilde{k}$ und $h \cdot w = \tilde{h}$.



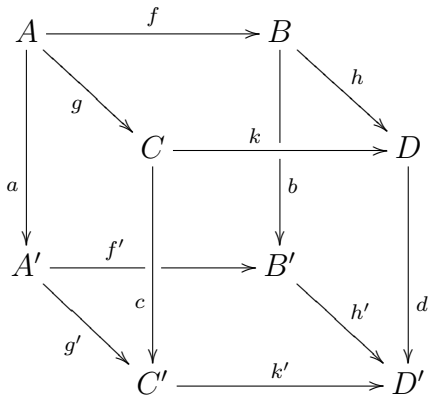
Aufgabe. Sei $A = \mathrm{D}_0$, mit nur einem Punkt und ohne Kanten. Man wähle Graphen B und C und Graphmorphismen $A \xrightarrow{f} B$ und $A \xrightarrow{g} C$. Man bilde den Pushout D. Wie kann man den Pushout D anschaulich beschreiben?

Sei nun ein kommutatives Diagramm wie folgt gegeben.



Seien darin die Vierecke (A, B, C, D) und (A', B', C', D') Pushouts.

Dank der universellen Eigenschaft des Pushouts (A, B, C, D) gibt es genau einen Graphmorphismus $D \xrightarrow{d} D'$ mit $k \cdot d = c \cdot k'$ und $h \cdot d = b \cdot h'$.



Projekt. Man finde Beispiele für solche Diagramme, in welchen die vertikal eingetragenen Morphismen a, b, c noch Eigenschaften haben, wie Faserung, azyklische Cofaserung, Quasiisomorphismus etc. zu sein. Man untersuche in diesen Beispielen, welche Eigenschaften d dann hat oder eben nicht hat.

Spezialfall: Die Pushouts (A, B, C, D) und (A', B', C', D') können aus azyklischen Cofaserungen bestehen. Oder teilweise aus azyklischen Cofaserungen bestehen. Wie ist die Lage mit den genannten Beispielen dann?

Magma-Werkzeuge:

```
RedSeq := function(S); // S: sequence, to be reduced and sorted
 return Sort(SetToSequence(SequenceToSet(S)));
end function;


Equivclasses := function(R,M) // R: relation on set M
 Rinv := [<r[2],r[1]> : r in R];
 Diag := [<m,m> : m in M];
 RR := [r : r in R cat Rinv | not r[1] eq r[2]];
 equivclasses := [];
 Mtodo := [m : m in M];
 while not #Mtodo eq 0 do
  k := Mtodo[1];
  kclassold := [];
  kclassnew := [k];
  while not #kclassnew eq #kclassold do
   kclassold := kclassnew;
   kclassnew cat:= [j[2] : j in RR | j[1] in kclassold];
   kclassnew := RedSeq(kclassnew);
  end while;
  equivclasses cat:= [kclassnew];
  Mtodo := [u : u in Mtodo | not u in kclassnew];
 end while;
 return equivclasses;
end function;


DisjointUnion := function(X,Y); // X, Y: Listen
 return [<1,x> : x in X] cat [<2,y> : y in Y];
end function;
```

```
PushoutSets := function(X,Y,X2,f,g); // f : X -> Y, g : X -> X2 maps
 M := DisjointUnion(X2,Y);
 R := [ [<<1, g_elt[2]>, <2, f_elt[2]>> : g_elt in g, f_elt in f
                                        | g_elt[1] eq x and f_elt[1] eq x][1] : x in X];
 equiv := Equivclasses(R,M);
 u := [ [<x2,t> : t in equiv | <1,x2> in t][1] : x2 in X2];
 v := [ [< y,t> : t in equiv | <2, y> in t][1] :  y in  Y];
 return <equiv, u, v>; // Pushout, u : X2 -> Pushout, v : Y -> Pushout
end function;

PushoutGraphs := function(X,Y,X2,f,g); // f : X -> Y, g : X -> X2 graph morphisms
 nodes := PushoutSets(X[1],Y[1],X2[1],f[1],g[1]);
 edges := PushoutSets(X[2],Y[2],X2[2],f[2],g[2]);
 N := [i : i in [1..#nodes[1]]];
 E := [i : i in [1..#edges[1]]]; // Edges noch ohne Start und Ziel, numeriert
 EE := [ <  Index(nodes[1], [n : n in nodes[1] |
                             <edges[1][e][1][1],edges[1][e][1][2][1]> in n][1]),e,
            Index(nodes[1], [n : n in nodes[1] |
                             <edges[1][e][1][1],edges[1][e][1][2][3]> in n][1])> :
         e in E];
       // Edges, numeriert, mit Start und Ziel, auch numeriert
 PP := <N,EE>;
 uN := [<x2, Index(nodes[1], [n[2] : n in nodes[2]
                                  | x2 eq n[1]][1]) > : x2 in X2[1]]; // <p,p@v>
 uE := [<x2, Index(edges[1], [e[2] : e in edges[2]
                                  | x2 eq e[1]][1]) > : x2 in X2[2]]; // <p,p@v>
       // zweiter Eintrag ist nur Nummer der Kante
 uEE := [ <x[1], <[ee[1] : ee in EE | ee[2] eq x[2]][1],x[2],
                  [ee[3] : ee in EE | ee[2] eq x[2]][1]>> : x in uE];
       // <p,p@u> // zweiter Eintrag bekommt noch source und target dazu
 u := <uN,uEE>;
 vN := [<y, Index(nodes[1], [n[2] : n in nodes[3]
                                 | y eq n[1]][1]) > : y in Y[1]]; // <p,p@v>
 vE := [<y, Index(edges[1], [e[2] : e in edges[3] | y eq e[1]][1]) > : y in Y[2]];
       // <p,p@v> // zweiter Eintrag ist nur Nummer der Kante
 vEE := [ <x[1], <[ee[1] : ee in EE | ee[2] eq x[2]][1],x[2],
                  [ee[3] : ee in EE | ee[2] eq x[2]][1]>> : x in vE];
       // <p,p@u> // zweiter Eintrag bekommt noch source und target dazu
 v := <vN,vEE>;
 return <PP,u,v>;
end function;

InducedMorphismGraphsPO := function(X,X2,Y,Y2,u,u2,v,v2)
 // u: X->X2, u2: X2->Y2, v: X->Y, v2: Y->Y2
 T := <Y2,u2,v2>;
 P := PushoutGraphs(X,Y,X2,v,u);
 x2_nodes := [<i,[t[2] : t in T[2][1] | t[1] in [r[1] : r in P[2][1] | r[2] eq i]]> : i in P[1][1]];
 x2_edges := [<i,[t[2] : t in T[2][2] | t[1] in [r[1] : r in P[2][2] | r[2] eq i]]> : i in P[1][2]];
 y_nodes := [<i,[t[2] : t in T[3][1] | t[1] in [r[1] : r in P[3][1] | r[2] eq i]]> : i in P[1][1]];
 y_edges := [<i,[t[2] : t in T[3][2] | t[1] in [r[1] : r in P[3][2] | r[2] eq i]]> : i in P[1][2]];
 c_nodes := [<r[1],r[2][1]> : r in x2_nodes | not #r[2] eq 0]
        cat [<r[1],r[2][1]> : r in y_nodes | not #r[2] eq 0];
 c_edges := [<r[1],r[2][1]> : r in x2_edges | not #r[2] eq 0]
        cat [<r[1],r[2][1]> : r in y_edges | not #r[2] eq 0];
 c := <Sort(SetToSequence(SequenceToSet(c_nodes))),Sort(SetToSequence(SequenceToSet(c_edges)))>;
 return c;
end function;
```

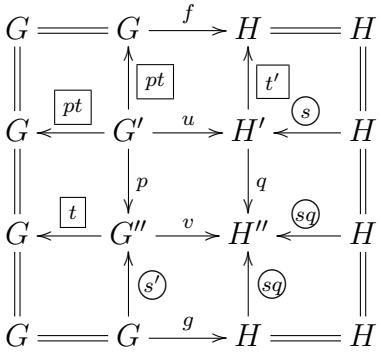
6 Homotopie

Wir kennen bereits azyklische Cofaserungen; vgl. §4.4.

Ferner heiße ein Graphmorphismus eine *azyklische Faserung*, wenn er zugleich eine Faserung und ein Quasiisomorphismus ist; vgl. §4.3, §4.2.

Seien nun G und H Graphen. Seien $G \overset{f}{\underset{g}{\rightrightarrows}} H$ Graphmorphismen.

Wir sagen, es ist f *homotop* zu g, falls es ein kommutatives Diagramm folgender Form von Graphen und Graphmorphismen gibt, in welchem s, sq und s' azyklische Cofaserungen, t, pt und t' azyklische Faserungen sind.



Vgl. [**4,** prop. 5.17.(a)]. Insbesondere sind dann p und q Quasiisomorphismen.

Projekt. Man finde Graphmorphismen, die homotop, aber nicht gleich sind.

Literatur

[1] Bisson, T.; Tsemo, A., *A homotopical algebra of graphs related to zeta series*, Homology, Homotopy and Applications 11(1), p. 171-184, 2009.

[2] Bosma, W.; Cannon, J.J.; Fieker, C.; Steel, A. (eds.), *Handbook of Magma functions,* Edition 2.16, 2010; cf. magma.maths.usyd.edu.au, magma.maths.usyd.edu.au/calc.

[3] Hess, J., master thesis, in Arbeit.

[4] Thomas, S., *On the* 3*-arrow calculus for homotopy categories*, Homology, Homotopy and Applications 13(1), pp. 89-119, 2011 (version arxiv.org/abs/1001.4536v2).