Vortrag: Persistenz und Algorithmen

Matthias Blank

2. Juli 2010

Zusammenfassung

Der Vortrag stellt im Wesentlichen die in [EH10, VII.1 und VII.2] behandelten Algorithmen zur Berechnung von Persistenz vor. Nach einer kurzen Wiederholung der theoretischen Grundlagen wird in 2.1 ein Algorithmus vorgestellt, der für einen beliebigen simplizialen Komplex dessen persistente Bettizahlen bestimmt. In 2.3 wird eine verbesserte Version des Algorithmus behandelt, die benutzt das die Randabbildungen im Allgemeinen dünn besetzt sind. In den Abschnitten 2.4, 2.5 und 2.6 wird der Union-Find-Algorithmus präsentiert und gezeigt, wie man mit dessen Hilfe die 0ten persistenten Bettizahlen in linearer Zeit berechnen kann.

1 Begriffe und Wiederholung

Im folgenden seien alle Komplexe, Graphen etc. als endlich vorausgesetzt. Mit Homologie sei stets simpliziale Homologie mit $\mathbb{F}_2$-Koeffizienten gemeint.

Definition 1.1.

- Im Folgenden sei stets K ein endlicher simplizialer Komplex und $f : K \to \mathbb{R}$ eine monotone Abbildung, d.h. $\sigma \leq \tau \Rightarrow f(\sigma) \leq f(\tau)$ für alle Simplizes $\sigma, \tau \in K$ (Wobei hier und im Folgenden $\sigma \leq \tau$ bedeuten soll das σ eine Seite von τ ist.)
- Wegen der Monotonizität von f sind die Mengen $K(a) := f^{-1}(]-\infty, a])$ Unterkomplexe von K. Sind $\sigma_1, \dots \sigma_n$ die Simplizes von K, $f(\sigma_i) = a_i$ und $K_i := K(a_i)$, so erhält man eine Sequenz

  $$\emptyset = K_0 \subset K_1 \subset \cdots \subset K_n = K$$

  von Unterkomplexen, die sogenannte *Filtrierung zu f*.
- Wichtiges Beispiel ist $f(\sigma_i) = i$ für eine Anordnung der Simplizes mit $\sigma_i \leq \sigma_j \Rightarrow i \leq j$. Diese sogenannte Indexfiltrierung soll unser zentrales Beispiel sein.
- Die Inklusionsabbildungen $K_i \to K_j$ induzieren in jeder Dimension p Abbildungen $f_p^{i,j} : H_p(K_i) \to H_p(K_j)$.
- In obiger Situation bezeichnet man $H_p^{i,j} := \operatorname{im} f_p^{i,j} \subset H(K_j)$ als *p-te persistente Homologiegruppe* (der Filtrierung $(K_i)_i$).
- Analog erklärt man persistente Betti-Zahlen, reduzierte persistente Homologie etc. Die zu $H_p^{i,j}$ gehörende persistente Betti-Zahl wird dabei entsprechend mit $\beta_p^{i,j}$ bezeichnet.
- Wir stellen uns die Elemente von $H_p^{i,j}$ als die Homologieklassen von K_i vor, „die in K_j noch am leben sind“. Wir sagen, eine Homologieklasse $\gamma \in H_p(K_i)$ wird zum Zeitpunkt i *geboren*, wenn $\gamma \notin H_p^{i-1,i}$. Wir sagen, dass ein zum Zeitpunkt i geborenes γ zum Zeitpunkt j *stirbt*, falls $f_p^{i,j}(\gamma) \in H_p^{i-1,j}$ und $f_p^{i,j-1}(\gamma) \notin H^{i-1,j-1}$.

- Möchte man vermeiden, dass Klasse unendliche lange leben, so kann man $K_{n+1} = pt$ setzen und in obige Sequenz aufnehmen, dass heißt man erzwingt, das alle Klassen spätestens bei $n+1$ sterben.
- Wenn γ zum Zeitpunkt i geboren wird und bei j stirbt, so nennt man $\text{pers}(\gamma) := a_j - a_i$, mit a_i und a_j wie oben, die *Persistenz* von γ.

2 Persistenz und Algorithmen

2.1 Matrixreduktion

Um mittels Matrixreduktion die persistente Homologie zu bestimmen, wählen wir zu-nächst eine Anordnung $\sigma_1, \ldots, \sigma_m$ der Simplizes in K die

$$f(\sigma_i) < f(\sigma_j) \Rightarrow i < j$$
$$\sigma_i \leq \sigma_j \Rightarrow i \leq j$$

erfüllt. Damit können wir nun die Randabbildungen geeignet in einer Matrix zusammenfassen:

Definition 2.1. In der obigen Situation definieren wir eine Gesamtrandmatrix ∂ durch:

$$\partial_{i,j} = \begin{cases} 1 & \text{Falls } \sigma_i \text{ eine Seite von } \sigma_j \text{ von Kodimension 1 ist} \\ 0 & \text{sonst} \end{cases}$$

Ist etwa $C_n \xrightarrow{\partial_n} C_{n-1} \xrightarrow{\partial_{n-1}} \ldots \xrightarrow{\partial_2} C_1 \xrightarrow{\partial_1} C_0 \to 0$ der simpliziale Kettenkomplex, so hat ∂ die folgende Gestalt:

	C_0	C_1	$\ldots$	C_n
C_0	0	$\boxed{\partial_1}$		0
$\vdots$			$\ddots$	
C_{n-1}				$\boxed{\partial_p}$
C_n	0			0

Definition 2.2. Für eine beliebige Matrix mit $\mathbb{F}_2$-Koeffizienten definieren wir $low(j)$ als größten Zeilenindex eines nicht-verschwindenden Eintrags in der j-ten Spalte der Matrix (d.h. als Zeilenindex der „am niedrigsten stehenden"1 in der j-ten Spalte) falls ein solcher Eintrag existiert. Andernfalls setzen wir $low(j) = -\infty$.

Wir nennen eine Matrix reduziert, falls für zwei verschiedene, nicht-verschwindende Spalten mit Indizes $i \neq j$ stets gilt, dass $low(i) \neq low(j)$.

Der folgende Algorithmus reduziert eine gegebene Matrix ∂ durch Links-Rechts-Spaltenadditionen und speichert die reduzierte Matrix als R.

Algorithmus 2.3.

```
R := ∂ ∈ Matrix_{F_2}(m, m)
for (j = 1 bis m)
{
        while(es gibt ein j_0 mit j_0 < j und 0 ≠ low(j_0) = low(j))
        {
                Addiere Spalte R_{j_0} zur Spalte R_j
        }
}
```

Die äußere und innere Schleife werden jeweils höchstens m mal durchlaufen, in der inneren Schleife werden zwei m-Spalten addiert, der Rechenaufwand ist also höchstens $O(m^3)$.

Bemerkung. Die Anzahl der zur Dimension p gehörenden nicht-verschwindenden Spaltenvektoren in R gibt $\operatorname{rk} B_{p-1}$ an, die Anzahl der zur Dimension p gehörenden 0-Spalten gibt $\operatorname{rk} Z_p$ an.

Proposition 2.4. *Seien $a_1, \ldots, a_{n+1} \in \mathbb{F}_2^m$ Spaltenvektoren, $A = (a_1, \ldots, a_n)$ eine $m \times n$-Matrix, $B = (a_1, \ldots, a_{n+1})$. Sei weiterhin $C \rightsquigarrow C^R$ obiger Reduktions-Algorithmus und $R = \partial^R = (r_1, \ldots, r_m)$ die bzgl. ^{R} reduzierte Randmatrix. Dann gilt*

i) Ist $B^R = (a'_1, \ldots, a'_{n+1})$ mit $a'_{n+1} = 0$, so gilt $\operatorname{rk} A = \operatorname{rk} B$, $\dim \ker A + 1 = \dim \ker B$.

ii) Ist $B^R = (a'_1, \ldots, a'_{n+1})$ mit $a'_{n+1} \neq 0$, so gilt $\operatorname{rk} A + 1 = \operatorname{rk} B$, $\dim \ker A = \dim \ker B$.

iii) Ist $r_j = 0$, so führt das Hinzufügen von σ_j zur Geburt eines $|\sigma_j|$-dimensionalen Zykels.

iv) Ist $r_j \neq 0$, so führt das Hinzufügen von σ_j zum Tod eines $(|\sigma_j| - 1)$-dimensionalen Zykels.

Genauer stirbt in iv) die zum Spaltenvektor r_j gehörende Homologieklasse und diese wurde zum Zeitpunkt $low(j)$ geboren.

Beweis: Da der Algorithmus von Links nach Rechts durchläuft, sieht man sofort, dass $((a_1, \ldots, a_n)^R, a'_{n+1}) = (a_1, \ldots, a_{n+1})^R$. Da Reduzieren den Rang einer Matrix nicht ändert und in einer reduzierten Matrix der Rang gleich der Anzahl der nicht verschwindenden Spaltenvektoren ist, erhält man damit i) und ii).

Der Algorithmus ist so konstruiert, dass er jede einzelne Randabbildungsmatrix in der Gesamtmatrix einzeln durch Links-Rechts-Spaltenumformungen reduziert. Wir können uns daher auf den Fall einer einzelnen Randabbildung beschränken. In diesem Fall sind Aussage iii) und iv) nur Umformulierungen von Aussage i) und ii) in die Sprache der persistenten Homologie.

Sei p die Dimension des j-ten Simplizes. Spaltenumformungen wie in unserem Algorithmus ändern nicht das Bild einer lineare Abbildung. Insbesondere lässt sich konkret ein Element angeben, das durch Hinzufügen des j-ten Simplexes neu im Bild der zugehörigen Randabbildung liegt, nämlich der durch den Spaltenvektor r_j dargestellte Zykel. Damit sieht man, dass eine der sterbenden Homologieklasse gerade $[r_j]$ ist. Diese wurde spätestens zum Zeitpunkt $i := low(j)$ geboren, denn die zu r_j gehörende Homologieklasse in $H_p(K^j)$ ist offensichtlich Bild der entsprechenden Homologieklasse in $H_p(K^i)$. Wäre $[r_j]$ älter als i), so gäbe es einen Zykel $s_j \in [r_j] \subset Z_p^j$, der älter als i wäre, also an der i-ten Stelle keinen Eintrag haben kann. Dann wäre $r_j - s_j$ nach Definition ein $(p-1)$-Rand, läge also im Bild der Randabbildung ∂_{p-1} und insbesondere auch im Bild der reduzierten Randabbildung ∂_{p-1}^R, wäre also Linearkombination der entsprechenden Spaltenvektoren von ∂_{p-1}^R. Nach Konstruktion von s_j wäre aber der niedrigste Eintrag von $r_j - s_j$ an der Stelle i, d.h. man könnte also mittels der gefundenen Linearkombination der Spaltenvektoren von ∂_{p-1}^R den Wert $low(j)$ weiter verringern, im Widerspruch zur Konstruktion von ∂^R mit unserem Algorithmus. □

Korollar 2.5. *Man erhält also: (i, j) ist ein endlicher Punkt im Persistenzdiagramm genau dann, wenn $low(j) = i \neq 0$. Ist $r_j = 0$, d.h. verschwindet die $j$te Spalte in der reduzierten Matrix, und gibt es kein j mit $low(j) = i$, so wird bei i eine Homologieklasse geboren, die nie stirbt, man erhält also einen unendlichen Punkt (i, ∞) im Persistenzdiagramm, bzw. in der Konstruktion im vorangegangenen Vortrag einen Punkt in $(i, m+1)$.*

Mittels der niedrigsten Einsen lässt sich also das Persistenzdiagramm direkt an der reduzierten Matrix ablesen. Wenn wir unseren Algorithmus verändern wollen sollten wir sicherstellen, dass diese Bedeutung der niedrigsten Einsen erhalten bleibt. Dazu:

Lemma 2.6 (Paarungslemma)**.** *Die niedrigsten Einsen sind unabhängig davon, mit welchem Algorithmus man die Randabbildung reduziert, solange man lediglich Links-Rechts-Spaltenumformungen verwendet.*

Übungsaufgabe 1. Sei R eine durch Links-Rechts-Spaltenumformungen aus ∂ erzeugte reduzierte Matrix, R_i^j die linke untere Blockmatrix deren rechte obere Ecke gerade $R_{i,j}$ ist. Analog sei ∂_i^j für ∂ statt R definiert.

i) Man zeige $\operatorname{rk} R_i^j = \operatorname{rk} \partial_i^j$.

ii) Man zeige für $r_R(i,j) := \operatorname{rk} R_i^j - \operatorname{rk} R_{i+1}^j + \operatorname{rk} R_{i+1}^{j-1} - \operatorname{rk} R_i^{j-1}$:
$$r_R(i,j) = \begin{cases} 1 & \text{falls } low(j) = i \\ 0 & \text{sonst} \end{cases}$$

iii) Man folgere das Paarungslemma.

2.2 Ein Beispiel

Beispiel 2.7. Als Beispiel berechnen wir die reduzierte persistente Homologie eines Voll-Dreiecks. Dies lässt sich als simplizialer Komplex mit einem 2-Simplex, drei 1-Simplizes und drei 0-Simplizes darstellen. Da wir an der reduzierten Homologie interessiert sind fügen wir noch formal einen (-1)-Simplex hinzu, der der Rand jedes 0-Simplizes sei. Wir erhalten folgenden reduzierten simplizialen Kettenkomplex mit $\mathbb{F}_2$-Koeffizienten:

$$\begin{array}{ccccccccc} C_2(K) & \xrightarrow{\partial_2} & C_1(K) & \xrightarrow{\partial_1} & C_0(K) & \xrightarrow{\partial_0} & C_{-1}(K) & \longrightarrow & 0 \\ \| & & \| & & \| & & \| & & \\ \mathbb{F}_2 & \longrightarrow & \mathbb{F}_2^3 & \longrightarrow & \mathbb{F}_2^3 & \longrightarrow & \mathbb{F}_2 & \longrightarrow & 0 \end{array}$$

Wir nummerieren nun die Simplizes von 0 bis 7 so, dass wir eine monotone Anordnung erhalten. Indem wir die Randabbildungen zusammensetzen erhalten wir ein ∂, das wir mittels unseres Algorithmus reduzieren können:

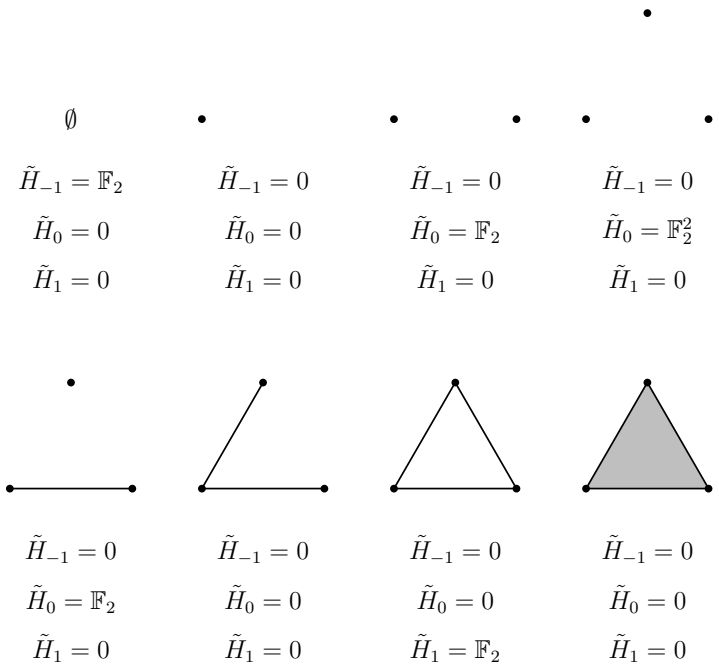
$$\partial = \begin{pmatrix} 0&1&1&1&0&0&0&0 \\ 0&0&0&0&1&1&0&0 \\ 0&0&0&0&1&0&1&0 \\ 0&0&0&0&0&1&1&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}; R = \begin{pmatrix} 0&\boxed{1}&0&0&0&0&0&0 \\ 0&0&0&0&1&1&0&0 \\ 0&0&0&0&\boxed{1}&0&0&0 \\ 0&0&0&0&0&\boxed{1}&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&\boxed{1} \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

Die markierten Kästchen geben die Positionen der niedrigsten Einsen in den nicht verschwindenden Spalten an. Wir haben in R genauso viele verschwindende wie nicht verschwindende Spalten, also keine Homologieklassen unendlicher Lebensdauer. Die Koordinaten der niedrigsten Einsen geben uns die Geburts- und Sterbezeitpunkte der Homologieklassen mit endlicher Lebensdauer an. Genauer erhalten wir in diesem Beispiel (wobei die Spalten und Zeilen von 0 bis 7 nummeriert seien)

$$low(1) = 0;\ low(4) = 2;\ low(5) = 3;\ low(7) = 6$$

Nach obigen Feststellungen erhalten wir also die Geburt eines -1-Zykels bei 0 welcher bei 1 stirbt, zweier 0-Zykel bei 2 und 3, welche bei 4 und 5 sterben, und

Abbildung 1: Aufbau des Simplizialkomplexes



eines 1-Zykels bei 6, welcher bei 7 stirbt. Dies deckt sich mit der geometrischen Beobachtung (siehe Abbildung 1).

Schließlich fassen wir das Ergebnis noch in den Persistenzdiagrammen zusammen. (Siehe nächste Seite).

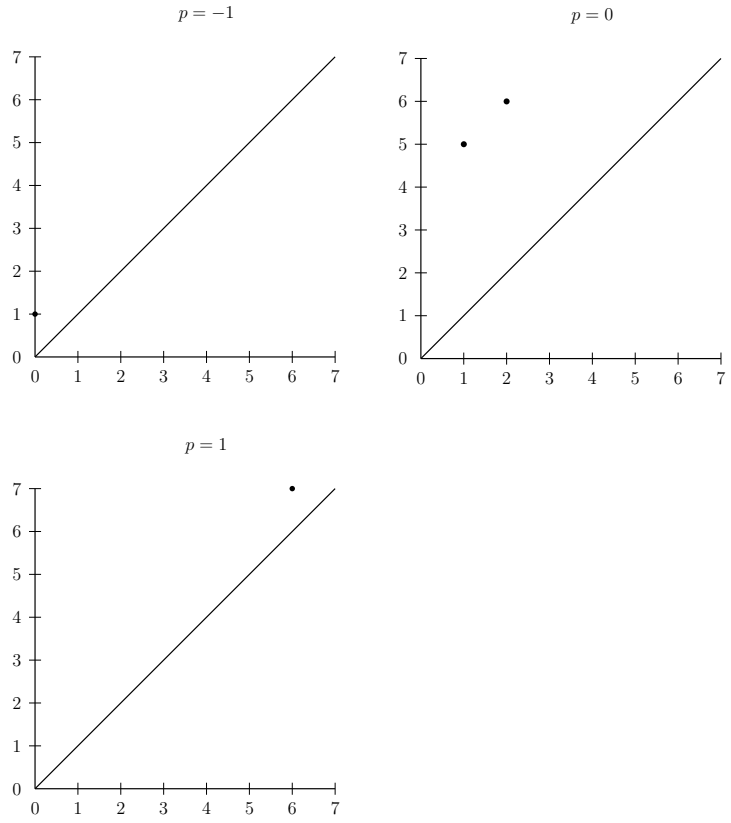
Übungsaufgabe 2. Berechne analog die persistente Homologie eines Simplizialkomplexes aus zwei an einer Seite verklebten Dreiecken.

2.3 Reduktion dünn besetzter Matrizen

Definition 2.8 (Array und Linked List)**.**

- Ein Array ist eine Datenstruktur, die eine a priori festgelegte Anzahl von Daten eines bestimmten Typs speichern kann. Beispielsweise würde das zum Befehl `int vektor[1000]` gehörende Array genau 1000 Daten des Typs int speichern können. In obigem Matrixreduktionsalgorithmus waren die Matrizen (implizit) als Arrays definiert mit stets m^2 Einträgen. Wichtig ist, dass dafür der gesamte Speicherplatz bei der Definition des Arrays zugewiesen wird und damit belegt ist, unabhängig davon, ob auch tatsächlich Daten gespeichert werden. Möchte man zudem z.B. einen 1001-ten Eintrag speichern, so muss man einen neuen Array mit 1001 Einträgen definieren und die ersten tausend Einträge kopieren.
- Eine sogenannte Linked List (oder verkettete Liste) speichert ebenfalls Daten. Im Gegensatz zum Array wird allerdings kein a priori festgelegter Speicherblock für die gesamte Struktur reserviert. Stattdessen werden die Daten folgendermaßen gespeichert: Für jeden Eintrag (*node*) der Linked List wird eigenständig Speicherplatz reserviert. Ein solcher Eintrag besteht aus dem eigentlichen Datum an dieser Stelle und aus einem Zeiger auf den

Abbildung 2: Die Persistenzdiagramme



nächsten Eintrag. Zusätzlich benötigt man noch einen sogenannten Kopf (*head*), der einen Zeiger auf den ersten Eintrag enthält. Zur Erinnerung: Ein Zeiger ist nichts anderes als ein Verweis auf eine bestimmte Stelle im Speicher. Für uns von Bedeutung ist, dass eine Linked List im wesentlichen nur den Speicher blockiert, den ihre Einträge auch wirklich belegen. Zudem ist es leicht möglich eine gegebene Linked List zu verlängern, indem man einfach in den letzten Eintrag einen Zeiger auf das neu eingefügte Segment einträgt.

Beide Datenstrukturen besitzen noch weitere relevante Vor- und Nachteile, die ihre Verwendung von der gegebenen Situation abhängig machen. Für unsere Zwecke soll aber obige Beschreibung ausreichen.

Bemerkung (Anwendung auf die Randmatrix)**.** Für große simpliziale Komplexe wird die Randmatrix schnell sehr groß, andererseits ist sie im Allgemeinen nur sehr dünn besetzt. Um Speicherplatz zu sparen, speichert man daher nicht die ganze Matrix als $m \times n$ Array, sondern erfasst lediglich die Plätze der Einsen. Da sich deren Anzahl im Laufe unseres Algorithmus ändern kann, ist es notwendig, zu Strukturen variabler Länge überzugehen. Man definiert dazu einen Array ∂ mit m Einträgen. An jeder Stelle $\partial[i]$ speichere man den i-ten Spaltenvektor der Randabbildung als Linked List. Allerdings speichert man nicht mehr den gesamten Spaltenvektor, sondern nur die Positionen der Einsen. Unser Algorithmus

lässt sich dann auch auf diese Situation übertragen:

Algorithmus 2.9.

```
R = ∂, pers = 0;
for{j = 1 bis m}
{
        L = ∂[j].cycle; R[j].cycle = 0;
        while{L ≠ NULL und R[i] ist besetzt für i = HEAD(L)}
        {
                L = L + R[i]
        }
        if{L ≠ NULL}
        {
                R[i] = L
                pers[j] = i
        }
}
```

In diesem Algorithmus wird mit `R[j]` auf die an der Stelle j stehenden Linked List zugegriffen. Addition ist als Vereinigung von Linked Lists unter Entfernung von in beiden Listen auftretenden Einträgen zu verstehen (entspricht mod 2 Arithmetik), HEAD gibt jeweils den Index des zuletzt hinzugekommenen Simplexes in der entsprechenden Liste aus und entspricht *low* in der ersten Version des Algorithmus. Der Algorithmus unterscheidet sich von der ersten Version zudem dadurch, dass die Information über die niedrigsten Einsen nicht in der Matrix R gespeichert wird. Stattdessen speichert der Vektor pers an der j-ten Stelle gerade den Wert von $low(j)$ (siehe Kapitel zur Matrixreduktion).

2.4 Persistenz in Dimension 0

Es ist möglich das 0te Persistenzdiagram sehr viel effizienter durch geometrische Argumente zu bestimmen. Ausgangspunkt unserer Überlegungen ist folgende elementare Feststellung:

Bemerkung. Die 0te Homologie eines zusammenhängenden Simplizialkomplexes ist stets $\mathbb{F}_2$, allgemeiner gibt die 0te Bettizahl genau die Anzahl der Zusammenhangskomponenten des Komplexes an.

Die 0te Homologie eines simplizialen Komplexes hängt nur von den Simplizes in Dimension 0 und 1 ab, es genügt also erst einmal simpliziale Graphen zu betrachten. Damit gibt es zunächst zwei Fälle zu betrachten:

i) Das Hinzunehmen eines 0-Simplizes erhöht stets die 0te Betti-Zahl um 1 (da jede 0-Kette stets ein 0-Zykel ist bzw. stets die Anzahl der Zusammenhangskomponenten erhöht.

ii) Nimmt man einen 1-Simplex hinzu, so gibt es zwei Möglichkeiten: Entweder gehören beide Ecken des 1-Simplizes zur selben Zusammenhangskomponente, dann ändert das Hinzunehmen nicht die Anzahl der Zusammenhangskomponenten und damit nicht die 0te Bettizahl. Andernfalls verringert sich die Anzahl der Zusammenhangskomponenten und damit die nullte Bettizahl um eins. Es stirbt dabei stets die zur jüngeren der beiden Komponenten gehörende Homologieklasse.

Wenn wir die nullte persistente Homologie ausrechnen wollen, brauchen wir also nur einen Algorithmus, der die in ii) vorkommenden Möglichkeiten unterscheidet und gegebenenfalls die ältere Komponente bestimmt. Dazu benutzen wir eine Variante des Union-Find-Algorithmus, den wir zunächst vorstellen.

2.5 Union-Find-Algorithmus

Definition 2.10. Ein *(endlicher simplizialer) Graph* ist ein 1-dimensionaler (endlicher) simplizialer Komplex. Ein *Baum* ist ein einfach-zusammenhängender Graph, d.h. ein Graph in dem sich zwei Ecken auf genau eine Weise durch Kantenpfade verbinden lassen (d.h. es gibt keine Kreise).

Lemma 2.11. *Jeder zusammenhängende Graph besitzt einen maximalen Teilbaum, der alle Ecken des Graphen enthält.*

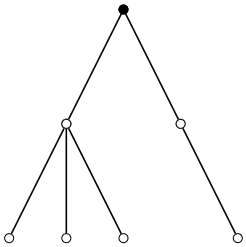
Übungsaufgabe 3.

i) Beweise Lemma 2.11.

ii) Benutze die Beweisidee, um einen Algorithmus anzugeben, der zu einem gegebenen Graphen einen maximalen Unterbaum angibt.

iii) Beweise Lemma 2.11 auch für unendliche Graphen.

Abbildung 3: Ein Wurzelbaum (mit schwarz markierter Wurzel)



Bemerkung. Ein Baum zusammen mit einer ausgezeichneten Ecke, der sogenannten Wurzel, heißt auch *Wurzelbaum*. Jede Ecke des Wurzelbaums liegt auf einem eindeutigen Kantenpfad der diese mit der Wurzel verbindet, und wir nennen eine Ecke i *Vorgänger (parent)* einer Ecke j, falls i auf dem zu j und der Wurzel gehörenden Kantenpfad direkt „oberhalb“ von j liegt (siehe Abbildung 3). Aus formalen Gründen definieren wir dabei noch den Vorgänger der Wurzel als die Wurzel selbst. Auf diese Weise besitzt jede Ecke einen eindeutig bestimmten Vorgänger.

Man kann einen Baum daher folgendermaßen als Array speichern: Man wähle zuerst eine Wurzelbaumdarstellung und speichere dann den Baum als Array $V(n)$ von Ecken, der an der $i$ten Stelle einen Zeiger auf den Vorgänger parent$[i]$ der Ecke i hat (und üblicherweise auch Zeiger auf die Nachfolger). Entsprechend speichert man einen Graphen, der disjunkte Vereinigung mehrerer Bäume ist (ein sogenannter *Wald*) in einem einzigen Array (siehe Abbildung 4).

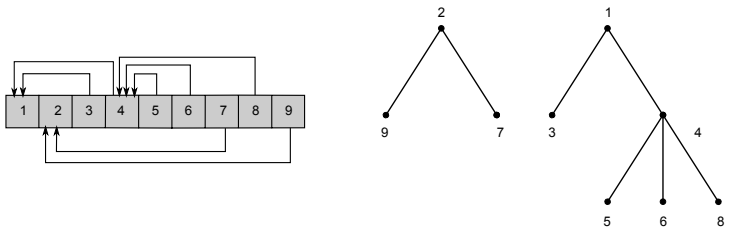
Beispiel 2.12.

Da wir nur an den Zusammenhangskomponenten interessiert sind, können wir statt der Komponenten eines Graphen maximale Unterbäume betrachten und diese auf obige Weise speichern. Wenn wir stets dafür sorgen, dass die Wurzel jeder Komponente ihre älteste Ecke ist, können wir direkt durch vergleichen der Wurzeln entscheiden, ob zwei Ecken in einer Komponente liegen, und falls nicht, welche von beiden älter ist. Die Wurzel zu einer Ecke i können wir mit einem einfachen Algorithmus bestimmen:

Algorithmus 2.13.

```
int Find(i)
if(V[i].parent ≠ i) return Find(V[i].parent)
elseif return i
endif
```

Abbildung 4: Speicherung eines Waldes



Der Algorithmus geht solange von einer Ecke zu deren Vorgänger, bis er die Wurzel findet (die ihr eigener Vorgänger ist) und diese dann ausgibt. Um die persistente Homologie bestimmen zu können benötigen wir noch einen weiteren Algorithmus, der zwei Bäume durch hinzunehmen einer Kante zwischen den Wurzeln vereinigt, den Union-Algorithmus:

Algorithmus 2.14.

```
void union(i,j)
x = Find(i); y = Find(j)
if(x≠y)
{
        V[x].parent = y
}
```

Wenn man den Find-Algorithmus wiederholt aufruft, hat man das Problem, dass lange Pfade unter Umständen wiederholt durchlaufen werden, was unnötig die Rechenzeit erhöht. Um dieses Problem zu umgehen verwenden wir einen modifizierten Algorithmus, der einmal durchlaufene Pfade abkürzt. Außerdem lässt sich dieses Problem auch im Union-Algorithmus verringern, indem dieser so angepasst wird, dass stets die Wurzel des kleineren Pfades Vorgänger der Wurzel des längeren Pfades wird. (Deren Länge erfasst `V[X].size`).

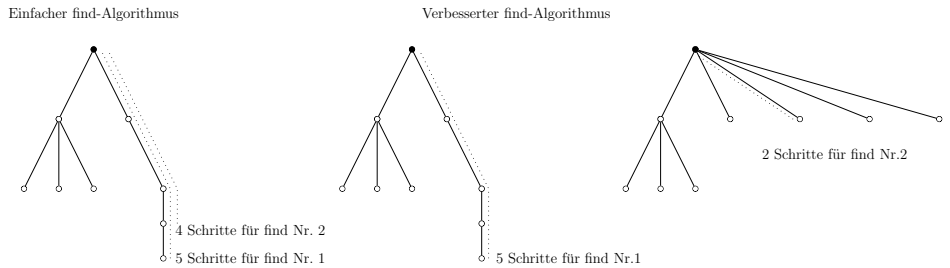
Algorithmus 2.15 (verbesserter Find)**.**

```
int Find(i)
if(V[i].parent ≠ i)
{
        V[i].parent = Find(V[i].parent)
}
return i
```

Algorithmus 2.16 (verbesserter Union)**.**

```
void union(i,j)
x = Find(i); y = Find(j)
if(x≠y)
{
        if(V[x].size > V[y].size)
                {
                d = x; x = y; y = d
                }
                V[x].parent = y
}
```

Abbildung 5: Vergleich der beiden find-Varianten



Für die verbesserten Algorithmen lässt sich zeigen, dass die Rechenzeit für m Union-Find-Durchläufe auf derselben Datenstruktur bei n Ecken in der Größenordnung $\alpha(n)m$ liegt. dabei ist α die inverse Ackermann-Funktion, welche für alle für uns relevanten Werte von n kleiner als 5 ist. Eine ausführliche Analyse des Union-Find-Algorithmus und eine Definition der Ackermann-Funktion findet man zum Beispiel in [CLRS01, Abschnitt 21.4].

2.6 Algorithmus für Dimension 0

Um das 0te Persistenzdiagramm zu bestimmen gehen wir nun folgendermaßen vor. Mit jedem neuen 0-Simplex entsteht eine neue Homologieklasse. Wir speichern die 0-Simplizes als Array, sortiert nach Alter, den wir als Menge von (einpunktigen) Bäumen auffassen. Wir gehen dann induktiv wie folgt vor: Zum Zeitpunkt k starten wir mit einer Menge von Bäumen, die wir als Simplizialkomplex zum Zeitpunkt $k-1$ interpretieren. Wir fügen nun den $k$ten 1-Simplex σ_k folgendermaßen hinzu: Seien etwa i, j seine Ecken. Mit Find entscheiden wir, ob diese zur selben Zusammenhangskomponente gehören. Falls ja, so ändern wir nichts und gehen zu $k+1$ über (Hinzufügen von σ_k hat keinen Einfluss auf die Zusammenhangskomponente). Andernfalls fügen wir eine Kante zwischen den zu i und j gehörenden Wurzeln ein (es spielt in diesem Fall keine Rolle, wo wir die Kante einfügen) so dass die ältere Wurzel Vorgänger der Jüngeren wird und merken uns, dass die jüngere Zusammenhangskomponente stirbt. Insgesamt erhalten wir wieder eine Menge von Bäumen deren älteste Ecken ihre Wurzeln sind und wir können induktiv fortfahren. Insgesamt konnten wir so die 0te persistente Homologie mit einem Rechenaufwand von $O(n\alpha(n))$ berechnen, wobei n die Anzahl der 0- und 1-Simplizes ist.

Bemerkung. Wir haben damit einen effizienten Algorithmus gefunden, um Persistenz in Dimension 0 direkt aus zurechnen. Für die wichtige Klasse geschlossener, triangulierte 2-Mannigfaltigkeiten (geschlossene Flächen) kann man zeigen, dass es auch für Dimension 1 einen Algorithmus gleicher Effizienz gibt, siehe auch [EH10, VII.2 und VII.3]. Für diese Mannigfaltigkeiten genügt das bereits um das Persistenzdiagramm zu bestimmen, da sie nur Homologie in Dimension $p = 0, 1, 2$ haben und der Fall $p = 2$ für diese Beispielklasse direkt ablesbar ist.

Literatur

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Intoduction to algorithms*, The MIT Press, 2001

[EH10] Herbert Edelsbrunner, John L. Harer, *Computational topology*, AMS, Providence, 2010