Dieser Abschnitt behandelt die versionsübergreifende Suche nach Umsteigern. Es geht also zum einen darum zu bestimmen, ob ein bestimmter Kode einer bestimmten Version sich überhaupt verändert hat, sowie zum anderen darum, abhängig von einer Start- und Zielversion zu ermitteln, welche Überleitungen von Kodes möglich sind. Die hier aufgeführten Algorithmen sind unabhängig von einer konkreten Implementation verfasst.

§ Wichtig: 1 Umsteiger nur Veränderungen 2 Datenstrukturen

1 Allgemeine Funktionen

Grundlegend ist davon auszugehen, dass Funktionen zum Lesen der Daten vorliegen nach erfolgreicher Integration aus dem vorherigen Abschnitt.

1.1 Daten Lesen

Konkret könnten zum Beispiel die Inhalte der Kodes- und Umsteiger-Dateien mit folgenden Parametern ausgelesen werden.

readData

- `$system`
  Das Kodiersystem, beispielsweise als Konstanten definiert `'icd10gm'` und `'ops'`.
- `$version`
  Die Version, beziehungsweise Jahreszahl.
- `$code`
  Ein Kode, nach dem gesucht wird. In Bezug auf die Notation von ICD-10-GM und OPS ist es sinnvoll automatisch an rechter Stelle einen Wildcard zu implizieren, das heißt einen Platzhalter für beliebige, weitere Zeichen. Dann werden mit einem leeren Kode-Wert auch alle Daten für die angegebenen Parameter gelesen.
- `$type`
  Die Art an Daten, die gelesen werden sollen. Für die Suche der Umsteiger werden folgende benötigt – angegeben ebenfalls wieder als mögliche Konstanten:
  - `'kodes'`
    Kodes-Daten: Kode und Titel.
  - `'umsteiger'`
    Umsteiger-Daten: alter Kode und neuer Kode, sowie Information über die automatische Überleitbarkeit in beide Richtungen. Es werden die Umsteiger von der angegebenen auf die nächstältere Version abgefragt. Suche über $code = neuer Kode.
  - `'umsteiger_join'`
    Wie Umsteiger-Daten, aber zusätzlich mit Titel für jeweils den alten und neuen Kode. „Join“ bezieht sich auf die entsprechende Datenbankoperation.
  - `'umsteiger_join_alt'`
    Wie oben, außer dass die Suche über $code = alter Kode und in die andere Richtung erfolgt. Das heißt es wird die angegebene Version mit der nächstneueren Version verglichen. „Alt“ für „alternate“.

1.2 Nächste Version

Außerdem wird eine Funktion benötigt, die ausgehend von System und Version die jeweils nächstältere und -neuere Version ermittelt. Wenn diese nicht existiert, bleibt der Wert leer.

Hierfür kann die in §TODO: Verweis im Kommentar erwähnte strukturierte Datei dienen, welche die Versionen und Abweichungen definiert.

nextNewerVersion / nextOlderVersion

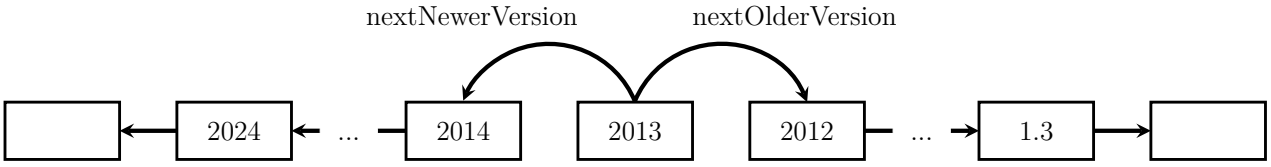


Abbildung 1: Ermitteln der nächstliegenden Versionen am Beispiel von ICD-10-GM.

1.3 Beispieldaten mit graphischer Darstellung

Um beispielsweise die Kodes M21.4 und M21.6 aus der ICD-10-GM Version 2016 abzufragen, sehen die Funktionsaufrufe dann so aus:

```
readData ('icd10gm', '2014', 'M21.4', 'kodes')
readData ('icd10gm', '2014', 'M21.6', 'kodes')
```

Und das Ergebnis als Tabelle:

code (Kode)	name (Titel)
M21.4	Plattfuß [Pes planus] (erworben)
M21.6	Sonstige erworbene Deformitäten des Knöchels und des Fußes

Für den restlichen Abschnitt wird angenommen, dass die Ergebnisse der readData-Funktion in einer zweidimensionalen Datenstruktur gespeichert werden die einer Tabelle ähneln, das heißt eine Array-Liste mit einem Eintrag pro Zeile beginnend mit Index = 0, sowie einem assoziativen Array (Map oder auch eine Klassenstruktur) mit Key = Spaltenüberschrift. Der Zugriff auf den Titel „Plattfuß“ in deren oberen Tabelle erfolgt dann über `$data[ 0 ][ 'name' ]`.

Analog zu den Kodes können die Umsteiger wie folgt abgefragt werden:

```
lies_daten ('icd10gm', '2014', 'M21.4', 'umsteiger')
lies_daten ('icd10gm', '2014', 'M21.6', 'umsteiger')
```

Das Ergebnis sieht wie folgt aus; “Auto” steht für die automatische Überleitbarkeit in die jeweilige Richtung, das heißt 2014←2013 und 2014→2013.

new (Kode Version 2014)	old (Kode Version 2013)	←Auto	→Auto
M21.4	M21.4	A	A
M21.6	M21.6	A	A

Von Version 2014 auf 2013 ändern sich die angegebenen ICD-10-GM Kodes nicht. Wie bereits in der Datenintegration erwähnt, müssen diese Einträge gar nicht aufgenommen werden. Sie sind hier nur zur Vereinfachung aufgelistet, weil die Einträge auch so in den BfArM-Dateien enthalten sind. Bei der versionsübergreifenden Suche nach Umsteigern sind nur die Veränderungen relevant. Anstatt alle nicht relevanten Umsteiger-Einträge abzuspeichern und bei jeder Abfrage auszuschließen, kann alternativ auch angenommen werden, weil eine Suche nach Umsteigern immer von einem vorhanden

Kode ausgeht, dass bei Nichtvorhandensein eines Umsteigers zwischen zwei Versionen dieser Kode einfach gleich bleibt.

Insgesamt treten bei M21.4 über alle Versionen keine Veränderungen in den Umsteigern auf. M21.6 hat allerdings folgende:

new (Kode Version 2015)	old (Kode Version 2014)	←Auto→	
M21.4	M21.4	A	A
M21.60	M21.6		A
M21.61	M21.6		A
M21.62	M21.6		A
M21.63	M21.6		A
M21.68	M21.6		A

new (Kode Version 2013)	old (Kode Version 2012)	←Auto→	
M21.4	M21.4	A	A
M21.6	M21.60	A	
M21.6	M21.67	A	
M21.6	M21.87	A	A

new (Kode Version 2.0)	old (Kode Version 1.3)	←Auto→	
M21.4	M21.4	A	A
M21.60	M21.6		A
M21.67	M21.6		A
M21.87	M21.8		A

Anmerkung: Von Kode M21.8, Version 1.3. ausgehend gibt es noch wesentlich mehr Umsteiger, aber diese werden hier nicht gelistet, weil sind von M21.6, Version 2014 aus nicht erreichbar sind.

Die Ergebnisse können als Graph dargestellt werden. Die Versionen sind horizontal angeordnet und die Kodes pro Version vertikal darunter als Knoten. Ausgangspunkt sind wie erwähnt die Kodes M21.4 und M21.6 der Version 2014. Die Knoten der anderen Versionen entsprechen den Kodes, die über Umsteiger erreichbar sind. Die Pfeilspitzen geben die Richtung der automatischen Überleitbarkeit an.

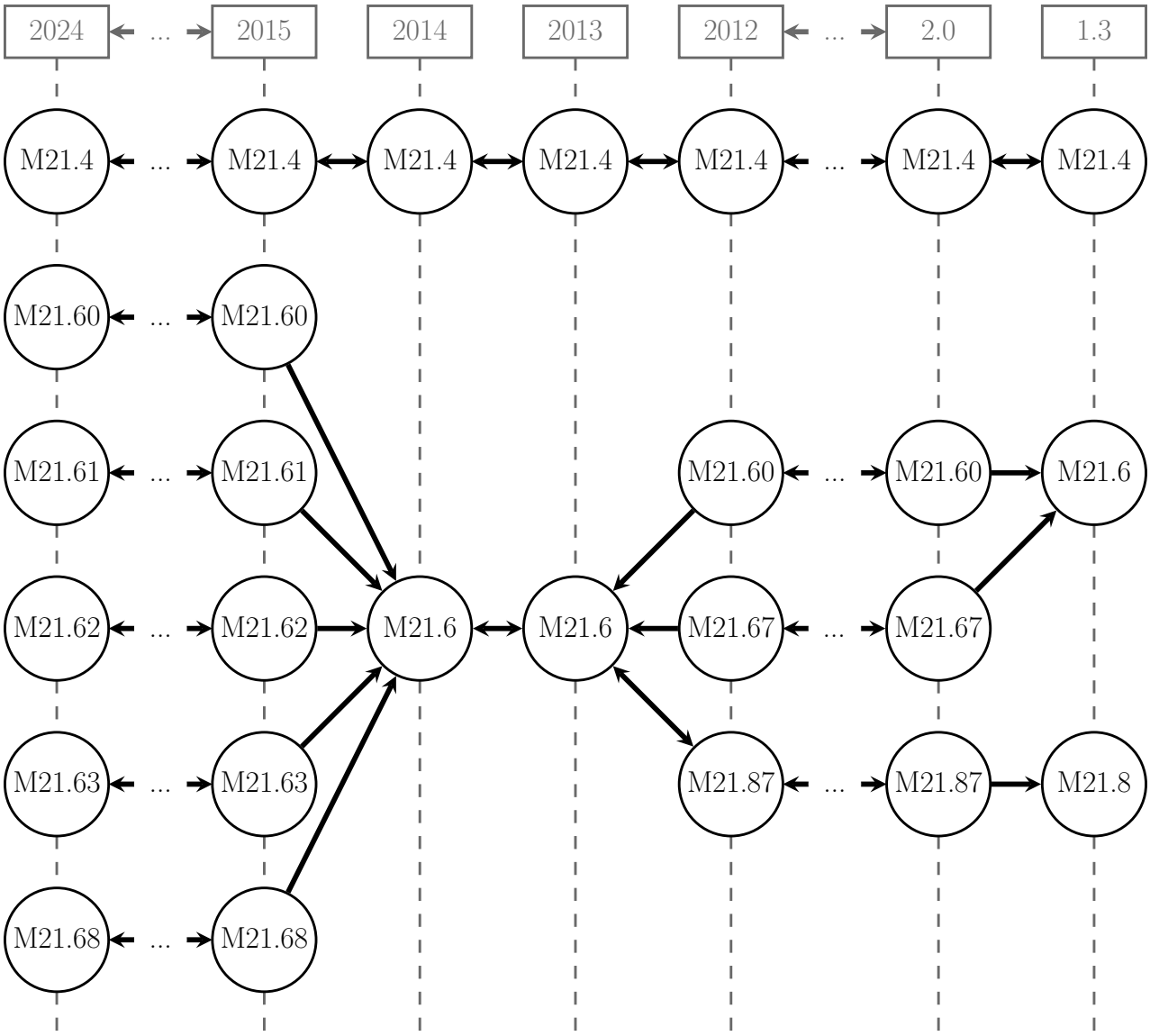


Abbildung 2: Graphische Repräsentation der Umsteiger ausgehend von den Kodes M21.4 und M21.6 der ICD-10-GM Version 2014.

2 Transitive Hülle

Wenn statt der graphischen Repräsentation oben:

1. Die nicht relevanten Umsteiger ausgeschlossen werden; also wie besprochen die Umsteiger-Einträge ohne Veränderung von Kodes.
2. Die Kanten im Graph die Suchrichtung anzeigen statt der automatischen Überleitbarkeit.

Dann ergibt sich ein gerichteter Graph. Das heißt ein Graph, der nur einfach gerichtete Kanten enthält und in dem jeder Knoten mit mindestens einem anderen Knoten über eine gerichtete Kante –auch Bogen genannt– verbunden ist. Die Suche nach über Umsteiger erreichbaren Kodes entspricht daher der Bestimmung der transitiven Hülle in der Graphentheorie.

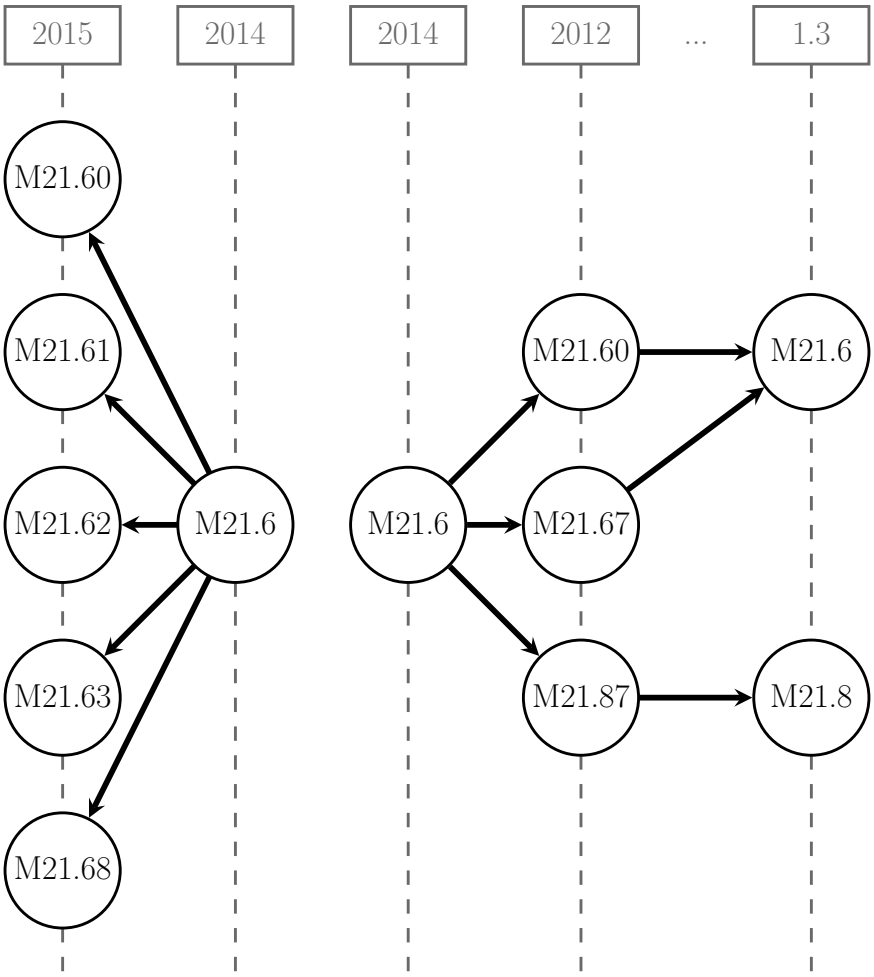


Abbildung 3: Von M21.6, Version 2014 erreichbare Kodes als gerichteter Graph.

Die transitive Hülle wird in (Gross et al., 2013, Seite 172) wie folgt definiert:

> **D26:** The ***transitive closure*** R^* of a binary relation R is the relation R^* defined by $(x, y) \in R^*$ if and only if there exists a sequence $x = v_0, v_1, v_2, \ldots, v_k = y$ such that $k \geq 1$ and $(v_i, v_{i+1}) \in R$, for $i = 0, 1, \ldots, k-1$. Equivalently, the transitive closure R^* of the relation R is the smallest transitive relation that contains R.
>
> **D27:** Let G be the digraph representing a relation R. Then the digraph G^* representing the transitive closure R^* of R is called the ***transitive closure of the digraph*** G. Thus, an arc (x, y), $x \neq y$, is in the transitive closure G^* if and only if there is a directed x-y path in G. Similarly, there is a self-loop in digraph D^* at vertex x if and only if there is a directed cycle in digraph G that contains x.

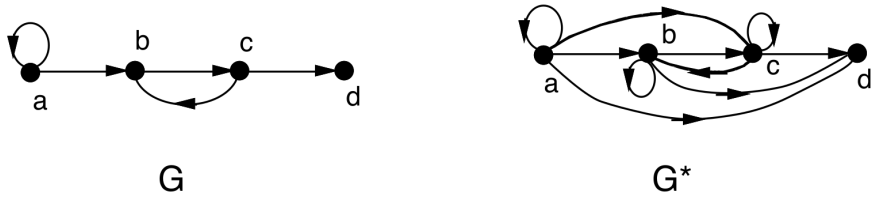
Oder kurz zusammengefasst: die transitive Hülle eines gerichteten Graphens G ist wiederum ein gerichteter Graph G^*, der von jedem Knoten einen Bogen zu allen von diesem Knoten aus in G erreichbaren Nachbarn besitzt.

In (Gross et al., 2013, Seite 172) ist hierzu folgendes Beispiel dargestellt:

E7: Suppose a relation R on the set $S = \{a, b, c, d\}$ is given by

$$\{(a, a), (a, b), (b, c), (c, b), (c, d)\}$$

Then the digraph G representing the relation R and the transitive closure G^* are as shown in Figure 3.1.7.



Es gibt mehrere Verfahren, um die transitive Hülle zu bestimmen. Es werden nun zwei Algorithmen vorgestellt, die anhand ihrer primären Suchrichtung benannt sind. Wie in den Graphen dargestellt, läuft die *horizontale* Suche primär über die Versionen –mit jeweils einzelnen Kodes– und die *vertikale* Suche primär über die Kodes –es werden zuerst alle Kodes einer Version gelesen.

2.1 Horizontale Suche

Ein intuitives Verfahren die transitive Hülle zu bestimmen wird in (Jakobsson, 1991, Seite 200) so beschrieben:

> **2.2 Search**
>
> A second approach for computing the transitive closure is to search the graph n times, each time starting from a different node, thereby determining what can be reached from that node. In this approach, we completely disregard any parts of the TC-relation that have already been computed for other starting nodes. Instead, we search the entire part of the graph that is reachable from the starting node by following every edge we can get to.

Ähnlich wie vorherigen Abschnitt für den Kode M24.6 dargestellt, werden also einfach ausgehend von einem Knoten so viele Suchen gestartet bis keine benachbarten Knoten mehr gefunden werden – oder im Anwendungsfall der Umsteiger bis alle Versionen verarbeitet wurden.

Ergebnisse für andere Knoten, beziehungsweise Kodes, werden ignoriert. Das hieße im Beispiel oben, dass eine rückwärts chronologische Suche nach Umsteigern von M21.60 und M21.61 der ICD-10-GM Version ab dem Vorgänger M21.6 der Version 2014 zweimal exakt gleich ablaufen würde.

Der Pseudocode für diese Vorgehensweise:

searchHorizontal

Funktionsparameter:

- `$system`, `$version`, `$code`
  Wie in Funktion 1.1 readData.

Lokale Variable:

- `$data` Rückgabewert, initial: leer.
  Eine assoziative Datenstruktur mit Key `fwd` ⇒ Umsteiger, die chronologisch vorwärts von dem Suchkode erreichbar sind und `rev` ⇒ chronologisch rückwärts.

$data['fwd'] = searchHorizontalRecursion ($system, $version, $code, true)
$data['rev'] = searchHorizontalRecursion ($system, $version, $code, false)
RETURN $data

Ausgehend von einem gegebenen Kode einer Version eines Kodiersystems wird chronologisch vorwärts und rückwärts eine rekursive Suche über alle Versionen gestartet.

searchHorizontalRecursion

Funktionsparameter:

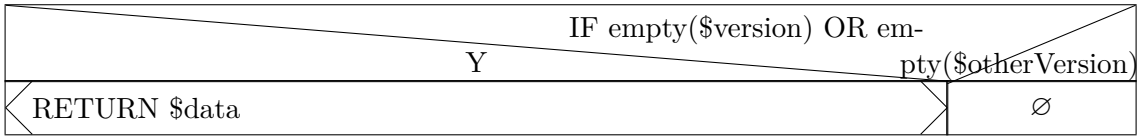
- `$system`, `$version`, `$code`
  Wie searchHorizontal.
- `$chronological`
  Bestimmt die Suchrichtung. TRUE: chronologisch vorwärts und FALSE: rückwärts.

Lokale Variablen:

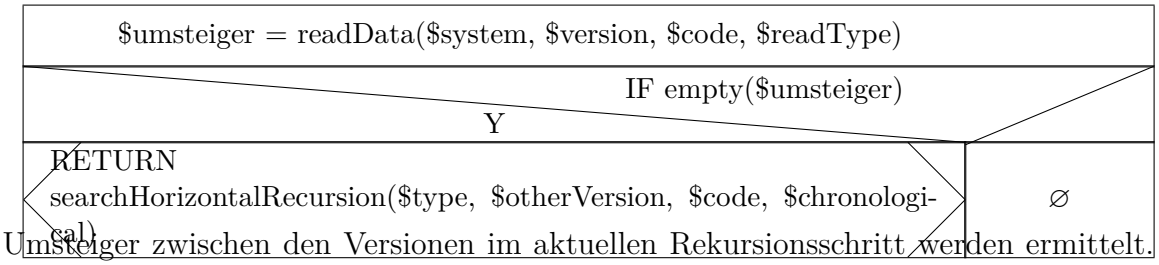
- `$data` Rückgabewert, initial: leer.
  Eine mehrdimensionale, assoziative Datenstruktur, die die Ergebnisse der rekursiven Suche enthält. Gefundene Umsteiger werden als Liste mit dem Key `umsteiger` gespeichert; zusätzlich zu der Listen auch die Versionen vorher und nachher. Jeder einzelne Umsteiger-Eintrag enthält wiederum die Informationen zur automatischen Überleitbarkeit, sowie die Titel der Kodes vorher und nachher. Falls ein Umsteiger weitere Umsteiger hat, werden diese unter dem einem Key `recursion` abgelegt und die Suche fortgesetzt. Das Beispielergebnis für M21.4 befindet sich im Anhang A.
- `$otherVersion`
  Die Version, zu der die Umsteiger ermittelt werden.
- `$otherCode`
  Die Kode, zu dem der aktuelle Kode übergeleitet wird.
- `$readType`
  Auf welche Art die Daten gelesen werden; siehe `readData`.
- `$umsteiger`, `$entry`
  Liste der Umsteiger, Schleifenvariable: Umsteiger-Eintrag.
- `$recursion`
  Ergebnis des rekursiven Funktionsaufrufs bei gefundenen Umsteigern.

IF $chronological	
Y	N
$otherVersion = nextNewerVersion($system, $version)	$otherVersion = nextOlderVersion($system, $version)
$readType = 'umsteiger_join_alt'	$readType = 'umsteiger_join'
$otherCode = 'new'	$otherCode = 'old'

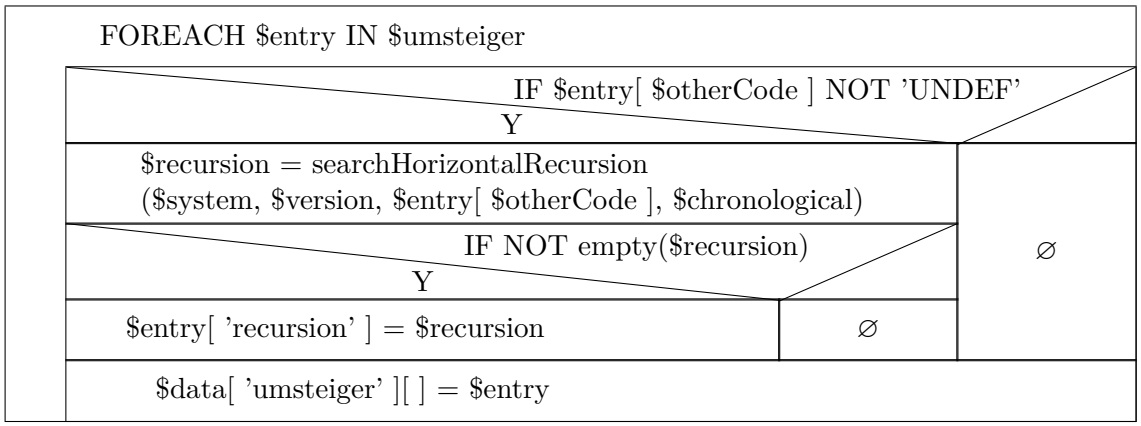
Lokale Variablen werden anhand der Suchrichtung unterschiedlich belegt.



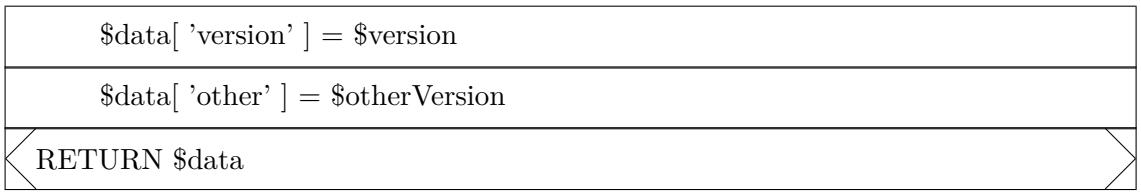
Falls es in der Suchrichtung keine weitere Version mehr gibt, endet die Rekursion.



Die Umsteiger zwischen den Versionen im aktuellen Rekursionsschritt werden ermittelt. Falls es keine gibt –also keine, die eine Veränderung ausdrücken– dann wird die Rekursion mit der nächsten Version fortgesetzt.



Für jeden Umsteiger wird die Rekursion mit dem veränderten Kode fortgesetzt. Allerdings nur falls dieser nicht UNDEF ist, weil es sich dann um einen entfernten oder neu hinzugefügten Kode handelt. Also wenn wie im Beispiel der M21.6 der ICD-10-GM Version 2013 die Umsteiger [M21.60, M21.67, M21.87] in der Version 2012 hat, dann wird die Suche in die rückwärts chronologische Richtung mit diesen drei Kodes fortgesetzt statt mit M21.6. Die Ergebnisse dieser Verzweigung, sofern vorhanden, werden abgespeichert mit dem Key `recursion` zusätzlich zu jedem Umsteiger-Eintrag. Die Liste aller Umsteiger-Einträge, inklusive der UNDEF-Umsteiger, wird in das Ergebnis mit dem Key `umsteiger` aufgenommen.



Falls es Umsteiger für einen Kode gibt, werden zusätzlich die Versionen vorher/nachher gespeichert. Dann endet auch hier die Rekursion.

Appendix A enthält das Ergebnis im JSON-Format für den Beispielaufruf der horizontale Suche für M21.6, Version 2014 entsprechend der Abbildung 3.

2.2 Vertikale Suche

Wie bereits erwähnt startet der Algorithmus der horizontalen Suche für jeden Kode bei Null. Gerade wenn die Umsteiger für alle Kodes einer Version – beziehungsweise eines Kodiersystems– gefunden werden sollen, ist diese Vorgehensweise nicht effizient.

Besonders für gerichtete Graphen ist allerdings Purdoms Algorithmus gut geeignet, um die transitive Hülle zu bestimmen. Folgende kurze Beschreibung des Algorithmus basiert auf Purdom (1970) und (Dar, 1993, Seite 77):

1. Ausgehend von einem Graphen G: Bestimme die stark zusammenhängenden Komponenten (SCC) in G und vereinige diese in jeweils einen einzelnen Knoten. Das ergibt einen zyklenfreien, verdichteten Graphen G_c.
2. Sortiere G_c topologisch.
3. Ermittele die transitive Hülle von G_c über dessen Adjazenzlisten in rückwärts topologischer Reihenfolge.
4. Bestimme die transitive Hülle des ursprünglichen Graphens G über die Nachbarschaftslisten der stark zusammenhängenen Komponenten in G_c: ein Knoten y ist ein erreichbarer Nachbar von Knoten x, falls $SCC(y) = SCC(x)$ oder falls $SCC(y)$ ein erreichbarer Nachbar von $SCC(x)$ ist.

Die Vereinigung der stark zusammenhängenden Komponenten aus Schritt eins wird in Purdom (1970) wie folgt dargestellt:

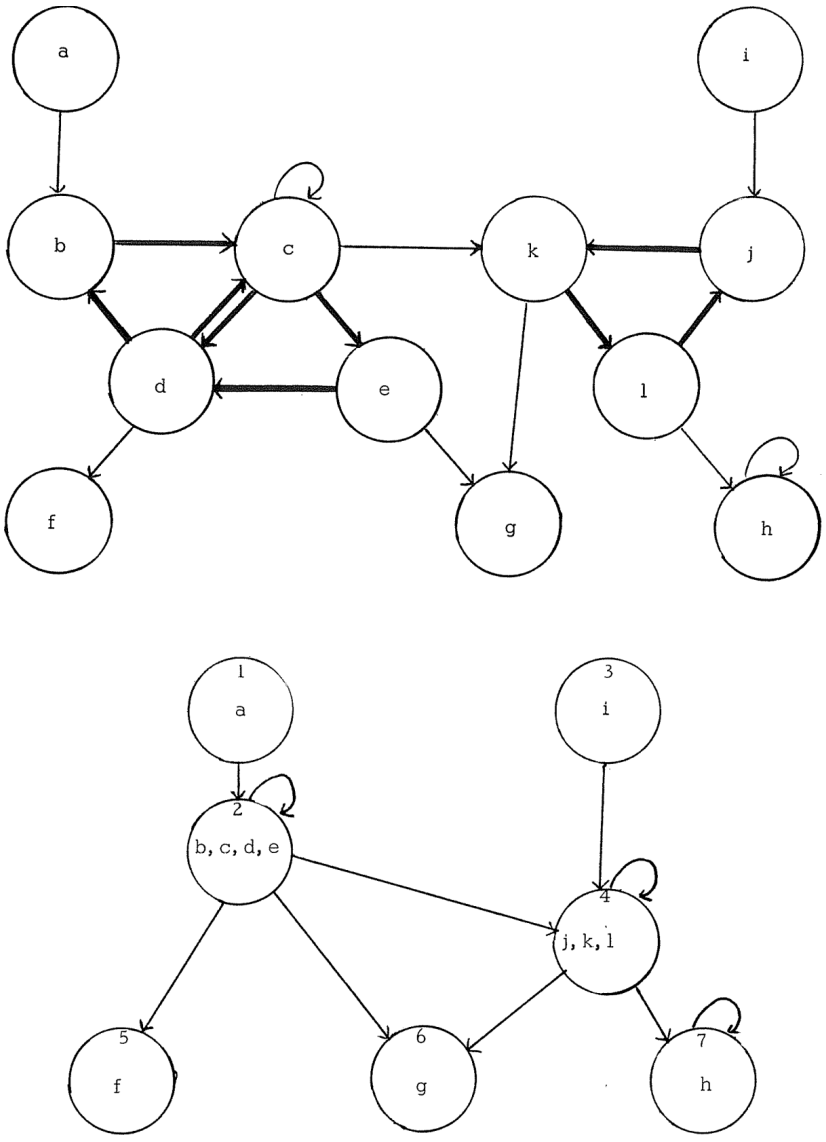


Abbildung 4: Beispielgraph G und Vereinigung der stark zusammenhängenden Komponenten aus (Purdom, 1970, Seite 78).

Die hier vorgestellte vertikale Suche nach Umsteigern entspricht nicht ganz Purdoms Algorithmus.

Zum Beispiel ist die topologische Sortierung aus Schritt zwei in der Anwendung der Suche nach Umsteigern nicht notwendig, weil die Versionen immer in einer bestimmte Richtung verglichen werden und die Kodes sich nur zwischen zwei Versionen ändern können. Die Daten sind also schon implizit topologisch sortiert.

Aber die vertikale Suche verwendet wesentlich zwei Besonderheiten von Purdoms Algorithmus, die in (Dar, 1993, Seite 76f) hervorgehoben werden:

> **5.3.1 The Purdom Algorithm**
>
> Purdom, in [Pur70], made two key observations:
>
> 1. During the computation of transitive closure of a directed acyclic graph, if node $s \prec_t t$, then additions to the successor list of node s cannot affect the successor list of node t. One should therefore compute the successor list of t first and then that of s. By processing nodes in reverse topological order, one need add to a node only the successor lists of its immediate successors since the latter would already have been fully expanded. We call this idea the *immediate successor* optimization [AgJ90].
>
> 2. All nodes within a strongly connected component (SCC) in a graph have identical reachability properties, and the condensation graph obtained by collapsing all the nodes in each strongly connected component into a single node is acyclic.

Konkret bedeuten diese Besonderheiten für die Umsteiger-Suche:

1. Topologisch rückwärts gerichtete Vorgehensweise: So ist zum Beispiel bei der Suche nach den Umsteigern eines ICD-GM-10 Kode der Version 1.3 die Reihenfolge, in der die anderen Versionen abgearbeitet werden: 2024, 2023, 2022 und so weiter. Oder anders ausgedrückt: Wenn die Überleitungen in chronologischer Reihenfolge bestimmt werden sollen, dann läuft die vertikale Suche umgekehrt, also chronologisch rückwärts.
2. Vereinigung der stark zusammenhängenden Komponenten: Nach jedem Schritt über eine Version, wird geprüft, ob ein neu gefundener Umsteiger eine Überleitung in einen Kode enthält, der bereits in den davor abgearbeitet Versionen als Umsteiger vorkommt. Falls ja, dann werden diese Umsteiger vereinigt, das heißt die Zwischenschritte werden zu einem Schritt zusammengefügt.

Einfaches Beispiel für die Vereinigung:

Angenommen ein abstraktes System hat Umsteiger von der Version 1.0 auf 2.0: $A \rightarrow B$ und $M \rightarrow N$, sowie von Version 2.0 auf 3.0: $B \rightarrow C$. Wenn nun in chronologischer Reihenfolge 1.0$\rightarrow$2.0$\rightarrow$3.0 Umsteiger ermittelt werden sollen, erfolgt die Suche sowie Vereinigung gefundener Umsteiger in umgekehrter Reihenfolge. Das heißt wenn im Schritt 3.0$\leftarrow$2.0 der Umsteiger $C \leftarrow B$ gefunden wird, dann wird danach bei der Vereinigung mit Version 1.0 der Umsteiger $B \leftarrow A$ in $C \leftarrow A$ umgewandelt. A in Version 1.0 entspricht C in Version 3.0.

Falls es Verzweigungen gibt, also es mehrere Umsteiger für einen Kode gibt, dann werden entsprechend Listen vereinigt. Ein reales Beispiel dazu folgt später.

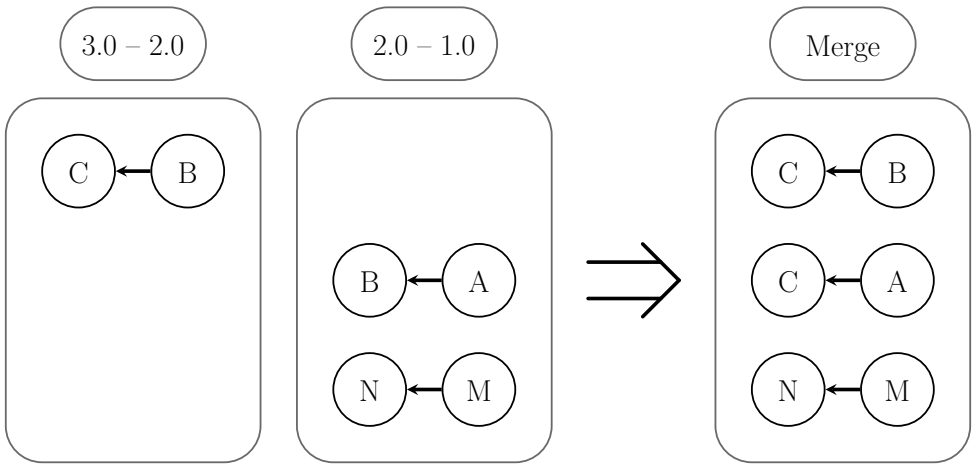


Abbildung 5: Graphische Veranschaulichung der zwei Besonderheiten von Purdoms Algorithmus, die in der vertikalen Suche zur Anwendung kommen.

Die gesamte Algorithmus für die vertikale Suche als Pseudocode:

searchVertical

Funktionsparameter:

- `$system`
  Das Kodiersystem, beispielsweise als Konstanten definiert `'icd10gm'` und `'ops'`.
- `$targetVersion`
  Die Zielversion, auf die von allen anderen Version die Umsteiger über alle Kodes ermittelt werden.
- `$function`
  Eine optionale, anonyme Funktion.

Lokale Variable:

- `$data` Rückgabewert, initial: leer.
  Eine zweidimensionale, assoziative Datenstruktur – wird nur befüllt wenn `$function` nicht gesetzt ist. Die erste Dimension hat als Keys alle Versionen außer der Zielversion. Die zweite Dimension hat als Keys die Kodes der Version und als Value die Liste der zugehörigen Umsteiger, inklusive Vereinigung. Kodes ohne Umsteiger werden nicht aufgenommen.

$data += searchVerticalSubroutine($system, $targetVersion, false, $function)
$data += searchVerticalSubroutine($system, $targetVersion, true, $function)
RETURN $data

Basierend auf der Zielversionen wird in beide Richtungen die vertikale Suche als Unterfunktion gestartet.

`$data +=` bedeutet, dass die assoziativen Datenstrukturen zusammengefasst werden. Wenn ein Key auf beiden Seiten der Addition existiert, wird der von der linken Seite für die Summe übernommen. Das kann allerdings hier bei den Versionen als Keys nicht passieren, weil jede Version nur einmal bearbeitet wird.

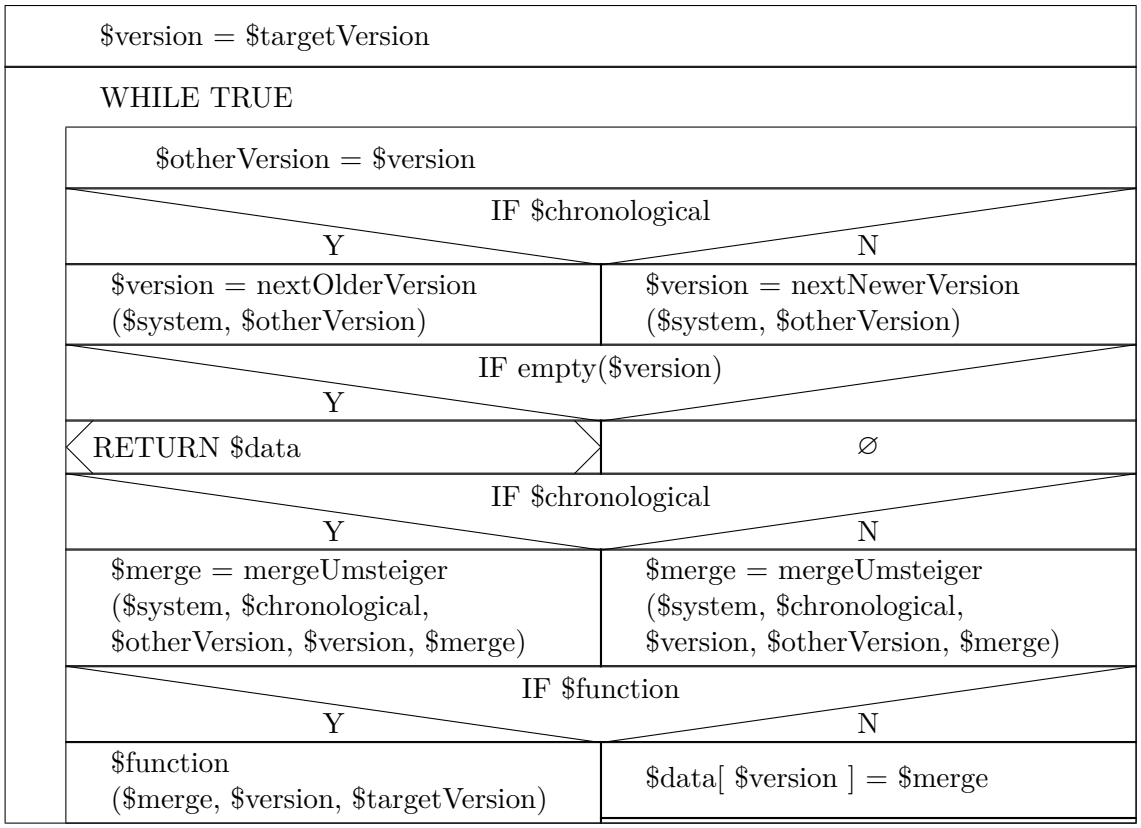
searchVerticalSub

Funktionsparameter:

- `$system`, `$targetVersion`, `$function`
  Wie searchVertical.
- `$chronological`
  Bestimmt die Richtung, in der die Versionen abgearbeitet werden. TRUE: chronologisch vorwärts und FALSE: rückwärts. Die Reihenfolge ist wie erwähnt topologisch rückwärts, das heißt ausgehend von der Zielversion.

Lokale Variablen:

- `$data` Rückgabewert, initial: leer.
  Eine zweidimensionale, assoziative Datenstruktur – wird nur befüllt wenn `$function` nicht gesetzt ist! Die erste Dimension hat als Keys alle Versionen außer der Zielversion. Die zweite Dimension hat als Keys die Kodes aller bisher verarbeiteten Umsteiger und als Values die Listen der Kodes, in die übergeleitet wird, inklusive Vereinigungen wie oben beschrieben. Das heißt im Gegensatz zur horizontalen Suche werden die Umsteiger als Key-Value Paare gespeichert: Kode der aktuellen Version ⇒ Kode/s in die Vergleichsversion. Kodes ohne Umsteiger werden nicht aufgenommen.
- `$merge`
  bla
- `$version`
  bla
- `$otherVersion`
  bla



Q

mergeUmsteiger

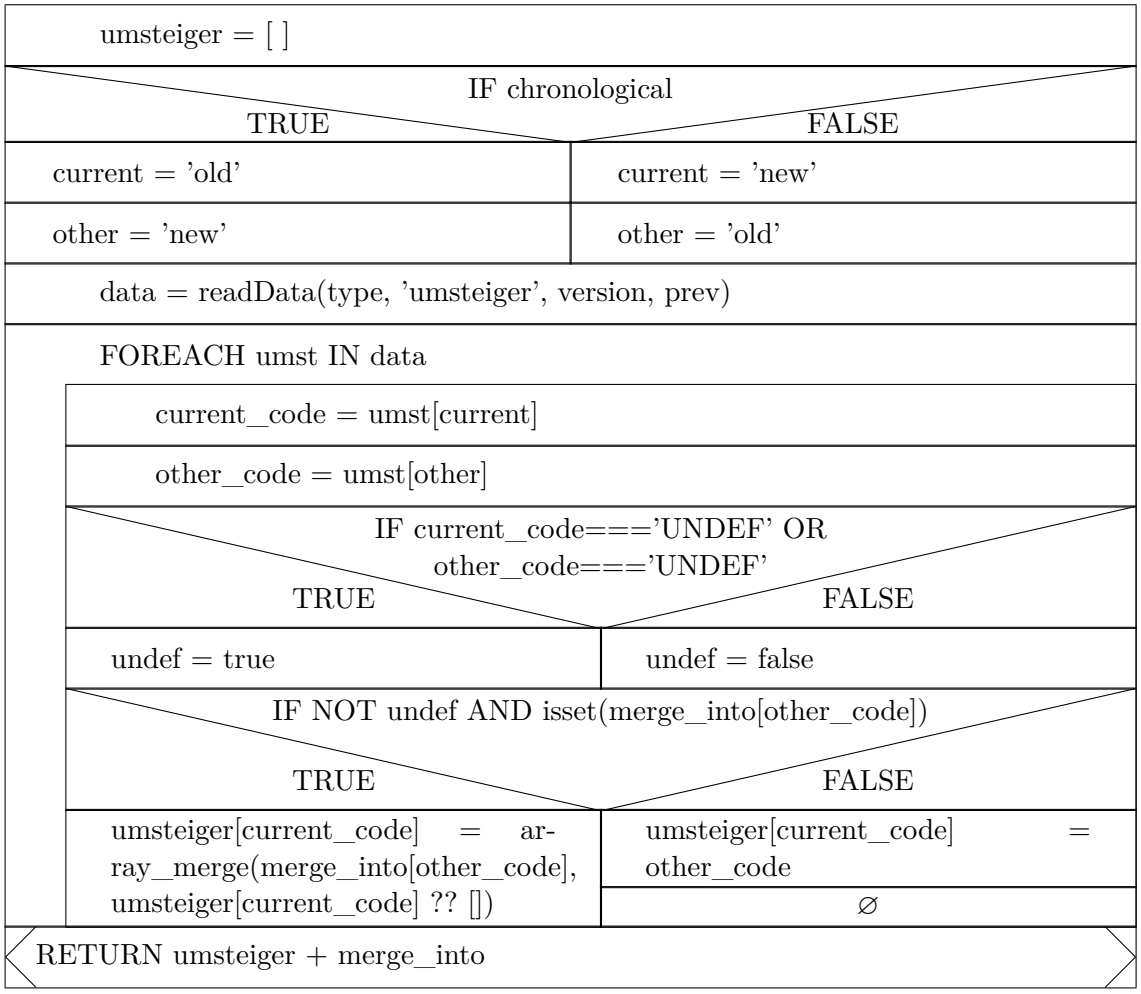
Funktionsparameter:

- `$system`, `$target`, `$function`
  Wie searchVertical.
- `$chronological`
  Bestimmt die Richtung, in der die Versionen abgearbeitet werden. TRUE: chronologisch vorwärts und FALSE: rückwärts. Die Reihenfolge ist wie erwähnt topologisch rückwärts, das heißt ausgehend von der Zielversion.

§TODO: ab hier weitermachen

Lokale Variable:

- `$data` Rückgabewert, initial: leer.
  Eine zweidimensionale, assoziative Datenstruktur – wird nur befüllt wenn `$function` nicht gesetzt ist. Die erste Dimension hat als Keys alle Versionen außer der Zielversion. Die zweite Dimension hat als Keys die Kodes der Version und als Value die Liste der zugehörigen Umsteiger, inklusive Vereinigung. Kodes ohne Umsteiger werden nicht aufgenommen.
- `$merge`
  bla, initial: leer.
- `$version`
  bla, initial = target.



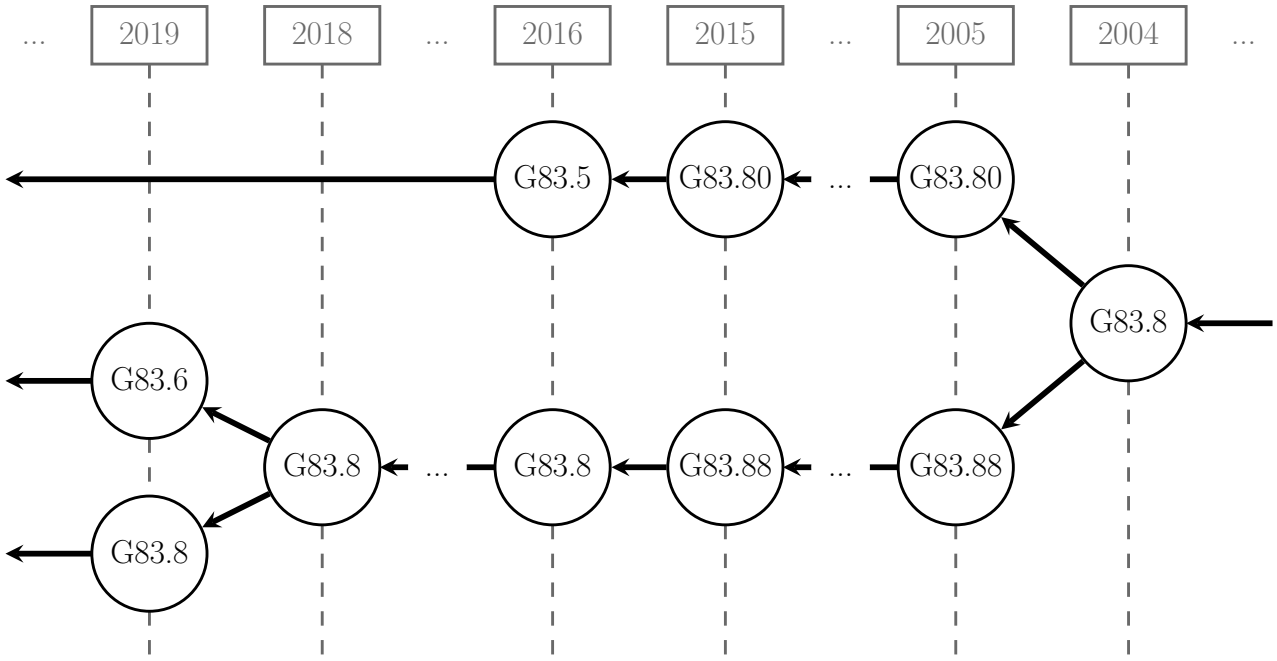


Abbildung 6: caption

test1 test2

3 Vergleich der Suchalgorithmen

Wenn die Daten aus einer Datenbank gelesen werden, dann ist der Menge pro readData Aufruf weniger entscheidend als die Anzahl solcher Aufrufe. Das heißt alle Umsteiger einer Version auf einmal zu lesen ist wesentlich schneller als sie einzeln zu lesen.

Außerdem redundant

In dem Fall ist der

Horizontal:

Schneller bei einem Kode

Mehr Daten hinzufügbar. Vertikale Suche mit Titeln würde schnell mehrere hundert MB groß werden, Arbeitsspeicher.

Vertikal:

Schneller bei allen Kodes

Verwendung für ConceptMap

Verwendung für Bestimmen ob ein Kodes Umsteiger haben für alle Kodes und Versionen

4 Schreiben der ConceptMap

Appendix XML **??**

Literatur

S. Dar, *Augmenting Databases with Generalized Transitive Closure*, ser. Computer sciences technical report. University of Wisconsin–Madison, 1993.

J. Gross, J. Yellen, and P. Zhang, *Handbook of Graph Theory, Second Edition*, ser. Discrete Mathematics and Its Applications. Taylor & Francis, 2013.

H. Jakobsson, "Mixed-approach algorithms for transitive closure," in *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1991, pp. 199–205. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/113413.113431

P. Purdom, "A transitive closure algorithm," *BIT Numerical Mathematics*, vol. 10, no. 1, pp. 76–94, 1970.

A Beispielergebnis der Horizontalen Suche

Ergebnis für den Aufruf `searchHorizontal('icd10gm', '2014', 'M21.6')` im JSON-Format. Um Platz zu sparen enthält nur der erste Umsteiger die zusätzlichen Informationen, die pro Eintrag ermittelt werden, also die automatische Überleitbarkeit sowie die Titel der Kodes. Außerdem sind für die Lesbarkeit die Sonderzeichen nicht kodiert.


```
{
  "fwd": {
    "year": "2014",
    "other": "2015",
    "umsteiger": [
      {
        "old": "M21.6",
        "new": "M21.60",
        "auto": "",
        "auto_r": "A",
        "old_name": "Sonstige erworbene
Deformitäten des Knöchels und des Fußes",
        "new_name": "Erworbener Hohlfuß
[Pes cavus]"
      },
      {
        "old": "M21.6",
        "new": "M21.61",
      },
      {
        "old": "M21.6",
        "new": "M21.62",
      },
      {
        "old": "M21.6",
        "new": "M21.63",
      },
      {
        "old": "M21.6",
        "new": "M21.68",
      }
    ]
  },
  "rev": {
    "year": "2013",
    "other": "2012",
    "umsteiger": [
      {
        "old": "M21.60",
        "new": "M21.6",
        "recursion": {
          "year": "2.0",
          "other": "1.3",
          "umsteiger": [
            {
              "old": "M21.6",
              "new": "M21.60",
            }
          ]
        }
      },
      {
        "old": "M21.67",
        "new": "M21.6",
        "recursion": {
          "year": "2.0",
          "other": "1.3",
          "umsteiger": [
            {
              "old": "M21.6",
              "new": "M21.67",
            }
          ]
        }
      },
      {
        "old": "M21.87",
        "new": "M21.6",
        "recursion": {
          "year": "2.0",
          "other": "1.3",
          "umsteiger": [
            {
              "old": "M21.8",
              "new": "M21.87",
            }
          ]
        }
      }
    ]
  }
}
```