Wo komme ich her, wo gehe ich hin? Ahnenforschung bei SAS Data Sets

Jörg Sahlmann
iOMEDICO AG
Ellen-Gottlieb-Straße 19
79106 Freiburg
joerg.sahlmann@iomedico.com

Zusammenfassung

Die Vorbereitung von Daten für Visualisierungen und statistische Analysen erfordert fast immer mehr oder weniger aufwändige Aufbereitung der Daten mit Merge- und Set-Befehlen in vielen Data Steps. Hier kann man gelegentlich die Übersicht verlieren, welcher Data Step welche Funktionen hat und wo sein Ergebnis weiterverarbeitet wird.
In dieser Arbeit wird die Abfolge von Data Steps als azyklischer gerichteter Graph (DAG, directed acyclic graph) dargestellt, so dass sich Abhängigkeiten leicht visualisieren lassen und verwaiste Data Sets, die nur erzeugt, aber nicht genutzt werden, leicht erkennen lassen.
Betrachtet man die Data Sets als Knoten des Graphen, kann man über Attribute Eigenschaften verwalten wie zum Beispiel die Sortierreihenfolge, so dass sich redundante Sortiervorgänge schneller entfernen lassen.
Der Graph und die Attribute können zur Dokumentation der Herleitung von Analysedatensätzen benutzt werden.
Diese Betrachtungsweise nutzt jeweils die aktuelle Version eines SAS Data Sets. Eine mögliche Historie, wie sie innerhalb eines Data Sets mit den Generation Data Sets dokumentiert wird, wird nicht berücksichtigt.

Schlüsselwörter: Graphen, SAS Programme, Historie

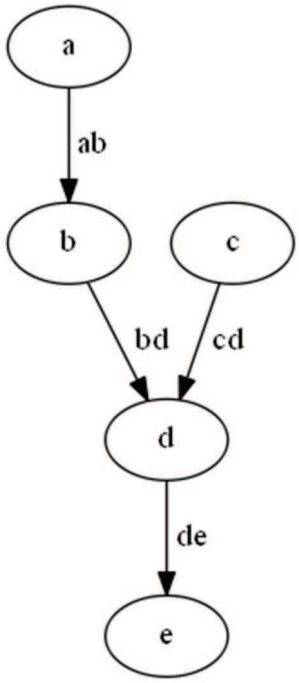
1 Grundlagen von Graphen

Die Graphentheorie ist ein Teilgebiet der Mathematik, das sich mit den Eigenschaften von Graphen beschäftigt.

1.1 Knoten und Kanten

Ein Graph ist dabei eine Menge von Knoten (Ecken) und eine Menge von Verbindungen (Kanten, Ecken) zwischen diesen Knoten.

Der Graph rechts hat fünf Knoten (a bis e) und vier Verbindungen (ab, bd, cd, de).
Die Verbindungen dieses Graphens haben eine Richtung, die durch den Pfeil angedeutet wird.
Graphen dieser Form nennt man gerichtete Graphen (directed graph). Da dieser Graph keinen Zyklus hat, nennt man ihn auch gerichteten azyklischen Graphen (directed acyclic graph, DAG).



1.2 Formale Darstellung

Ein (gerichteter) Graph ist ein Paar G = (V, E), hierbei ist V eine endliche Menge von Knoten und E ist die Menge der Kanten, die in V × V (eine Relation auf V) enthalten ist.

Für den Graphen in 1.1 ist V = {a, b, c, d, e} und E = {(a, b), (b, d), (c, d), (d, e)}.

Für den Knoten a und die Verbindung ab gilt, dass a der Vorgänger von b und b der Nachfolger von a ist.

Der Ausgangsgrad eines Knotens ist die Zahl der Nachfolger. Der Knoten d hat den Ausgangsgrad 1. Der Eingangsgrad eines Knotens ist die Zahl der Vorgänger. Der Knoten d hat den Eingangsgrad 2.

Ein Knoten heißt isoliert, wenn er den Eingangsgrad 0 und den Ausgangsgrad 0 hat.

Pfad p = (a, b) (b, d) (d, e) heißt Pfad der Länge 3 mit dem Anfangsknoten a und dem Endknoten e. Wenn der Endknoten gleich dem Anfangsknoten ist, ist der Pfad geschlossen und wird als Zyklus bezeichnet.

Wenn der Pfad keine Kante mehrfach durchläuft, heißt er einfach. Wenn er keinen Knoten mehrfach durchläuft, heißt er Weg. Er heißt Kreis, wenn er ein Weg und geschlossen ist. Weitere Details finden sich in [2] und [3].

1.3 Algorithmen

Es gibt einige Standardalgorithmen, die grundlegende Aufgaben für Graphen berechnen.

- Erreichbarkeit
- Test auf Zyklen

- Minimaler Spannbaum
- Kürzeste Wege

2 Übertragung der Konzepte auf SAS-Programme

2.1 Ableitung eines Datensatzes aus einem anderen Datensatz

Jeder Datensatz entspricht einem Knoten.
Jede Verbindung entspricht einer Ableitung eines Datensatzes aus einem anderen. In den beiden folgenden Beispielen (Tabelle 1) wird einmal ein neuer Datensatz über einen Datenschritt und einmal als Ausgabe einer Prozedur erzeugt.

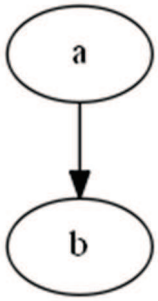
Tabelle 1: Musterableitung 1 und 2

```
* Muster 1;

data b;
  set a;
  …;
run;
```

```
* Muster 2;

proc means data = a;
  var v1;
  output out = b …;
run;
```



2.2 Ableitung eines Datensatzes aus mehreren anderen Datensätzen

Auch hier entspricht jeder Datensatz einem Knoten. Wenn ein Datensatz von mehreren Datensätzen abhängt, wird von jedem der unabhängigen Datensätzen eine Verbindung zu dem abhängigen Datensatz gezogen (Tabelle 2).

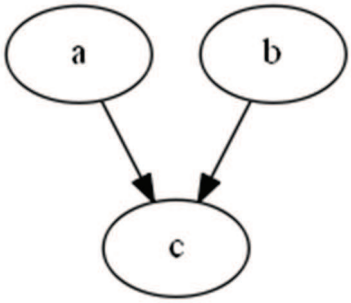
Tabelle 2: Musterableitung 3 und 4

```
* Muster 3;

data c;
  set a b;
  …;
run;
```

```
* Muster 4;

data c;
  merge a b;
  by v1;
run;
```



2.3 Ableitung eines Analysedatensatzes ADSL aus den SDTMs

Wenn man die oben angeführten Muster jetzt auf ein studienspezifisches Programm zur Erzeugung eines Analysedatensatzes ADSL anwendet, der nur auf SDTM Datensätzen aufgebaut ist, erhält man den folgenden Graph (Abbildung 1).

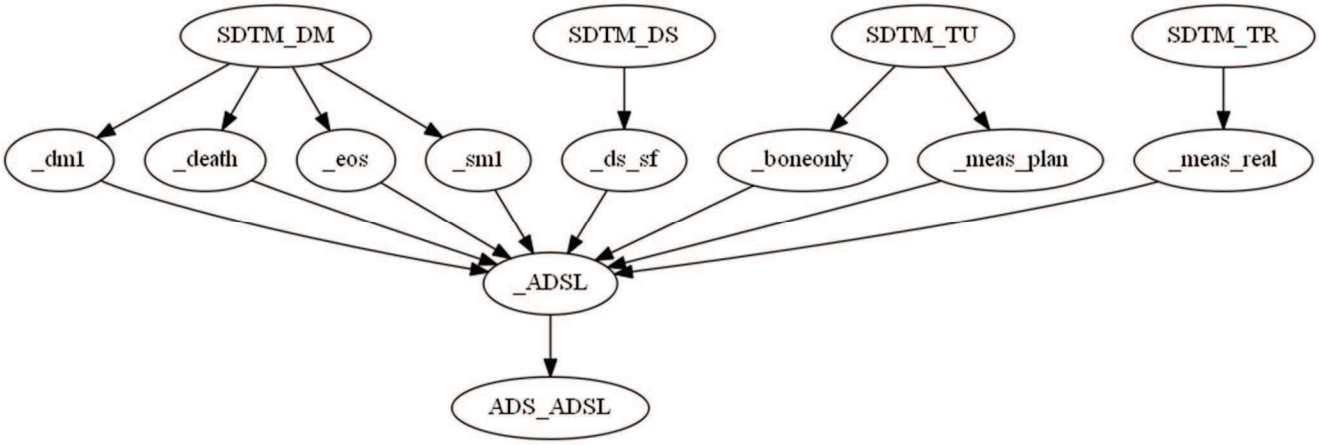


Abbildung 1: Ableitungsgraph ADSL aus SDTMs

Aus vier verschiedenen SDTM Domänen (DM, DS, TU, TR) wird der ADSL zusammengestellt. Die temporären Datensätze sind als Zwischenschicht gut erkennbar.

2.4 Ergänzung des Ableitungsgraphen durch Attribute

Der Übersichtlichkeit wegen werden im folgenden Text Teilgraphen des o.g. Graphen benutzt.
Wenn der letzte Sortierungsschritt als Label an die Kante geschrieben wird, sieht man, dass alle Datensätze für den Match-Merge korrekt sortiert sind (Abbildung 2).

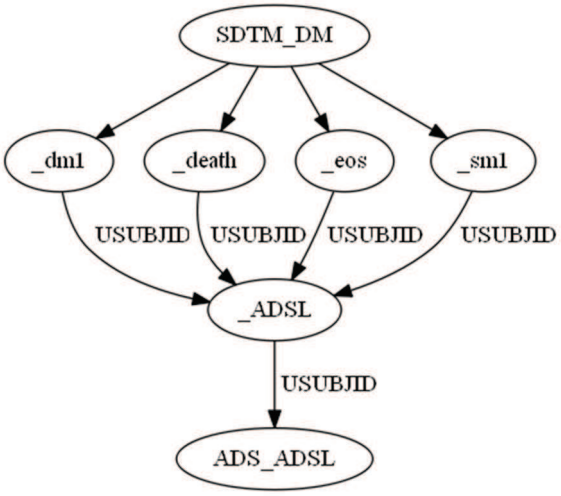


Abbildung 2: Ableitungsgraph (Teildarstellung) mit Sortierung als Label

3 Algorithmen und Anwendung

Es gibt in der Graphentheorie viele Standardalgorithmen, die sich dann auf Fragestellungen in der Programmierung übertragen lassen.

Bevor ich einen Algorithmus auf einen Graphen anwenden kann, muss ich den Graph zunächst in eine bearbeitbare Datenstruktur überführen. Vom Grundsatz gib es drei übliche Datenstrukturen:

- Edge List Structure
- Adjacency List Structure
- Adjacency Matrix Structure

Auf Details kann an dieser Stelle nicht eingegangen werden. Wesentliche Unterschiede ergeben sich im Speicherplatzbedarf und im Laufzeitverhalten. Eine gute Einführung findet sich bei Goodrich et al. [1].

3.1 Brauche ich alle erzeugten Datensätze für den Zieldatensatz?

Zur Beantwortung dieser Frage muss ich sehen, ob es einen Weg von jeden Knoten (Datensatz) zum Zielknoten (zum zu erzeugenden Datensatz) gibt, ob also der Zielknoten von jedem Knoten erreichbar ist (Frage der Erreichbarkeit, Reachability).

Hierzu wird dann der Algorithmus BFS (Breadth-first search) auf die oben ausgewählte Datenstruktur angewendet.

- Nimm die Liste aller n verfügbaren Knoten.
- Für jeden Knoten i ∈ {1 .. n} führe die folgenden Schritte aus:
  - Setze den Knoten i als Startknoten.
  - Addiere diesen Knoten zur Warteschlange.
  - Setze das Attribut „Erreichbar“ für diesen Knoten i auf False.
  - Arbeite, bis die Warteschlange leer ist oder bis „True“:
    - Nimm den nächsten Knoten von der Warteschlange.
    - Füge diesen Knoten zur Menge der besuchten Knoten hinzu.
    - Wenn dieser Knoten der Zielknoten ist, setze das Attribut „Erreichbar“ auf „True“ und beende diesen Schritt. Wenn er nicht der Zielknoten ist, dann füge die Nachfolger dieses Knotens der Warteschlange hinzu.

Alle Knoten, von denen aus der Zielknoten erreichbar ist, haben das Attribut „Erreichbar“ mit dem Status „True“. Die Knoten (Datensätze), von denen der Zielknoten nicht erreichbar ist, haben entsprechend den Status „False“. Ihre Notwendigkeit wäre in diesem Skript zu überdenken. Sie können ohne Bedenken weggelassen werden.

Der Algorithmus DFS (Depth-first search) eignet sich genauso, unterscheidet sich nur durch die Besuchsreihenfolge der Knoten.

3.2 Welches sind meine Ausgangsdatensätze?

Diese Frage wird durch die Auflistung aller Datensätze beantwortet, die in meinem Skript keine Kante haben, die zu diesen Datensätzen hinführen.

Eine mögliche Verarbeitungsvorschrift lautet:

- Nimm die Liste aller n verfügbaren Knoten.
- Erstelle eine Menge a aus diesen Knoten.
- Für jeden Knoten i ∈ {1, ... n} führe den folgenden Schritt aus:
  - Wenn der Knoten i mindestens einen Vorgänger hat, entferne diesen Knoten aus der Menge a.
- Gib die Menge a aus. Sie enthält die Knoten, die keinen Vorgänger haben.

3.3 Erzeuge ich zwei oder mehr Zieldatensätze?

Zur Beantwortung dieser Frage muss ich sehen, ob es zwei oder mehr Knoten gibt, die keine Nachfolger haben.

Eine mögliche Verarbeitungsvorschrift lautet:

- Nimm die Liste aller n verfügbaren Knoten.
- Erstelle eine Menge b aus diesen Knoten.
- Für jeden Knoten i ∈ {1, ... n} führe den folgenden Schritt aus:
  - Wenn der Knoten i mindestens einen Nachfolger hat, entferne diesen Knoten aus der Menge b.
- Gib die Menge b aus. Sie enthält die Knoten, die keinen Nachfolger haben.

4 Ausblick

Bisher ist diese Betrachtungsweise sehr theoretisch. Bei dem einfachen Beispiel erschließt sich das Potential noch nicht.

Wenn man jedoch mehrere Dutzend temporäre Datensätze erzeugt und eine programmübergreifende Abhängigkeit berücksichtigen will, kann der Ansatz seine Stärke entfalten.

Voraussetzung hierfür ist die Möglichkeit, die Skripte durchzuparsen und die Graphdefinitionen automatisch zu erzeugten. Hier dran wird gerade gearbeitet.

Literatur

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser: Data Structures and Algorithms in Python. Wiley, 2013.

[2] https://de.wikipedia.org/wiki/Graphentheorie, letzter Zugriff 22.01.2019.

[3] http://www.iti.fh-flensburg.de/lang/algorithmen/grundlagen/graph.htm, letzter Zugriff 22.01.2019.

[4] https://graphviz.org/, letzter Zugriff 22.01.2019

Anhang A: Source Code GraphViz

Die Abbildungen der Graphen lassen sich mit den folgenden Code-Abschnitten generieren. Details siehe [4]

Tabelle 1:

```
digraph tabelle1
 {
     a -> b;
 }
```

Tabelle 2:

```
digraph tabelle2
{
  a -> c;
  b -> c;
}
```

Abbildung 1:

```
digraph ADSL
{
  SDTM_DM -> _dm1;
  SDTM_DM -> _death;
  SDTM_DM -> _eos;
  SDTM_DM -> _sm1;
  SDTM_DS -> _ds_sf;
  SDTM_TU -> _boneonly;
  SDTM_TU -> _meas_plan;
  SDTM_TR -> _meas_real;
  _dm1 -> _ADSL;
  _death -> _ADSL;
  _eos -> _ADSL;
  _sm1 -> _ADSL;
  _ds_sf -> _ADSL;
  _boneonly -> _ADSL;
  _meas_plan -> _ADSL;
  _meas_real -> _ADSL;
  _ADSL -> ADS_ADSL;
}
```

Abbildung 2:

```
digraph ADSL_teil
{
  SDTM_DM -> _dm1;
  SDTM_DM -> _death;
  SDTM_DM -> _eos;
  SDTM_DM -> _sm1;
  _dm1 -> _ADSL [label = " USUBJID"];
  _death -> _ADSL [label = " USUBJID"];
  _eos -> _ADSL [label = " USUBJID"];
```

```
  _sm1 -> _ADSL [label = " USUBJID"];
  _ADSL -> ADS_ADSL [label = " USUBJID"];
}
```

Anhang B: Source Code Simple Graph-Implementation

Einfache Datenstrukturen und Fragestellungen lassen sich mit SAS Base darstellen und bearbeiten. An der kompletten Umsetzung des abstrakten Datentyps für Graphen, wie er in [1] beschrieben wird, wird gearbeitet.

```
* Import of gv file;
proc import datafile=" adsl.gv"
   out=dag dbms=dlm replace;
   delimiter='#';
   getnames=no;
run;

* Simple data structure for edges;
data edges (keep = von nach);
  attrib
    von  length = $20.
    nach length = $20.
  ;
  set dag;
  von = "";
  nach = "";
  var1 =tranwrd(var1, '->', '>');
  var1 =tranwrd(var1, ';', ' ');
  var1 = compress(var1);
  von = scan(var1, 1, '>');
  nach = scan(var1, 2, '>');
  if nach NE "" then output;
run;

proc sort data = edges;
  by von nach;
run;

* simple list of nodes;
data nodes (keep = node);
  set
    edges (rename = (von = node))
    edges (rename = (nach = node))
  ;
run;

proc sort data = nodes nodup;
  by node;
run;
```

```
data nodes;
  attrib
    i    label = "Node number"
    node label = "Node label"
  ;
  set nodes;
  i = _N_;
run;

* looking for nodes without successor;
data no_successor (keep = i node);
  merge
    nodes
    edges (rename = (von = node))
  ;
  by node;
  if nach NE "" then delete;
run;

proc sort data = edges;
  by nach von;
run;

* looking for nodes without predecessor;
data no_predecessor (keep = i node);
  merge
    nodes
    edges (rename = (nach = node))
  ;
  by node;
  if von NE "" then delete;
run;
```