LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Institut für Informatik

A Testbed and Experimental Evaluation for Accessing Interval-Compressed Graph Data

Nicolai Roth

Diplomarbeit

Beginn der Arbeit:	01. Juli 2010
Abgabe der Arbeit:	14. Januar 2011
Betreuer:	Prof. Dr. François Bry
	Simon Brodt

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 4. Januar 2011 .....

Nicolai Roth

Zusammenfassung

Diese Diplomarbeit setzt die Arbeiten von Tim Furche und Simon Brodt fort. Es wird ein Parser implementiert um Graphen aus XML- und RDF-Daten in die von Simon Brodt geschaffenen Datenstrukturen einzulesen. Um Abfragen auf die eingelesenen Daten zu ermöglichen, wird eine eigene, aber an SQL angelehnte, Anfragesprache definiert. Mit Hilfe einer einfachen Konsolenanwendung ist es möglich, das Parsen von Daten und das Durchführen von Anfragen vorzunehmen.

Als Teil der Optimierung der Anfragebearbeitung wird der Edmonds–Algorithmus benötigt, der in dieser Arbeit implementiert und an Beispielen veranschaulicht wird. Dabei wird eine Optimierung von Joins und Selektionen auf Basis eines statischen Kostenmodels vorgenommen. Zusätzlich wird die Anfrageauswertung verbessert, indem die Anordnung der Operatoren für die Auswertung optimiert wird.

Des Weiteren wird der PC–Tree–Algorithmus implementiert, der für die Funktionsweise der in der Arbeit von Simon Brodt eingeführten Datenstrukturen in Verbindung mit Circular–Continuous–Image–Relationen (CCIRs) notwendig ist und bisher nicht zur Verfügung stand. Auch diese Implementierung wird an Beispielen illustriert.

Die Korrektheit der verschiedenen Programmstücke und der implementierten Algorithmen wird anhand einiger Tests überprüft. Diese Tests können auch verwendet werden um den Programmcode besser zu verstehen.

Abstract

This diploma thesis continues the work of Tim Furche and Simon Brodt. A parser for XML and RDF data is implemented to read these formats into the data structures created by Simon Brodt. Moreover, to permit queries, a new query language is defined which is similar to SQL. With the help of a simple console application it possible to parse the data and to enter queries.

In order to optimize query processing the Edmonds–Algorithm is needed, which is implemented in this work and which we illustrate by examples. An optimization of joins and selections based on a static cost model is made. In addition, the query evaluation is imporved by optimizing the arrangement of the operators for the evaluation.

Furthermore, the PC–Tree–Algorithm is implemented, which is necessary for the functioning of the data structures created by Simon Brodt in combination with Circular-Continuous-Image-Relations (CCIRS) and which was previously unavailable. This implementation is illustrated with examples, too.

To check that the various program segments and the algorithms used are correct, additional tests are created, an extra advantage of which is that they can also be used to better understand the code.

Inhaltsverzeichnis

1. Einleitung **9**

2. Einlesen von Graphstrukturen **10**
2.1. Das Zusammenspiel mit dem Kern . . . 10
2.2. Neue Hilfsklassen . . . 13
2.2.1. Hilfsklasse *TwoWayDictionary* . . . 13
2.2.2. Hilfsklasse *GraphDescriptor* . . . 13
2.2.3. Hilfsklasse *ParserSettings* . . . 14
2.3. Der XML-Parser . . . 15
2.3.1. Die Hilfsklasse *XMLParserSettings* . . . 17
2.3.2. Die Klasse *XMLParser* . . . 17
2.4. Der RDF–Parser . . . 19
2.4.1. Die Hilfsklasse *RDFParserSettings* . . . 20
2.4.2. Die Klasse *RDFParser* . . . 21

3. Anfragesprache und Anfrageauswertung **22**
3.1. Eine SQL-nahe Anfragesprache . . . 22
3.2. Syntax und Grammatik der Anfragesprache . . . 23
3.3. Die Klasse *QueryParser* . . . 24
3.4. Die Klasse *Query* . . . 24
3.4.1. Erstellung des Anfragegraphen . . . 25
3.4.2. Erstellung des Anfragegraphen — Beispiel . . . 27
3.4.3. Auswertung des Anfragegraphen . . . 32
3.4.4. Optimierte und nicht optimierte Auswertung des Teilanfragegraphen . . . 35
3.4.5. Auswertung eines Teilanfragegraphen — nicht optimiert . . . 35
3.4.6. Auswertung des Anfragegraphen — Beispiel . . . 36
3.4.7. Optimierung der Anwendungsreihenfolge der Suboperatoren . . . 49
3.4.8. Allgemeiner Operatorbaum . . . 52

4. Die Konsole **53**
4.1. Die Klasse *GraphDescriptorDataBase* . . . 53
4.2. Die Klasse *QueryConsole* . . . 54
4.3. Beispielsitzung . . . 56

5. Die Algorithmen **57**
5.1. Der Edmonds–Algorithmus . . . 57
5.1.1. Einführungsbeispiel . . . 57
5.1.2. Der Algorithmus als Pseudocode . . . 61
5.1.3. Konstruktion des Hilfsgraphen B_i aus G_i . . . 63
5.1.4. Konstruktion von G_{i+1} aus G_i und dem Hilfsgraphen B_i . . . 63
5.1.5. Fälle der Expansion . . . 65
5.1.6. Deterministisches Verhalten . . . 67
5.1.7. Weitere Beispiele . . . 67
5.1.8. Implementierung . . . 68

5.2. Der PC–Tree–Algorithmus . . . 69
5.2.1. CIG–Beispiel . . . 69
5.2.2. Der PC-Tree . . . 71
5.2.3. Konstruktion des PC-Trees . . . 71
5.2.4. Einführungsbeispiel . . . 73
5.2.5. Der Terminal-Pfad . . . 90
5.2.6. Probleme mit dem Terminal-Pfad . . . 91
5.2.7. Weitere Beispiele . . . 95
5.2.8. Implementierung . . . 95

6. Schluss **102**

Literatur **103**

A. Gesamtübersicht über implementierte Klassen / Dateien **104**

B. Weitere Beispiele für die Anwendung des Edmonds–Algorithmus **107**
B.1. oneNodeOnly . . . 107
B.2. example1 . . . 107
B.3. example2 . . . 109
B.4. example3 . . . 111
B.5. example4 . . . 112
B.6. example5 . . . 113
B.7. example6 . . . 114
B.8. example7 . . . 116

C. Weitere Beispiele für die Anwendung des PC–Trees **119**
C.1. example1 Error1 . . . 119
C.2. example2 Error2a . . . 126
C.3. example3 Error2b . . . 136
C.4. example4 . . . 138

1. Einleitung

Die zunehmendes Digitalisierung der Welt bringt eine nicht mehr aufzuhaltende Flut an Daten mit sich. Es handelt sich dabei jedoch nicht lediglich um unabhängige Fakten, sondern oft um Daten, die den Zusammenhang verschiedener Einzelinformationen beschreiben. Ein sehr anschauliches Beispiel sind soziale Netzwerke, wie sie heute in zunehmender Zahl im Internet existieren. Hier findet man nicht Informationen über Personen oder Ereignisse, man erfährt vielmehr in welcher Form diese zueinander in Beziehung stehen. Graphen sind eine Abstraktion solcher Strukturen. Ein Graph ist anschaulich eine Menge von Punkten oder Knoten die über Kanten verbunden sein können. Im Falle des sozialen Netzwerkes wären die Personen, Ereignisse oder Abbildungen die Knoten eines Graphen. Für jede Beziehung, die zwischen zwei solchen Objekten existiert, gibt es eine Kante, die die beiden Objekte verbindet. Jedoch sind Graphdaten nicht nur in so anschaulichen Beispielen wie einem Plan eines U–Bahn–Netzes, in einem Stammbaum oder eben in sozialen Netzwerken zu finden, sondern auch in Bereichen wie der Molekülphysik oder der DNA-Sequenzierung. Damit sind Graphdaten für verschiedenste Bereiche der Wissenschaft und ihrer Datenspeicherung relevant.

Für die unterschiedlichen Anwendungsgebiete und Ausprägungen, die ein Graph oder allgemein eine Relation haben kann, hat Simon Brodt in seiner Diplomarbeit[1] eine Reihe von Speicherstrukturen zur Repräsentation von Graphen (oder allgemeiner Binärerrelationen) implementiert auf die mittels einer einheitlichen Schnittstelle effizient zugegriffen werden kann. Diese Diplomarbeit baut auf diesem „Kern“auf und erweitert ihn um Strukturen, die das Einlesen und vor allem auch Anfragen von Daten ermöglichen sollen. Man bedenke, dass in der reinen Zunahme von Informationen noch kein wirklicher Informationsgewinn steckt. Dieser entsteht erst, wenn es möglich ist, Daten zu speichern und Anfragen sinnvoll durchzuführen. Somit wird sich der erste Teil dieser Arbeit mit dem Einlesen und Speichern von Daten in die vorhandenen Speicherstrukturen beschäftigen. Später wollen wir Anfragen auf die eingelesenen Daten ermöglichen. Letztlich soll sowohl das Einlesen als auch Anfragen in Form einer einfachen Konsolenapplikation möglich sein.

Neben der eigentlichen Implementierung wird auch Wert darauf gelegt die Funktionsweise nicht nur zu kommentieren, sondern die Ideen und Abläufe möglichst detailliert und auf Beispiele bezogen vorzuführen. Dies machen wir auch vor dem Hintergrund, dass eine Weiterentwicklung dieser Implementierung leicht möglich sein soll.

Die Implementierungssprache ist C#, da auch der Kern in dieser Sprache geschrieben ist. C# ist eine objektorientierte und plattformunabhängige Sprache, die aufgrund ihrer hohen Leistung und ihrem direkten Speicherzugriff für den Kern gewählt wurde.

2. Einlesen von Graphstrukturen

Der erste Teil dieser Arbeit beschäftigt sich damit, wie wir Graphen in die vorhandenen Datenstrukturen einlesen können. Als Quelldatenformate wurden XML und RDF gewählt, da diese bedingt durch das semantische Web von großer Bedeutung sind. Um aus XML allgemeine Graphen zu erhalten, soll es zusätzlich möglich sein, ID– und IDREF–Beziehungen anzugeben und auszuwerten. RDF kann im XML– oder N3–Format vorliegen. Bevor wir genauer auf den XML-Parser und RDF-Parser eingehen, ist zuerst ein Blick auf die Klassen und Methoden, die uns der Kern zur Verfügung stellt, und auf die neuen Komponenten sinnvoll.

2.1. Das Zusammenspiel mit dem Kern

Bei der Speicherung von Relationen arbeitet der Kern mit sogenannten Compact Integer Domains. Die Idee dahinter ist, dass sich jede endliche Domain mit d Elementen auf einen Wertebereich $0, \ldots, d-1$ abbilden lässt. Vorteile bestehen unter anderem darin, dass sich Integer als Knotentyp sehr effizient verarbeiten lassen oder die Integer selbst als Index in Arrays oder Matrizen verwendet werden können. Der Kern konzentriert sich vor allem auf die effiziente Speicherung solcher Relationen über Compact Integer Domains. Strukturen, um die Abbildung von Strings auf die Compact Integer Domains durchzuführen, bietet der Kern nicht an. Hierzu werden wir später das *TwoWayDictionary* einführen, das diese Aufgabe übernehmen wird.

Datenstrukturen für das Einlesen von Graphen oder Relationen

Für das Einlesen der Graphen bietet der Kern die *UnorderedAddableAdjacencyArray*–Klasse, die eine Unterklasse der abstrakten Klasse *ABinaryIntRelation* ist. Diese erlaubt es, mit ihren Methoden AddVertex() und AddEdge(int *vertex1*, int *vertex2*) Knoten und Kanten anzulegen. Der folgende Codeabschnitt soll die Verwendung illustrieren:

```
UnorderedAddableAdjacencyArray rel =
         new UnorderedAddableAdjacencyArray();

rel.AddVertex();       // legt ersten Knoten an
rel.AddVertex();       // legt zweiten Knoten an
rel.AddEdge(0, 1);     // legt Kante vom ersten
                       // zum zweiten Knoten an
```

Wahl der richtigen Speicherstruktur

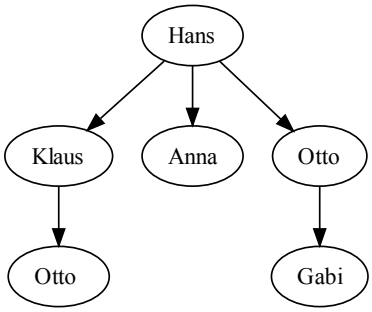
Eine der wichtigsten Methoden des Kerns ist die Encode-Methode der statischen *GRAPH*–Klasse. Diese bekommt als Parameter eine Instanz eines *ABinaryIntRelation*–Objektes, das den Graph repräsentiert, in unserem Fall also eine Instanz der *UnorderedAddableAdjacen-*

cyArray–Klasse. Die Encode-Methode sucht dann nach effizienten Repräsentationen für die Kind–, Nachkommen–, Eltern– und Vorfahrenrelation und stellt diese in einer Instanz der Klasse *AGraphIntRelationSet* zur Verfügung:

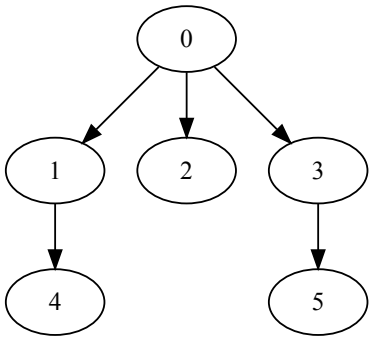
```
AGraphIntRelationSet relSet= GRAPH.Encode(rel);
```

Abbildung der Graphen auf Relationen über Compact Integer Domains

Die Abbildung der Originalknoten mit ihren Attributen auf die Compact Integer Knoten gestalten wir indirekt: Für jedes Attribut des Originalknotens bekommt der Compact Integer Knoten ein Label zugeordnet. In unseren Fällen der XML– bzw. RDF–Daten bekommt jeder Compact Integer Knoten lediglich ein einziges String-Label zugeordnet, das den Knotennamen des Originalknotens beinhaltet. Für die Speicherung und den Zugriff auf die Knotennamen implementieren wir ein Wörterbuch, das jedem String einen eindeutigen Compact Integerwert zuordnet: Das erste Wort, das in das Wörterbuch geschrieben wird, erhält den Integer 0, das zweite den Integer 1 usw.. Zudem soll das Wörterbuch Anfragen in beide Richtungen beantworten können, man soll also zu einem String den Integerwert bekommen können und umgekehrt. Die Implementierung des Wörterbuchs findet sich in der Klasse *TwoWayDictionary*. Hierzu später mehr. Die Beziehung zwischen den Compact Integer Knoten und den Wörterbucheinträgen wird durch eine weitere zweistellige Relation ausgedrückt. Hierzu bietet der Kern die Klasse *VertexLabel1IntRelationSet*. Betrachten wir das Ganze an einem einfachen Beispiel. Gegeben sei folgender Graph:



Die beiden „Otto“-Knoten seien verschiedene Knoten, die lediglich über die gleiche Beschriftung verfügen. Wir machen einen Breitendurchlauf. Dabei werden die Knoten in folgender Reihenfolge besucht:



Unser Graph wird somit durch drei Objekte repräsentiert:

UnorderedAddableAdjacencyArray Objekt	*VertexLabel1IntRelationSet* Objekt	*TwoWayDictionary* Objekt
Knoten: {0,. . . ,5} Kanten: {(0,1), (0,2), (0,3), (1,4), (3,5)}	{(0,0), (1,1), (2,2), (3,3), (4,3), (5,4)}	{(0,"Hans") (1,"Klaus"), (2,"Anna"), (3,"Otto"), (4,"Gabi")}

Durch den Aufruf der GRAPH.Encode-Methode ersetzen wir das *UnorderedAddableAdjacencyArray*–Objekt durch ein *AGraphIntRelationSet*–Objekt, das dann effiziente Repräsentationen für die Kind-, Nachkommen-, Eltern- und Vorfahrenrelation bereitstellt:

AGraphIntRelationSet Objekt	*VertexLabel1IntRelationSet* Objekt	*TwoWayDictionary* Objekt
Knoten: {0,. . . ,5} Kindrelation: {(0,1),(0,2),(0,3),(1,4),(3,5)} Nachkommenrelation: {(0,1),(0,2),(0,3),(1,4), (3,5),(0,4),(0,5)} Elternrelation: {(1,0),(2,0),(3,0),(4,1),(5,3)} Vorfahrenrelation: {(1,0),(2,0),(3,0),(4,1),(5,3), (4,0),(5,0)}	{(0,0), (1,1), (2,2), (3,3), (4,3), (5,4)}	{(0,"Hans") (1,"Klaus"), (2,"Anna"), (3,"Otto"), (4,"Gabi")}

Dabei gilt:

- (x, y) stehen in der Kindrelation, wenn y ein Kind von x ist.
- (x, y) stehen in der Nachkommenrelation, wenn y ein Nachkomme von x ist.
- (x, y) stehen in der Elternrelation, wenn x ein Kind von y ist.
- (x, y) stehen in der Vorfahrenrelation, wenn x ein Nachkomme von y ist.

Nehmen wir zu diesen drei Objekten noch einen String hinzu, um dem Graph einen Namen zu geben, dann haben wir insgesamt vier Objekte, die unseren Graph repräsentieren. Die Repräsentation des Graphen übernimmt die Klasse *GraphDescriptor*. Die eben genannten vier

Objekte stellen die wesentlichen Attribute einer jeden Instanz der Klasse *GraphDescriptor* dar.

2.2. Neue Hilfsklassen

2.2.1. Hilfsklasse *TwoWayDictionary*

Die Klasse *TwoWayDictionary* wird für die Abbildung von Strings auf die Compact Integer Domains benötigt, und stellt ein Wörterbuch für die Speicherung und den Zugriff auf Knotennamen zur Verfügung. Jeder Knotennamen ist ein String. Diesem wird ein eindeutiger Compact Integerwert zuordnet: Das erste Wort, das in das Wörterbuch geschrieben wird, erhält den Integer 0, das zweite den Integer 1 usw.. Damit das Wörterbuch Anfragen in beide Richtungen beantworten kann, besteht es aus einem *SArrayStack*<string> für die „Integer-nach-String“ Richtung und aus einem herkömmlichen C# *Dictionary*<string, int> für die Richtung „ String-nach-Integer“. *SArrayStack* ist eine Klasse des Kerns und implementiert ein Array, das allerdings seine Größe nach Bedarf dynamisch anpasst. Da wir das Wörterbuch vor dem Parsen anlegen müssen, können wir noch nicht wissen, wie groß das Wörterbuch und damit das Array sein muss. Daher ist eine dynamische Größenanpassung notwendig. Zudem verfügt die *SArrayStack*–Klasse über eine push-Methode, so dass sich neue Wörterbucheinträge einfach hinzufügen lassen. Im Nachfolgenden sei der Integerwert als *Key* bezeichnet, der String als *Value*. Die Methode zum Befüllen des Wörterbuchs heißt AddAndGetKey, bekommt als Parameter den einzufügenden String (*Value*) und gibt den dazugehörigen Compact Integer zurück. Die Methode überprüft dabei, ob der einzufügende String bereits im Wörterbuch enthalten ist. Ist dies der Fall, wird aus dem *Dictionary*<string, int> der Integerwert ermittelt. Ist der einzufügende String noch nicht im Wörterbuch enthalten, wird dem String der nächste Compact Integer Wert zugeordnet und dieser auch zurückgegeben. Des Weiteren verfügt das Wörterbuch über die Methoden ContainsKey, ContainsValue, getValue und getKey, die von ihrem Namen her selbsterklärend sein sollten.

2.2.2. Hilfsklasse *GraphDescriptor*

Aufgabe eines Objekts der *GraphDescriptor*–Klasse ist es, einen Graph vollständig zu repräsentieren. Zu den Objektattributen gehören ein String, der als Name des Graphen zur eindeutigen Bezeichnung und Identifizierung benutzt wird, ein *AGraphIntRelationSet*-Objekt für die Repräsentation der Relationen, ein *VertexLabel1IntRelationSet*-Objekt und ein *TwoWayDictionary*-Objekt für die Speicherung der Label. Diese vier Objektattribute (vgl. Seite 12) sind für die vollständige Repräsentation eines Graphen, bei dem nur Knoten, aber keine Kanten eine Beschriftung haben dürfen, ausreichend. Die Klasse *GraphDescriptor* bietet einen Konstruktor mit diesen vier Attributen als Parameter an, um *GraphDescriptor*–Objekte erstellen zu können. In der Regel wird es jedoch so sein, dass bis auf den Namen des Graphen die anderen Objekte aus dem Parsen einer XML- oder RDF-Quelle hervorgehen. Für diesen Fall kann ein *GraphDescriptor*–Objekt als Attribut ein *ParserSettings*–Objekt besitzen, welches alle nötigen Informationen für das Parsing zur Verfügung stellt und dazu einen passenden Konstruktor, der als Attribute nur den Namen und das *ParserSettings*–Objekt

erhält. Um das Parsing zu starten, muss dann die parse-Methode des *GraphDescriptor*–Objekts aufgerufen werden. Diese führt das Parsing der jeweiligen Datenquelle durch und berechnet die Objekte *AGraphIntRelationSet*, *VertexLabel1IntRelationSet* und *TwoWayDictionary*. Die Idee hinter dieser Konstruktion ist, dass ein Graph durch eine einzige Instanz der *GraphDescriptor*–Klasse, die alle Attribute speichert, repräsentiert wird. Später können weitere Attribute leicht ergänzt werden. Die Verwaltung eines Graphen in Form eines einzigen *GraphDescriptor*–Objekts ist somit einfacher als die Verwaltung mehrerer einzelner Objekte.

2.2.3. Hilfsklasse *ParserSettings*

Die Klasse *ParserSettings* ist abstrakt. Das einzige Objektattribut speichert den Pfad zu der zu parsenden Datei als String ab. Alle weiteren Informationen, die für das Parsing erforderlich sind, müssen die Spezialisierungen der *ParserSettings*–Klasse liefern.

2.3. Der XML-Parser

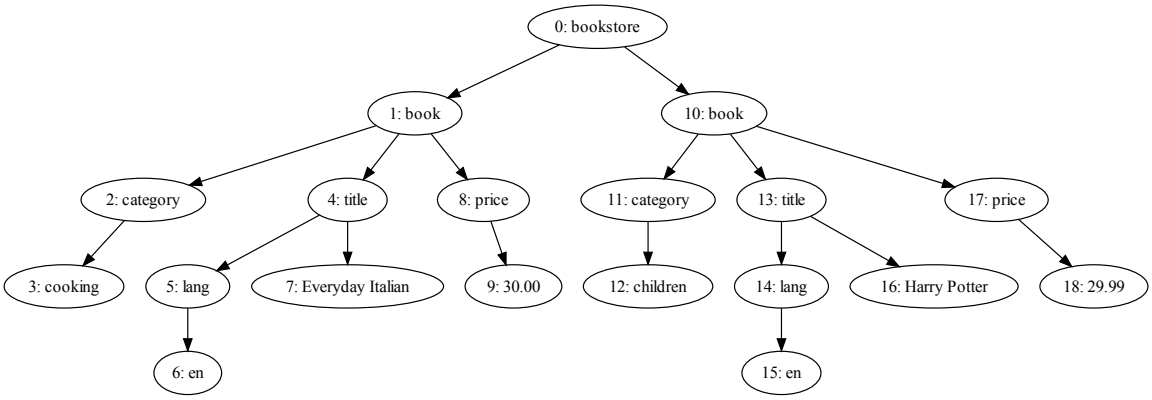
Mit Hilfe der Extensible Markup Language (XML) lässt sich leicht die Bedeutung von Daten, Texten oder Informationen definieren. Im Gegensatz zu anderen Sprachen kann XML durch zusätzliche „Befehle“(Tags) erweitert werden und bietet durch eine streng festgelegte Syntax trotzdem den Vorteil, sich als einheitliches und allgemein verwendbares Format zum Datenaustausch zu eignen. Damit stellt XML ein Datenformat dar, das auch von uns eingelesen werden können soll. Für das Parsen von XML–Daten gilt es, als Erstes festzulegen, wie der logische Aufbau eines XML Dokumentes als Graph aufgefasst wird. Wir gehen folgendermaßen vor: Die Baumstruktur der XML-Elemente übernehmen wir direkt, wobei jedes XML–Element durch einen Knoten repräsentiert wird, der als Namen den Namen des XML–Tags trägt. Der Graph, in den wir unser XML–Dokument übersetzen, wird also wieder ein Baum sein. Für jedes XML-Attribut a eines XML–Elementes e hängen wir an den e repräsentierenden Knoten einen Kindknoten n mit dem Attributnamen von a. Den Attributwert w von a hängen wir als Kindknoten an n. XML–Verarbeitungsanweisungen und XML-Kommentare werden nicht übersetzt. Der normale Text eines XML–Elementes e wird als Kindknoten an den e repräsentierenden Knoten angehängt.

Betrachten wir dazu ein einfaches XML–Dokument:

2books.xml

```
<bookstore>
   <book category="cooking">
      <title lang="en">Everyday Italian</title>
      <price>30.00</price>
   </book>
   <book category="children">
      <title lang="en">Harry Potter</title>
      <price>29.99</price>
   </book>
</bookstore>
```

Übersetzt sieht das XML–Dokument als Baum folgendermaßen aus:



Die Nummerierung vor den Knotennamen dient lediglich der Orientierung und kann ignoriert werden.

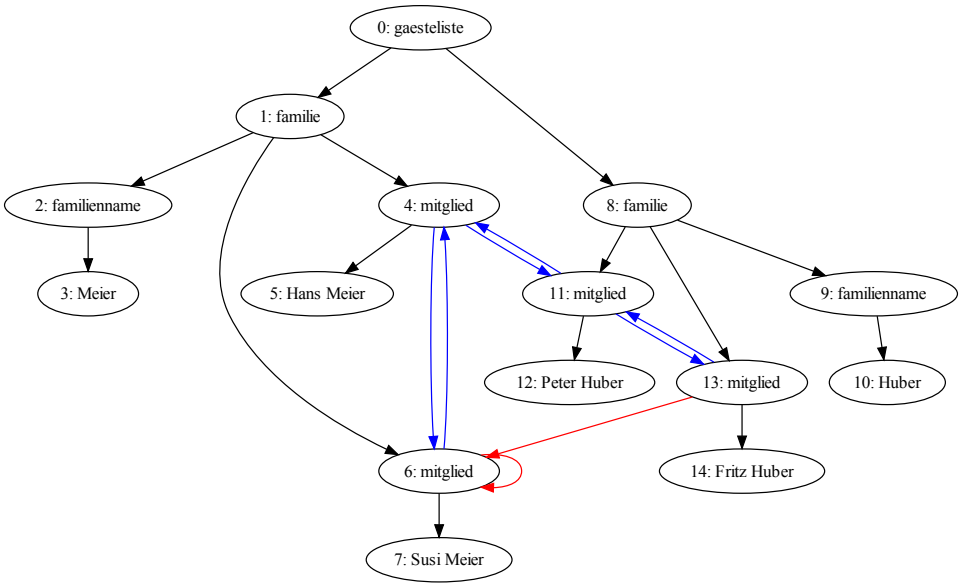
ID–IDREF–Beziehungen

Mit Hilfe von ID–IDREF–Beziehungen lassen sich auch in XML allgemeine Graphen repräsentieren. Somit steht uns eine einfache Möglichkeit zur Verfügung diese einzulesen und damit auch die Speicherstrukturen des Kerns für allgemeine Graphen zu testen.

gaesteliste.xml

```
<gaesteliste>
  <familie familienname="Meier">
   <mitglied id="m0" schulpartner="m1␣m2">Hans Meier
   </mitglied>
   <mitglied id="m1" schulpartner="m0" liebt="m1">Susi Meier
   </mitglied>
  </familie>
  <familie familienname="Huber">
   <mitglied id="m2" schulpartner="m0␣m3">Peter Huber
   </mitglied>
   <mitglied id="m3" schulpartner="m2" liebt="m1">Fritz Huber
   </mitglied>
  </familie>
</gaesteliste>
```

Das Attribut *id* wird als ID-Attribut verwendet werden, die Attribute *schulpartner* und *liebt* seien IDREF-Attribute. Wir erhalten somit keinen Baum mehr, sondern einen allgemeinen Graphen:



Die Abbildung zeigt den Graphen, der die XML–Datei „gaesteliste.xml“ repräsentiert. Die schwarzen Kanten bilden die Baumstruktur des XML–Dokumentes ab. Die ID–IDREF–Beziehung *liebt* erzeugt die roten Kannten, die ID–IDREF–Beziehung *schulpartner* die blauen.

Quelle der ID-IDREF Beziehungen

Um die ID–IDREF–Beziehungen verwenden zu können, müssen diese aus einer Quelle gewonnen werden, die festlegt, welche Attribute als Knoten aufgefasst werden sollen. Gegenwärtig werden folgende ID–IDREF–Optionen unterstützt:

- useInlineXSD: Das XML–Dokument enthält selbst ein XSD Schema
- useXSDSchemaLocation: Das XML–Dokument verweist auf eine externe XSD–Schema–Quelle
- useXSDFile: Die ID–IDREF–Beziehungen werden aus einer angegebenen Datei bezogen
- useIDIDREFDescriptorList: Die ID–IDREF–Attribute werden dem Parser in Form von von *IDIDREFDescriptor*–Objekten übergeben.
- dontUseIDIDREF: ID–IDREF–Beziehungen werden ignoriert

2.3.1. Die Hilfsklasse *XMLParserSettings*

Die Hilfsklasse *XMLParserSettings* ist eine Spezialisierung der abstrakten *ParserSettings*–Klasse. Ihre Instanzen speichern neben dem Pfad der zu parsenden XML–Datei die ID–IDREF–Option und gegebenenfalls den Pfad zu einer XSD–Datei. Der Standardkonstruktor bekommt den Pfad der XML–Datei als String und die ID–IDREF–Option in Form eines *IDIDREFOption*–Objektes. Zudem stehen zwei Setter–Methoden zum Setzen des XSD–Pfades und der IDIDREFDescriptorList zur Verfügung.

2.3.2. Die Klasse *XMLParser*

Hier wird das eigentliche Parsing durchgeführt. Alle benötigten Klassen stellt C# bereits zur Verfügung. Diese heißen *XmlTextReader*, *XmlReaderSettings*, *XmlSchemaValidationFlags*, *XmlReader*, *XmlDocument*, *XmlElement*, *XmlAttribute* und *XmlTypeCode*. Der Ablauf des Parsings wird nun kurz beschrieben: Ausgangspunkt ist ein *XmlTextReader*–Objekt. Dieses bekommt den Pfad der XML–Datei und liest diese ein. Im Falle der Gewinnung der ID-IDREF Beziehungen aus einem XSD Schema wird ein neues *XmlReaderSettings*–Objekt erzeugt. Die Parameter für die XSD Validierung werden über ein *XmlSchemaValidationFlags*–Objekt gesetzt. Aus dem *XmlTextReader*–Objekt und dem *XmlReaderSettings*–Objekt wird ein *XmlReader*–Objekt erzeugt. Dabei findet eine Validierung des XML–Dokuments gegenüber der XSD–Datei statt. Nebeneffekt dieser Validierung ist der Gewinn von Schema–Informationen, die für die Erkennung der ID– und IDREF–Attribute notwendig sind. Aus dem *XmlReader*–Objekt wird ein *XmlDocument*–Objekt erstellt. Das *XmlDocument*–Objekt bietet Methoden an um unter anderem auf die Wurzel des XML–Dokumentes zuzugreifen

sowie auf die Kinder eines Knotens. Somit ist ein Breiten– oder Tiefendurchlauf durch das XML–Dokument möglich. Gegenwärtig ist ein Breitendurchlauf implementiert. Stößt dieser auf einen Knoten mit Attributen, so lässt sich für jedes Attribut dank der Validierung ein *XmlTypeCode* bestimmen. Entspricht dieser dem *XmlTypeCode* für ID oder IDREF ist ein ID– bzw. IDREF–Attribut gefunden und anstatt neuer Knoten werden die entsprechenden ID–IDREF Kanten erstellt. Im Falle der Gewinnung der ID–IDREF–Beziehungen aus der IDIDREFDescriptorList kann selbstverständlich keine Validierung ausgeführt werden. Daher bleibt das Setzen des *XmlSchemaValidationFlags* aus. Der restliche Ablauf zum Erstellen des *XmlDocument*–Objekts ist identisch. Für die Erkennung von ID–IDREF–Attributen findet ein Abgleich mit der IDIDREFDescriptorList statt. Für den Fall, dass gar keine ID–IDREF–Beziehungen beachtet werden sollen oder ein Attribut kein ID– oder IDREF–Attribut ist, wird jedes Attribute durch je einen Knoten für Attributnamen und Attributwert, wie in Abschnitt 2.3 beschrieben, übersetzt. Der Breitendurchlauf lässt sich anhand der Knotennummerierungen in den Abbildungen auf Seiten 15 und 16 erkennen.

Einlesen einer XML Datei mit XSDSchemaLocation Validierung

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.ValidationFlags |=
XmlSchemaValidationFlags.ProcessSchemaLocation;

XmlTextReader xmlTextReader =
      new XmlTextReader("C:\...\meine_xml_datei.xml");

XmlReader XMLreader =
      XmlReader.Create(xmlTextReader, settings);

XmlDocument doc = new XmlDocument();
doc.Load(XMLreader);

XmlElement root = doc.DocumentElement;

foreach (XmlNode childNode in root.ChildNodes)
   {
      (...)
   }

foreach (XmlAttribute att in node.Attributes)
   {
      (...)
   }
```

2.4. Der RDF–Parser

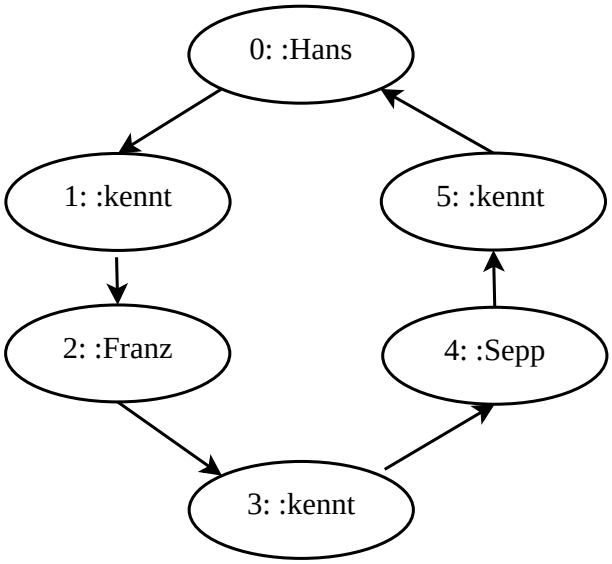
Das Resource Description Framework (RDF) dient der formalen Beschreibung von Informationen über Objekte und stellt eine wichtige Komponente des Sematischen Webs dar. Daher darf RDF auch in unserer Implementierung nicht fehlen. Ähnlich wie bei der XML–Verarbeitung müssen wir auch bei RDF Daten festlegen, wie wir diese als Graphdaten interpretieren wollen. RDF modelliert Aussagen von Form von Tripeln. Dabei spricht man von Subjekt, Prädikat und Objekt. Das Subjekt ist die Ressource über die eine Aussage getroffen wird, das Prädikat kennzeichnet eine Eigenschaft des Subjekts und das Objekt ist ein Argument des Prädikats. Betrachten wir dies an einem Beispiel in der Notation 3 kurz N3. N3 ist gleichwertig zur XML–Syntax von RDF jedoch einfacher zu schreiben und oft besser zu lesen.

friends.n3

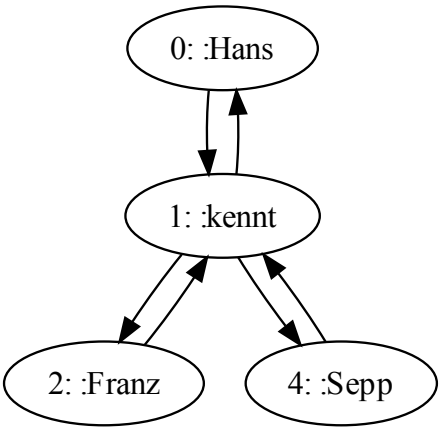
```
<Hans>      <kennt>      <Franz> .
<Franz>     <kennt>      <Sepp>  .
<Sepp>      <kennt>      <Hans>  .
```

Im Gegensatz zur XML–Verarbeitung, bei der auch Elemente des gleichen Namens durch verschiedene Knoten übersetzt wurden, möchten wir bei RDF für gleiche Subjekte und Objekte denselben Knoten verwenden. Lediglich die Prädikate bekommen immer einen eigenen Knoten. Dabei wird ein RDF–Tripel so übersetzt, dass der Knoten, der das Subjekt repräsentiert, Vater des Knotens wird, der aus dem Prädikat entstanden ist, der wiederum Vater des aus dem Objekt hervorgegangenen Knotens wird.

Nach diesen Vorgaben übersetzt sieht unser RDF–Dokument folgendermaßen aus:

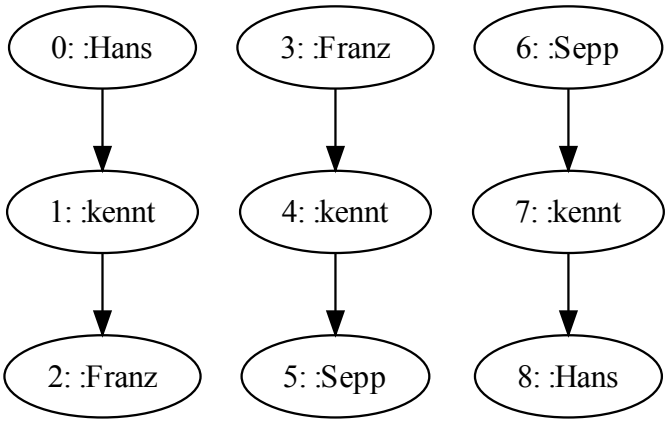


Zum Vergleich ein Blick auf den Graphen, wie er aussehen würde, wenn auch gleiche Prädikate durch nur einen einzigen Knoten repräsentiert werden würden:



Hierbei würden falsche Informationen entstehen. Beispielsweise, dass Hans den Sepp kennt, was jedoch nicht der Fall ist.

Zuletzt betrachten wir den Graphen der entstehen würde, wenn für jedes Subjekt, Prädikat oder Objekt ein eigener Knoten angelegt würde. Ergebnis wäre ein Wald mit trivialen Bäumen, die jeweils nur aus drei Knoten bestehen würden. Die Anzahl der Bäume entspräche der Anzahl der RDF–Aussagen. Es gäbe keinen Informationsgewinn, da jede Aussage unabhängig von bereits vorhandenen Aussagen behandelt werden würde.



2.4.1. Die Hilfsklasse *RDFParserSettings*

Wie bei der *XMLParserSettings*–Klasse (siehe 2.3.1) ist auch die Klasse *RDFParserSettings* eine Spezialisierung der abstrakten *ParserSettings*–Klasse. Ihre Instanzen speichern neben dem Pfad der zu parsenden RDF–Datei noch deren Format ab. Unterstützt wird RDF als XML und N3.

2.4.2. Die Klasse *RDFParser*

Für die RDF–Verarbeitung stehen in C# im Gegensatz zur XML keine Standardklassen zur Verfügung. Daher weicht die *RDFParser*–Klasse hier auf die SemWeb.NET–Bibliothek[1] von Joshua Tauberer aus. Diese kann neben dem Lesen von RDF in N3 und XML auch für das Schreiben von RDF–Daten verwendet werden. In unserem Fall reicht das Lesen. Der Umgang mit dieser Bibliothek ist sehr einfach: Als Erstes legt man ein *MemoryStore*–Objekt an. Dieses bietet eine Import Methode an, welche als Argument ein *RdfXmlReader*–Objekt oder einen *N3Reader*–Objekt akzeptiert. Die beiden Reader–Objekte bekommen als Argument jeweils den Pfad zum RDF–Dokument als String. Nach dem Import kann man über alle Aussagen des RDF–Dokumentes in Form von *Statement*–Objekten iterieren. Die *Statement*–Objekte bieten dann direkten Zugriff auf Subjekt, Prädikat und Objekt jeder Aussage des RDF–Dokumentes.

Einlesen von N3 mit Hilfe der SemWeb.NET Bibliothek

```
MemoryStore store = new MemoryStore();

store.Import(new N3Reader("C:\...\meine_rdf_datei.n3"));

foreach (Statement statement in store)
{
    Console.WriteLine("Subjekt:␣"   + statement.Subject);
    Console.WriteLine("Prädikat:␣"  + statement.Predicate);
    Console.WriteLine("Objekt:␣"    + statement.Object);
}
```

[1] http://razor.occams.info/code/semweb/

3. Anfragesprache und Anfrageauswertung

3.1. Eine SQL-nahe Anfragesprache

Nachdem wir uns mit dem Einlesen von Daten beschäftigt haben, gilt es nun, eine einfache Anfragesprache zu definieren, um Anfragen auf die eingelesenen Daten tätigen zu können. Ziel ist es zum einen, Informationen aus den eingelesenen Graphen bzw. Dokumenten zu erhalten. Zum anderen, Tests und eine Evaluierung der Speicherstrukturen des Kerns zu ermöglichen. Wir haben uns dafür entschieden, eine an SQL angelehnte Syntax zu verwenden, um einen leichten Zugang zu unserer Anfragesprache zu ermöglichen. SQL hat sich im Laufe der Zeit durch den Gebrauch vieler Anwender und verschiedener Softwarehersteller als sehr nützlich erwiesen. Zudem ist die SQL-Syntax vielen Anwendern vertraut und stellt damit eine niedrige Einstiegshürde dar.

Gemeinsamkeiten und Unterschiede gegenüber SQL

Wie in SQL, unterstützen wir einfache Abfragen über alle Spalten, Abfragen mit Spaltenauswahl (Projektion) und Abfragen bei gleichzeitiger Umbenennung der Spalten. Eine Eliminierung von Duplikaten ist gegenwärtig nicht implementiert. Bei den Datenquellen verpflichten wir den Benutzer, im Gegensatz zu SQL, auch bei nur einfach vorkommenden Quellrelationen, diese mit einem eindeutigen Namen zu bezeichnen. In SQL wäre SELECT * FROM Professor eine gültige Anfrage. In unserer Anfragesprache erwarten wir dagegen beispielsweise SELECT * FROM Professor AS P1. Diese Einschränkung würde aufgrund einer einfacheren Implementierbarkeit eingefügt. Im Bereich der Selektionen sind Equi-Joins und einfache Gleichheitsfilter auf String oder Integer möglich. Alle anderen Join-Arten werden nicht unterstützt, ebenso keine Selektionen, die nicht auf Gleichheit beruhen. Da das eigentliche Ziel der Anfragesprache die Auswertung der Datenstrukturen des Kerns ist, wurde auf Gruppierungen, Aggregatfunktionen oder Sortierungen sowie auf Subqueries verzichtet.

3.2. Syntax und Grammatik der Anfragesprache

Die Syntax unserer Anfragesprache sei durch folgende EBNF-Grammatik definiert:

```
Anfrage          ::= "SELECT" Projektionen "FROM" Relationen
                     ["WHERE" Selektionen].
Projektionen     ::= "*"
                   | Projektion {"," Projektion}.
Projektion       ::= Attribut
                   | Attribut "AS" Name.
Relationen       ::= Relation
                   | Relation {"," Relation}.
Relation         ::= Graphname "." Relationstyp
                     "AS" Relationsname.
Selektionen      ::= Selektion {"AND" Selektion}.
Selektion        ::= JoinSelektion
                   | StringSelektion
                   | IntegerSelektion.
Attribut         ::= Relationsname "." Knoten.
Knoten           ::= "V"
                   | "LABEL"
                   | "V1"
                   | "V2".
Relationstyp     ::= "ADJACENCY"
                   | "REACHABILITY"
                   | "LABEL".
JoinSelektion    ::= Attribut "=" Attribut.
StringSelektion  ::= Attribut "=" "'" Name "'".
IntegerSelektion ::= Attribut "=" Zahl.
Graphname        ::= Name.
Relationsname    ::= Name.
```

Die Definitionen der ganzen Zahlen und der Namen werden vorausgesetzt. Das heißt, Zahl und Name werden als Terminalsymbole betrachtet.

Beispiele für gültige Anfragen

- SELECT * FROM Graph1.ADJACENCY AS R
- SELECT R.V2 FROM Graph1.ADJACENCY AS R
- SELECT R.V2, Label.LABEL FROM Graph1.ADJACENCY AS R, Graph1.LABEL AS Label WHERE R.V2 = Label.V
- SELECT R.V2, Label.LABEL FROM Graph1.ADJACENCY AS R, Graph1.LABEL AS Label WHERE R.V2 = Label.V AND Label.LABEL = ”Hans”
- SELECT R.V2, Label.LABEL AS Vorname FROM Graph1.ADJACENCY AS R,

Graph1.LABEL AS Label WHERE R.V2 = Label.V AND Label.LABEL = ”Hans”

- SELECT R.V2, Label.LABEL AS Vorname FROM Graph1.ADJACENCY AS R, Graph1.LABEL AS Label WHERE R.V2 = Label.V AND Label.V = 14

3.3. Die Klasse *QueryParser*

Für Testzwecke ist es möglich, eine Anfrage auch ohne Benutzung der Anfragesprache zu erstellen. Dabei müssen alle Objekte, in die der Anfragestring sonst übersetzt würde, selbst erstellt werden. Da dies in der Regel jedoch nicht der Fall sein wird, müssen wir den Anfragestring zuerst parsen. Die Klasse *QueryParser* stellt dazu die Methode parseComplete() zur Verfügung. Da die endgültige Grammatik erst im Laufe der Implementierung entstanden ist, arbeitet die *QueryParser*–Klasse vor allem mit regulären Ausdrücken, um den Anfrage-String nach und nach zu zerteilen. Endergebnis sind drei Listen von Objekten: die Relationsobjekte, Selektionsobjekte und Projektionsobjekte. Ein *Relation*–Objekt speichert den Namen des Graphen und den Relationstyp sowie den Alias-Namen der Relation ab. Die Selektionsobjekte können *TrueIntSelection*–, *TrueStringSelection*– oder *JoinSelection*–Objekte sein. Gemeinsam ist ihnen, dass sie alle über ein linkes *RelationAttribute* verfügen, da sie Spezialisierungen der abstrakten *ASelection*–Klasse sind. Ein *RelationAttribute*–Objekt ist ein *Relation*–Objekt, erweitert um ein Attribut aus der Menge {V1, V2, V, LABEL}. Die Bezeichnung linkes *RelationAttribute*–Objekt ist gültig, da unsere Grammatik für alle Selektionen als Erstes ein Attribut erwartet. Eine Selektion wie beispielsweise „5 = R.V1“, bei der auf der linken Seite des Gleicheitszeichens kein Attribut steht, erlaubt die Grammatik nicht. Das *TrueIntSelection*–Objekt speichert zusätzlich einen Integer, das *TrueStringSelection*–Objekt einen String und das *JoinSelection*–Objekt ein weiteres *RelationAttribute*–Objekt. Da bei den Projektionen auch die Möglichkeit der Umbenennung des Projektionsattributs existiert, werden die Projektionen durch zwei Klassen vertreten, die Spezialisierungen der abstrakten *AProjektion*–Klasse sind. Die Klasse *Projection* steht für die Projektionen ohne Umbenennung zur Verfügung. Projektionen mit Umbenennung des Projektionsattributs werden durch die Klasse *AsProjection* übernommen, die zusätzlich den String speichert, in den das Projektionsattribut umbenannt werden soll. Beide Spezialisierungen speichern das *Relationattribute*, auf das projiziert werden soll.

3.4. Die Klasse *Query*

Die Klasse *Query* stellt die Methoden zum Ausführen der Anfrage zur Verfügung. Dazu bekommt ein *Query*–Objekt eine Liste von Relationsobjekten, Selektionsobjekten und Projektionsobjekten, die in der Regel über ein *QueryParser*–Objekt aus einem String gewonnen wurden. Zusätzlich wird ein Zugriff auf die geparsten Graphen benötigt. Dazu erhält das *Query*–Objekt ein *GraphDesciptorDataBase*–Objekt, welches im Wesentlichen aus einer Liste mit *GraphDesciptor*–Objekten besteht, die aus dem Parsing hervorgegangen sind. Zur Klasse *GraphDesciptorDataBase* folgt später mehr. Für die Ausführung der Anfrage erstellen wir einen neuen Graphen, den Anfragegraphen. Der Aufbau dieses Graphen und dessen Auswertung sollen erst stichpunktartig und später an einem konkreten Beispiel aufgezeigt werden. Die run()-Methode der *Query*–Klasse übernimmt sowohl Aufbau des Anfragegraphen als auch dessen Auswertung. Beginnen wir mit der Erstellung des Anfragegraphen.

3.4.1. Erstellung des Anfragegraphen

1. Für alle binären Relationen, die aus der Liste der Relationsobjekte hervorgehen, fügen wir ihre beiden Relationsattribute in eine neue Liste ein In dieser sind dann alle Relationsattribute aller Relationen enthalten. Nachdem jede Relation binär ist, wird die Liste der Relationsattribute doppelt so viele Elemente enthalten wie die Liste der Relationsobjekte Relationen.
2. Wir legen in einem Hilfsgraphen für jedes Relationsattribut aus der im vorherigen Schritt ermittelten Liste einen Knoten mit dem Namen des Relationsattributs an.
3. Die Selektionen werden durchlaufen und für jede Join-Selektion wird im Hilfsgraphen eine ungerichtete und ungelabelte Kante zwischen den Relationsattributen angelegt, deren Werte durch die Join-Selektion identisch sein sollen.
4. Mit Hilfe der *STRONG_COMPONENTS*–Klasse des Kerns bestimmen wir die Zusammenhangskomponenten des Hilfsgraphen. Dafür übersetzen wir jede ungerichtete Kante in zwei gerichtete Kanten.
5. Wir legen einen Anfragegraph in Form eines *QueryGraph*–Objektes an. Für jede Zusammenhangskomponente erzeugen wir ein *QueryGraphNode*–Objekt, welches dem *QueryGraph*–Objekt hinzugefügt wird. In dem *QueryGraphNode*–Objekt werden in einer Liste alle Relationsattribute gespeichert, deren Knoten in der Zusammenhangskomponente enthalten waren.

Zusammenfassung bis hier hin: Der Anfragegraph besteht bisher lediglich aus Knoten und einer leeren Menge von Kanten. Die Knoten enthalten Listen. In den Listen stehen die Relationsattribute, deren Werte gleich sein sollen.

6. Für jede Relation R aus der Liste der Relationsobjekte fügen wir dem Anfragegraphen zwei Cut–Kanten[2] in Form von *QueryGraphCutEdge*–Objekten hinzu. *QueryGraphCutEdge* –Objekte sind gerichtete Kanten, die zusätzlich über ein Kantengewicht verfügen. Eine Kante führt vom Anfrageknoten, in dem das erste Relationsattribut von R enthalten ist, zum Anfrageknoten mit dem zweiten Relationsattribut von R. Die andere Kante verläuft in die Gegenrichtung. Jedem *QueryGraphCutEdge*–Objekt werden die Relationsattribute, der Relationstyp, der Name des Graphen und der Typ des Cuts (CUT1 oder CUT2) zugeordnet. Das Gewicht einer Kante entspricht den Kosten des jeweiligen Cuts, von denen die Kosten einer Contains-Anfrage subtrahiert werden. Beide Kosten werden vom Kern bereitgestellt. Die Idee hinter dieser Berechnung ist, dass wir im nächsten Schritt die Wahl der Kanten optimieren möchten. Dabei ist es möglich, dass die Optimierung bzgl. zweier Relationsattribute weder die CUT1 noch die CUT2-Kante enthält. Dann ist es erforderlich, eine Contains–Kante zwischen diesen Relationsattributen einzufügen, die anstelle des Joins eine Selektion durchführt. Die Kosten für diese Contains-Kante müssen jedoch in der Optimierung mit beachtet werden. Eine Cut–Kante wird also leichter (besser), wenn dafür auf eine teurere

[2] Die Bezeichnung Cut greift auf die Nomenklatur innerhalb des Kerns zurück. Dieser bietet für zweistellige Integer Relationen die Methoden Cut1(int x) und Cut2(int y) an. Angewandt auf eine zweistellige Integer Relation R liefert R.Cut1(x) alle y, so dass $(x, y) \in R$. Dementsprechend liefert R.Cut2(y) alle x, so dass $(x, y) \in R$.

Contains–Kante verzichtet werden kann. Dies drückt die Subtraktion der Contains–Kosten von den Cut–Kosten aus.

7. Mit Hilfe des Edmonds-Algorithmus führen wir eine statische Optimierung auf den Anfragegraphen durch. Ziel ist es, ein „quasi-minimales“ Branching[3] des Anfragegraphen zu bestimmen. Der Edmonds-Algorithmus liefert ein maximales Branching[4]. Ein minimales Branching wäre durch die leere Menge gegeben. Das „quasi–minimale“ Branching soll über die gleiche Anzahl an Kanten verfügen wie das maximale Branching, jedoch die aufsummierte Kantensumme nicht maximieren, sondern minimieren. Die einfachste Methode dies zu erreichen, erhält man, indem die Kantengewichte so verändert werden, dass man das maximale Branching bestimmen kann. Dazu bestimmt man die neuen Kantengewichte nach folgender Formel:

   $$\text{Kantengewicht}_{neu} = (\text{Kantengewicht}_{alt} - max\{\text{Kantengewichte}\} - 1) \cdot (-1)$$

8. Wir bestimmen das maximale Branching auf dem Anfragegraph mit Hilfe des Edmonds–Algorithmus.
9. Wir stellen die alten Kantengewichte wieder her.
10. Für alle Relationen R überprüfen wir, ob zwischen den Anfrageknoten, die die Relationsattribute von R enthalten, eine Cut1– oder Cut2–Kante existiert. Existiert keine Kante, so fügen wir eine Contains-Kante[5] zwischen den beiden Anfrageknoten ein. Die Relationsattribute, der Relationstyp und der Name des Graphen werden von einer der Cut–Kanten übernommen.

Zusammenfassung bis hier hin: Der QueryGraph besteht nun aus der unveränderten Menge von Knoten und einer Menge von Kanten. Die Kanten sind Cut-Kanten oder Contains-Kanten. Führt eine Cut-Kante vom Knoten A zum Knoten B, so gibt es keine Cut-Kante in der Gegenrichtung von Knoten B zu A. Gab es nach Schritt 6 zwei Knoten K_1 und K_2, die durch Cut-Kanten($K_1 \to K_2$ und $K_2 \to K_1$) verbunden waren, und wurden beide Cut-Kanten durch die Edmonds-Optimierung entfernt, so existiert jetzt eine Contains-Kante zwischen K_1 und K_2 welche die Joins durch eine Selektion ersetzt.

Betrachten wir den QueryGraph mit der Menge seiner Knoten und der Menge seiner Cut–Kanten, sehen wir einen Wald. Idee der später folgenden Auswertung wird es sein, jeden Baum des Waldes als Teilanfrage anzusehen, wobei sich das Endergebnis der Anfrage als Kreuzprodukt aller Teilanfragen zusammensetzt. Die Notwendigkeit des Kreuzprodukts resultiert aus der Tatsache, dass bei Anfragen, die in voneinander unabhängige Teilanfragen zerfallen sich, das Ergebnis aus allen möglichen Kombinationen der Teilanfrageergebnisse kombinieren lässt. Das ist genau das, was das Kreuzprodukt ausdrückt.

[3] Ein Branching in einem gerichteten Graphen sei definiert als eine Menge gerichteter Kanten mit der Eigenschaft, dass die Menge keine Zyklen enthält und dass kein Knoten zwei eingehende Kanten besitzt.

[4] Für einen gerichteten Graphen mit Kantengewichten sei ein maximales Branching definiert als ein Branching, bei dem die Summe aller Kantengewichte maximal ist.

[5] Die Bezeichnung Contains greift auf die Nomenklatur innerhalb des Kerns zurück. Dieser bietet für zweistellige Integer–Relationen die Methode Contains(int x, int y). Angewandt auf eine zweistellige Integer–Relation R liefert R.Contains(x, y) den Wahrheitswert, ob $(x, y) \in R$.

3.4.2. Erstellung des Anfragegraphen — Beispiel

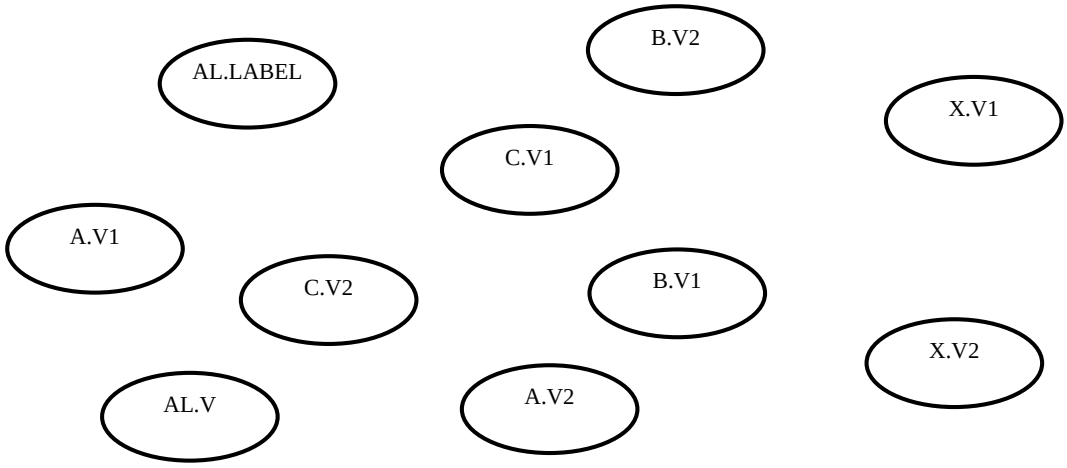
Führen wir nun die Schritte 1 bis 10 einmal an einem Beispiel durch. Gegeben sei folgende Anfrage:

```
SELECT X.V1,
       AL.V,
       AL.LABEL,
FROM   G1.LABEL AS AL,
       G1.ADJACENCY AS A,
       G1.ADJACENCY AS B,
       G1.ADJACENCY AS C,
       G1.ADJACENCY AS X
WHERE  A.V1 = AL.V    AND
       A.V2 = B.V1    AND
       B.V2 = C.V1    AND
       C.V2 = A.V1    AND
       A.V1 = 4       AND
       X.V1 = 0
```

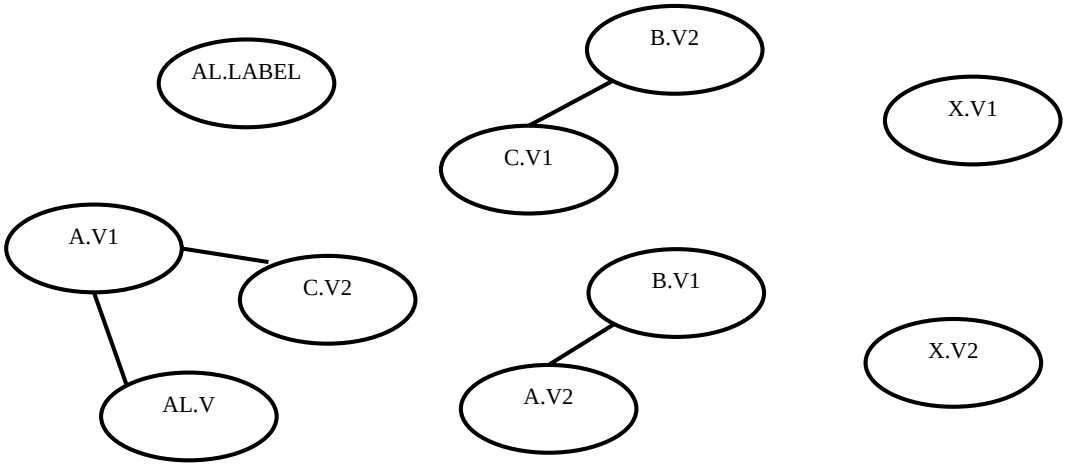
Schritt 1: Bestimmung der Liste der Relationsattribute

Relationsattribute = {AL.V, AL.LABEL, A.V1, A.V2, B.V1, B.V2, C.V1, C.V2, X.V1, X.V2}. Die Mächtigkeit der Menge beträgt 10.

Schritt 2: Knoten des Hilfsgraphen



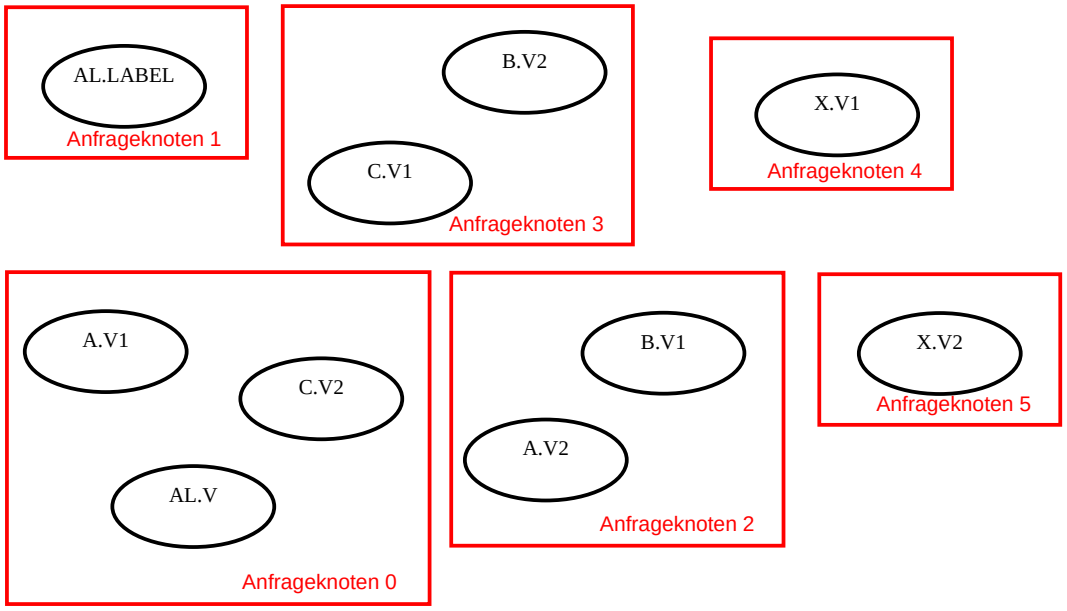
Schritt 3: Kanten des Hilfsgraphen
Liste der Joinselektionen = {A.V1 = AL.V, A.V2 = B.V1, B.V2 = C.V1, C.V2 = A.V1}



Schritt 4: Bestimmung der Zusammenhangskomponenten

Zusammenhangskomponenten des Hilfsgraphen = $\{Z_0, Z_1, Z_2, Z_3, Z_4\}$ mit Z_0 = {AL.V, A.V1, C.V2}, Z_1 = {Al.LABEL}, Z_2 = {A.V2, B.V1}, Z_3={B.V2, C.V1}, Z_4 = {X.V1} und Z_5={X.V3}

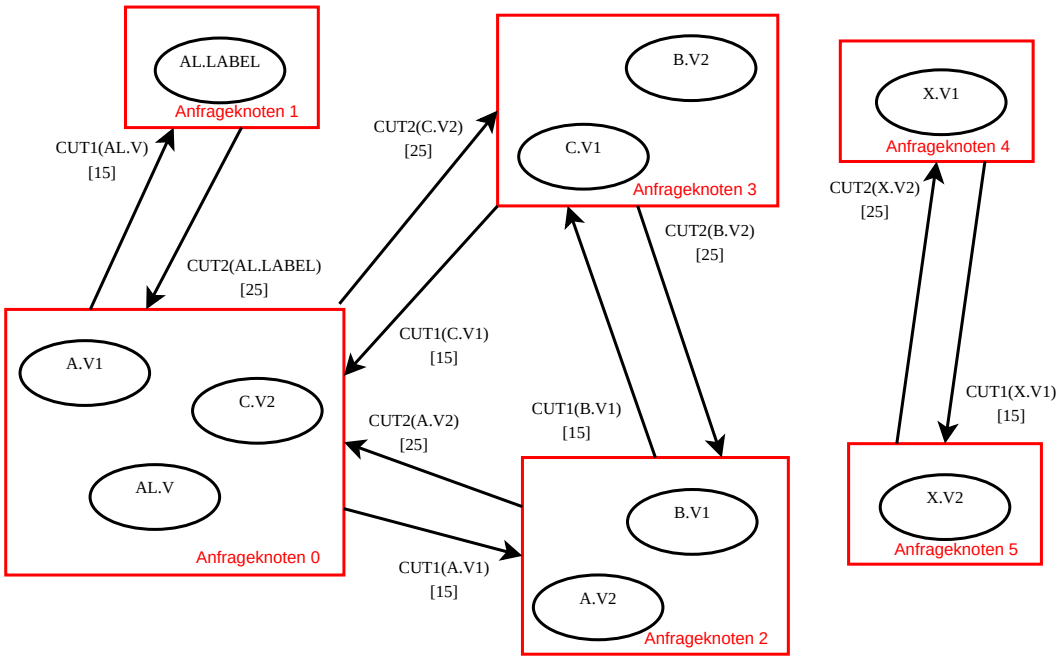
Schritt 5: Anfragegraph anlegen und Anfrageknoten hinzufügen



Schritt 6: Hinzufügen der Cut-Kanten
Um die Lesbarkeit des Graphen beizubehalten, sind die Kanten nur mit dem Cut-Typ (CUT1 oder CUT2)[a], dem Relationsattribut, das später das Argument der jeweiligen Cut–Methode sein wird, und den Kantengewichten beschriftet. Die Kantengewichte stehen in eckigen Klammern. Die Symmetrie und Gleichheit der Kantengewichte bei gleichem Cut-Typ beruht auf der Tatsache, dass der Kern für die Cut– und Contains–Kosten bisher nur statische Werte liefert. Eine von der jeweiligen Relationsausprägung abhängige Kostenbestimmung ist bisher nicht implementiert. Die Kosten sind bisher wie folgt definiert:

- Kosten(Cut1) = 20
- Kosten(Cut2) = 30
- Kosten(Contains) = 5

Daher ergeben sich beispielsweise für eine Cut2–Kante folgende Kosten:
Kantengewicht(Cut2-Kante) = Kosten(Cut2) + Kosten(Contains) = 30 − 5 = 25



[a]Der Kern bietet für zweistellige Integer Relationen die Methoden Cut1(int x) und Cut2(int y) an. Angewandt auf eine zweistellige Integer Relation R liefert R.Cut1(x) alle y, so dass $(x, y) \in R$. Dementsprechend liefert R.Cut2(y) alle x, so dass $(x, y) \in R$.

Schritt 7: Änderung der Kantengewichte

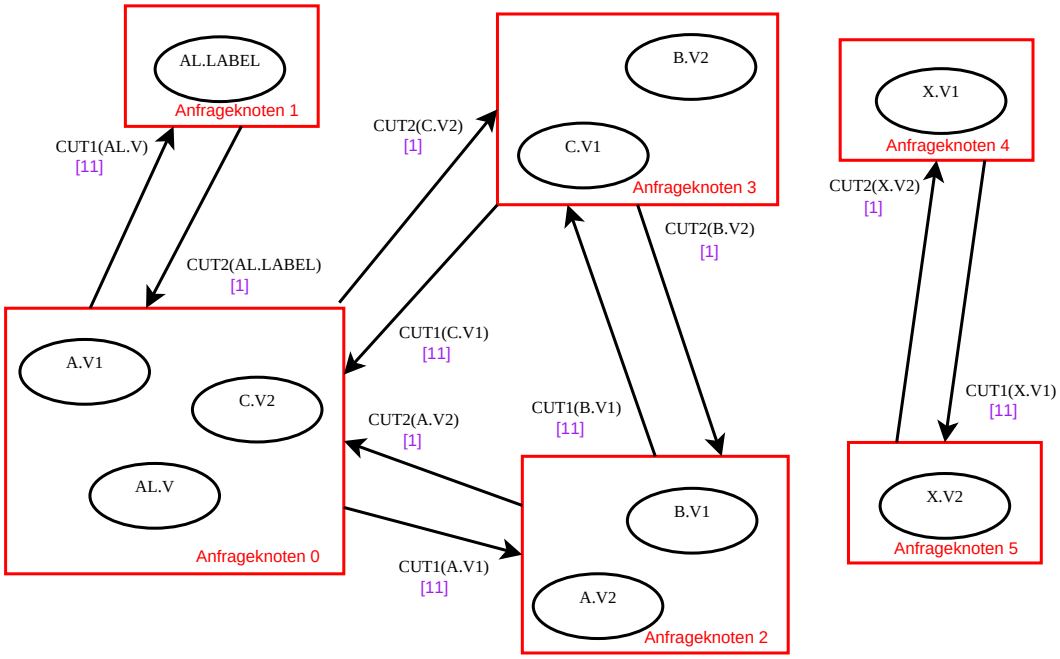
$\text{Kantengewicht}_{neu} = (\text{Kantengewicht}_{alt} - max\{\text{Kantengewichte}\} - 1) \cdot (-1)$

$max\{\text{Kantengewichte}\} = 25$

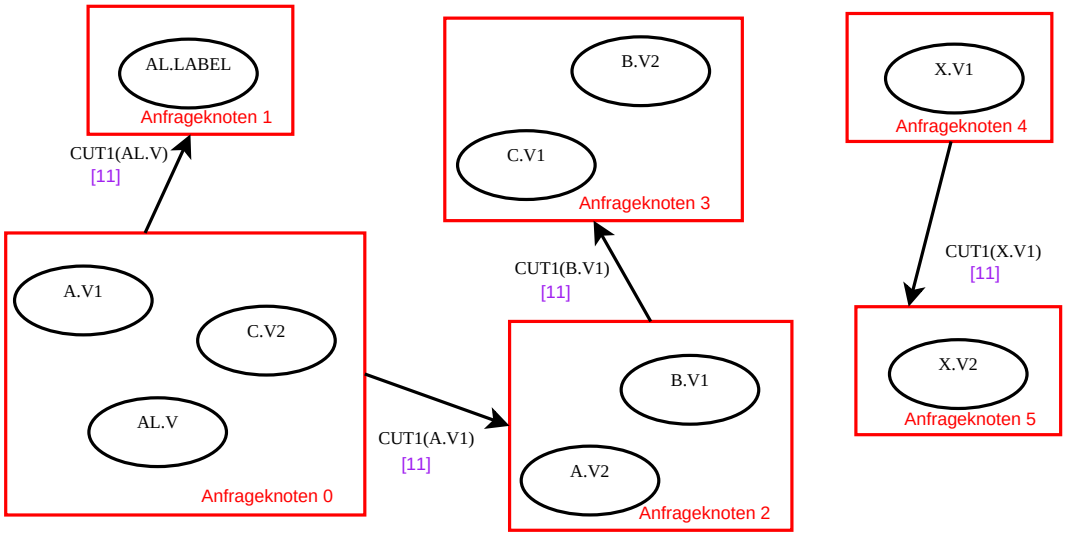
Beispiel: $\text{Kantengewicht}_{neu}(\text{CUT2(AL.LABEL)}) =$

$= (\text{Kantengewicht}_{alt}(\text{CUT1(AL.V)}) - max\{\text{Kantengewichte}\} - 1) \cdot (-1) =$

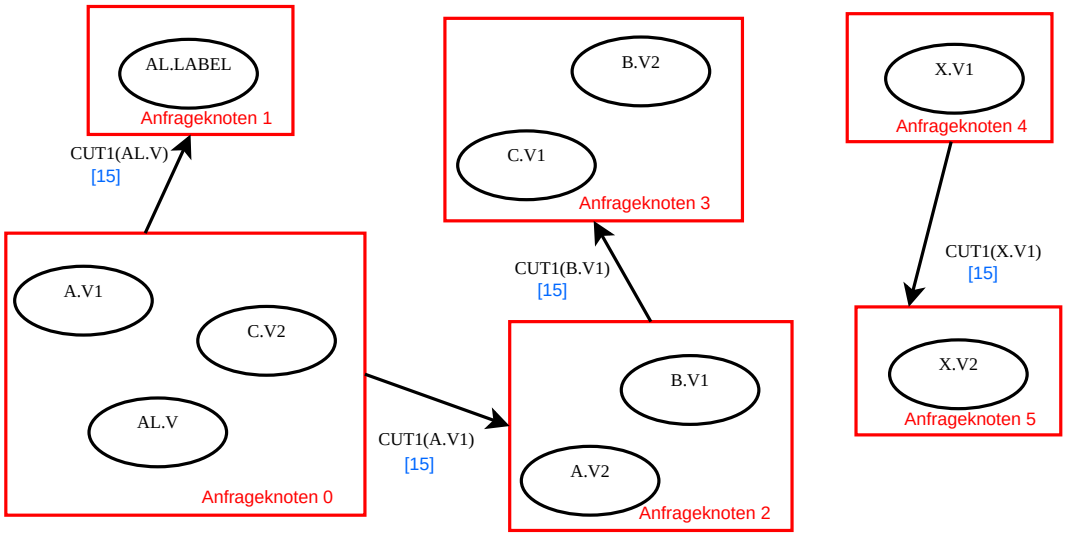
$= (15 - 25 - 1) \cdot (-1) = 11$



Schritt 8: Bestimmung des maximalen Branching

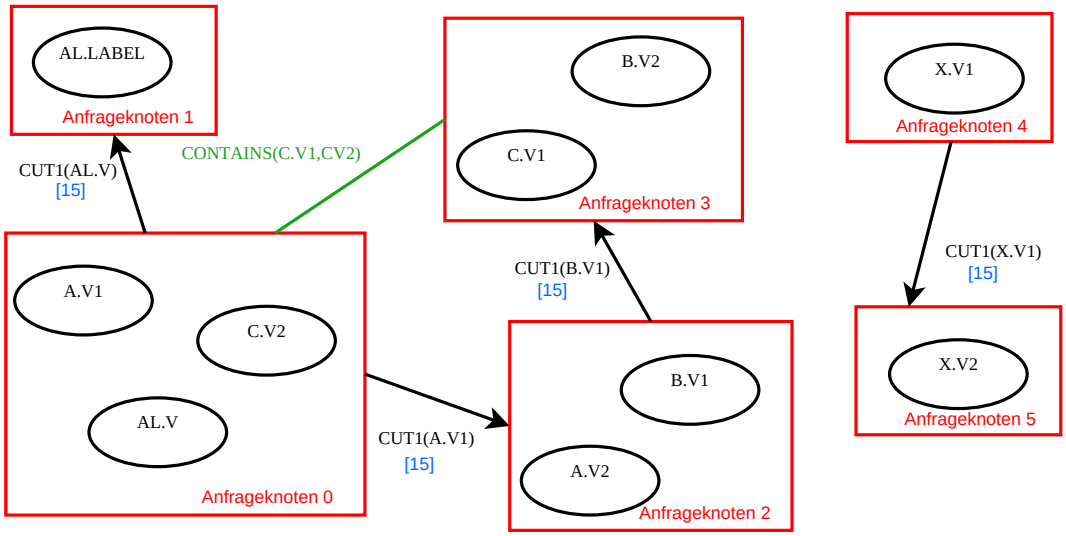


Schritt 9: Wiederherstellung der alten Kantengewichte

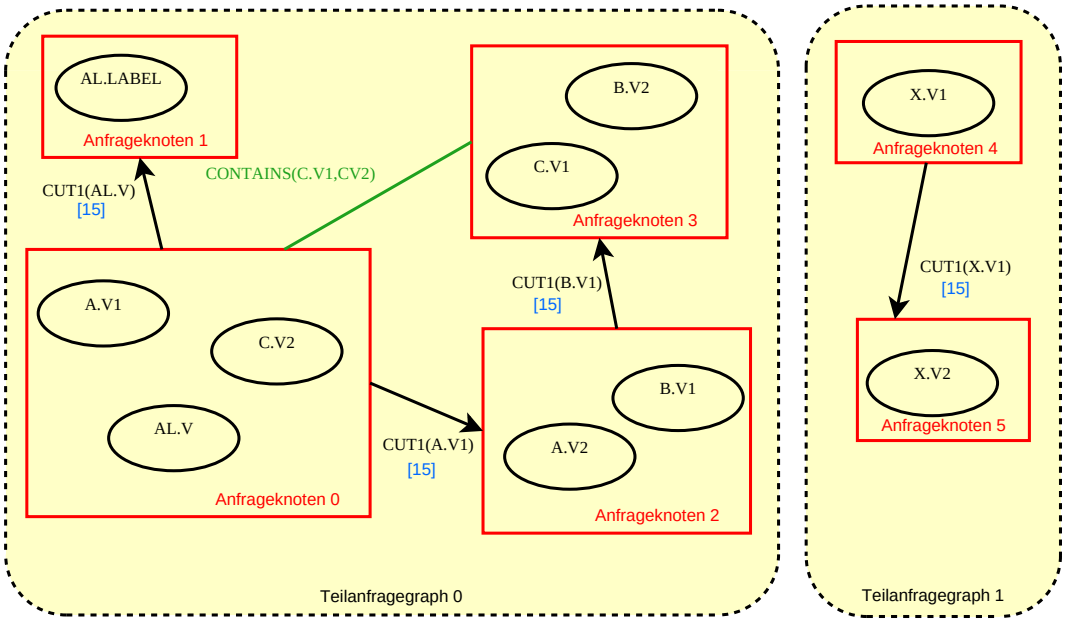


Schritt 10: Hinzufügen von Contains-Kanten

Vor der Edmonds-Optimierung existierten zwischen Anfrageknoten 0 und Anfrageknoten 3, genauer genommen zwischen den Relationsattributen der C Relation, zwei Cut–Kanten. Beide Kanten wurden durch die Edmonds-Optimierung entfernt. Dafür ergänzen wir nun eine Contains-Kante:



Zwischenergebnis



Der jetzt entstandene Anfragegraph lässt sich in zwei Teilanfragegraphen zerteilen. Die Auswertung des Anfragegraphen hin zum Ergebnis wird auf dem Ansatz beruhen, dass alle Teilanfragegraphen für sich ausgewertet werden und deren Ergebnisse dann über ein Kreuzprodukt zum Endergebnis der Anfrage zusammengesetzt werden.

3.4.3. Auswertung des Anfragegraphen

Angelehnt an SQL soll die Auswertung der Anfrage eine Ergebnistabelle liefern. Jede Zeile der Tabelle stellt dabei ein Ergebnistupel dar, das die Bedingungen der Anfrage erfüllt. Jedes Ergebnistupel wird während der Auswertung als reines Integer–Array behandelt. Die Ersetzung in Strings für die jeweiligen Attribute erfolgt erst am Ende der Auswertung. Ein Zwischenergebnis oder Endergebnis ist damit eine Auflistung von gleichlangen Integer-Arrays. Auf jedes Ergebnis (Auflistung von Integer-Arrays) kann ein Operator angewandt werden, der wiederum ein Ergebnis (Auflistung von Integer–Arrays) liefert. Die Operatoren lassen sich somit aufeinander stapeln. Die Anwendung eines Operators kann dabei die Auflistung der Integer-Arrays und die Integer-Arrays selbst verändern. Das „Initialergebnis“ ist eine leere Auflistung. Wurden auf das Initialergebnis alle Operatoren, die aus dem Anfragegraphen, den Projektionen und Selektionen gewonnen wurden, angewandt, erhalten wir das Endergebnis der Anfrage. Eine optische Korrektur ist schließlich die Ersetzung der Integer in Strings, die die eigentlichen Attributwerte repräsentieren.

Codeschnipsel zur Verdeutlichung der Operatorfunktionsweise

```
(...)
Operator op1 = new Operator(...);
result = op1.applyOn(result);

Operator op2 = new Operator(...);
result = op2.applyOn(result);

(...)
```

Die Operatoren sind in zwei Gruppen unterteilt. Die eine Gruppe der „Suboperatoren“ besteht aus den Operatoren, die für die Erstellung und Modifikation der Teilanfragegraphen verantwortlich sind. Die andere Gruppe enthält den Operator für die Zusammensetzung der Teilanfragegraphenergebnisse zum Gesamtergebnis sowie einen Operator für die Umordnung der Integer–Arrays.

Übersicht über die Operatoren

Anfrageoperatoren (Operatoren):

- *CartesianProductOperator*: bekommt als Argument eine Auflistung von Integer-Arrays. Erstellt aus dem Argument und der Auflistung, auf die der Operator angewandt wurde ein kartesisches Produkt der beiden Auflistungen.
- *ChangeOrderOperator*: bekommt als Argument ein Integer–Array. Dieses hat dieselbe Länge wie die Integer–Arrays in der Auflistung, auf die der Operator angewandt wurde. Das Argument stellt eine Permutation der Indizes dar. Bezüglich dieser Permutation sollen die Integer-Arrays der Auflistung neu geordnet werden, was somit einer einfachen Umordnung der Attribute entspricht.

Teilanfrageoperatoren (Suboperatoren):

- *InitialOperator*: bekommt als Argument einen Integer *domSize*, der die Domaingröße des ersten Attributs des Anfrageteilgraphen angibt. Daraus wird eine neue Auflistung von Integer-Arrays erstellt. Die Auflistung erhält $domSize - 1$ Integerarrays der Länge 1. Das erste Array hat an der Indexposition 0 den Wert 0. Das letzte Array hat an der Indexposition 0 den Wert $domSize - 1$.
- *CopyAtributeOperator*: bekommt als Argument einen Integer *index*, der die zu kopierende Indexposition angibt. Der Operator vergrößert jedes Array der Auflistung, auf die er angewandt wurde, um eins. An die neue Indexposition wird der Wert der Indexposition *index* kopiert.
- *Cut1Operator*: bekommt als Argument einen Integer *index* und ein *ABinaryIntRelation*–Objekt *relation*. Für jedes Array der Auflistung, auf die der Operator angewandt wurde, bestimmt er den Wert des Arrays an der Stelle *index*. Mit diesem Wert x wird die Cut1-Methode des *ABinaryIntRelation*–Objekts aufgerufen, um alle y zu erhal-

ten, für die $(x, y) \in relation$ gilt. Die Ergebnisauflistung enthält für jedes Array a der Originalauflistung mit x an der Indexposition *index* und jedes y, das $(x, y) \in relation$ erfüllt, ein neues um eins vergrößertes Array, das die Werte des Arrays a enthält und an der neuen Indexposition den Wert von y trägt.

- *Cut2Operator*: Vgl. auch *Cut1Operator*: bekommt als Argument einen Integer *index* und ein *ABinaryIntRelation*–Objekt *relation*. Für jedes Array der Auflistung, auf die der Operator angewandt wurde, bestimmt er den Wert des Arrays an der Stelle *index*. Mit diesem Wert x wird die Cut2-Methode des *ABinaryIntRelation*–Objekts aufgerufen, um alle y zu erhalten, für die $(y, x) \in relation$ gilt. Die Ergebnisauflistung enthält für jedes Array a der Originalauflistung mit x an der Indexposition *index* und jedes y, das $(y, x) \in relation$ erfüllt, ein neues um eins vergrößertes Array, das die Werte des Arrays a enthält und an der neuen Indexposition den Wert von x trägt.
- *DeleteAttributeOperator*: bekommt als Argument einen Integer *index*. Der Operator löscht aus jedem Array der Auflistung, auf die er angewandt wurde, den Wert an der Stelle *index* und verkürzt dabei jedes Array um eins.
- *LabelOperator*: bekommt als Argument einen Integer *index* und ein Objekt *labelRelation* vom Typ *VertexLabel1IntRelationSet*. Für jedes Array der Auflistung, auf die er angewandt wurde, bestimmt er den Wert x des Arrays an der Stelle *index*. Zu diesem Wert x wird mit der Label1(int x)-Methode des *VertexLabel1IntRelationSet*–Objekts der dazugehörige Labelwert y bestimmt. Der Operator vergrößert jedes Array der Auflistung, auf die er aufgerufen wurde, um eins. Der Wert der neuen Indexposition ist y.
- *RevLabelOperator*: Umkehrung des *LabelOperator*: bekommt als Argument einen Integer *index* und ein *VertexLabel1IntRelationSet*–Objekt *labelRelation*. Für jedes Array der Auflistung, auf die er angewandt wurde, bestimmt er den Wert x des Arrays an der Stelle *index*. Zu diesem Labelwert x wird mit der LabelRelation(int x)-Methode des *VertexLabel1IntRelationSet*–Objekts jeder Knotenwert y bestimmt, der das Label x trägt. Die Ergebnisauflistung enthält für jedes Array a der Originalauflistung mit x an der Indexposition *index* für jedes y, das $(y, x) \in$ LabelRelation($labelRelation$) erfüllt, ein neues um eins vergrößertes Array, das die Werte des Arrays a enthält und an der neuen Indexposition den Wert von y trägt.
- *SelectionOperator*: bekommt als Argumente zwei Integer, *index* und *value*. Der Operator entfernt aus der Auflistung, auf die er angewandt wurde, alle Arrays, deren Wert an der Indexposition *index* ungleich *value* ist.
- *BinaryContainsOperator*: bekommt als Argumente zwei Integer, *index1* und *index2*, sowie ein *ABinaryIntRelation*–Objekt relation. Der Operator entfernt aus der Auflistung, auf die er angewandt wurde, alle Arrays a, für die gilt,
  dass $(a[index1], a[index2]) \notin relation$.

Mit den nun vorgestellten Operatoren möchten wir, wie bereits erwähnt, als Erstes die Teilanfragegraphen auswerten. Das Endergebnis der Anfrage setzen wir aus den Teilergebnissen der Teilanfragegraphen mit Hilfe des CartesianProductOperator zusammen. Danach wenden wir den ChangeOrderOperator an, um die Reihenfolge der Spalten der Ergebnistabelle anzupassen.

3.4.4. Optimierte und nicht optimierte Auswertung des Teilanfragegraphen

Ziel der Auswertung des Anfragegaphen ist es, die Liste der Operatoren zu bestimmen, die aufeinander angewandt das Ergebnis der Anfrage liefern. In der optimierten Auswertung bestimmen wir zunächst alle Operatoren und führen dann eine statische Optimierung derart durch, dass wir eine Umordnung der Operatoren vornehmen, um Projektionen und Selektionen möglichst früh durchzuführen. Die nicht optimierte Auswertung führt keine solche Umordnung durch. Hier wird jeder Operator nach seiner Erstellung sofort angewandt. Diese Variante eignet sich daher besser, um die Funktionsweisen der Operatoren zu betrachten.

3.4.5. Auswertung eines Teilanfragegraphen — nicht optimiert

Der Zusatz — nicht optimiert — bedeutet, dass wir zwar eine Liste der Operatoren aufbauen, diese Reihenfolge der Operatoren jedoch nicht optimieren. Jeder Operator wird nach seiner Erstellung sofort auf das aktuelle Zwischenergebnis angewandt. Eine optimierte Auswertung betrachten wir später.

1. Vorbereitung: Wir bestimmen die Wurzel des Teilanfragegraphen. Die Wurzel ist der Anfrageknoten, der keine eingehende Cut-Kante besitzt. Wir erzeugen eine leere Liste für die Operatoren.
2. Initialisierung: Wir nehmen das erste Relationsattribut des Wurzel-Anfrageknotens, bestimmen dessen domsize und initialisieren unser erstes Zwischenergebnis über den *InitialOperator*. Der Liste der Operatoren fügen wir den *InitialOperator* hinzu.
3. Rekursive Tiefensuche mit Join-Anwendung: Wir starten eine rekursiv aufgebaute Tiefensuche an der Wurzel. Die Tiefensuche geht die Liste der Relationsattribute des aktuellen Knotens durch. Für jedes im Ergebnis noch nicht enthaltende Relationsattribut wird der *CopyAttributOperator* auf den Index eines im Ergebnis bereits enthaltenen Relationsattributs angewandt. Der *CopyAttributOperator* wird der Liste der Operatoren hinzugefügt. Danach werden die ausgehenden Kanten betrachtet. Für jede Cut–Kante wird der Index des Quell-Relationsattributs, der Graphname, der Cut–Typ und der Relationstyp bestimmt und ein entsprechender Operator (*Cut1Operator*, *Cut2Operator*, *LabelOperator* oder *RevLabelOperator*) erstellt und auf das Zwischenergebnis angewandt. Der Operator wird zur Liste der Operatoren hinzugefügt. Danach erfolgt der rekursive Aufruf auf den Anfrageknoten, der das Ziel-Relationsattribut enthielt. Die Rekursion terminiert an Anfrageknoten, die keine ausgehenden Kanten besitzen.
4. Anwendung „echter“ Selektionen: Wir gehen die Liste der Selektionsobjekte unserer Anfrage durch und führen dabei die „echten“ Selektionen (Integer- und Stringselektionen) durch. Dazu wird der SelectionOperator verwendet. Jeder *SelectionOperator* wird der Liste der Operatoren hinzugefügt.
5. Anwendung der Contains-Selektionen: Alle Selektionen der Contains-Kanten des Graphen werden über den *BinaryContainsOperator* ausgeführt. Jeder *BinaryContains-Operator* wird der Liste der Operatoren hinzugefügt.
6. Anwendung der Projektionen: Wir gehen die Liste der Projektionsobjekte unserer

Anfrage durch. Für jedes Relationsattribut, auf das nicht projiziert wird, wird eine *DeleteAttributeOperator* mit dem Index des Relationsattributs auf das Zwischenergebnis angewandt. Die Operatoren werden in absteigender Reihenfolge bezüglich ihres zu löschenden Indices geordnet und dann der Liste hinzugefügt. Der Grund hierfür soll an einem Beispiel veranschaulicht werden: Seien die zu löschenden Indices 2,3 und 5. Löscht man zuerst die Indexposition 2, so müssten 3 und 5 jeweils um eins reduziert werden. Um diese Korrektur(en) vorerst einzusparen, löschen wir in der Reihenfolge 5, dann 3 und zuletzt 2.

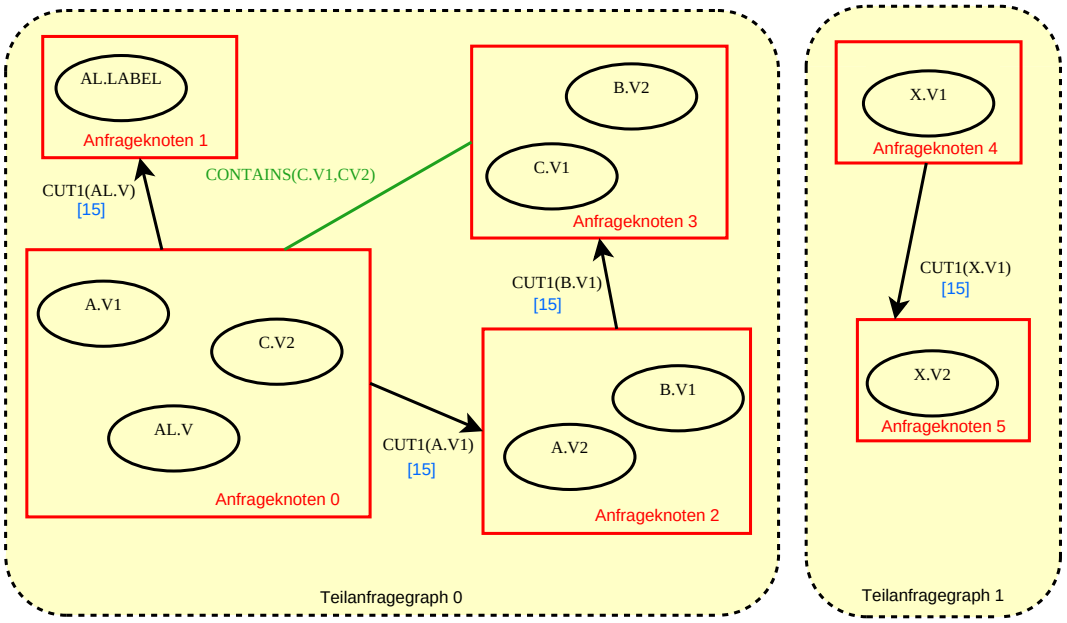
3.4.6. Auswertung des Anfragegraphen — Beispiel

Die gegenwärtige Implementierung bietet die Möglichkeit, sich bei der nicht optimierten Anfrageauswertung jeden Einzelschritt ausgeben zu lassen. Dies möchten wir nun tun, um das Verständnis der nicht optimierten Anfrageauswertung zu verbessern.

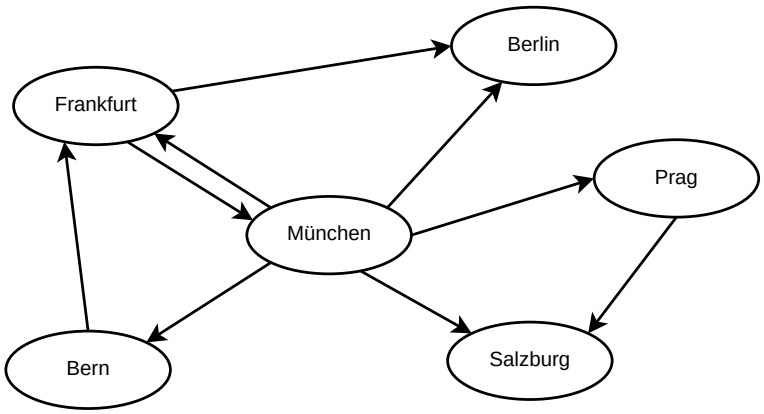
Bisher war uns die folgende Anfrage gegeben:

```
SELECT X.V1,
       AL.V, AL.LABEL
FROM   G1.LABEL AS AL, G1.ADJACENCY AS A,
       G1.ADJACENCY AS B, G1.ADJACENCY AS C,
       G1.ADJACENCY AS X
WHERE  A.V1 = AL.V    AND    A.V2 = B.V1    AND
       B.V2 = C.V1    AND    C.V2 = A.V1    AND
       A.V1 = 4       AND    X.V1 = 0
```

Diese hatten wir zu folgendem Anfragegraph als Zwischenergebnis übersetzt:



Die Anfrage wollen wir nun auf diesen Graphen anwenden:



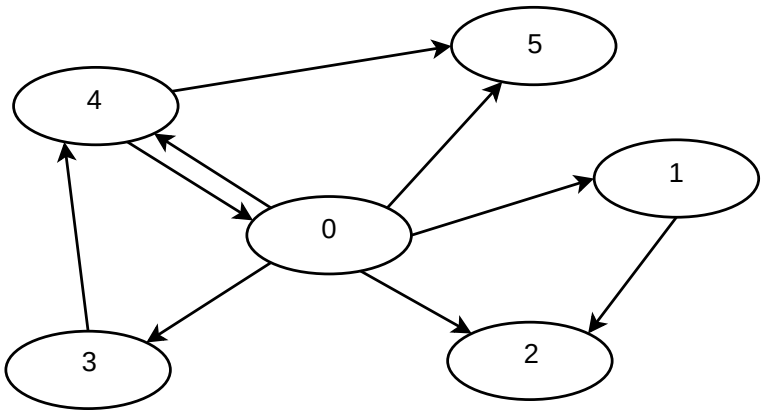
Gespeichert als GraphDescriptor benötigen wir für unsere Anfrage die Kindrelation des *AGraphIntRelationSet*–Objekt, das *VertexLabel1IntRelationSet*–Objekt sowie das *TwoWayDictionary*–Objekt. Um die Begrifflichkeiten etwas zu vereinfachen sprechen wir im folgenden Abschnitt lediglich vom Wörterbuch, der Kindrelation und der Labelrelation:

Labelrelation: {(0,0), (1,1), (2,2), (3,3), (4,4) (5,5)}

Wörterbuch:
{(0,"München"), (1,"Prag"), (2,"Salzburg"), (3,"Bern"), (4,"Frankfurt"), (5,"Berlin")}

Kindrelation: {(0,1), (0,2), (0,3), (0,4), (0,5), (1,2), (3,4), (4,0), (4,5)}

Ersetzen wir im Graphen die Städtenamen durch ihre Integerwerte bzgl. des Wörterbuches, dann erhalten wir folgenden Integer–Graphen:



Teilanfragegraph 0:

Zwischenergebnis für Teilanfragegraph 0:

Liste der Operatoren: {}

Schritt 1: Vorbereitung — Wurzelbestimmung

Die Wurzel von Teilanfragegraph 0 ist der Anfrageknoten 0.

Schritt 2: Initialisierung

Das erste Relationsattribut des Anfrageknotens 0 sei A.V1. Die domsize der Kindrelation ist 6. Wir erzeugen einen *InitialOperator* mit domsize = 6, wenden ihn an und fügen ihn der Liste der Operatoren hinzu:

Zwischenergebnis für Teilanfragegraph 0:

```
AL.V
0
1
2
3
4
5
AL.V
```

Liste der Operatoren: { *InitialOperator* mit domsize = 6 }

Schritt 3: Rekursive Tiefensuche mit Join-Anwendung

Aufruf auf die Wurzel:
Betrachtung der Relationsattribute der Wurzel:
AL.V ist bereits im Zwischenergebnis enthalten.
A.V1 ist nicht im Zwischenergebnis enthalten –> *CopyAttributeOperator* mit SourceIndex 0.

Zwischenergebnis für Teilanfragegraph 0:

```
AL.V A.V1
0    0
1    1
2    2
3    3
4    4
5    5
AL.V A.V1
```

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0 }

C.V2 ist nicht im Zwischenergebnis enthalten –> *CopyAttributeOperator* mit SourceIndex 0.

Zwischenergebnis für Teilanfragegraph 0:

```
AL.V A.V1 C.V2
0    0    0
1    1    1
2    2    2
3    3    3
4    4    4
5    5    5
AL.V A.V1 C.V2
```

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0 }

Die Liste der Relationsattribute der Wurzel ist abgearbeitet.

Betrachtung der ausgehenden Kanten der Wurzel:

Kante: CUT1(AL.V) –> *LabelOperator* auf Index 0 mit der Labelrelation

Für jeden Wert von AL.V wird AL.LABEL aus der Labelrelation bestimmt. Betrachten wir erneut die Labelrelation:

Labelrelation: {(0,0), (1,1), (2,2), (3,3), (4,4) (5,5)}

Bsp.: Knoten AL.V = 3 trägt Label AL.LABEL = 3. Das Zwischenergebnis sieht aus wie nach einem *CopyAttributeOperator*, ist jedoch aus dem *LabelOperator* auf Index 0 entstanden. Die Symmetrie folgt aus der Eigenschaft speziellen Eigenschaft dieser Beispiel–Labelrelation, dass nämlich für alle $(x, y) \in$ Labelrelation gilt, dass $x = y$.

Zwischenergebnis für Teilanfragegraph 0:

AL.V	A.V1	C.V2	AL.LABEL
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
AL.V	A.V1	C.V2	AL.LABEL

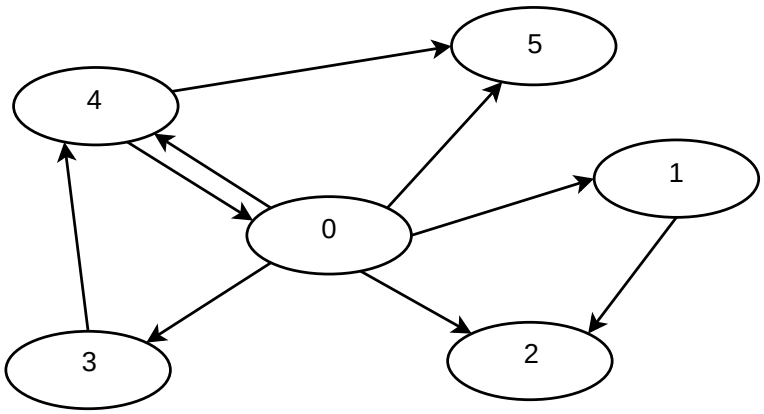
Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation }

Rekursiver Aufruf auf Anfrageknoten 1:
Betrachtung der Relationsattribute des Anfrageknoten 1: nichts zu tun!
Betrachtung der ausgehenden Kanten des Anfrageknoten 1: nichts zu tun!
Der rekursive Aufruf auf Anfrageknoten 1 ist beendet.

Fortsetzung der Betrachtung der ausgehenden Kanten der Wurzel:

Kante: CUT1(A.V1) –> *Cut1Operator* auf Index 1 mit der Kindrelation

Für jeden Wert von A.V1 wird A.V2 aus der Kindrelation bestimmt. Blicken wir dazu einfach auf den Graphen:



Oder auf die Kindrelation: {(0,1), (0,2), (0,3), (0,4), (0,5), (1,2), (3,4), (4,0), (4,5)}

Bsp.: Knoten A.V1 = 4 liefert A.V2 = {0,5)}. A.V1 = 4 erzeugt zwei Zeilen im Zwischenergebnis. Bsp.: Knoten A.V1 = 5 liefert A.V2 = {} und erzeugt keine Zeile.

Zwischenergebnis für Teilanfragegraph 0:

AL.V	A.V1	C.V2	AL.LABEL	A.V2
0	0	0	0	1
0	0	0	0	3
0	0	0	0	4
0	0	0	0	2
0	0	0	0	5
1	1	1	1	2
3	3	3	3	4
4	4	4	4	5
4	4	4	4	0
AL.V	A.V1	C.V2	AL.LABEL	A.V2

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation,
Cut1Operator auf Index 1 mit der Kindrelation }

Rekursiver Aufruf auf Anfrageknoten 2:
Betrachtung der Relationsattribute des Anfrageknoten 2:
A.V2 ist bereits im Zwischenergebnis enthalten.
B.V1 ist nicht im Zwischenergebnis enthalten –> *CopyAttributeOperator* mit SourceIndex 4.

Zwischenergebnis für Teilanfragegraph 0:

AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1
0	0	0	0	1	1
0	0	0	0	3	3
0	0	0	0	4	4
0	0	0	0	2	2
0	0	0	0	5	5
1	1	1	1	2	2
3	3	3	3	4	4
4	4	4	4	5	5
4	4	4	4	0	0
AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1

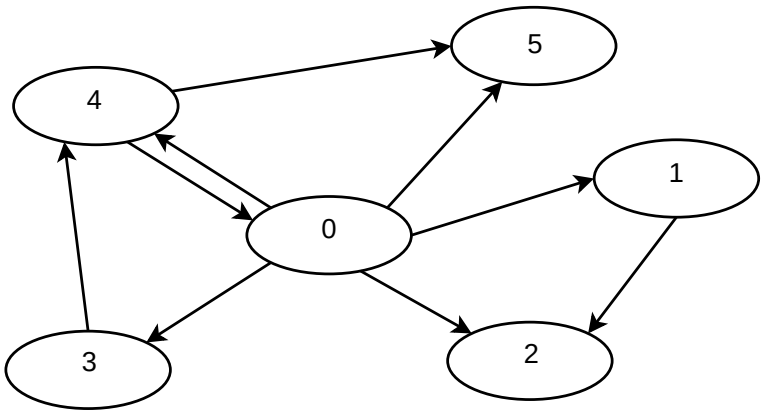
Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation,
Cut1Operator auf Index 1 mit der Kindrelation,
CopyAttributeOperator mit SourceIndex 4 }

Die Liste der Relationsattribute des Anfrageknoten 2 ist abgearbeitet.

Betrachtung der ausgehenden Kanten des Anfrageknoten 2:

Kante: CUT1(B.V1) –> *Cut1Operator* auf Index 5 mit der Kindrelation

Betrachten wir dazu wieder den Graphen:



Oder die Kindrelation: {(0,1), (0,2), (0,3), (0,4), (0,5), (1,2), (3,4), (4,0), (4,5)}

Zwischenergebnis für Teilanfragegraph 0:

AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1	B.V2
0	0	0	0	1	1	2
0	0	0	0	3	3	4
0	0	0	0	4	4	5
0	0	0	0	4	4	0
3	3	3	3	4	4	5
3	3	3	3	4	4	0
4	4	4	4	0	0	1
4	4	4	4	0	0	3
4	4	4	4	0	0	4
4	4	4	4	0	0	2
4	4	4	4	0	0	5
AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1	B.V2

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation,
Cut1Operator auf Index 1 mit der Kindrelation,
CopyAttributeOperator mit SourceIndex 4,
Cut1Operator auf Index 5 mit der Kindrelation }

Rekursiver Aufruf auf Anfrageknoten 3:
Betrachtung der Relationsattribute des Anfrageknoten 3:
B.V2 ist bereits im Zwischenergebnis enthalten.
C.V1 ist nicht im Zwischenergebnis enthalten –> *CopyAttributeOperator* auf Index 6.

Zwischenergebnis für Teilanfragegraph 0:

AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1	B.V2	C.V1
0	0	0	0	1	1	2	2
0	0	0	0	3	3	4	4
0	0	0	0	4	4	5	5
0	0	0	0	4	4	0	0
3	3	3	3	4	4	5	5
3	3	3	3	4	4	0	0
4	4	4	4	0	0	1	1
4	4	4	4	0	0	3	3
4	4	4	4	0	0	4	4
4	4	4	4	0	0	2	2
4	4	4	4	0	0	5	5
AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1	B.V2	C.V1

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation,
Cut1Operator auf Index 1 mit der Kindrelation,
CopyAttributeOperator mit SourceIndex 4,
Cut1Operator auf Index 5 mit der Kindrelation,
CopyAttributeOperator auf SourceIndex 6 }

Betrachtung der ausgehenden Kanten des Anfrageknoten 3: nichts zu tun!
Der rekursiver Aufruf auf Anfrageknoten 3 ist beendet.

Betrachtung der ausgehenden Kanten des Anfrageknoten 2:
keine weiteren ausgehenden Kanten –> nichts zu tun!
Der rekursiver Aufruf auf Anfrageknoten 2 ist beendet.

Betrachtung der ausgehenden Kanten der Wurzel:
keine weiteren ausgehenden Kanten –> nichts zu tun!

Schritt 4: Anwendung der „echten“ Selektionen

Liste der Selektionen: (A.V1 = 4, X.V1 = 2)

Selektion A.V1 = 4 –> *SelectionOperator* mit Index 1 und Wert 4

Jede Zeile, in der A.V1 ungleich 4 ist, wird entfernt.

Zwischenergebnis für Teilanfragegraph 0:

AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1	B.V2	C.V1
4	4	4	4	0	0	1	1
4	4	4	4	0	0	3	3
4	4	4	4	0	0	4	4
4	4	4	4	0	0	2	2
4	4	4	4	0	0	5	5
AL.V	A.V1	C.V2	AL.LABEL	A.V2	B.V1	B.V2	C.V1

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation,
Cut1Operator auf Index 1 mit der Kindrelation,
CopyAttributeOperator mit SourceIndex 4,
Cut1Operator auf Index 5 mit der Kindrelation,
CopyAttributeOperator auf SourceIndex 6,
SelectionOperator mit Index 1 und Wert 4 }

Die Selektion $X.V1 = 2$ ist für diesen Teilanfragegraphen irrelevant.

Schritt 5: Anwendung der Contains-Selektionen

Der Teilanfragegraph 0 verfügt über eine Contains-Kante. Diese verbindet Anfrageknoten 0 und Anfrageknoten 3 bzw. die Relationsattribute der C Relation.

Wir erzeugen einen *BinaryContainsOperator* mit den Indices 7 für C.V1 und 2 für C.V2 und der Kindrelation.

Jede Zeile des Zwischenergebnisses enthält ein Paar (C.V1, C.V2). Ist dieses Paar nicht Element der Kindrelation, wird die Zeile entfernt. Bsp.: Die Zeile ⟨4440044⟩ enthält das Paar (4,4). $(4, 4) \notin$ Kindrelation. Die Zeile wird entfernt.

Zwischenergebnis für Teilanfragegraph 0:

```
AL.V A.V1 C.V2 AL.LABEL A.V2 B.V1 B.V2 C.V1
4    4    4    4        0    0    3    3
AL.V A.V1 C.V2 AL.LABEL A.V2 B.V1 B.V2 C.V1
```

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation,
Cut1Operator auf Index 1 mit der Kindrelation,
CopyAttributeOperator mit SourceIndex 4,
Cut1Operator auf Index 5 mit der Kindrelation,
CopyAttributeOperator auf SourceIndex 6,
SelectionOperator mit Index 1 und Wert 4,
BinaryContainsOperator mit Indices 7 und 2 }

Schritt 6: Anwendung der Projektionen

Liste der Projektionen: (AL.V, AL.LABEL, X.V1)

Index von AL.V: 0
Index von AL.LABEL: 3
Index von X.V1: für diesen Teilanfragegraphen irrelevant.

Anlegen und Anwenden von *DeleteAttributeOperator* mit Index x,
wobei $x \in \{0, \ldots, 7\} \backslash \{0, 3\}$

Wir fügen diese in absteigender Reihenfolge der Liste der Operatoren hinzu.

Endergebnis für Teilanfragegraph 0:

```
AL.V AL.LABEL
4    4
AL.V AL.LABEL
```

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
CopyAttributeOperator mit SourceIndex 0,
CopyAttributeOperator mit SourceIndex 0,
LabelOperator auf Index 0 mit der Labelrelation,
Cut1Operator auf Index 1 mit der Kindrelation,
CopyAttributeOperator mit SourceIndex 4,
Cut1Operator auf Index 5 mit der Kindrelation,
CopyAttributeOperator auf SourceIndex 6,
SelectionOperator mit Index 1 und Wert 4,
BinaryContainsOperator mit Indices 7 und 2,
DeleteAttributeOperator mit Index 7,
DeleteAttributeOperator mit Index 6,
DeleteAttributeOperator mit Index 5,
DeleteAttributeOperator mit Index 4,
DeleteAttributeOperator mit Index 2,
DeleteAttributeOperator mit Index 1 }

Teilanfragegraph 1:

Da eine ausführliche Behandlung des Teilanfragegraphen 1 keinen wesentlicher Verständniszuwachs generieren würde, geben wir das Endergebnis der Auswertung des Teilanfragegraphen 1 direkt an.

Endergebnis für Teilanfragegraph 1:

```
X.V1
0
0
0
0
0
X.V1
```

Liste der Operatoren: { *InitialOperator* mit domsize = 6,
Cut1Operator auf Index 0 mit der Kindrelation,
SelectionOperator mit Index 0 und Wert 0,
DeleteAttributeOperator mit Index 1 }

Nachdem nun alle Teilanfragegraphen ausgewertet sind, können deren Ergebnisse jetzt über den *CartesianProductOperator* zusammengeführt werden. Dabei wird der *CartesianProductOperator* auf die erste Auflistung von Integerarrays aufgerufen. Die zweite Auflistung bekommt er als Argument.

Zwischenergebnis des Anfragegraphen:

```
AL.V AL.LABEL X.V1
4    4        0
4    4        0
4    4        0
4    4        0
4    4        0
4    4        0
AL.V AL.LABEL X.V1
```

Um die Ergebnisspalten so anzuordnen, dass sie die Reihenfolge der Projektionen der Anfrage erfüllen, bestimmen wir für jedes Relationsattribut den Index bezüglich der Reihenfolge der Anfrage.

Reihenfolge der Relationsattribute in der Anfrage: (X.V1, AL.V, AL.LABEL)

Index von AL.V: 1 Index von Al.LABEL: 2 Index von X.V1: 0

Die Permutation (1,2,0) wird einem *ChangeOrderOperator* in Form eines Arrays übergeben und auf des Zwischenergebnis angewandt.

Endergebnis des Anfragegraphen ohne Strings:

```
X.V1 AL.V AL.LABEL
0    4    4
0    4    4
0    4    4
0    4    4
0    4    4
X.V1 AL.V AL.LABEL
```

Damit ist die Anwendung aller Operatoren beendet. Die Werte der Relationsattribute, die eigentlich vom Typ String sind, werden mit Hilfe des Wörterbuchs übersetzt. AL.LABEL ist ein String Relationsattribut. Das Wörterbuch liefert für den Integer 4 den String „Frankfurt“.

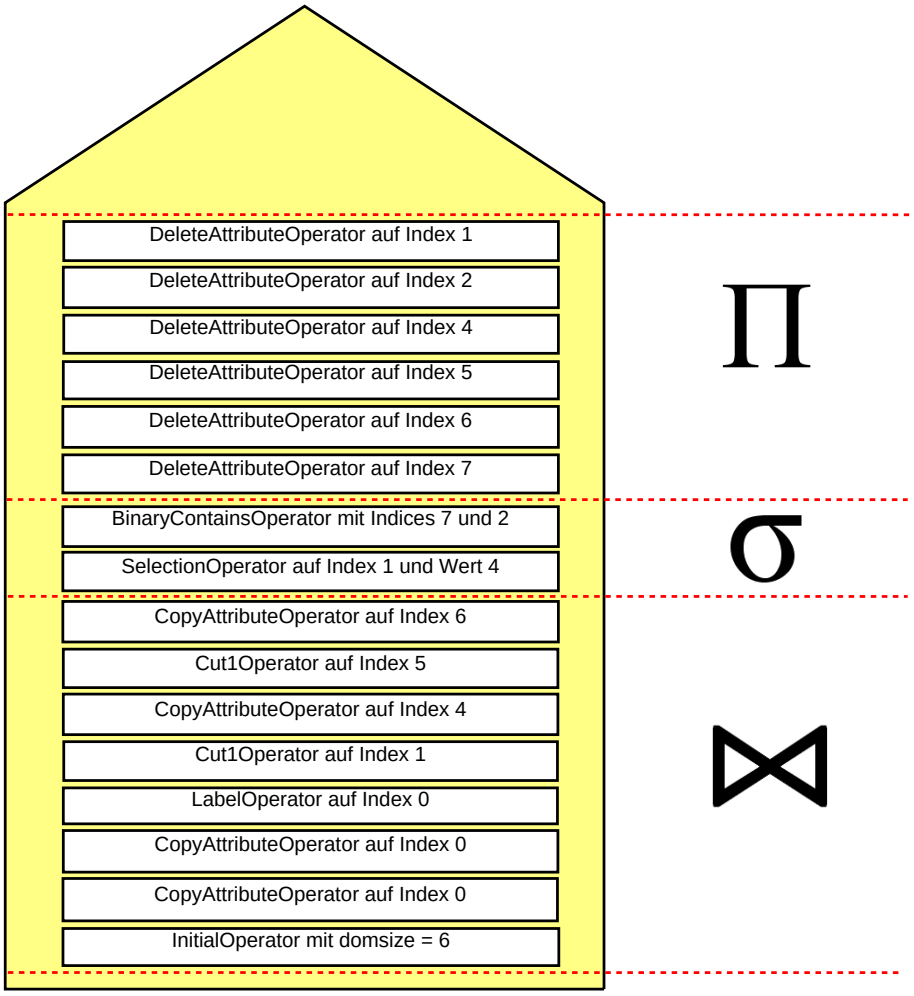
Endergebnis des Anfragegraphen mit Strings:

```
X.V1 AL.V AL.LABEL
0    4    Frankfurt
0    4    Frankfurt
0    4    Frankfurt
0    4    Frankfurt
0    4    Frankfurt
X.V1 AL.V AL.LABEL
```

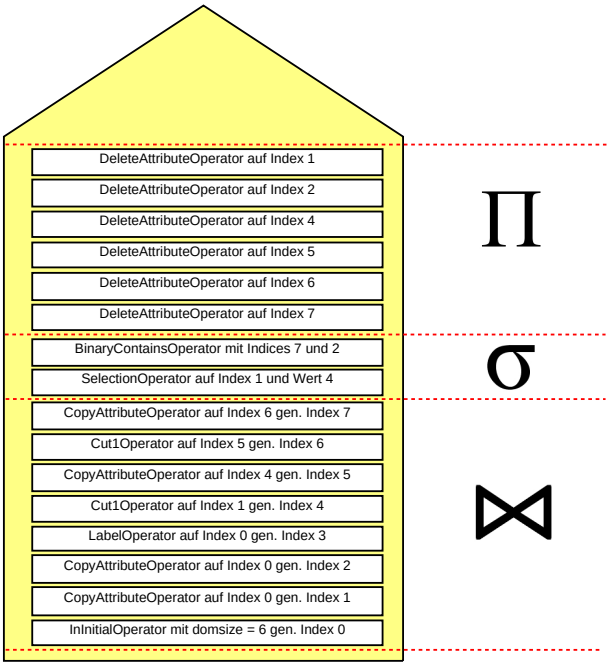
3.4.7. Optimierung der Anwendungsreihenfolge der Suboperatoren

Betrachten wir noch einmal die Auswertung des Anfrageteilgraphen 0.

Dazu stellen wir die Liste der Operatoren als Ast eines Operatorbaums grafisch dar. Dabei lassen sich die Operatoren in drei Gruppen aufteilen: die unterste Operatorengruppe führt die Joins durch, die mittlere Gruppe die Selektionen und die oberste Gruppe die Projektionen:

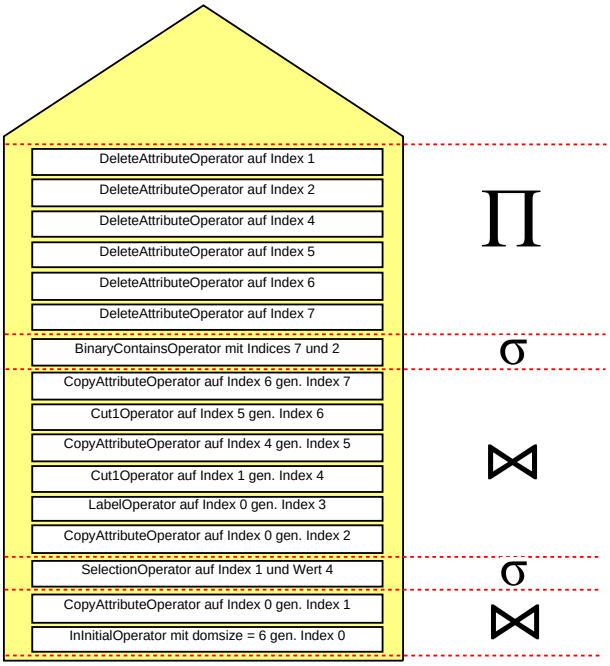


Ziel der Optimierung der Reihenfolge soll es sein, die Selektionen und Projekten möglichst früh auszuführen. Damit würden sie im Operatorbaum nach unten wandern. Als Vorbereitung bestimmen wir für die Operatoren der Joingruppe die Indexpositionen, die sie generieren. Diese Indexposition ist gleich ihrer eigenen Indexposition in der Liste der Operatoren:

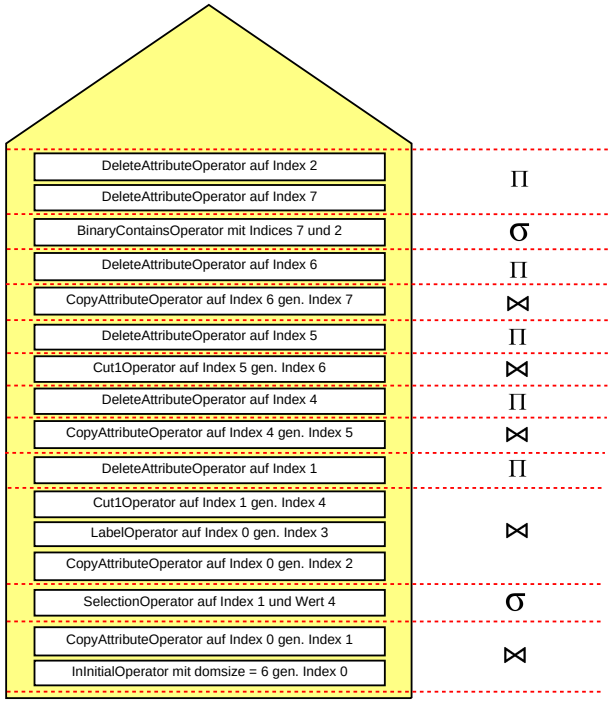


Nun können wir die Operatoren der Gruppe der Selektionen nach folgenden Regeln verschieben:

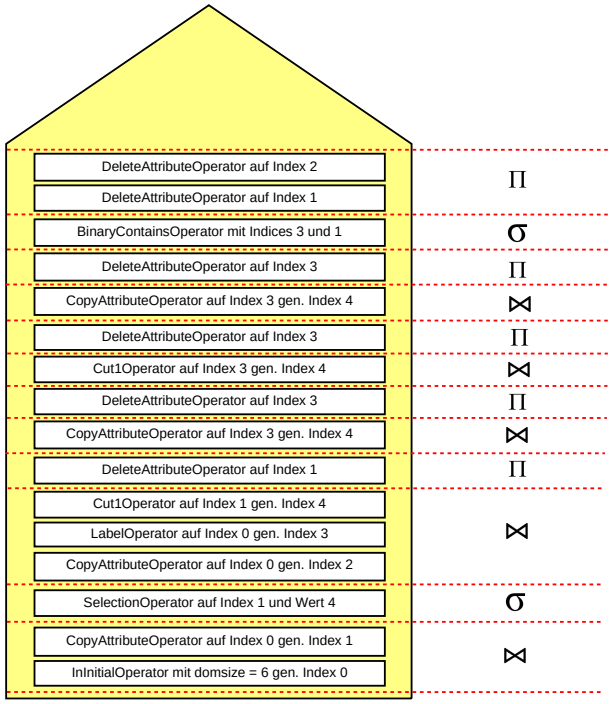
- Jeder *SelectionOperator* wird solange um eine Position nach unten verschoben, bis der untere Nachbar den Index generiert, auf den der SelectionOperator angewandt werden soll.
- Jeder *BinaryContainsOperator* wird solange um eine Position nach unten verschoben, bis der untere Nachbar einen der Indices generiert, auf den der BinaryContainsOperator angewandt werden soll.



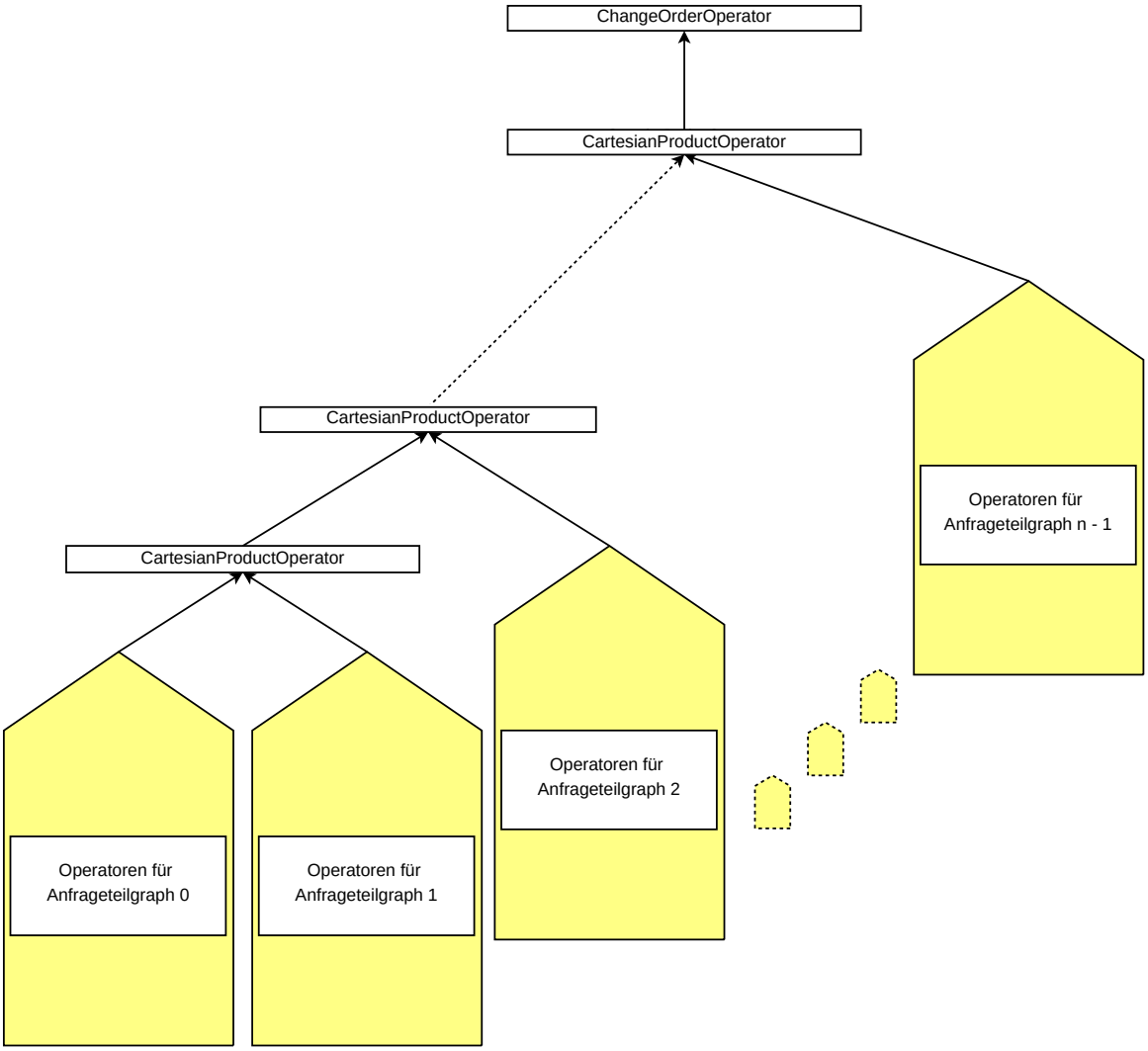
Schließlich verschieben wir noch die Projektionsoperatoren. Dazu wird jeder *DeleteAttributeOperator* auf Index x solange um eine Position nach unten verschoben, bis der untere Nachbar entweder auf dem Index x eine Operation durchführt oder den Index x selbst erzeugt.



Als Letztes müssen wir eine Korrektur der Indexpositionen vornehmen. Wir gehen dazu die Operatoren von unten nach oben durch. Treffen wir auf einen DeleteAttributeOperator auf Index x, so reduzieren wir bei allen nachfolgenden Operatoren deren Indexpositionen, sofern sie größer als x sind, um eins:



3.4.8. Allgemeiner Operatorbaum



4. Die Konsole

Nachdem wir uns mit dem Einlesen und Abfragen von Daten vertraut gemacht haben, möchten wir nun eine einfache Konsole zur Verfügung stellen, mit der später Tests und Auswertungen durchgeführt werden können. Die Konsole sei eine C# Konsole, in die alle Eingaben in Form von Strings gemacht werden. Alle Ausgaben (Ergebnisse, Bestätigungen, Fehler) werden ebenfalls auf die Konsole zurückgegeben. Die Konsole läuft als Endlosschleife und kann lediglich durch einen festgelegten Befehl beendet werden. Gefolgt auf ein Größerzeichen („>“) am Zeilenanfang können Befehle eingegeben werden. Der Konsole stellen wir ein *GraphDescriptorDataBase*–Objekt zur Seite, welches die eingelesenen und für Anfragen zu Verfügung stehenden Graphen verwaltet.

4.1. Die Klasse *GraphDescriptorDataBase*

Die Klasse *GraphDescriptorDataBase* führt eine Liste von *GraphDescriptor*–Objekten. *GraphDescriptor*–Objekte entstehen aus dem Parsing von XML– oder RDF–Daten (siehe 2.2.2). Auf die Liste wird kein direkter Zugriff ermöglicht, es stehen jedoch folgende Methoden zur Verfügung:

- hasGraphWithName(string *name*) gibt den Wahrheitswert zurück, ob sich in der Liste bereits ein *GraphDescriptor* mit Namen *name* befindet.
- addableDescriptor(GraphDescriptor *desc*) gibt den Wahrheitswert zurück, ob sich der *GraphDescriptor desc* in die Liste der *GraphDescriptor*–Objekte aufnehmen lässt. Dabei wird der Name von *desc* bestimmt und die hasGraphWithName-Methode benutzt.
- addDescriptor(GraphDescriptor *desc*) überprüft mit Hilfe der addableDescriptor-Methode, ob *desc* in die Liste der *GraphDescriptor*–Objekte aufnehmen lässt. Ist dies der Fall, wird *desc* der Liste hinzugefügt.
- removeDescriptorByName(string *name*), sofern sich in der Liste der *GraphDescriptor*–Objekte mit Namen *name* befindet, wird dieses aus der Liste entfernt.
- getGraphDescriptorByName(string *name*), sofern sich in der Liste der *GraphDescriptor*–Objekte mit Namen *name* befindet, wird dieses zurückgegeben.

Zusätzlich implementiert die Klasse *GraphDescriptorDataBase* das *IEnumerable*–Interface, so dass mit einer foreach Schleife über die *GraphDescriptor*-Objekte iteriert werden kann. So lassen sich sehr einfach alle verfügbaren *GraphDescriptor*–Objekte der *GraphDescriptorDataBase* auflisten.

4.2. Die Klasse *QueryConsole*

Die Behandlung der Eingaben auf der Konsole wird in der Klasse *QueryConsole* behandelt. Folgende Befehle werden unterstützt:

EXIT

Beendet das Programm und schließt die Konsole.

LOADXML

Unter Angabe einer Adresse der zu parsenden XML-Datei, eines Namens für den Graphen und einer IDIDREF-Option lädt LOADXML einen Graphen aus einer XML-Datei in die GraphDescriptorDataBase.

Syntax:

```
LOADXML "Graphname ohne Leerzeichen" "Pfad zur XML Datei"
<IDIDREFOption>
```

Gültige IDIDREF-Optionen sind: DONTUSEIDIDREF, USEINLINEXSD, USEXSDSCHEMALOCATION, IGNOREIDIDREF

Beispiel:

```
LOADXML "Graph_G1" "Datei1.xml" USEINLINEXSD
```

Kann der Graph erfolgreich aufgenommen werden, antwortet die Konsole mit „Graph added“. Im Fehlerfall erhält man: „Name of graph NOT addable. Exists allready!“

LISTGRAPHS

LISTGRAPHS listet alle in der GraphDescriptorDataBase enthaltenden Graphen auf.

Syntax:

```
LISTGRAPHS
```

DELETEGRAPH

Unter Angabe des zu löschenden Graphen wird dieser aus der GraphDescriptorDataBase entfernt

Syntax:

```
DELETEGRAPH "Graphname ohne Leerzeichen"
```

Beispiel:

```
DELETEGRAPH "Graph_G1"
```

Kann der Graph erfolgreich gelöscht werden antwortet die Konsole mit „Graph with name <Graph_G1> removed.“. Im Fehlerfall erhält man: „There is no graph with name <Graph_G1> to remove!“

SELECT

Um eine Anfrage einzugeben, benutzt man die in 3.2 angegebene Syntax.

Beispiel:

```
SELECT * FROM G1.ADJACENCY WHERE G1.R1 = 2
```

SETDEBUG

Um ausführliche Informationen über das Ergebnis des Parsens oder die Auswertung der Anfrage zu erhalten, ist es möglich, einen Debug-Modus zu aktivieren. Dies geschieht über den SETDEBUG–Befehl. Beim Starten der Konsole ist der Debug–Modus standardmäßig deaktiviert.

Syntax:

```
SETDEBUG ON|OFF
```

Beispiel:

```
SETDEBUG OFF
```

Die Konsole antwortet mit „Debug modus active.“ oder „Debug modus inactive.“.

4.3. Beispielsitzung

```
> LOADXML "G1" "2books.xml" IGNOREIDIDREF
Graph added.

> LISTGRAPHS
Known graphs: 1
Name: <G1> #Vertices: <19>

> SELECT LABEL1.LABEL, LABEL2.LABL FROM G1.ADJACENCY AS R,
G1.LABEL AS LABEL1, G1.LABEL AS LABEL2 WHERE LABEL1.V = R.V1
AND LABEL2 = R.V2 AND LABEL2.LABEL = "Harry␣Potter"

LABEL1.LABEL LABEL2.LABEL
title        Harry Potter
LABEL1.LABEL LABEL2.LABEL

> DELETEGRAPH "G1"
Graph with name <G1> removed.

>LISTGRAPHS
Known graphs: 0

>
```

Die benutzte XML–Datei kennen wir bereits aus Kapitel 2.3. Eine Abbildung der XML–Datei als Graph findet sich auf Seite 15. Dass der Graph über 19 Knoten verfügt lässt sich in der Abbildung leicht nachsehen. Die gestellte Anfrage sucht für den Knoten mit Namen „Harry Potter“ den Namen des Vaterknotens. Auch dies lässt sich aus der Abbildung nachvollziehen.

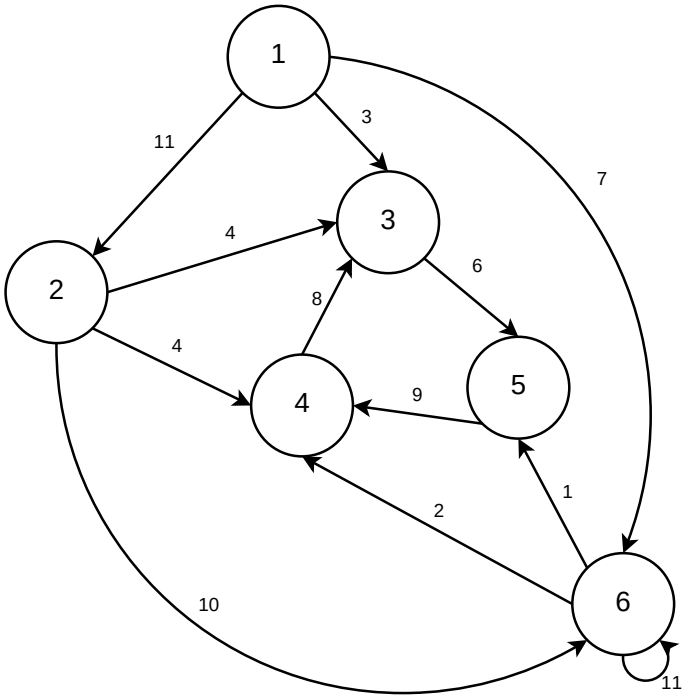
5. Die Algorithmen

5.1. Der Edmonds–Algorithmus

Der Edmonds–Algorithmus[2] bestimmt auf einem gerichteten Graphen ein maximales Branching unter Verwendung einer minimalen Anzahl von Kanten. Ein Branching in einem gerichteten Graphen sei definiert als eine Menge gerichteter Kanten mit der Eigenschaft, dass die Menge keine Zyklen enthält und dass kein Knoten mehr als eine eingehende Kante besitzt. Für einen gerichteten Graphen mit Kantengewichten sei ein maximales Branching definiert als ein Branching, bei dem die Summe aller Kantengewichte maximal ist.

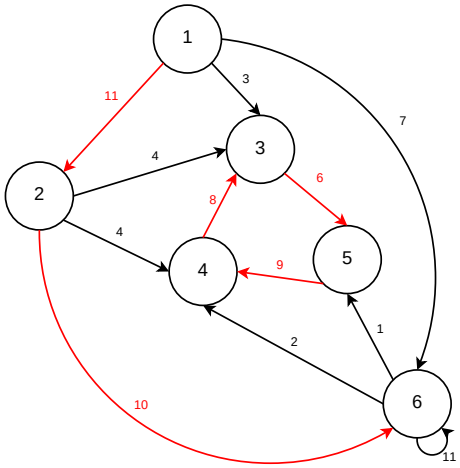
5.1.1. Einführungsbeispiel

Wir wollen den Algorithmus als Erstes an einem Beispiel betrachten. Gesucht sei das resultierende Branching R_0 zu folgendem gegebenen Graph $G_0 = (V(G_0), E(G_0))$ mit $V(G_0) = \{1, 2, 3, 4, 5, 6\}$ und $E(G_0) = \{$(1,2), (1,3), (1,6), (2,3), (2,4), (2,6), (3,5), (4,3), (5,4), (6,4), (6,5), (6,6)$\}$. Die Kantengewichte entnehme man der Abbildung:

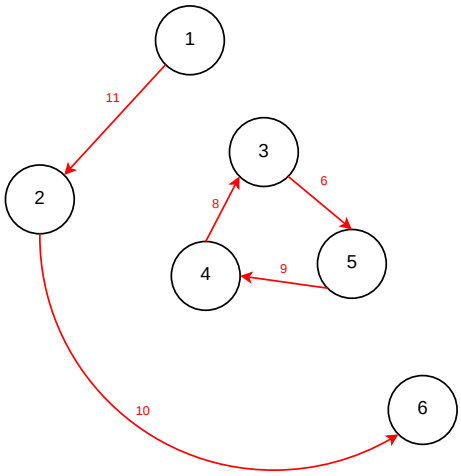


Die Gewichte der Kanten sind bewusst als echt positiv gewählt. Kanten mit negativem Gewicht oder Gewicht gleich Null können die Summe der Kantengewichte nicht vergrößern und fänden keine Beachtung.

Wir bestimmen als Erstes einen Hilfsgraphen B_0 aus G_0. B_0 bekommt alle Knoten von G_0. Für jeden Knoten k in B_0 nehmen wir jeweils nur die schwerste, echt positive, nicht zyklische Kante aus G_0 auf, die nach k führt:



Wir überprüfen, ob der Graph B_0 Zyklen enthält. Wenn nicht, ist B_0 das gesuchte maximale Branching mit minimaler Anzahl an Kanten. B_0:



B_0 enthält den Zyklus $3 \to 5 \to 4 \to 3$. Die eigentliche Idee des Algorithmus ist es den Zyklus zu einem Schrumpfknoten zu kontrahieren, das maximale Branching auf dem Graph mit Schrumpfknoten zu bestimmen und später in einem Expansionsvorgang den Schrumpfknoten wieder in seine Ursprungsknoten aufzulösen.

Wir legen für den gefundenen Zyklus einen neuen Knoten V_C an. Jede Kante $e = (x, y)$, die in einen Knoten des Schrumpfknoten V_C führte, ersetzen wir durch $e' = (x, V_C)$. Zudem merken wir uns e als Vorgänger von e': $\Psi(e') = e$. Für jede e' Kante bestimmen wir das Kantengewicht neu. Um beim Anschluss des Zyklus an den Restgraphen ein maximales Gewicht zu erzeugen, benötigen wir eine möglich schwere Kante in den Zyklus hinein, die dann eine möglich leichte Kante im Zyklus überflüssig macht.

Beispiele:

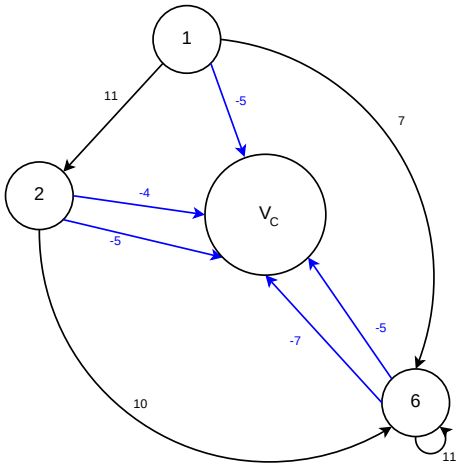
Schließen wir den Zyklus über die Kante (6,4) an, erhält man für den Anschluss des Zyklus ein Gewicht von: $c(6,4) + c(4,3) + c(3,5) = 2 + 8 + 6 = 16$

Schließen wir den Zyklus über die Kante (2,3) an, erhält man für den Anschluss des Zyklus ein Gewicht von: $c(2,3) + c(3,5) + c(5,4) = 4 + 6 + 9 = 19$

Um die Eignung einer in den Zyklus führenden Kante e' zu vergleichen, reicht es also, das Gewicht c für e' nach folgender Formel zu bestimmen:

$c(e') = c(e) - c(\overline{e})$ mit $\overline{e}$ = Kante im Zyklus nach y

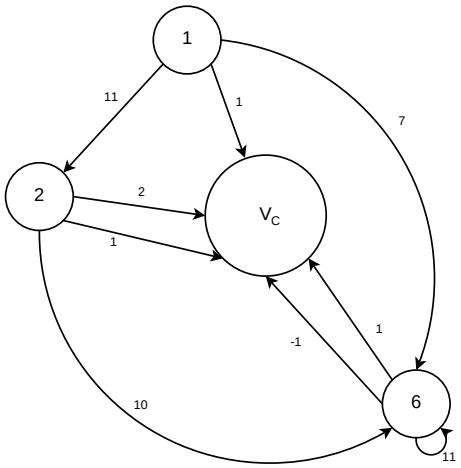
Je größer die Kosten für e', desto größer das summierte Kantengewicht.



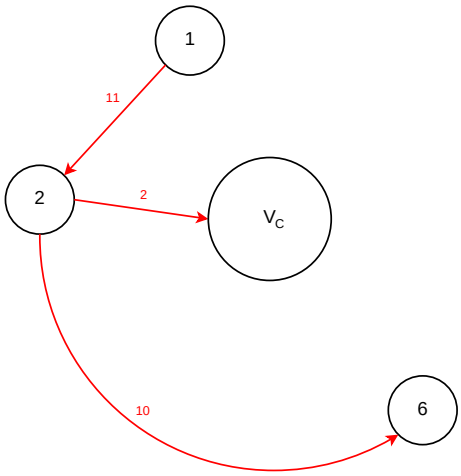
Die „teuerste“ Kante in den Zyklus hinein wäre jetzt die Kante $(2, V_C)$ mit Kantengewicht -2. Jede Kante, die in den Schrumpfknoten führt, bringt uns mindestens das Kantengewicht des leichtesten Kante im Zyklus. Daher normieren wir grundsätzlich alle e' Kanten, indem wir zusätzlich das kleinste Kantengewicht aller Kanten des Zyklus addieren. In diesem Fall 6. Die korrekte Formel für $c(e')$ lautet somit:

$c(e') = c(e) - c(\overline{e}) + c(e_c)$ mit e_c = leichteste Kante im Zyklus

Wir erhalten damit den Graphen G_1:



Wir bestimmen den Hilfsgraphen B_1 aus G_1:



Wir überprüfen, ob der Graph B_1 Zyklen enthält. B_1 enthält keine Zyklen. Damit ist $R_1 = B_1$ ein maximales Branching für G_1. Es muss kein weiterer Schrumpfknoten angelegt werden. Die Kontraktionsphase ist damit beendet.

Für die Expansionsphase initialisieren wir R_0 mit den Kanten von R_1. Dann betrachten wir die Zyklen in B_0. Für jede Kante $e' \in E(R_0)$, die in einen Zyklus C von B_0 führt, ändern wir $E(R_0)$:

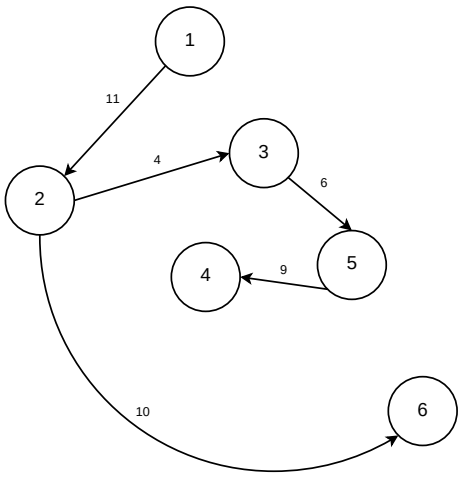
$$E(R_0) := (E(R_0) \backslash e') \cup \Psi(e') \cup (E(C) \backslash \overline{\Psi(e')}$$

Dabei ist $\Psi(e')$ die Vorgängerkante e von e', die wir uns bei der Kontraktion gemerkt hatten und $\overline{\Psi(e')}$ die Kante im Zyklus C, die in den gleichen Knoten führt wie $\Psi(e')$. $E(C)$ ist die Menge der Kanten des Zyklus C.

Die Knoten von R_0 sind gleich den Knoten von G_0:

$$V(R_0) := V(G_0)$$

Somit erhalten wir folgenden Graphen als maximales Branching für G_0:



5.1.2. Der Algorithmus als Pseudocode

INPUT: Gerichteter Graph $G = (V(G), E(G))$. Jede Kante e aus $E(G)$ hat ein Kantengewicht $c(e) > 0$.
OUTPUT: Ein maximales Branching R in G.

```
BEGIN
Setze i := 0
      G_0 := G

while (der aus G_i konstruierte Hilfsgraph B_i
       enthält mindestens einen Zyklus) do
    Kontraktion()
end while

R_i := B_i

while (i > 0) do
    Expansion()
end while

R := R_0
END.
```

Kontraktion()

```
BEGIN
   konstuiere G_{i+1} aus G_i und B_i
   i = i + 1
END.
```

Expansion()

```
BEGIN
R_{i-1} := R_i

for jeden Zyklus C in B_{i-1} do
   bool zyklusBehandelt = false;

   for jede Kante e' in E(R_{i-1}) do

      // Fall 1:
      if e' ist eingehend in C then
         zyklusBehandelt = true;
         E(R_{i-1}) := (E(R_{i-1})\e') ∪ Ψ(e') ∪ (E(C)\overline{Ψ(e')}
      end if

      // Fall 2:
      if e' ist ausgehend aus C then
         zyklusBehandelt = true;
         E(R_{i-1}) := (E(R_{i-1})\e') ∪ Ψ(e') ∪ (E(C)\e_c)
      end if

      // Fall 3:
      if e' kommt mit C nicht in Berührung then
         // nichts zu tun
      end if

   end for

   // Fall 4:
   if (zyklusBehandelt == false)
      E(R_{i-1}) := E(R_{i-1}) ∪ (E(C)\e_c)
   end if

end for
V(R_{i-1}) := V(G_{i-1})
i := i - 1

END.
```

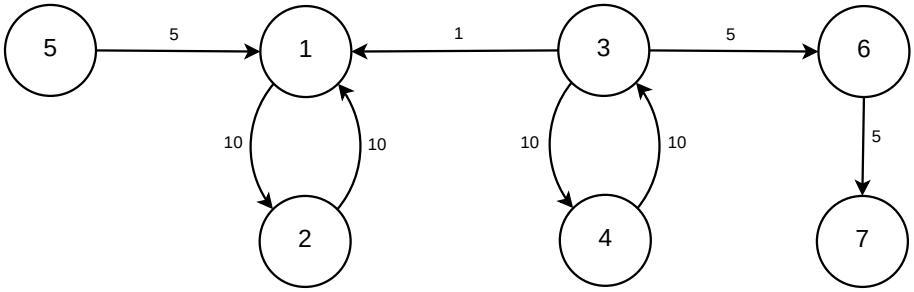
5.1.3. Konstruktion des Hilfsgraphen B_i aus G_i

B_i bekommt alle Knoten von G_i. Für jeden Knoten v in $V(B_i)$ nehmen wir jeweils nur die schwerste, echt positive, nicht zyklische Kante aus $E(G_i)$ auf, die nach v führt.

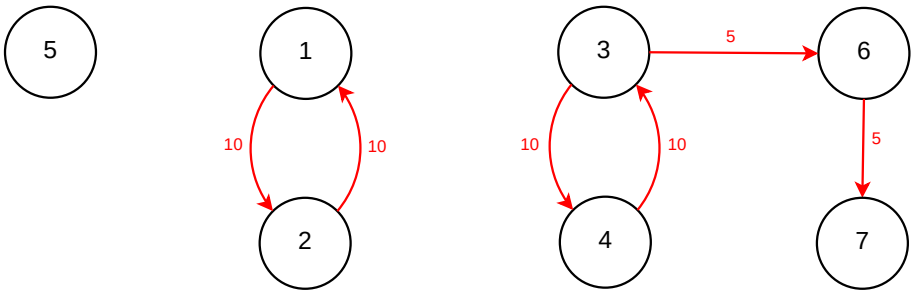
5.1.4. Konstruktion von G_{i+1} aus G_i und dem Hilfsgraphen B_i

G_{i+1} bekommt alle Knoten von G_i, außer jene, die in einem Zyklus in B_i liegen. G_{i+1} bekommt zusätzlich für jeden Zyklus in B_i einen neuen Knoten V_C (Schrumpfknoten) hinzu.

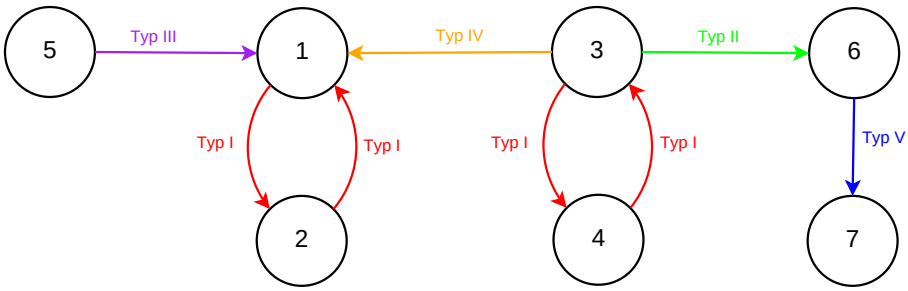
Die Kanten von G_{i+1} bedürfen einer genaueren Betrachtung an einem Beispiel. Gegeben sei G_i:



Dazu konstruieren wir B_i:



Um die Kanten von G_{i+1} zu bestimmen, bestimmen wir den Typ jeder Kante von G_i bezüglich B_i. Es gibt fünf verschiedene Typen:



- Typ I: Kante in einem Zyklus
- Typ II: Kante von einem Knoten eines Zyklus zu einem normalen Knoten
- Typ III: Kante von normalem Knoten zu einem Knoten eines Zyklus
- Typ IV: Kante von einem Knoten eines Zyklus zu einem Knoten in einem anderen Zyklus
- Typ V: Kante ohne Zykluskontakt — normaler Knoten zu normalem Knoten

Behandlung der Kantentypen

Typ I:

Für jede in einem Zyklus C von B_i liegende Kante $e = (x, y)$ mit $x \in V(C)$ und $y \in V(C)$: tue nichts!

Typ II:

Für jede aus einem Zyklus C von B_i führende Kante $e = (x, y)$ mit $x \in V(C)$ und $y \notin \bigcup_n V(C_n)$ erzeuge eine neue Kante $e' = (V_C, y)$ mit $c(e') = c(e)$ und füge e' den Kanten von G_{i+1} hinzu. Speichere e als Vorgängerkante von $e' : \Psi(e') = e$.

Typ III:

Für jede von einem Zyklus C_1 von B_i zu einem anderen Zyklus C_2 von B_i führende Kante $e = (x, y)$ mit $x \in V(C_1)$ und $y \in V(C_2)$ erzeuge eine neue Kante $e' = (V_{C_1}, V_{C_2})$ mit $c(e') = c(e) - c(\overline{e}) + c(e_c)$, wobei e_c die leichteste Kante im Zyklus C_2 und $\overline{e}$ die Kante im Zyklus C_2 nach y ist. Füge e' den Kanten von G_{i+1} hinzu und speichere e als Vorgängerkante von $e' : \Psi(e') = e$.

Typ IV:

Für jede in einen Zyklus C von B_i führende Kante $e = (x, y)$ mit $x \notin \bigcup_n V(C_n)$ und $y \in V(C)$ erzeuge eine neue Kante $e' = (x, V_C)$ mit $c(e') = c(e) - c(\overline{e}) + c(e_c)$, wobei e_c die leichteste Kante im Zyklus C und $\overline{e}$ die Kante im Zyklus C nach y ist. Füge e' den Kanten von G_{i+1} hinzu und speichere e als Vorgängerkante von $e' : \Psi(e') = e$.

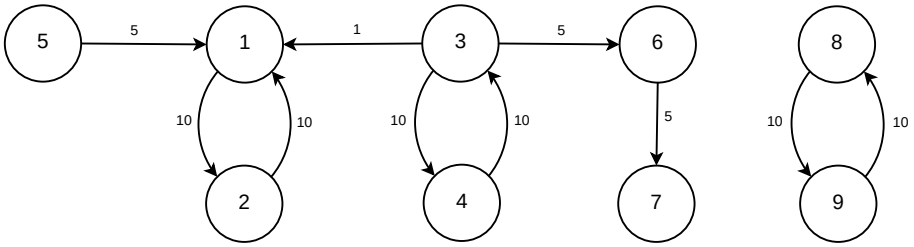
Typ V:

Für jede ohne Zykluskontakt mit einem Zyklus von B_i liegende Kante $e = (x, y)$ mit $x \notin \bigcup_n V(C_n)$ und $y \notin \bigcup_n V(C_n)$: füge e den Kanten von G_{i+1} hinzu.

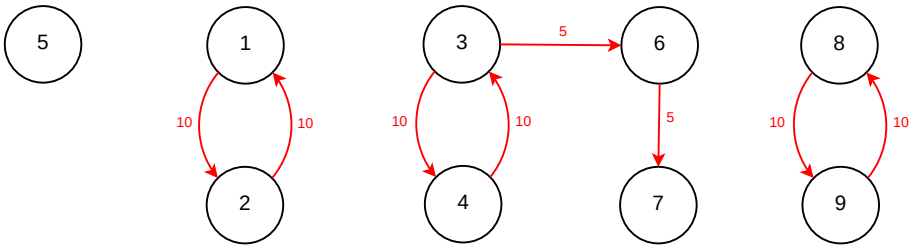
5.1.5. Fälle der Expansion

Im Pseudocode der Expansion auf Seite 62 gibt es vier Fälle für die Neubestimmung der Kantenmenge $E(R_{i-1})$.

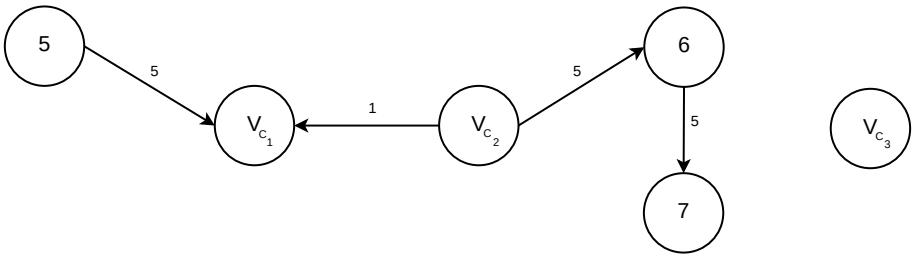
Wir möchten uns an einem weiteren Beispiel ansehen, wann welcher Fall durchlaufen wird. Dazu nehmen wir den Graph G_i aus 5.1.4, den wir um einen weiteren Zyklus, bestehend aus den Knoten 8 und 9 erweitern. Gegeben sei G_0:



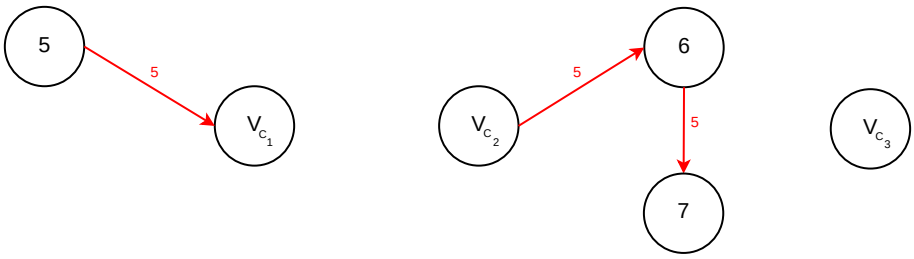
Wir konstruieren den Hilfsgraphen B_0:



B_0 enthält Zyklen. Die Kontraktion wird fortgesetzt. Dazu konstruieren wir G_1 aus B_0 und G_0:



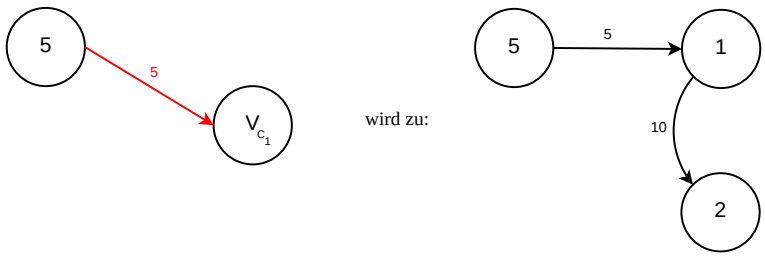
Wir konstruieren den Hilfsgraphen B_1:



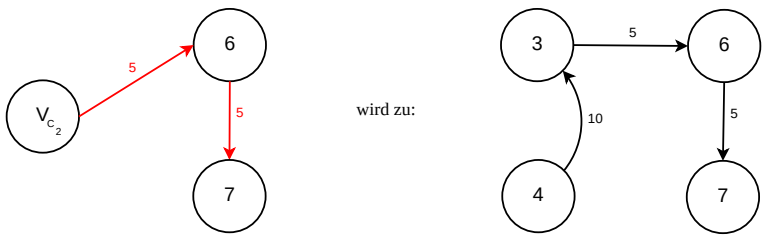
B_1 enthält keine Zyklen. Damit ist $R_1 = B_1$ das maximale Branching zu G_1. Wir beenden die Kontraktion und beginnen mit der Expansion. Wir setzen $R_0 := R_1$ und iterieren über die Zyklen C_1, C_2 und C_3 aus B_0. Für jeden Zyklus iterieren wir über alle Kanten von $R_0 = \{(5, V_{C_1}), (V_{C_2}, 6), (6, 7)\}$.

Zyklus	Kante	Fall
C_1	$(5,V_{C_1})$	1
	$(V_{C_2},6)$	3
	(6,7)	3
C_2	$(5,V_{C_1})$	3
	$(V_{C_2},6)$	2
	(6,7)	3
C_3	$(5,V_{C_1})$	3
	$(V_{C_2},6)$	3
	(6,7)	3

- **Fall 1:** bearbeitet eine in den Zyklus C führende Kante e'. Gemäß der Zuweisung $E(R_{i-1}) := (E(R_{i-1})\backslash e') \cup \Psi(e') \cup (E(C)\backslash \overline{\Psi(e')}$ wird e' aus der Kantenmenge $E(R_{i-1})$ entfernt und dafür der Vorgänger e von e' mit $\Psi(e') = e = (x, y)$ der Kantenmenge $E(R_{i-1})$ hinzugefügt. Weiter werden alle Kanten des Zyklus C, bis auf die Kante, die in den Knoten y führte, in die Kantenmenge $E(R_{i-1})$ aufgenommen. Der Zyklus C gilt als „behandelt“.



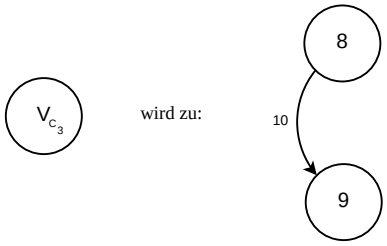
- **Fall 2:** bearbeitet eine aus dem Zyklus C führende Kante e'. Gemäß der Zuweisung $E(R_{i-1}) := (E(R_{i-1})\backslash e') \cup \Psi(e') \cup (E(C)\backslash e_c)$ wird e' aus der Kantenmenge $E(R_{i-1})$ entfernt und dafür der Vorgänger e von e' mit $\Psi(e') = e$ der Kantenmenge $E(R_{i-1})$ hinzugefügt. Weiter werden alle Kanten des Zyklus C, bis auf die leichteste Kante, in die Kantenmenge $E(R_{i-1})$ aufgenommen. Der Zyklus C gilt als „behandelt“.



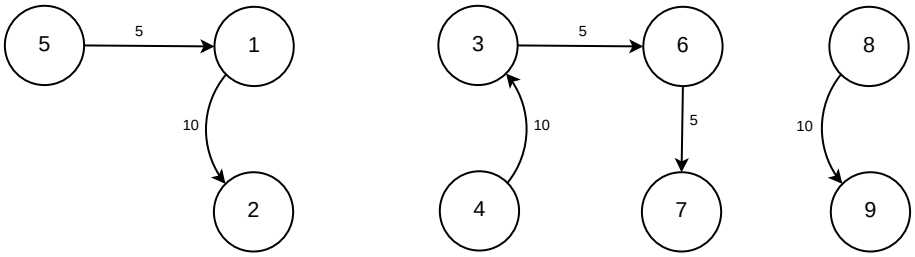
- **Fall 3:** bearbeitet eine Kante, die mit dem Zyklus C nicht in Kontakt kommt. Die Kante muss nicht ersetzt werden. Die Kantenmenge $E(R_{i-1})$ bleibt unverändert.

Wird für einen Zyklus C nie Fall 1 oder Fall 2 durchlaufen, gibt es keine in oder aus dem Zyklus C führende Kante. Somit gilt der Zyklus C als „nicht behandelt“.

- **Fall 4:** bearbeitetet einen „nicht behandelten“ Zyklus C. Gemäß der Zuweisung $E(R_{i-1}) := E(R_{i-1}) \cup (E(C) \backslash e_c)$ fügt er alle Kanten des Zyklus C, bis auf die leichteste, der Kantenmenge $E(R_{i-1})$ hinzu.



Damit ist R_0 bestimmt:



5.1.6. Deterministisches Verhalten

Bei der Konstruktion des Hilfsgraphen B_i kann es passieren, dass in einen Knoten y mehrere Kanten $e = (x, y)$ mit gleichem Kantengewicht $c(e)$ führen. Um dem Algorithmus ein eindeutiges Verhalten zu geben, wähle man die Kante, deren Quellknoten den kleinsten Wert trägt. Auch diese Wahl kann jedoch noch nicht eindeutig sein, wenn es mehr als eine Kante von x nach y gibt (Multigraph), die auch noch über das gleiche Kantengewicht verfügt. In diesem Fall betrachten wir die Kanten nicht als Menge, sondern als Liste, die Kante für Kante abgearbeitet wird. Eine Kante wird dann nur durch eine andere Kante ersetzt, wenn sie ein größeres Kantengewicht besitzt oder bei gleichem Kantengewicht ihr Ursprungsknoten einen kleineren Wert trägt.

5.1.7. Weitere Beispiele

Weitere Beispiele befinden sich im Anhang ab Seite 107.

5.1.8. Implementierung

Die Implementierung richtet sich im Wesentlichen nach dem Pseudocode von Seite 61 und den Beschreibungen der Konstruktionen der Folgegraphen. Daher wird hier nicht weiter auf die Implementierung eingegangen. Zudem ist die Implementierung weniger auf Effizienz als auf Korrektheit und Verständlichkeit des Algorithmus angelegt. Dies war besonders eine Folge der Suche nach einer fertigen Implementierung für den Edmonds–Algorithmus, die jedoch mit dem enttäuschenden Ergebnisse endete, dass alle gefundenen vorhandenen Implementierungen nicht korrekt arbeiteten. Daher erfolgte die Gewichtung auf Korrektheit und Verständlichkeit.

Ein Punkt, der bisher noch nicht beleuchtet wurde, ist die Frage, wie man in einem Graphen am einfachsten einen Zyklus sucht. Hier haben die Hilfsgraphen B_i im Gegensatz zu einem allgemeinen Graphen die angenehme Eigenschaft, dass die Knoten des Hilfsgraphen über maximal eine eingehende Kante verfügen. Somit kann man den Pfad der eingehenden Kanten schnell verfolgen und dabei die Knoten markieren. Trifft man auf einen bereits markierten Knoten, hat man einen Zyklus gefunden.

5.2. Der PC–Tree–Algorithmus

Ziel der Anwendung des PC–Tree–Algorithmus ist es, zu bestimmen, ob eine gegebene Relation die Eigenschaft einer Circular-Continuous-Image-Relation (CCIR) erfüllt.

Die Circular–Continuous–Image–Relationen sind eine Obermenge der Continuous–Image–Relationen (CIR). Die formalen Definitionen entnehme man der Arbeit von Simon Brodt[**?**]. Die Eigenschaften von CIRs und CCIRs wollen wir nun an Graphen betrachten.

Ein Continuous-Image-Graph (CIG) ist ein Spezialfall einer CIR insofern, dass seine Relation $R \subseteq V \times W$ die Eigenschaft hat, dass $V = W$. Ein CIG hat die Eigenschaft, dass eine Ordnung seiner Knoten existiert, so dass für alle Knoten die Kinder eines Knotens in einem zusammenhängenden Intervall in der Ordnung liegen.

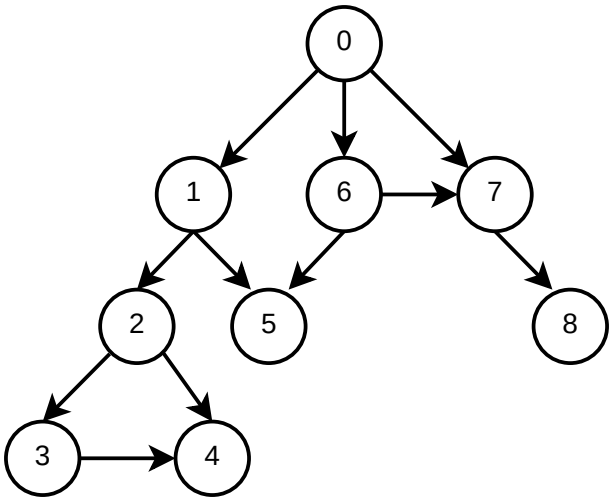
Eine Ordnung könnte gegeben sein durch 016752348. Ein gültiges Intervall wäre [167[. Bei einem CIG gäbe es jedoch bei dieser Ordnung kein Intervall, in dem 4801 liegen würde.

Diese Einschränkung, dass Intervalle nicht über die Grenzen der Ordnung gehen dürfen, liegt bei CCIRs nicht vor.

Der Kern bietet bisher eine statische *CCIR*–Klasse, deren Compute-Methode jedoch nicht implementiert ist. Der PC–Tree–Algorithmus wird bei der Suche nach einer optimalen Datenstruktur zur Speicherung eines Graphen benutzt und stand bisher nicht zur Verfügung. Somit wurden bisher bei Graphen, die als CCIR gespeichert werden können, weniger geeignete Strukturen benutzt. Mit der Implementierung des PC–Tree–Algorithmus vervollständigen wir somit den Kern und verbessern die Speicherung für CCIRs.

5.2.1. CIG–Beispiel

Betrachten wir folgenden Graphen:



Eine mögliche Ordnung der Knoten wäre gegeben durch: 016752348

Knoten x	Kinder von x
0	016752348
1	016752348
2	016752348
3	016752348
4	016752348
5	016752348
6	016752348
7	016752348
8	016752348

Wie an der Tabelle zu sehen ist, befinden sich alle Kinder eines Knotens in einem zusammenhängenden Intervall. Kein Intervall geht dabei über die Grenzen der Ordnung hinaus. Jedes Intervall lässt sich durch die Indexposition bzgl. der Ordnung ausdrücken:

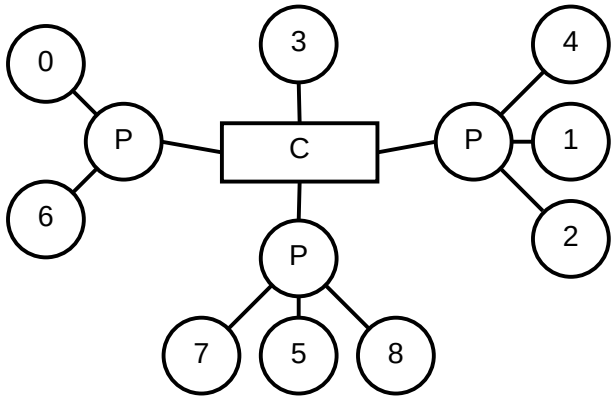
Ordnung	0	1	6	7	5	2	3	4	8
Indexposition	0	1	2	3	4	5	6	7	8

Knoten x	Indexposition der Kinder von x bzgl. der Ordnung
0	[1,3]
1	[4,5]
2	[6,7]
3	[5,5]
4	[]
5	[]
6	[3,4]
7	[8,8]
8	[]

5.2.2. Der PC-Tree

Um zu einem gegebenen Graphen oder einer gegebenen Relation eine Ordnung zu bestimmen, wird der PC-Tree[4] verwendet. Ein PC-Tree ist ein Baum, der aus P- und C-Knoten sowie aus Blättern besteht und ungerichtet ist. Somit besitzt er keine ausgezeichnete Wurzel. Ein P-Knoten ist ein Knoten mit mindestens zwei Kindern, wobei seine Kinder in beliebiger Reihenfolge angeordnet werden dürfen. Ein C-Knoten besitzt mindestens drei Kinder. Seine Kinder dürfen in ihrer Reihenfolge umgedreht oder zyklisch verschoben werden. Die Blätter repräsentieren die eigentlichen Daten.

Beispiel für einen gültigen PC-Tree:



Die von uns gesuchte Ordnung liest man ab, indem man an einem beliebigen Blatt beginnt und im Uhrzeigersinn alle Blätter besucht. Beginnt man beispielsweise bei der 5, so erhält man: 576034128. Die Tatsache, dass man an einem beliebigen Blatt beginnen kann, folgt aus Eigenschaft der CCIR, dass die Intervalle über die Grenzen der Ordnung hinausgehen dürfen.

5.2.3. Konstruktion des PC-Trees

Gegeben sei ein Graph G:

1. Bestimme die Adjazenzmatrix des Graphen G.
2. Initialisiere den PC-Tree mit einem P-Knoten, an dem für jeden Knoten des Graphen ein Blatt mit dem Namen des Knotens hängt.
3. Arbeite jede Zeile der Adjazenzmatrix des Graphen ab:
   - Markiere jedes Blatt mit 0 oder 1, entsprechend der aktuellen Zeile. Ein mit 1 markiertes Blatt wird als „voll“ bezeichnet, ein mit 0 markiertes Blatt als „leer“.
   - Bestimme den Terminal-Pfad (falls er existiert, sonst scheitere).
   - Modifiziere den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen. (falls möglich, sonst scheitere).

- Jeden Knoten auf dem Terminal-Pfad teile in zwei Knoten gleichen Typs auf. An den einen Knoten hänge die Teilbäume mit den leeren Blättern, an den anderen Knoten hänge die Teilbäume mit den vollen Blättern.
- Ersetze den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält.
- Verschmelze alle benachbarten C-Knoten.
- Entferne alle P- oder C-Knoten, die nur einen Nachbarn besitzen.
- Entferne alle Knoten, die genau zwei Nachbarn besitzen, und verbinde die Nachbarn direkt.

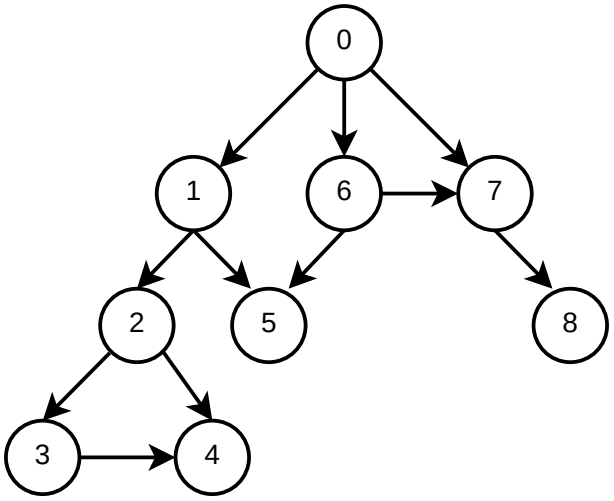
Bestimmung des Terminal-Pfades:

Für das Verständnis der Einführungsbeispiele sei der Terminal-Pfad der Pfad, an dem der Baum zerschnitten werden muss, so dass alle vollen Blätter auf der einen Seite und alle leeren Blätter auf der anderen Seite liegen können. Die genaue Bestimmung des Terminal-Pfades erklären wir später in 5.2.5.

Es ist möglich, dass sich kein Terminal-Pfad bestimmen lässt. In diesem Fall lässt sich der PC–Tree nicht zerschneiden, und die aktuell bearbeitete Zeile kann nicht angewandt werden. Der Algorithmus scheitert. Somit kann keine Ordnung für den Graphen bestimmt werden und dieser folglich nicht als CIG gespeichert werden. Dazu auch später mehr in 5.2.6.

5.2.4. Einführungsbeispiel

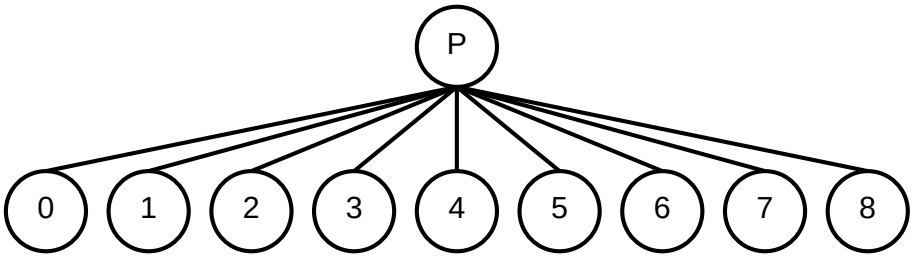
Gegeben sei folgender Graph G, bereits bekannt aus 5.2.1:



Wir bestimmen die Adjazenzmatrix zu G:

	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	0
2	0	0	0	1	1	0	0	0	0
3	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	0	1	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0

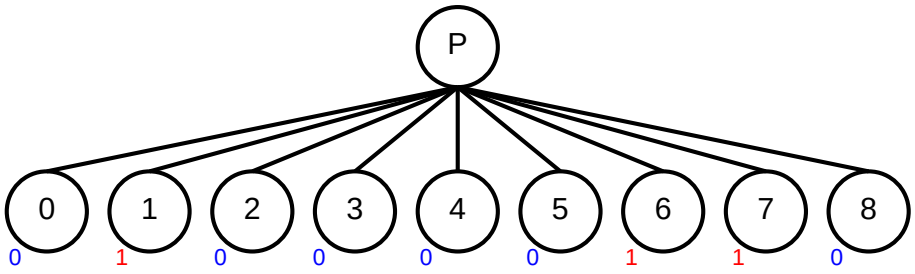
Wir initialisieren den PC-Tree:



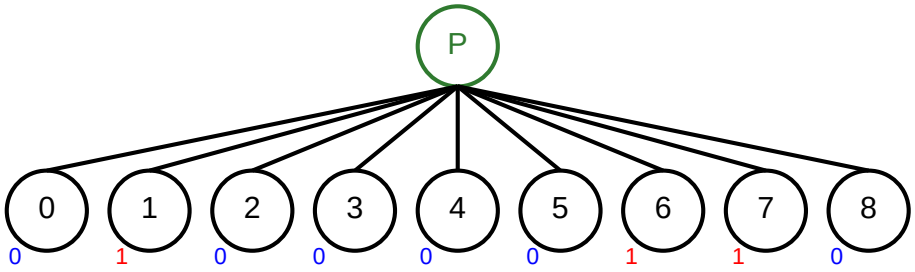
Wir arbeiten jede Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 0:	0	1	0	0	0	0	1	1	0

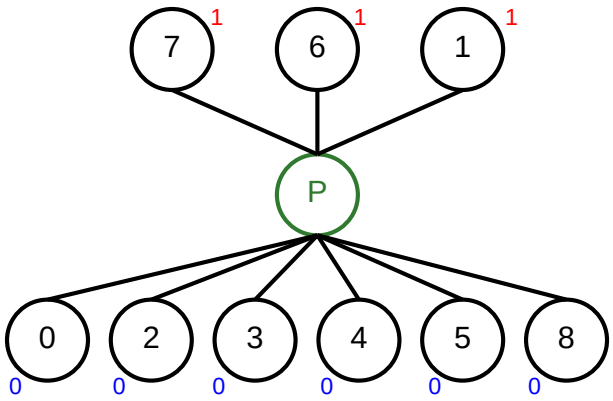
Wir markieren entsprechend der Zeile 0:



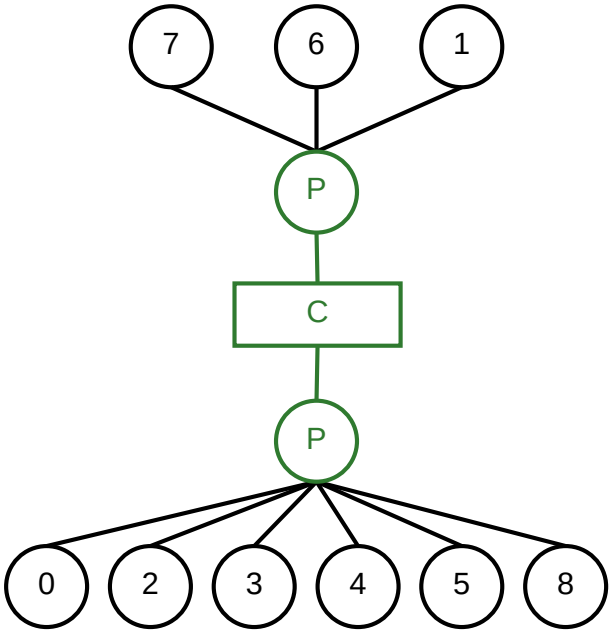
Wir bestimmen den Terminal-Pfad. Dieser besteht lediglich aus einem P-Knoten:



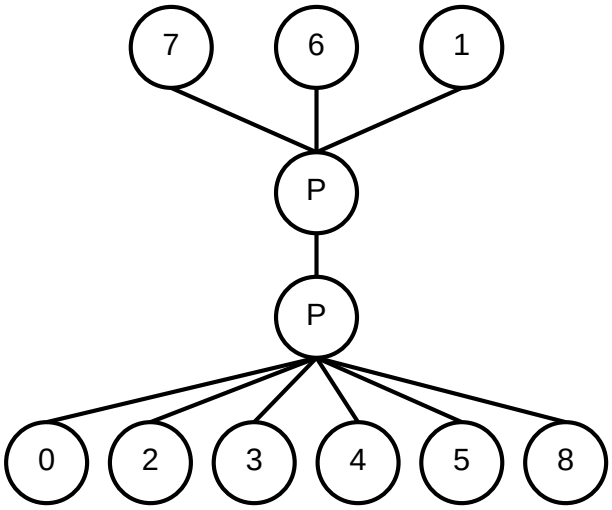
Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen Seite:



Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



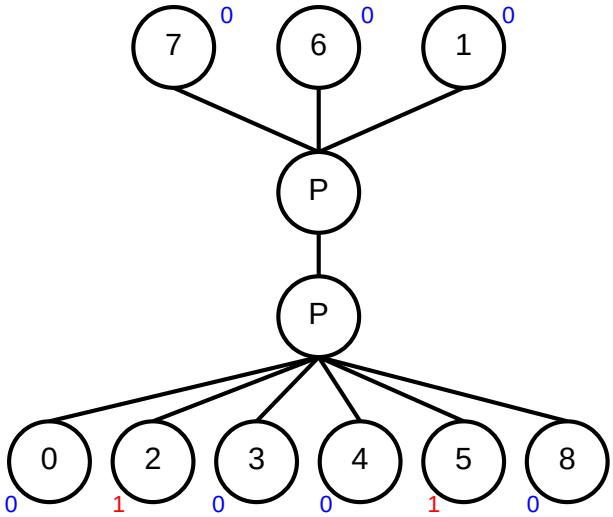
Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!
Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen, und verbinden die Nachbarn direkt:



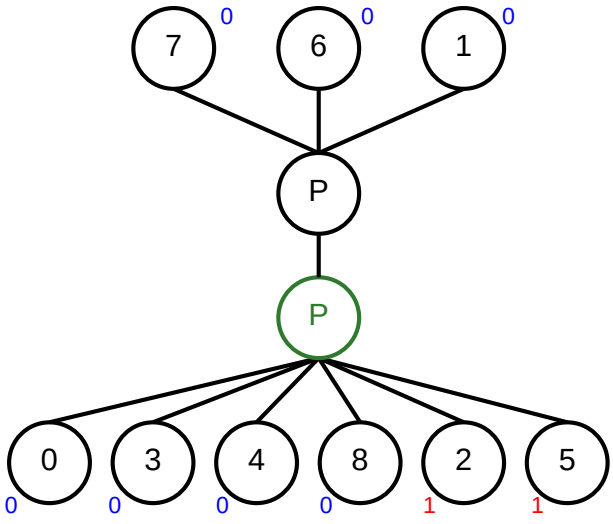
Wir arbeiten die 1. Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 1:	0	0	1	0	0	1	0	0	0

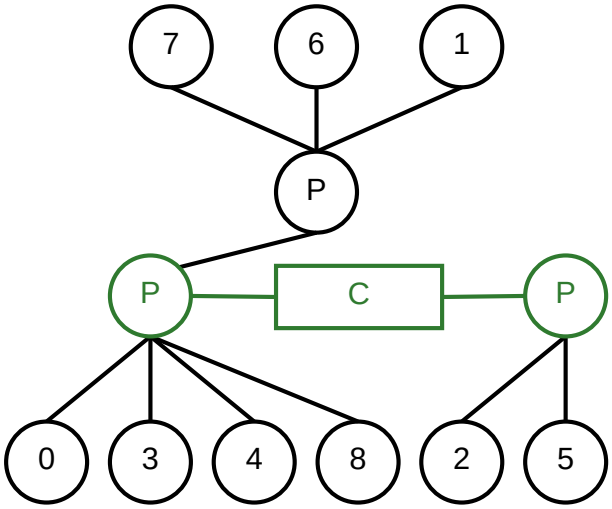
Wir markieren entsprechend der Zeile 1:



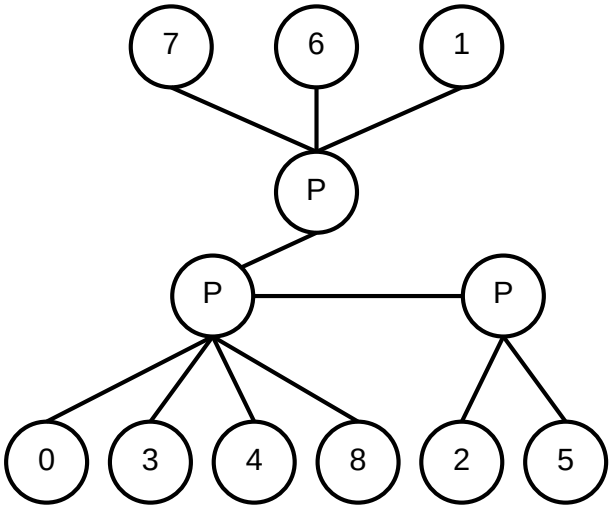
Wir bestimmen den Terminal-Pfad und modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



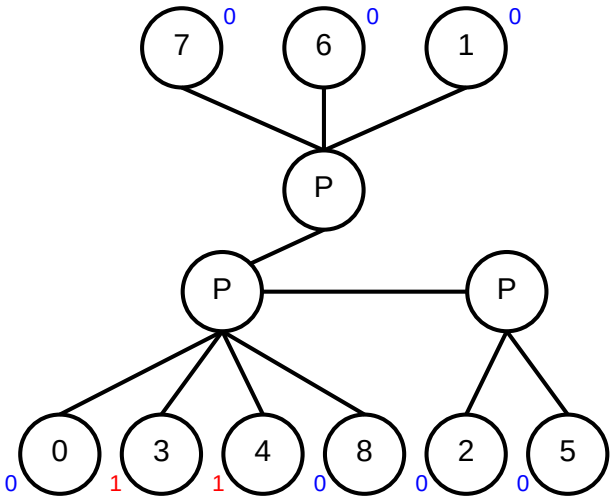
Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!
Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen, und verbinden die Nachbarn direkt:



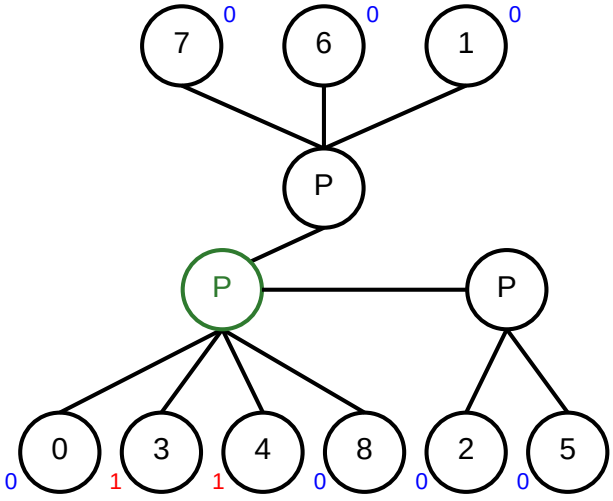
Wir arbeiten die 2. Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 2:	0	0	0	1	1	0	0	0	0

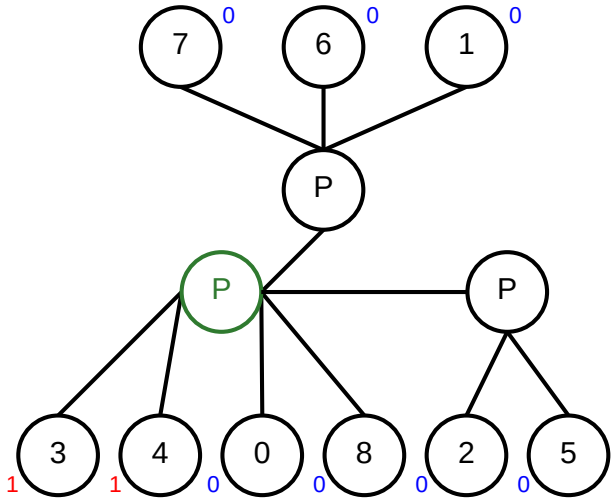
Wir markieren entsprechend der Zeile 2:



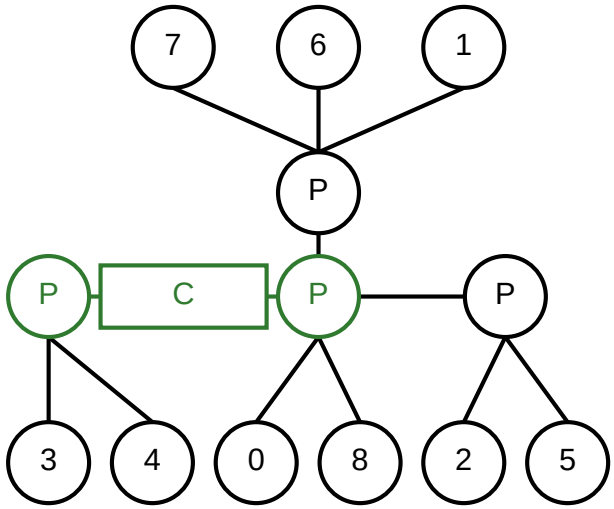
Wir bestimmen den Terminal-Pfad. Dieser besteht lediglich aus einem P-Knoten:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:

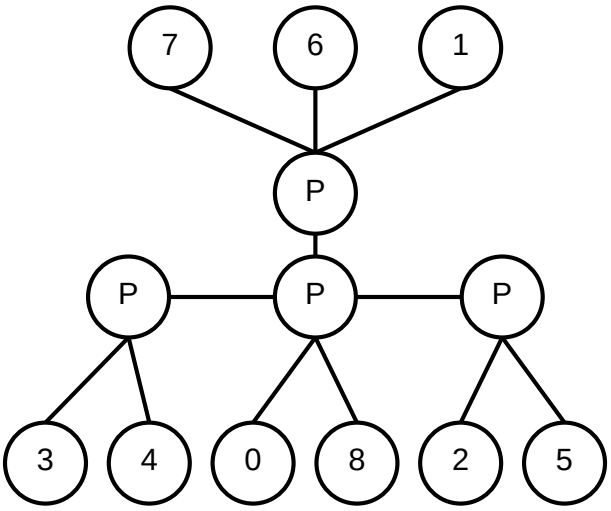


Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!
Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

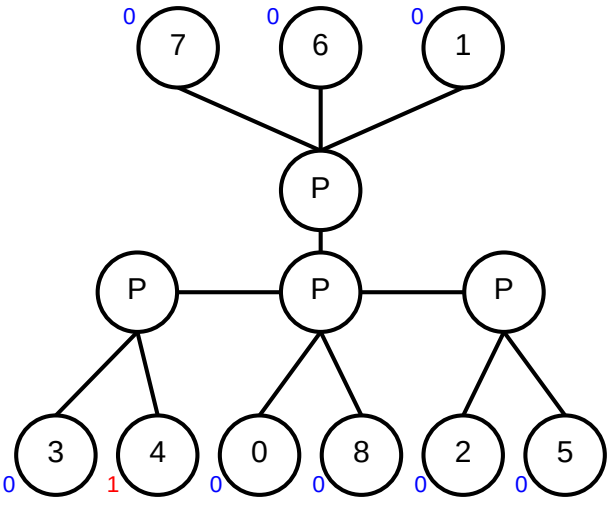
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt:



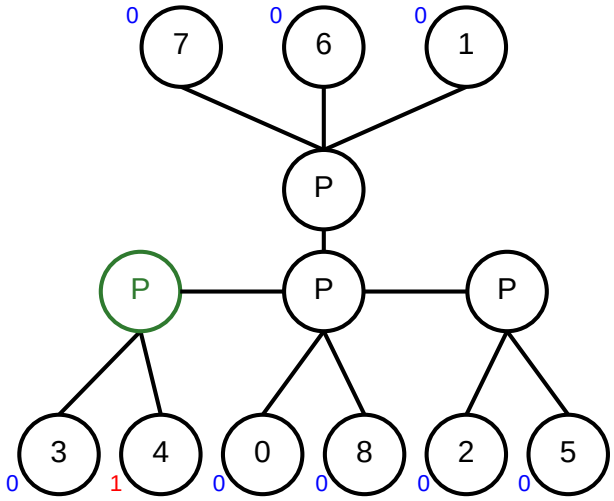
Wir arbeiten die 3. Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 3:	0	0	0	0	1	0	0	0	0

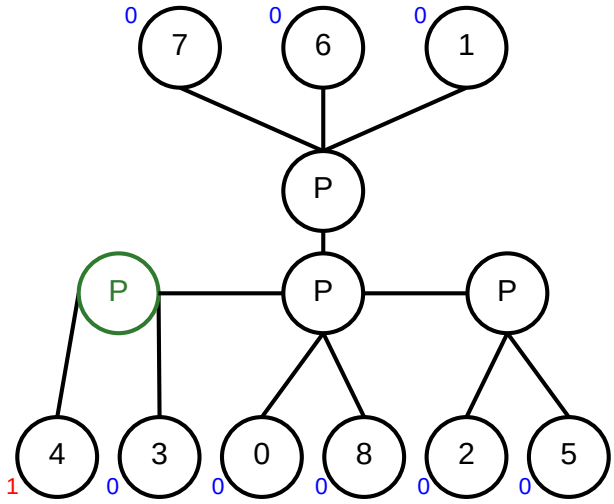
Wir markieren entsprechend der Zeile 3:



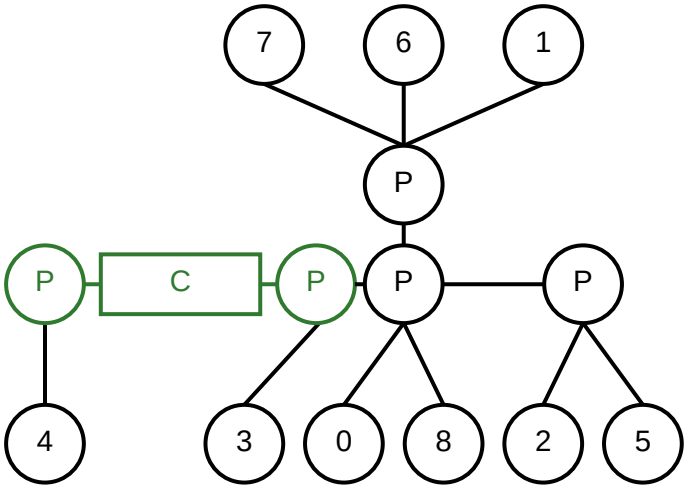
Wir bestimmen den Terminal-Pfad:



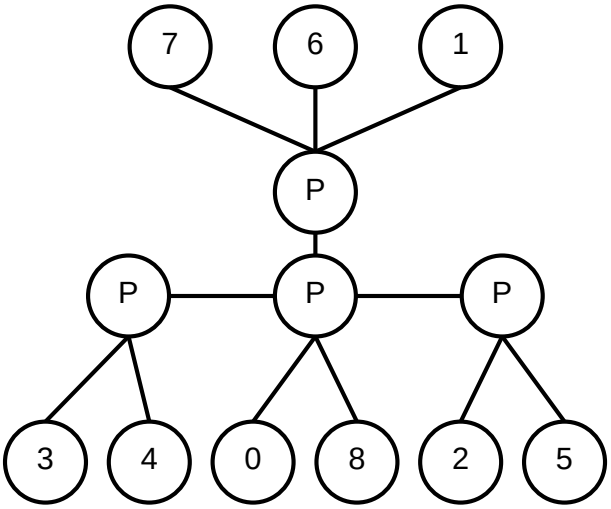
Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!
Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt:



Wir arbeiten die 4. Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 4:	0	0	0	0	0	0	0	0	0

Bei einer Zeile, die nur aus Nullen oder nur aus Einsen besteht, ist nichts zu tun.

Wir arbeiten die 5. Zeile der Adjazenzmatrix des Graphen ab:

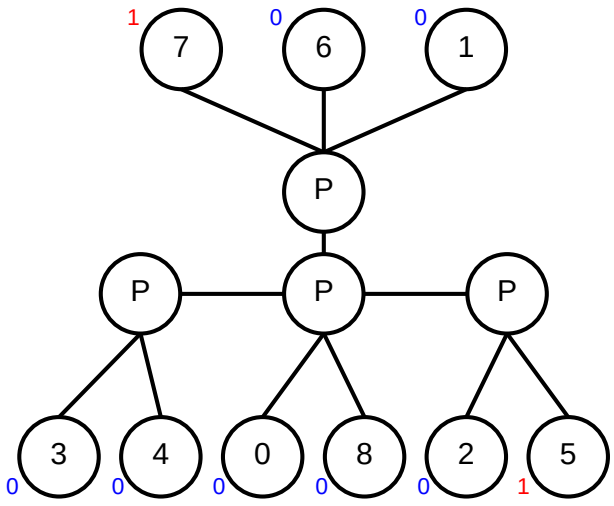
	0	1	2	3	4	5	6	7	8
Zeile 5:	0	0	0	0	0	0	0	0	0

Bei einer Zeile, die nur aus Nullen oder nur aus Einsen besteht, ist nichts zu tun.

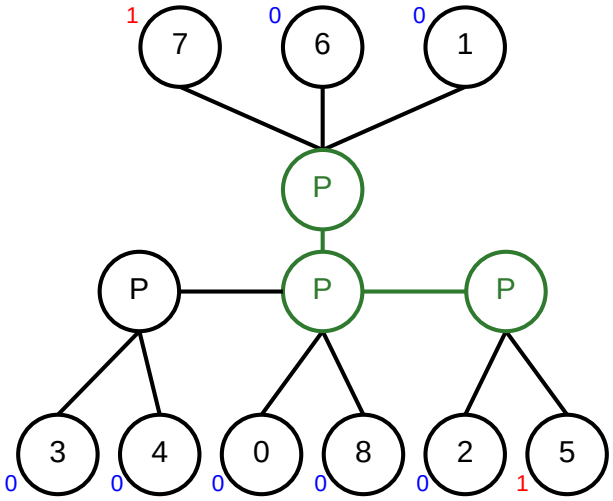
Wir arbeiten die 6. Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 6:	0	0	0	0	0	1	0	1	0

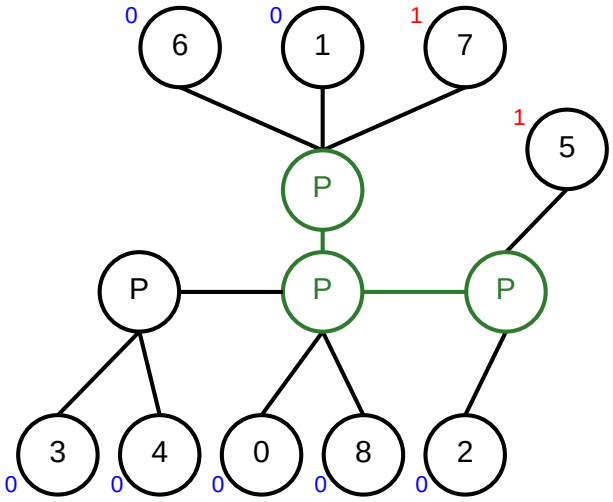
Wir markieren entsprechend der Zeile 6:



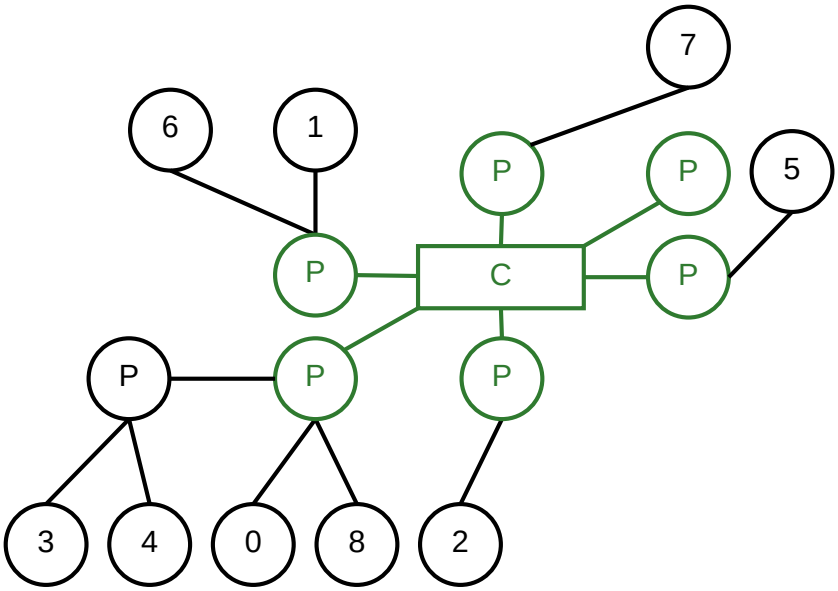
Wir bestimmen den Terminal-Pfad:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:

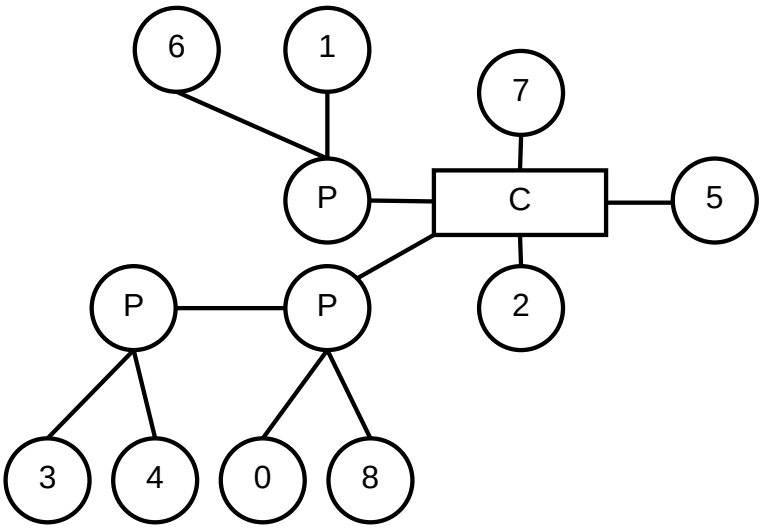


Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!

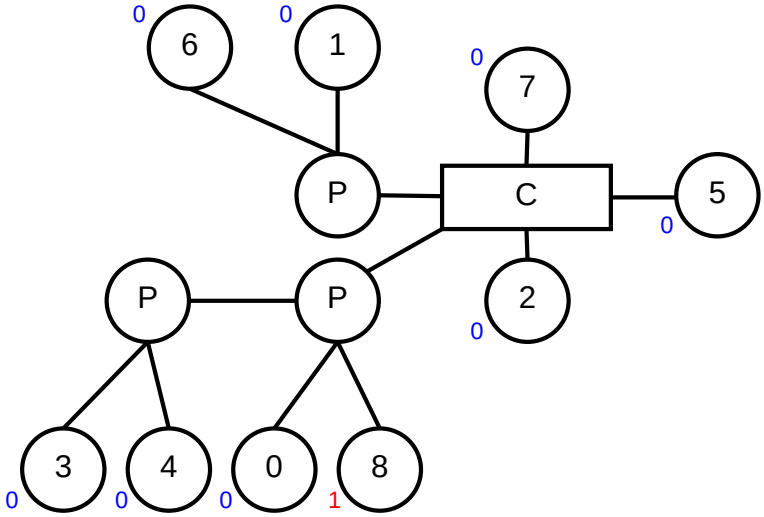
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt. Zudem entfernen wir alle P- oder C-Knoten, die nur einen Nachbarn besitzen:



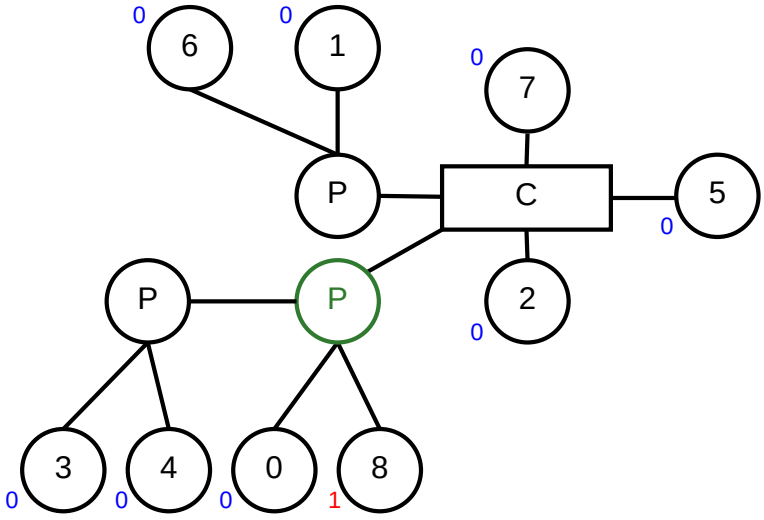
Wir arbeiten die 7. Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 7:	0	0	0	0	0	0	0	0	1

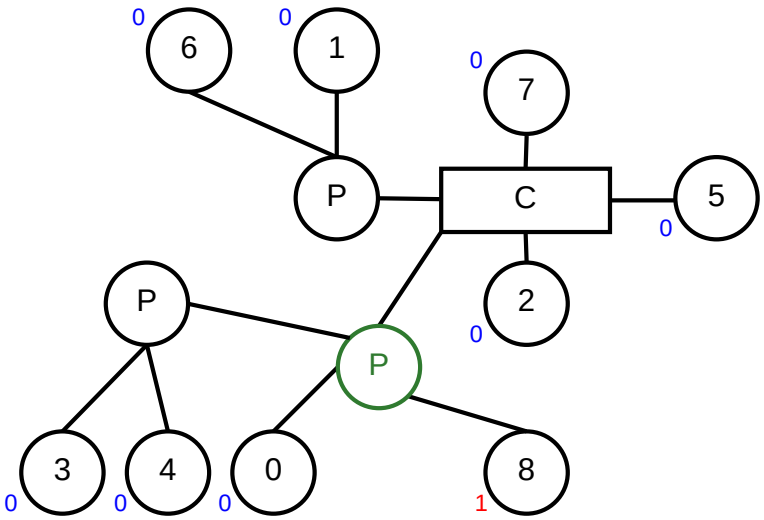
Wir markieren entsprechend der Zeile 7:



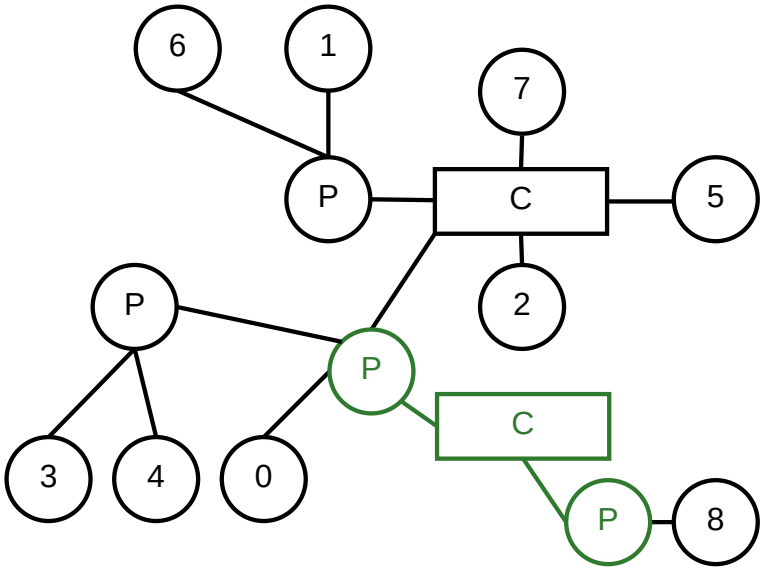
Wir bestimmen den Terminal-Pfad:



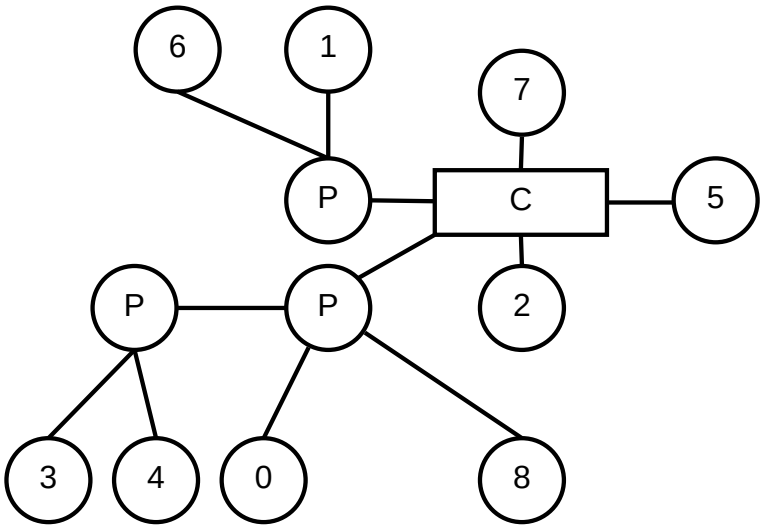
Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!
Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt:

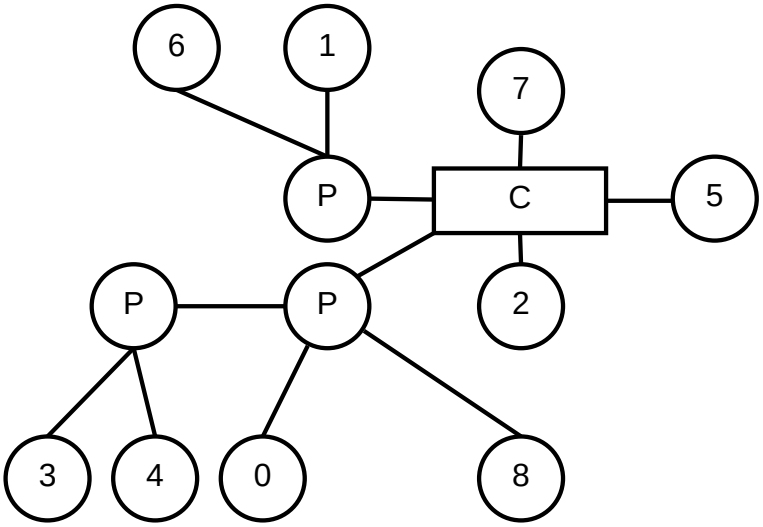


Wir arbeiten die 8. Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7	8
Zeile 8:	0	0	0	0	0	0	0	0	0

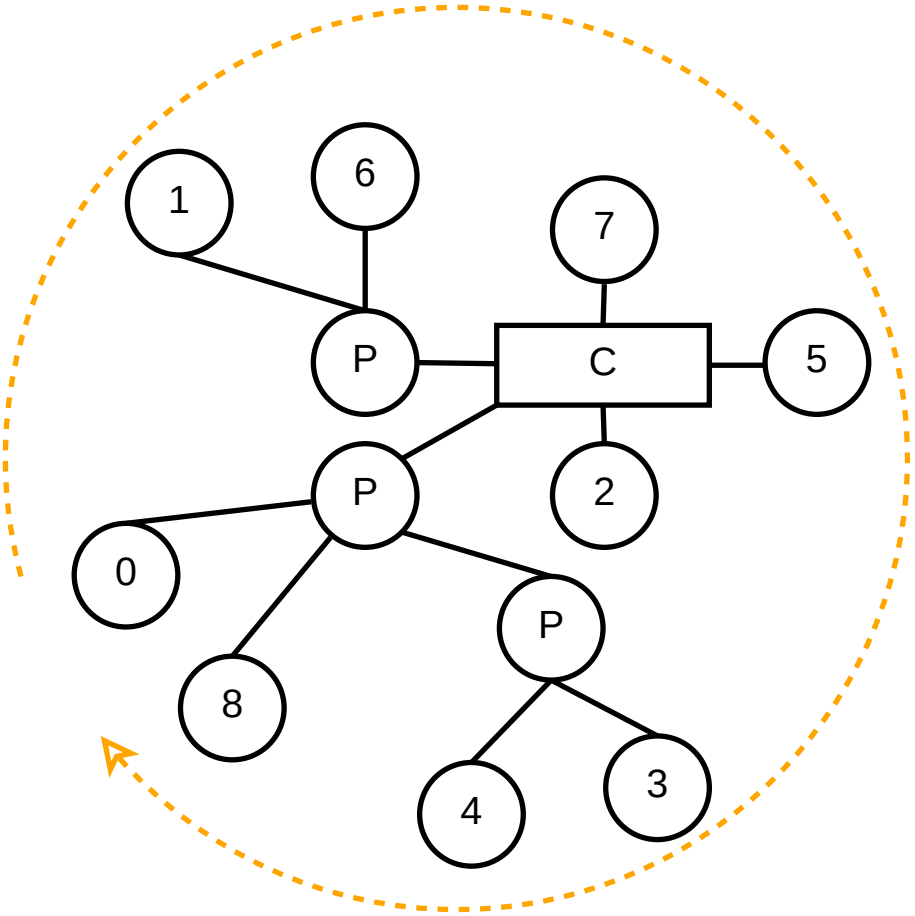
Bei einer Zeile, die nur aus Nullen oder Einsen besteht, ist nichts zu tun.

Wir haben alle Zeilen abgearbeitet. Das Endergebnis sieht folgendermaßen aus:



Nun wollen wir noch einige zulässige Umordnungen vornehmen:

- An dem P-Knoten, an dem die 6 und die 1 hängen vertauschen wir 6 und 1.
- An dem P-Knoten, an dem die 8, die 0, der C-Knoten und ein weiter P-Knoten hängen, verschieben wir den P-Knoten mit 3 und 4 zwischen den C-Knoten und die 8.
- An dem P-Knoten, an dem die 3 und die 4 hängen vertauschen wir 3 und 4.



Beginnen wir nun im Uhrzeigersinn bei der 0 damit die Blätter aufzulisten erhalten wir 016752348. Im CIG–Beispiel in 5.2.1 hatten wir bereits gesehen, dass dies eine mögliche Ordnung für den Graphen ist.

5.2.5. Der Terminal-Pfad

Trotz der Tatsache, dass ein PC–Tree ein ungerichteter Baum ist, werden wir ihn als gerichteten Baum implementieren. Dies vereinfacht die Verarbeitung und erhöht die Effizienz. Ausgehend davon möchten wir nun auch auf die konkrete Bestimmung des Terminal–Pfades eingehen.

Bei der Abarbeitung einer Zeile wird jedes Blatt mit einer Null oder Eins beschriftet und damit als „leer“ oder „voll“ bezeichnet. Auf dieser Basis sollen nun auch die anderen Knoten des PC-Trees, also die P- oder C-Knoten beschriftet werden. Dazu bekommt jeder P- oder C-Knoten einen Zähler, der mit 0 initialisiert wird und die Anzahl der vollen Nachbarn angeben soll. Wird der Zähler auf 1 erhöht so wird der Knoten als „partiell“ bezeichnet. Erreicht der Zähler eines Knotens den Wert, der nur um eins kleiner ist als der Grad des Knotens, nennt man ihn „voll“ und man erhöht die Zähler aller nicht-vollen Nachbarn des Knotens. Begonnen wird mit der Markierung der Blätter entsprechend der aktuellen Zeile. Jede Markierung eines Blattes als voll kann weitere Markierungen im Baum auslösen.

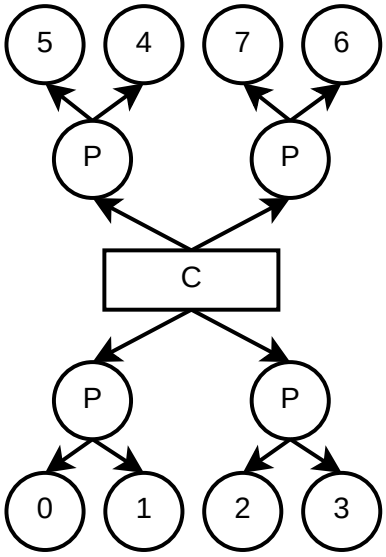
Für den Terminal-Pfad sind die partiellen Knoten entscheidend: jeder partielle Knoten liegt auf dem Terminal-Pfad, aber nicht jeder Knoten auf dem Terminal-Pfad muss partiell sein. Der Terminal-Pfad kann als Sonderfall auch nur aus einem einzigen Knoten bestehen. Diesen Sonderfall erhält man beispielsweise immer beim ersten Aufteilen des PC-Trees nach der Initialisierung. Um die Menge der Kanten des Terminal-Pfades zu bestimmen, wenn der Sonderfall nicht vorliegt, laufen wir von jedem partiellen Knoten den Baum hinauf und sammeln erstmal alle Kanten auf. Dann bestimmen wir den „Scheitel“. Der Scheitel ist der Knoten, der Vorfahre von allen partiellen Knoten ist und den größten Abstand zur Wurzel besitzt. Von den vorhin aufgesammelten Kanten entfernen wir nun alle Kanten, die oberhalb des Scheitels liegen. Über diese Menge von Kanten erhält man jedoch nicht zwangsläufig einen Pfad. Es ist ebenso möglich, dass wir einen Baum erhalten. In diesem Fall ist die Bestimmung des Terminal-Pfades nicht möglich.

Betrachten wir nun also noch die Fälle, in denen wir keinen gültigen Terminal-Pfad erhalten.

5.2.6. Probleme mit dem Terminal-Pfad

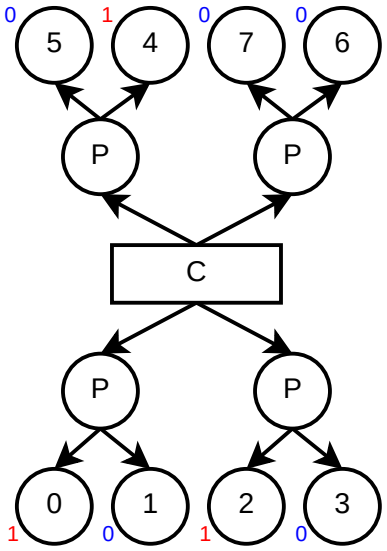
Fehler-Fall 1: Der Terminal-„Pfad“ ist kein Pfad.

Gegeben sei der folgende PC-Tree:

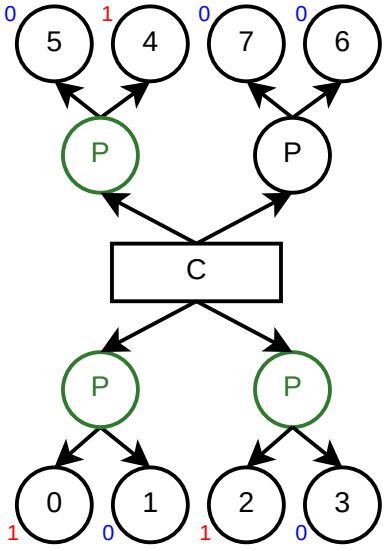


Wie man auf den gezeigten PC-Tree kommt, entnehme man dem Anhang ab Seite 119
Nun wird folgende Zeile eingearbeitet:

	0	1	2	3	4	5	6	7
Zeile:	1	0	1	0	1	0	0	0

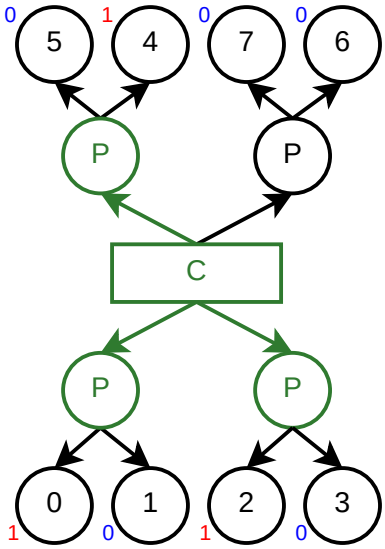


Durch die vollen Blätter 0, 2 und 4 werden drei P-Knoten partiell gekennzeichnet (grün markiert):



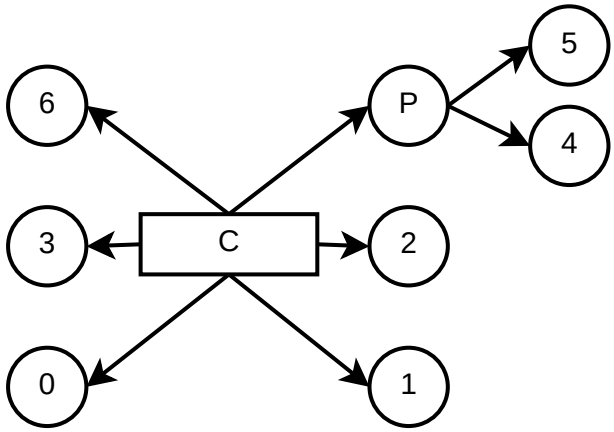
Ausgehend von den drei partiellen P-Knoten laufen wir nun den Baum hinauf. Dabei werden alle Kanten aufgesammelt. Danach wird der Scheitel bestimmt. Dieser ist in diesem Fall die

Wurzel selbst. Daher müssen wir keine Kante wieder entfernen. An den grün markierten Kanten ist nun zu sehen, dass die Kanten keinen Pfad bilden. Es ist somit nicht möglich alle vollen Blätter auf die eine Seite des Pfades und alle leeren Blätter auf die andere Seite des Pfades zu bringen. Der PC-Tree kann nicht zerschnitten werden. Die Abarbeitung endet hier:



Fehler-Fall 2: Der Pfad enthält einen C-Knoten, dessen volle und leere Nachbarn nicht konsekutiv angeordnet sind.

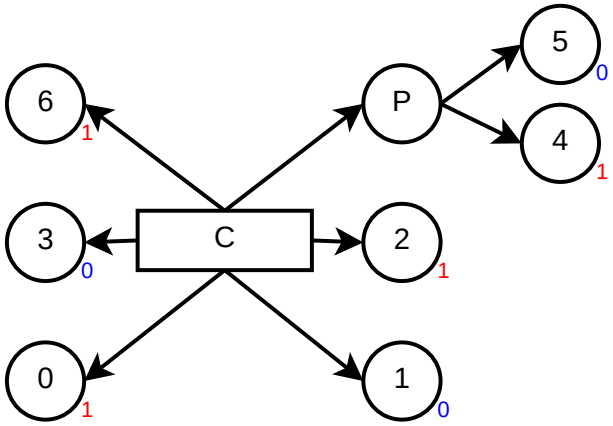
Gegeben sei der folgende PC-Tree:



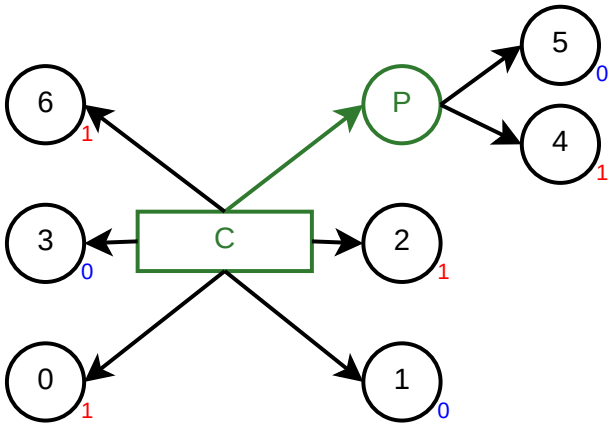
Wie man auf den gezeigten PC–Tree kommt, entnehme man dem Anhang ab Seite 126.

Nun wird folgende Zeile eingearbeitet:

	0	1	2	3	4	5	6
Zeile:	1	0	1	0	1	0	1



Wir bestimmen den Terminal-Pfad:



Als nächstes würden wir den Baum so modifizieren, dass alle vollen Blätter auf der einen Seite des Terminal–Pfads lägen, und alle leeren Blätter auf der anderen. An einem P-Knoten kann dies nie ein Problem darstellen, da wir die Nachbarn eines P-Knotens beliebig anordnen dürfen. Anders ist es jedoch bei einem C-Knoten, dessen Aufgabe es ja gerade ist, die zyklische Reihenfolge der Knoten zu bewahren. Damit sich ein C-Knoten überhaupt korrekt teilen lässt, müssen seine leeren und vollen Nachbarn jeweils in unmittelbar nachfolgender Reihenfolge liegen (konsekutiv).

An dem C-Knoten in unserem Beispiel liegen die vollen und leeren Teilbäume nicht in konsekutiver Reihenfolge. Es ist somit nicht möglich, alle vollen Blätter auf die eine Seite des Pfades und alle leeren Blätter auf die andere Seite des Pfades zu bringen. Der PC–Tree kann nicht zerschnitten werden. Die Abarbeitung endet hier.

5.2.7. Weitere Beispiele

Weitere Beispiele befinden sich im Anhang ab Seite 119.

5.2.8. Implementierung

Die Implementierung besteht aus den Klassen *Node*, *PNode*, *CNode*, *NNode* und *PCTree*. Da der PC–Tree vom Kern verwendet wird befinden sich die Klassen dort. Die *Node*–Klasse ist eine abstrakte Oberklasse für die Klassen *PNode*, *CNode* und *NNode*. Sie bietet eine nicht abstrakte foundFullNeighbor-Methode an, die auf einen Knoten aufgerufen wird, an dem ein voller Nachbar gefunden wurde. Wird der Knoten dadurch selbst voll, wird die foundFullNeighbor-Methode auch auf alle Nachbarknoten aufgerufen. Alle weiteren interessanten Methoden stellt die Klasse *PCTree* bereit. *PNode* und *CNode* sollten vom Namen her selbsterklärend sein, die Klasse *NNode* repräsentiert die „normalen“ Knoten, also die Blätter.

Über die execute-Methode der *PCTree*–Klasse, der eine Adjazenzmatrix in Form eines zweidimensionalen Integerarrays übergeben wird, wird die Abarbeitung der einzelnen Zeilen der Adjazenzmatrix gestartet. Die execute-Methode gibt die gesuchte Ordnung in Form eines eindimensionalen Arrays zurück oder wirft eine Exception, sofern die Bestimmung einer Ordnung aus bekannten Gründen (siehe 5.2.6) nicht möglich war.

Der folgende Codeabschnitt soll die wesentlichen Programmstücke der execute-Methode zeigen:

```
public static int[] execute(int[][] rows)
{
   PCTree pctree = new PCTree(rows[0].Count());
   // Initialisiert den PCTree mit so vielen Blättern,
   // wie die erste Zeile der Adjazenzmatrix Elemente hat

   // Behandle alle Zeilen:
   for (int i = 0; i < rows.Length; ++i)
   {
      List<Node> partNodes = pctree.applyRow(rows[i]);
      // Bestimmt die partiellen Knoten. Dabei kommt die
      // foundFullNeighbor-Methode der Node Klasse
      // zum Einsatz

      List<Node> termPath = pctree.getPathNodes(partNodes);
      // Bestimmt den Terminal-Pfad aus den partiellen Knoten

      pctree.split(termPath);
      // Führt den Splitt entsprechend dem Terminal-Pfad durch

      pctree.correct();
      // Verschmilzt C-Knoten, entfernt P-Knoten
      // mit nur einem Nachbarn, entfernt P-Knoten
      // mit zwei Nachbarn ...
   }

   int[] returnOrder = pctree.getOrder()
   // Bestimmt die Ordnung nach Anwendung aller Zeilen
   // der Adjazenzmatrix

   return returnOrder;
}
```

Betrachten wir nun noch einige Methoden etwas genauer.

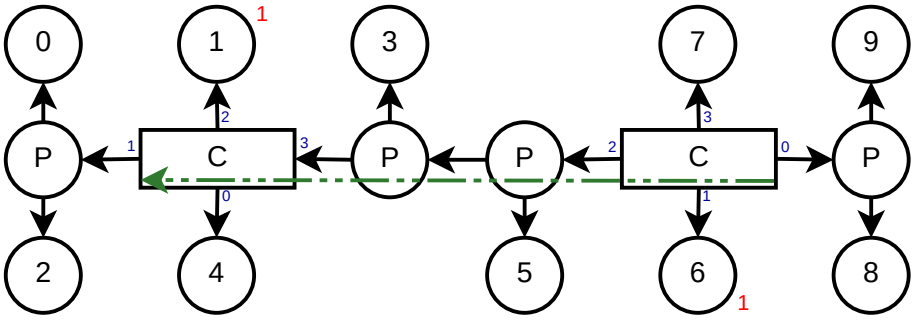
Die getPathNodes-Methode

Die getPathNode–Methode dient der Bestimmung des Terminal-Pfades. Ihr wird eine Liste aller gefundenen partiellen Knoten übergeben (siehe 5.2.5). Im ersten Schritt werden alle Knoten entfernt, die sowohl partiell als auch voll sind. Anschließend wird der Lauf von jedem übriggebliebenen Knoten hoch zur Wurzel durchgeführt und es werden die dabei durchlaufenen Kanten bzw. Knoten zwischengespeichert. Zusätzlich speichern wir bei

jedem Knoten seine Terminal–Pfad–Nachbarknoten. Nach der Bestimmung des Scheitels werden die Kanten bzw. Knoten, die oberhalb des Scheitels liegen, wieder aus der Zwischenspeicherung entfernt. Für die restlichen Knoten wird überprüft, ob es einen Knoten gibt, der mehr als zwei Terminal–Pfad–Nachbarknoten besitzt. Gleichzeitig suchen wir noch einen Knoten, der über genau einen Terminal–Pfad–Nachbarknoten verfügt. Dieser kann als Startknoten des Terminal–Pfades fungieren. Wird ein Knoten mit mehr als zwei Terminal–Pfad–Nachbarknoten gefunden, wird eine Exception geworfen. Gibt es also einen Knoten mit beispielsweise drei Terminal–Pfad–Nachbarknoten, kann die betrachtete Knotenmenge keinen Pfad bilden. Dies ist die Behandlung von Fehler–Fall 1 aus 5.2.6. Liegt der Fehler–Fall nicht vor, kann nun ausgehend vom Startknoten über die Terminal-Pfad Nachbarknoten ein Pfad erstellt werden.

Als Vorbereitung auf die Zerteilung des Terminal-Pfades wollen wir den PC-Tree noch so „hindrehen“, dass bezüglich des Terminal-Pfades alle vollen Blätter auf der einen Seite des Pfades liegen und alle leeren Blätter auf der anderen. Wir legen fest, dass die vollen Blätter in Pfadrichtung immer auf der linken Seite des Pfades liegen sollen. Zudem legen wir fest, dass eine Liste von Nachbarknoten immer im Uhrzeigersinn um den Knoten herum anzuordnen ist. In Wirklichkeit drehen wir den PC-Tree nur bezüglich der C-Knoten, da nur bei diesen die Reihenfolge der Nachbarknoten wichtig ist. Außerdem betrachten wir nur die rechte Seite des Terminal-Pfades. Finden wir hier einen vollen Knoten, so drehen wir. Dass wir dabei ggf. auch einen vollen Knoten von der linken auf die rechte Seite drehen, ignorieren wir erst einmal. Diese Überprüfung führen wir erst vor dem Splitt durch. Hier wäre eine Verbesserung des Algorithmus möglich.

Beispiel für die Drehung des PC-Trees bzgl. eines C-Knotens:



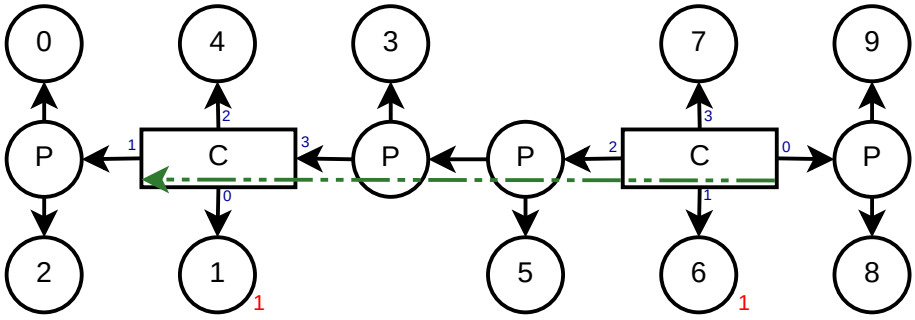
In der Abbildung ist der Terminal-Pfad grün eingezeichnet. Auf ihm liegen zwei C-Knoten, deren Nachbarknoten–Indicies in blau angegeben sind. An dem linken C-Knoten hängt ein volles Blatt, das sich allerdings rechts des Terminal-Pfades befindet. Der Terminal-Pfad führt in den C-Knoten über dessen Indexposition 3. Wir müssen also den C-Knoten bezüglich der Indexposition 3 drehen. Die Elemente der neuen Indexpositionen berechnen sich wie folgt:

$$index_{neu} = drehindexpostion + (drehindexpostion - index_{alt}) \mod Anzahl_{Nachbarn}$$

$$Nachbar_{neu}[0] = Nachbar_{alt}[3 + (3 - 0) \mod 4] = Nachbar_{alt}[2]$$
$$Nachbar_{neu}[1] = Nachbar_{alt}[3 + (3 - 1) \mod 4] = Nachbar_{alt}[1]$$

$$Nachbar_{neu}[2] = Nachbar_{alt}[3 + (3 - 2) \mod 4] = Nachbar_{alt}[0]$$
$$Nachbar_{neu}[3] = Nachbar_{alt}[3 + (3 - 3) \mod 4] = Nachbar_{alt}[3]$$

Führen wir diese Änderungen durch, liegen jetzt beide vollen Blätter auf der linken Seite des Terminal-Pfades:



Die splitt-Methode

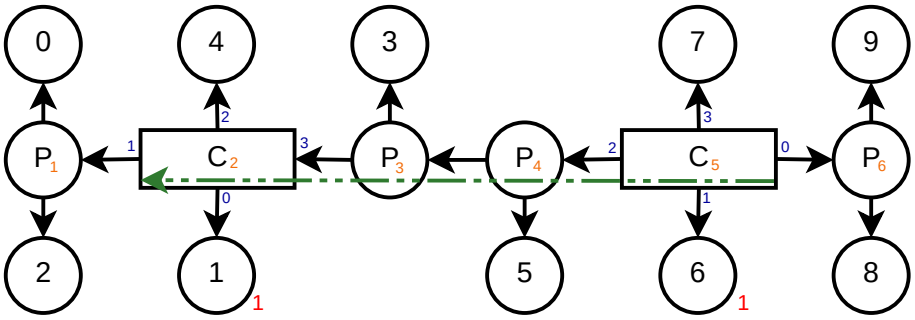
Die splitt-Methode bekommt den Terminal-Pfad als Liste von Knoten übergeben. Sie legt als Erstes einen neuen C-Knoten an, an den dann die vollen und leeren Teilbäume gehängt werden. Für jeden Splitt eines Knotens erstellen wir jedoch nicht zwei neue Knoten, sondern nur einen neuen für die vollen Teilbäume. Die leeren Teilbäume lassen wir an dem zu splittenden Knoten hängen, damit dieser wiederverwendet werden kann. Die eigentliche Operation ist es also, von einem zu splittenden Knoten die vollen Teilbäume an einen neuen Knoten zu hängen.

Für jeden C-Knoten auf dem Terminal-Pfad überprüfen wir zuerst, ob alle vollen Nachbarn konsekutiv angeordnet sind. Ist dies nicht so, erhalten wir Fehler-Fall 2a. Sind die vollen Nachbarn zwar konsekutiv angeordnet, die leeren aber nicht, führt dies zu Fehler–Fall 2b. Fehler-Fall 2 siehe auch 5.2.6.

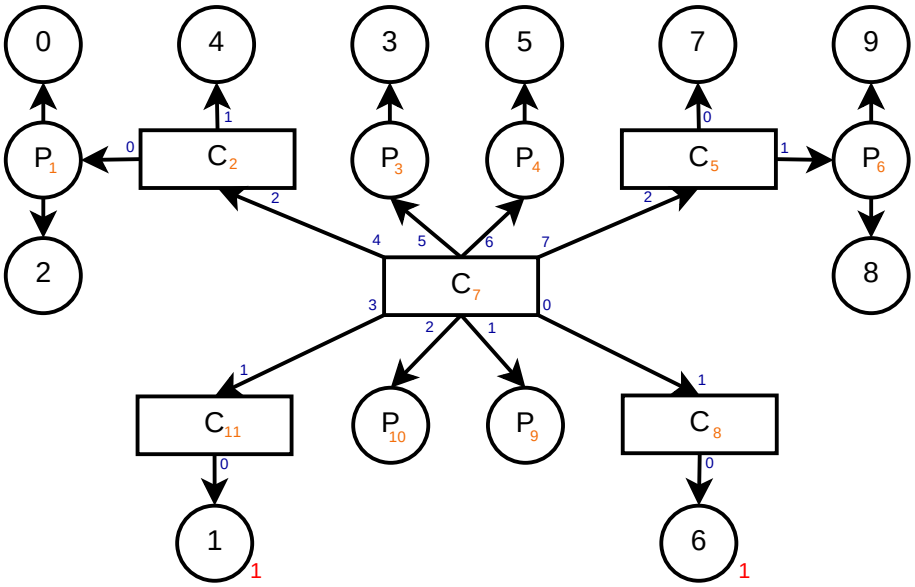
Für einen Terminal-Pfad bestehend aus n Knoten $K_1 \dots K_n$ hängen wir diese wie folgt an den neu erstellten C-Knoten: $V(K_1), \dots, V(K_n), K_n \dots K_1$ wobei $V(K)$ ein neuer Knoten ist, an dem die vollen Teilbäume des Knotens K hängen.

Diese Anordnung resultiert aus der von uns festgelegten Eigenschaft, dass die vollen Blätter in Pfadrichtung immer auf der linken Seite des Terminal-Pfades liegen sollen sowie aus dem Ziel, bei einer späteren Verschmelzung von C-Knoten ohne Reihenfolgenänderung der Nachbarknoten auszukommen.

Betrachten wir dies an einem Beispiel. Um die Wiederverwendung der Knoten erkennen zu können, sind die Knoten nummeriert. Der Terminal-Pfad sei $C_5 \rightarrow P_4 \rightarrow P_3 \rightarrow C_2$]:



Zunächst wird der neue C-Knoten (7) angelegt. An diesen C-Knoten kommen jetzt die neuen Knoten mit den vollen Teilbäumen, danach die alten Knoten, von denen die vollen Teilbäume abgehängt wurden, in umgekehrter Terminal–Pfad Reihenfolge, also: C_8, P_9, P_{10}, C_{11}, C_2, P_3, P_4, P_5:

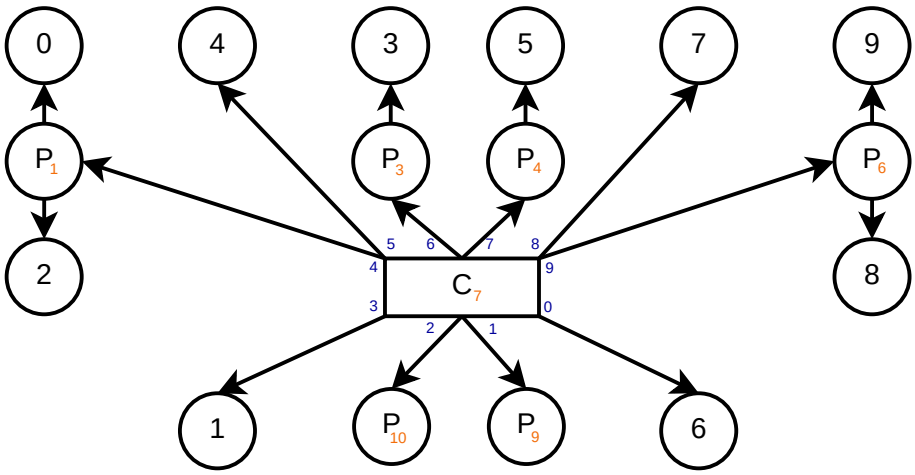


Die Regeln, nach denen bestimmt wird, welcher Knoten nach einem Splitt Vater eines anderen wird, sind sehr komplex. Eine ausführliche Behandlung würde den Rahmen dieser Arbeit sprengen. Daher sei hier auf dem Quellcode selbst verwiesen.

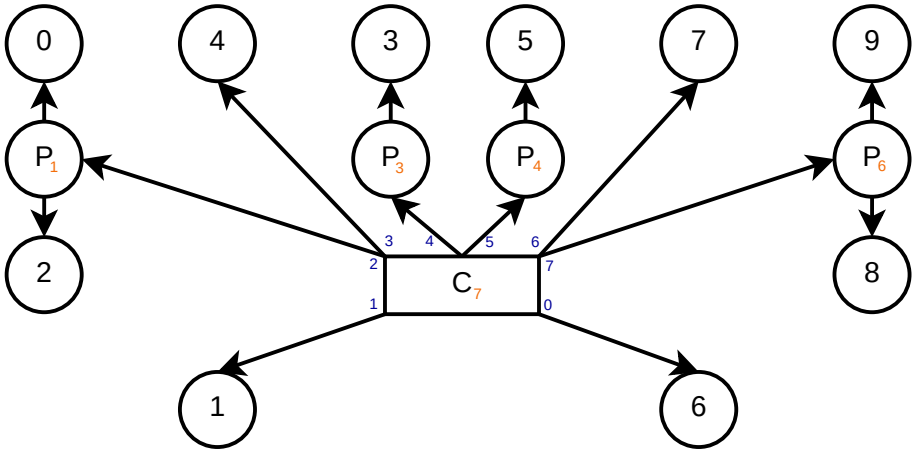
Die correct-Methode

Durch die correct–Methode werden die PC–Tree–Eigenschaften wiederhergestellt, die als Folge des Splitts verletzt wurden.

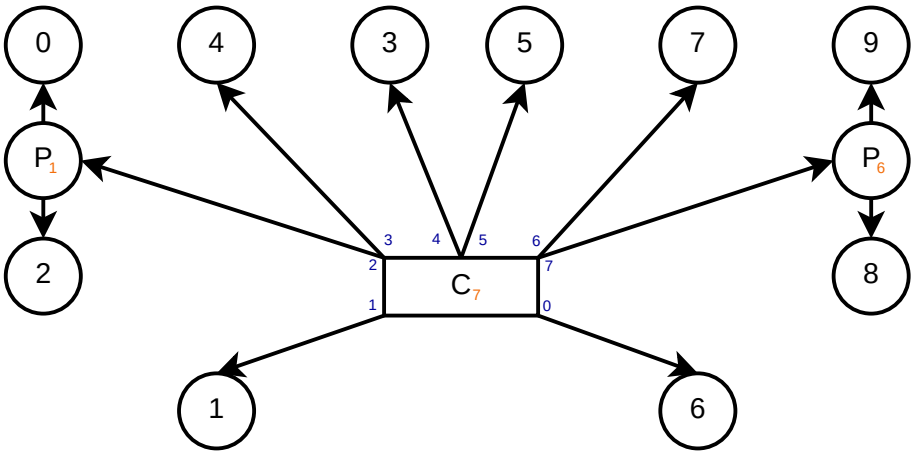
Als Erstes verschmelzen wir alle benachbarten C-Knoten:



Als Nächstes werden alle P- oder C-Knoten entfernt, die nur einen Nachbarn besitzen. Solche Knoten können entstanden sein, wenn auf dem Terminal-Pfad Knoten mit nur vollen und nur leeren Teilbäumen existierten. Folglich wurde entweder an den neu erstellen Knoten kein Teilbaum umgehängt oder es wurden alle Teilbäume umgehängt, so dass an dem zu splittenden Knoten nun keine Teilbäume mehr hängen. Egal um welchen Fall es sich handelt, die Knoten werden einfach entfernt:



Im letzten Schritt entfernen wir noch alle P- oder C-Knoten, die genau zwei Nachbarn besitzen, und verbinden ihre Nachbarn direkt:



6. Schluss

Diese Diplomarbeit stellt eine Konsolenanwendung zur Verfügung, mit der Daten im XML– oder RDF–Format in die Datenspeicherungsstrukturen des Kerns eingelesen werden können. Die Konsolenanwendung bietet einfache Operationen wie das Laden oder Entfernen eines Graphen an. Darüber hinaus können über eine SQL ähnliche Anfragesprache Anfragen auf die Graphdaten vorgenommen werden. Über einen Debug–Modus ist es zusätzlich möglich, eine detaillierte Ausgabe der Anfragebearbeitung auf der Konsole auszugeben. Als Teil der Anfragebearbeitung erstellen wir einen Anfragegraphen, auf dem wir mit Hilfe des Edmonds–Algorithmus eine statische Optimierung vornehmen. Da für den Edmonds–Algorithmus (unseres Wissens) keine eine korrekte, freie Implementierung verfügbar ist und zudem seine Dokumentation schlecht und unvollständig ist, haben wir den Edmonds Algorithmus selbst implementiert. Die Gewichtung bei der Implementierung lag auf Korrektheit und Verständlichkeit. Um die Verständlichkeit des Algorithmus zu maximieren, haben wir versucht, sowohl seine Idee als auch seine Anwendung an vielen Beispielen zu zeigen. Der Großteil dieser Beispiele befindet sich im Anhang. Die Konsolenanwendung bildet eine Testumgebung als Basis für eine zukünftige experimentelle Leistungsauswertung.

Losgelöst von der Konsolenanwendung haben wir eine vom Kern zur Behandlung von CCIRs benötigte Komponente erstellt. Der PC–Tree–Algorithmus ermöglicht es zu bestimmen, ob ein gegebener Graph die CCIR–Eigenschaft besitzt. Sofern dies der Fall ist, lässt sich mit ihm eine (zyklische) Ordnung der Knoten des Graphen bestimmen in der die Kinder jedes Knotens in einem zusammenhängenden Intervall liegen. Ähnlich wie bei dem Edmonds–Algorithmus haben wir diesen an vielen Beispielen vorgeführt und gezeigt, unter welchen Voraussetzungen der Algorithmus scheitert und sich keine Ordnung bestimmen lässt.

Für die elementaren Programmstücke und die Algorithmen sind einfache Tests verfügbar. Diese eignen sich auch sehr gut, um den Zugang zu unserer Implementierung und dessen Funktionsweise zu erleichtern.

Die ausführlichen Beispiele für den Edmonds– und PC–Tree–Algorithmus sollen eine Grundlage für eine zukünftige effizientere Implementierung dieser beiden Algorithmen bieten.

Als zukünftige Arbeit ist eine Erweiterung der einlesbaren Datenquellen und die Verarbeitung der RDF–Daten über die Konsolenapllikation notwendig. Zudem sollte der Parser für die Anfragesprache verbessert bzw. die Anfragesprache selbst erweitert werden. Durch die Einbindung des ITCH Joins[3] sollte eine effizientere Anfrageauswertung möglich sein, die in einer experimentelle Leistungsauswertung der Datenspeicherungsstrukturen des Kerns belegt werden sollte. Zusätzlich sollten der Edmonds–Algorithmus und der PC–Tree–Algorithmus jeweils durch effizientere Implementierung ersetzt werden.

Literatur

[1] S. Brodt. Efficient and practical access to any graph data: Data structures for continuous image graphs and beyond. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2009.

[2] J. Edmonds. Optimum branchings. *Jour. of Research of the National Bureau of Standards*, 71B:233 – 240, 1967.

[3] T. Furche. *Implementation of Web Query Language Reconsidered: Beyond Tree and Single-Language Algebras at (Almost) No Cost*. Dissertation/doctoral thesis, Ludwig-Maxmilians University Munich, 2008.

[4] W.-L. Hsu and R. M. McConnell. Pc trees and circular-ones arrangements. *Theoretical Computer Science*, 296(1):99 – 116, 2003.

A. Gesamtübersicht über implementierte Klassen / Dateien

```
.
|-- Edmonds/
|   |-- EdmondsEdge.cs
|   |-- EdmondsAlgorithm.cs
|   |-- EdmondsGraph.cs
|   `-- EdmondsNode.cs
|-- GraphDescriptor/
|   |-- ExampleGraphDescriptors.cs
|   |-- GraphDescriptor.cs
|   `-- GraphDescriptorDataBase.cs
|-- Hilfsklassen/
|   |-- ListPrinter.cs
|   |-- SubOperatorsSorter.cs
|   `-- TwoDimensionalStringArrayPrinter.cs
|-- Parser/
|   |-- IDIDREFDescriptor.cs
|   |-- IDIDREFOption.cs
|   |-- IDREFTriples.cs
|   |-- ParserSettings.cs
|   |-- RDFParser.cs
|   |-- RDFParserSettings.cs
|   |-- TwoWayDictionary.cs
|   |-- XMLParser.cs
|   `-- XMLParserSettings.cs
|-- Query/
|   |-- Operators/
|   |   |-- SubOperators/
|   |   |   |-- BinaryContainsIntArrayEnumerable.cs
|   |   |   |-- BinaryContainsOperator.cs
|   |   |   |-- CopyAttributeIntArrayEnumerable.cs
|   |   |   |-- CopyAttributeOperator.cs
|   |   |   |-- Cut1IntArrayEnumerable.cs
|   |   |   |-- Cut1Operator.cs
|   |   |   |-- Cut2IntArrayEnumerable.cs
|   |   |   |-- Cut2Operator.cs
|   |   |   |-- DeleteAttributeIntArrayEnumerable.cs
|   |   |   |-- DeleteAttributeOperator.cs
|   |   |   |-- InitialIntArrayEnumerable.cs
|   |   |   |-- InitialOperator.cs
|   |   |   |-- LabelIntArrayEnumerable.cs
|   |   |   |-- LabelOperator.cs
|   |   |   |-- RevLabelIntArrayEnumerable.cs
|   |   |   |-- RevLabelOperator.cs
```

```
|   |   |   |-- SelectionIntArrayEnumerable.cs
|   |   |   `-- SelectionOperator.cs
|   |   |-- CartesianProductIntArrayEnumerable.cs
|   |   |-- CartesianProductOperator.cs
|   |   |-- ChangeOrderIntArrayEnumerable.cs
|   |   |-- ChangeOrderOperator.cs
|   |   |-- IntArrayEnumerable.cs
|   |   |-- Operator.cs
|   |   `-- SubOperator.cs
|   |-- Projection/
|   |   |-- AProjection.cs
|   |   |-- AsProjection.cs
|   |   `-- Projection.cs
|   |-- QueryGraph/
|   |   |-- AQueryGraphEdge.cs
|   |   |-- QueryGraph.cs
|   |   |-- QueryGraphContainsEdge.cs
|   |   |-- QueryGraphCutEdge.cs
|   |   `-- QueryGraphNode.cs
|   |-- Relation/
|   |   |-- Attribute.cs
|   |   |-- Relation.cs
|   |   |-- RelationAttribute.cs
|   |   `-- RelationType.cs
|   |-- Selection/
|   |   |-- ASelection.cs
|   |   |-- JoinSelection.cs
|   |   |-- TrueIntSelection.cs
|   |   `-- TrueStringSelection.cs
|   |-- Query.cs
|   `-- QueryParser.cs
|-- QueryConsole/
|   |-- QueryConsole.cs
|   `-- QueryConsoleBatches.cs
|-- Testfiles/
|   |-- 2books.xml
|   |-- friends.n3
|   `-- gaesteliste.xml
`-- Unit-Tests/
    |-- EdondsEdgeUnitTest.cs
    |-- EdmondsAlgorithmUnitTest.cs
    |-- EdmondsGraphUnitTest.cs
    |-- NodeUnitTest.cs
    |-- PCTreeUnitTest.cs
    |-- QueryUnitTest.cs
    |-- RunCSUnitTest.cs
```

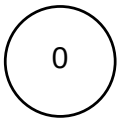
```
        `-- SubOperatorsSorterUnitTest.cs


.
`-- PCTree/
    |-- CNode.cs
    |-- NNode.cs
    |-- Node.cs
    |-- PCTree.cs
    `-- PNode.cs
```

B. Weitere Beispiele für die Anwendung des Edmonds–Algorithmus

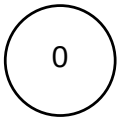
In der Klasse *EdmondsAlgorithmUnitTest* sind Beispielgraphen abgelegt, die zum Testen der Korrektheit des Edmonds–Algorithmus verwendet werden können. Diese Beispiele werden hier Schritt für Schritt ausgeführt.

B.1. oneNodeOnly

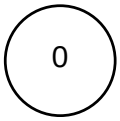
Gegeben sei der folgende Graph G_0:



Wir bestimmen den Hilfsgraphen B_0:

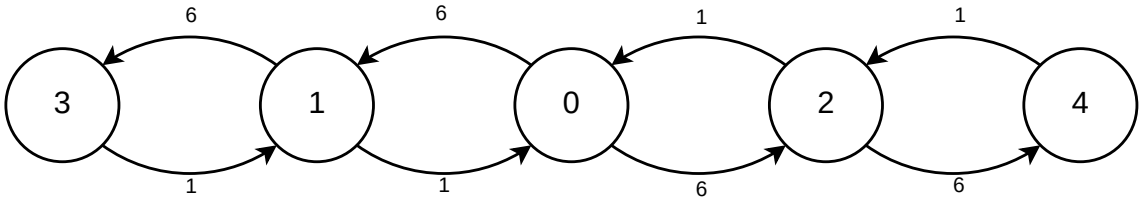


B_0 enthält keine Zyklen. Damit ist $R_0 = B_0$ ein maximales Branching zu G_0. R_0:

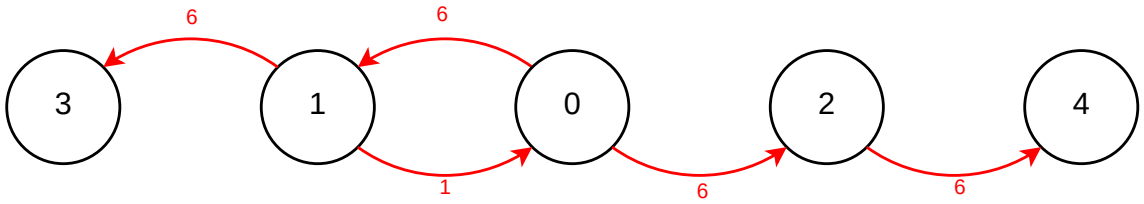


B.2. example1

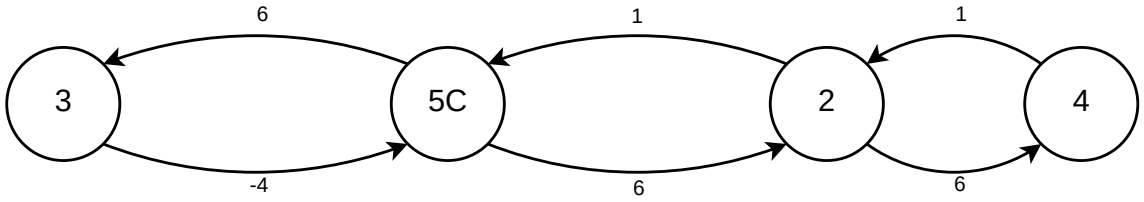
Gegeben sei der folgende Graph G_0:



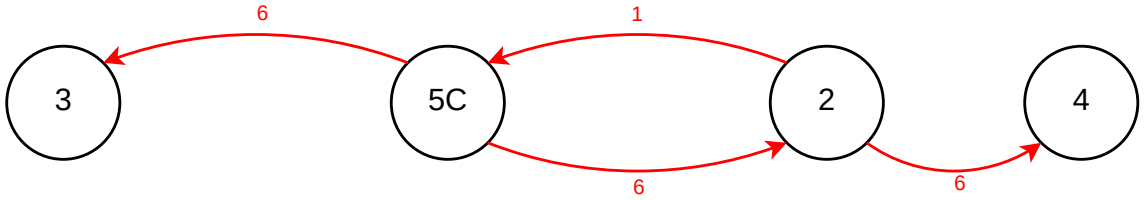
Wir bestimmen den Hilfsgraphen B_0:



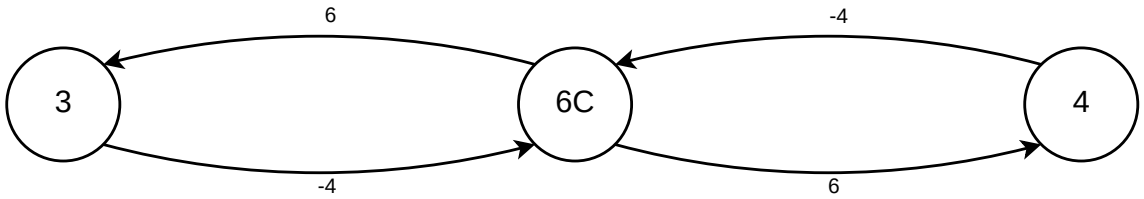
B_0 enthält den Zyklus $0 \to 1 \to 0$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 5C. Wir merken uns $\Psi(2, 5C) = (2, 0)$, $\Psi(5C, 2) = (0, 2)$, $\Psi(3, 5C) = (3, 1)$, $\Psi(5C, 3) = (1, 3)$. Wir bestimmen das neue Gewicht der Kanten $(2, 5C)$ und $(3, 5C)$. $c(2, 5C) = 1$, $c(3, 5C) = -4$. Durch diese Änderungen erhalten wir G_1:



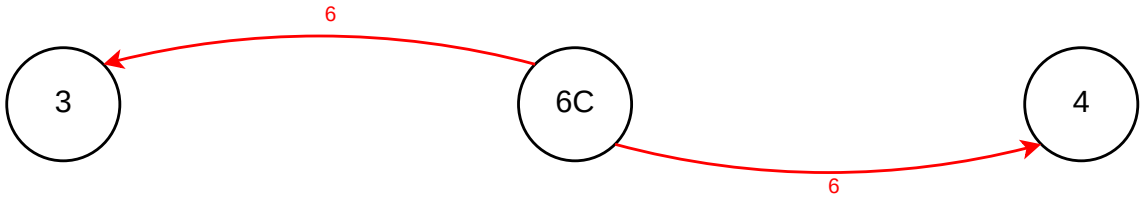
Wir bestimmen den Hilfsgraphen B_1:



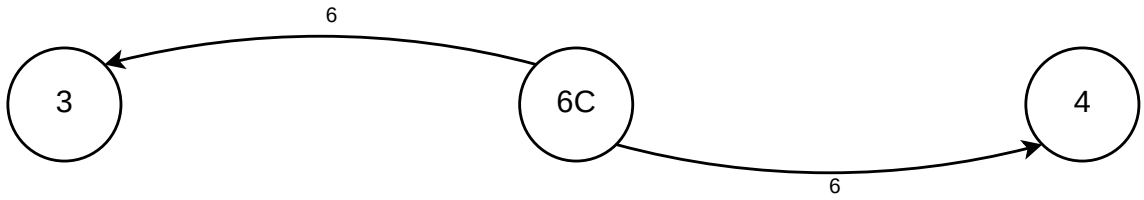
B_1 enthält den Zyklus $2 \to 5C \to 2$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 6C. Wir merken uns $\Psi(3, 6C) = (3, 5C)$, $\Psi(6C, 3) = (5C, 3)$, $\Psi(4, 6C) = (4, 2)$, $\Psi(6C, 4) = (2, 4)$. Wir bestimmen das neue Gewicht der Kanten $(3, 6C)$ und $(4, 6C)$. $c(3, 6C) = -4$, $c(4, 6C) = -4$. Durch diese Änderungen erhalten wir G_2:



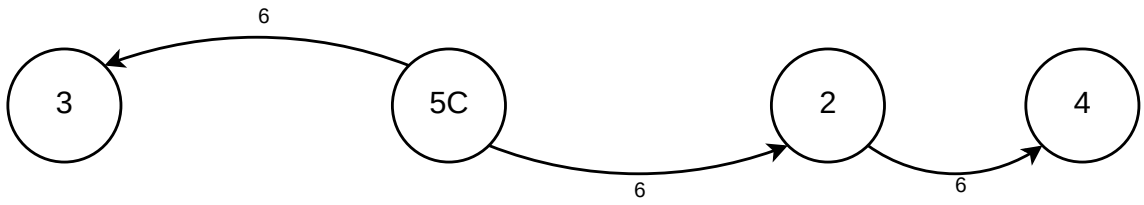
Wir bestimmen den Hilfsgraphen B_2:



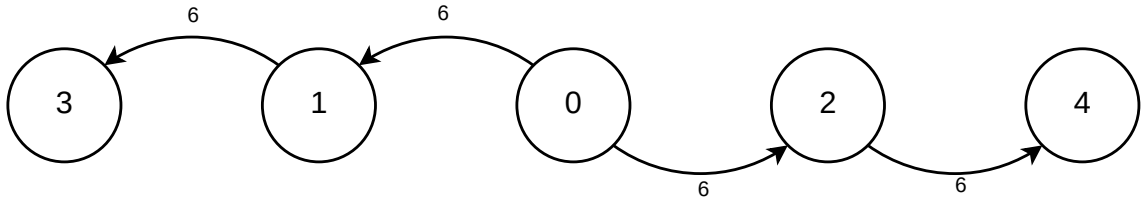
B_2 enthält keine Zyklen. Wir beenden die Kontraktion. Damit ist $R_2 = B_2$ ein maximales Branching zu G_2. R_2:



Wir beginnen die Expansion. Wir übernehmen die Knoten von G_1. Da in den Knoten $6C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $2 \to 5C \to 2$ bis auf die leichteste Kante. Die Kanten (6C,3) und (6C,4) ersetzen wir durch ihre Vorgänger (5C,3) und (2,4). Damit erhalten wir R_1:



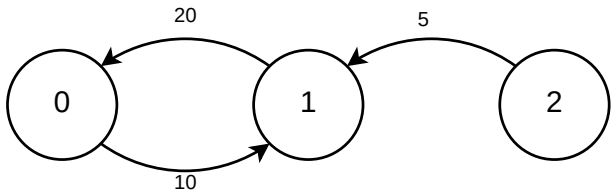
Wir setzen die Expansion fort. Wir übernehmen die Knoten von G_0. Da in den Knoten $5C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $0 \to 1 \to 0$ bis auf die leichteste Kante. Die Kanten (5C,2) und (5C,3) ersetzen wir durch ihre Vorgänger (0,2) und (1,3). Damit erhalten wir R_0:



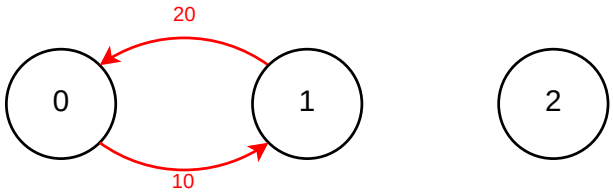
Die Expansion ist damit beendet. R_0 ein maximales Branching zu G_0.

B.3. example2

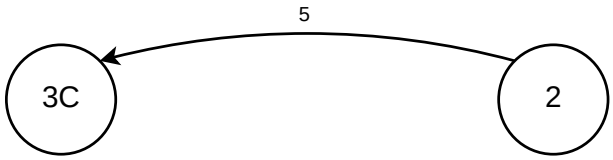
Gegeben sei der folgende Graph G_0:



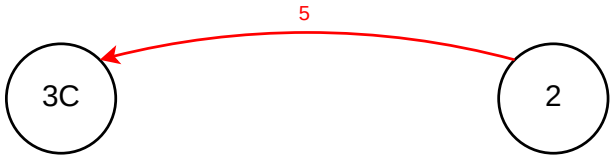
Wir bestimmen den Hilfsgraphen B_0:



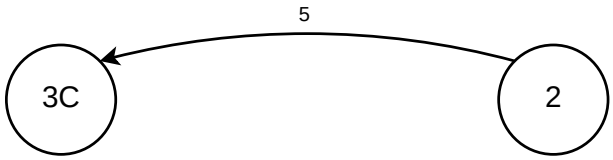
B_0 enthält den Zyklus $0 \to 1 \to 0$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 3C. Wir merken uns $\Psi(2, 3C) = (2, 1)$. Wir bestimmen das neue Gewicht der Kanten $(2, 3C)$. $c(2, 3C) = 5$. Durch diese Änderungen erhalten wir G_1:



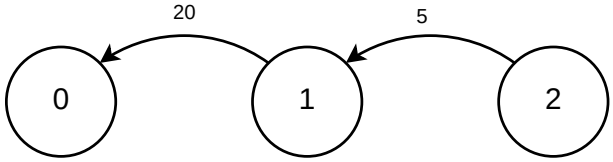
Wir bestimmen den Hilfsgraphen B_1:



B_1 enthält keine Zyklen. Wir beenden die Kontraktion. Damit ist $R_1 = B_1$ ein maximales Branching zu G_1. R_1:



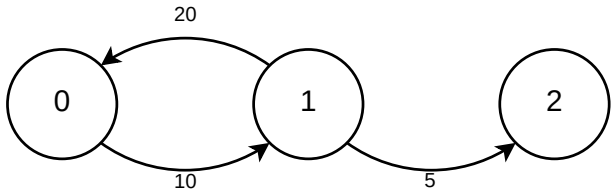
Wir beginnen die Expansion. Wir übernehmen die Knoten von G_0. Da in den Knoten $3C$ eine Kante führt, nehmen wir alle Kanten des Zyklus $0 \to 1 \to 0$ bis auf die Kante, die in den gleichen Knoten wie $\Psi(2, 3C)$ führt. Die Kante (2,3C) wird durch ihren Vorgänger (2,1) ersetzt. Damit erhalten wir R_0:



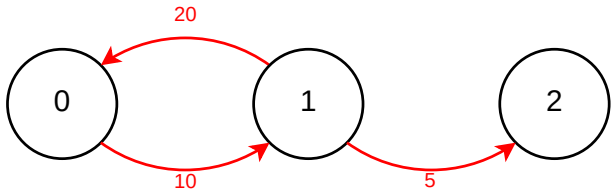
Die Expansion ist damit beendet. R_0 ist ein maximales Branching zu G_0.

B.4. example3

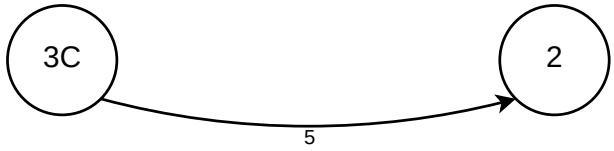
Gegeben sei der folgende Graph G_0:



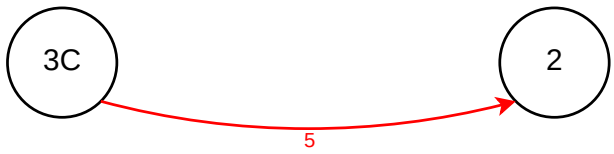
Wir bestimmen den Hilfsgraphen B_0:



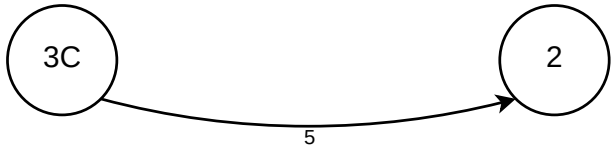
B_0 enthält den Zyklus $0 \to 1 \to 0$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 3C. Wir merken uns $\Psi(3C, 2) = (1, 2)$. Durch diese Änderungen erhalten wir G_1:



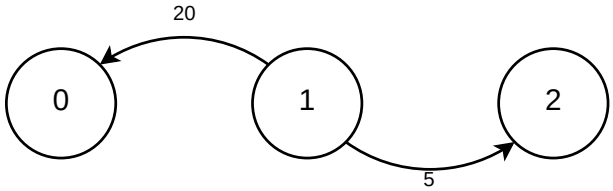
Wir bestimmen den Hilfsgraphen B_1:



B_1 enthält keine Zyklen. Wir beenden die Kontraktion. Damit ist $R_1 = B_1$ ein maximales Branching zu G_1. R_1:



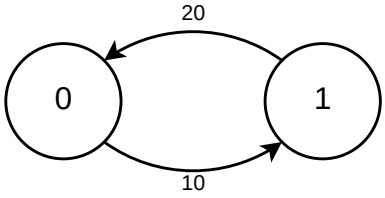
Wir beginnen die Expansion. Wir übernehmen die Knoten von G_0. Da in den Knoten $3C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $0 \to 1 \to 0$, bis auf die leichteste Kante. Die Kante (3C,2) wird durch ihren Vorgänger (1,2) ersetzt. Damit erhalten wir R_0:



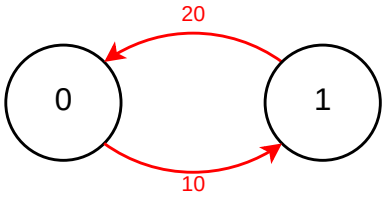
Die Expansion ist damit beendet. R_0 ist ein maximales Branching zu G_0.

B.5. example4

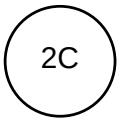
Gegeben sei der folgende Graph G_0:



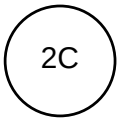
Wir bestimmen den Hilfsgraphen B_0:



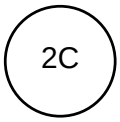
B_0 enthält den Zyklus $0 \to 1 \to 0$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 2C. Durch diese Änderungen erhalten wir G_1:



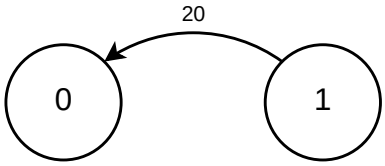
Wir bestimmen den Hilfsgraphen B_1:



B_1 enthält keine Zyklen. Wir beenden die Kontraktion. Damit ist $R_1 = B_1$ ein maximales Branching zu G_1. R_1:



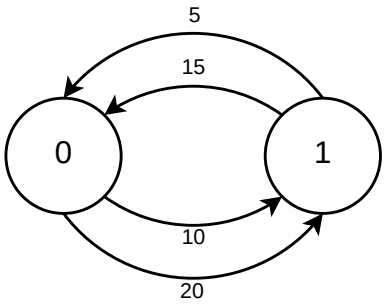
Wir beginnen die Expansion. Wir übernehmen die Knoten von G_0. Da in den Knoten $2C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $0 \to 1 \to 0$, bis auf die leichteste Kante. Damit erhalten wir R_0:



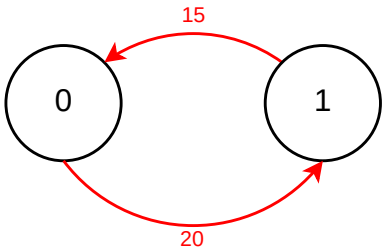
Die Expansion ist damit beendet. R_0 ist ein maximales Branching zu G_0.

B.6. example5

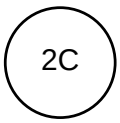
Gegeben sei der folgende Graph G_0:



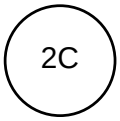
Wir bestimmen den Hilfsgraphen B_0:



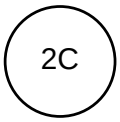
B_0 enthält den Zyklus $0 \to 1 \to 0$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 2C. Durch diese Änderungen erhalten wir G_1:



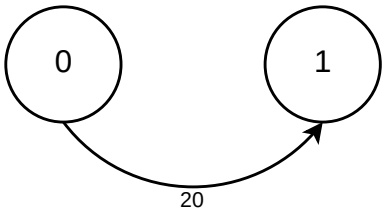
Wir bestimmen den Hilfsgraphen B_1:



B_1 enthält keine Zyklen. Wir beenden die Kontraktion. Damit ist $R_1 = B_1$ ein maximales Branching zu G_1. R_1:



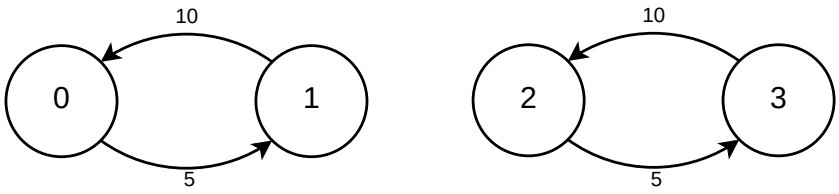
Wir beginnen die Expansion. Wir übernehmen die Knoten von G_0. Da in den Knoten $2C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $0 \to 1 \to 0$ von B_0, bis auf die leichteste Kante. Damit erhalten wir R_0:



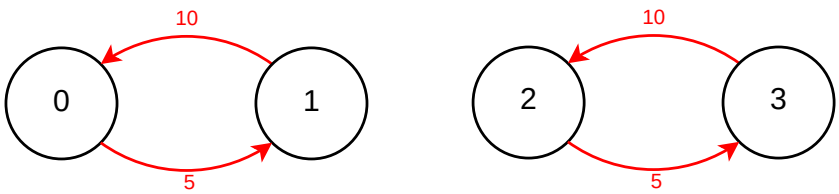
Die Expansion ist damit beendet. R_0 ist ein maximales Branching zu G_0.

B.7. example6

Gegeben sei der folgende Graph G_0:



Wir bestimmen den Hilfsgraphen B_0:

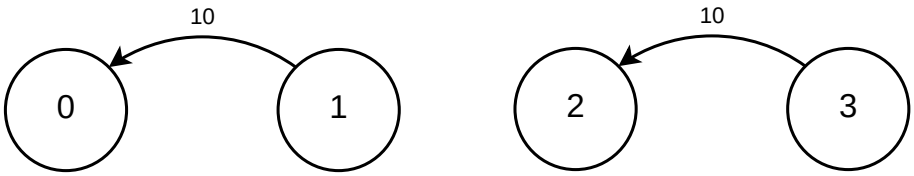


B_0 enthält den Zyklus $0 \to 1 \to 0$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 4C. B_0 enthält den Zyklus $2 \to 3 \to 2$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 5C. Durch diese Änderungen erhalten wir G_1:

Wir bestimmen den Hilfsgraphen B_1:

B_1 enthält keine Zyklen. Wir beenden die Kontraktion. Damit ist $R_1 = B_1$ ein maximales Branching zu G_1. R_1:

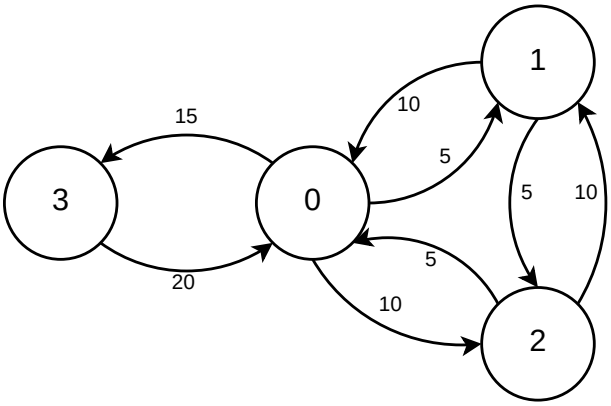
Wir beginnen die Expansion. Wir übernehmen die Knoten von G_0. Da in den Knoten $4C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $0 \to 1 \to 0$ von B_0, bis auf die leichteste Kante. Da in den Knoten $5C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $2 \to 3 \to 2$ von B_0, bis auf die leichteste Kante. Damit erhalten wir R_0:



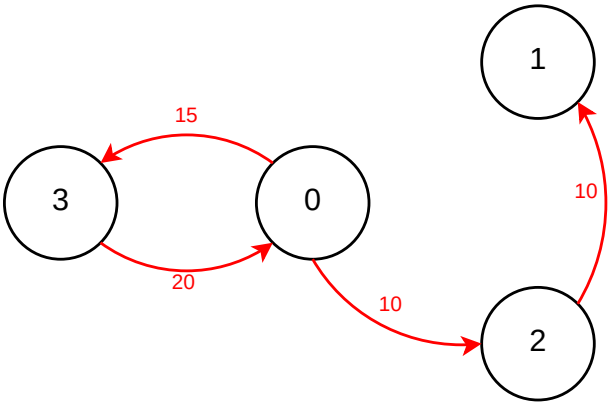
Die Expansion ist damit beendet. R_0 ist ein maximales Branching zu G_0.

B.8. example7

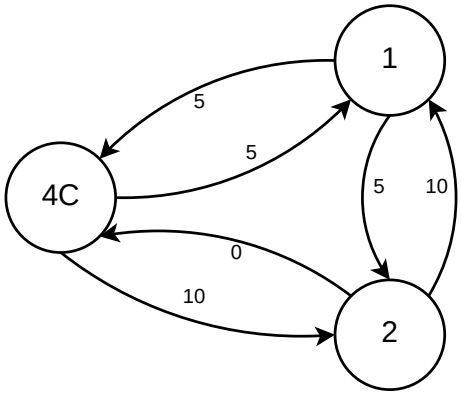
Gegegen sei der folgende Graph G_0:



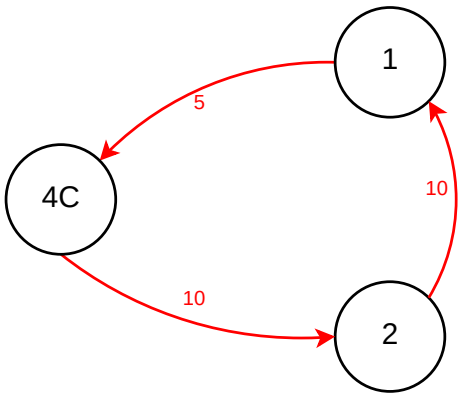
Wir bestimmen den Hilfsgraphen B_0:



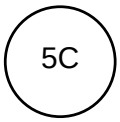
B_0 enthält den Zyklus $0 \to 3 \to 0$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 4C. Wir merken uns $\Psi(1, 4C) = (1, 0)$, $\Psi(4C, 1) = (0, 1)$, $\Psi(2, 4C) = (2, 0)$, $\Psi(4C, 2) = (0, 2)$. Wir bestimmen das neue Gewicht der Kanten $(1, 4C)$ und $(2, 4C)$. $c(1, 4C) = 10 - 20 + 15 = 5$, $c(2, 4C) = 5 - 20 + 15 = 0$. Durch diese Änderungen erhalten wir G_1:



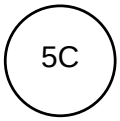
Wir bestimmen den Hilfsgraphen B_1:



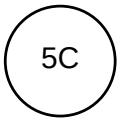
B_1 enthält den Zyklus $1 \to 4C \to 2 \to 1$. Wir ersetzen diesen Zyklus durch einen neuen Schrumpfknoten 5C. Durch diese Änderungen erhalten wir G_2:



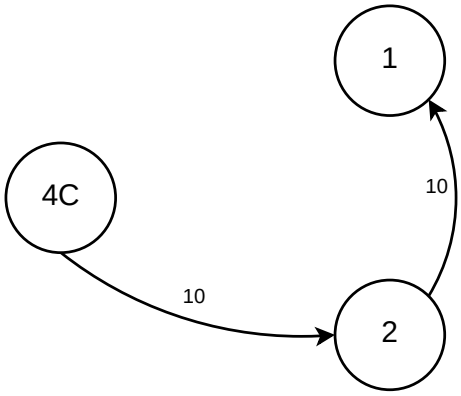
Wir bestimmen den Hilfsgraphen B_2:



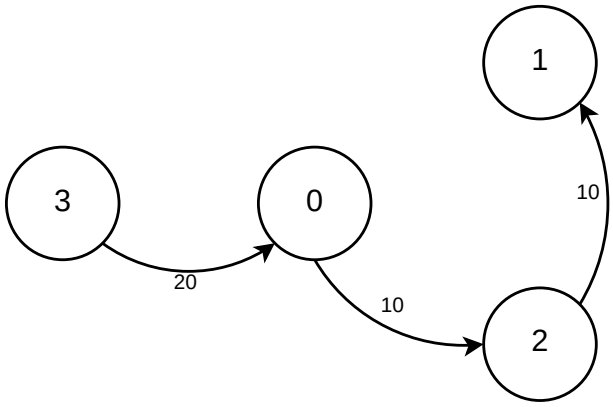
B_2 enthält keine Zyklen. Wir beenden die Kontraktion. Damit ist $R_2 = B_2$ ein maximales Branching zu G_2. R_2:



Wir beginnen die Expansion. Wir übernehmen die Knoten von G_1. Da in den Knoten $5C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $1 \to 4C \to 2 \to 1$ von B_1, bis auf die leichteste Kante. Damit erhalten wir R_1:



Wir setzen die Expansion fort. Wir übernehmen die Knoten von G_0. Da in den Knoten $4C$ keine Kante führt, nehmen wir alle Kanten des Zyklus $0 \to 3 \to 0$ von B_0, bis auf die leichteste Kante. Die Kante (4C,2) ersetzen wir durch ihre Vorgängerkante (0,2). Damit erhalten wir R_0:



Die Expansion ist damit beendet. R_0 ist ein maximales Branching zu G_0.

C. Weitere Beispiele für die Anwendung des PC–Trees

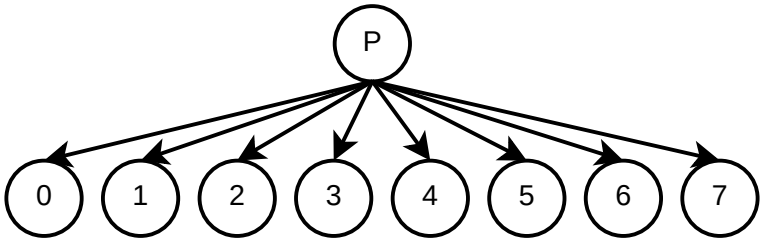
In der Klasse *PCTreeUnitTest* sind Beispiele abgelegt, die zum Testen der Korrektheit des PC–Tree Algorithmus verwendet werden können. Diese Beispiele werden hier Schritt für Schritt ausgeführt.

C.1. example1 Error1

Gegeben sei ein Graph G, durch seine Adjazenzmatrix:

	0	1	2	3	4	5	6	7
0	1	1	1	1	0	0	0	0
1	1	1	0	0	1	1	0	0
2	1	0	1	0	1	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

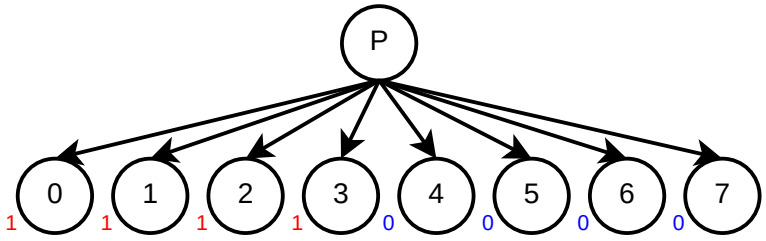
Wir initialisieren den PC-Tree:



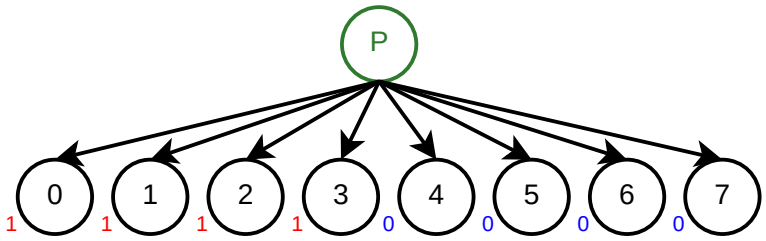
Wir arbeiten jede Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7
Zeile 0:	1	1	1	1	0	0	0	0

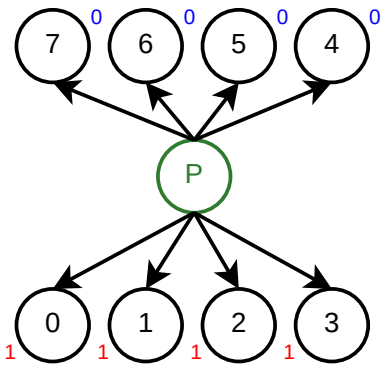
Wir markieren entsprechend der Zeile 0:



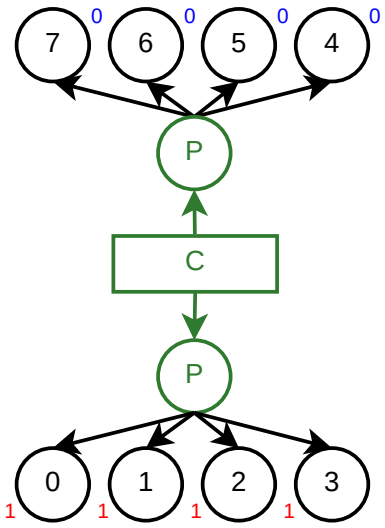
Wir bestimmen den Terminal-Pfad. Dieser besteht lediglich aus einem P-Knoten:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



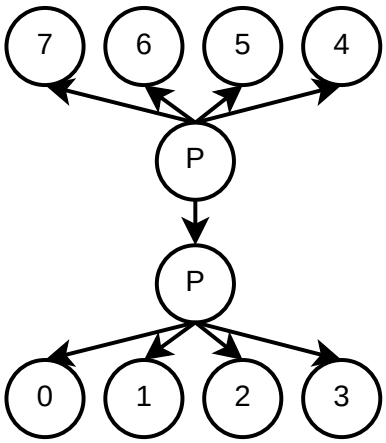
Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!

Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

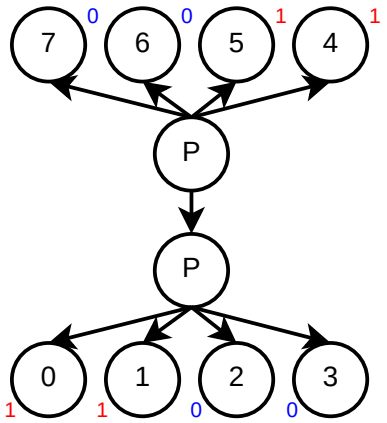
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen, und verbinden die Nachbarn direkt:



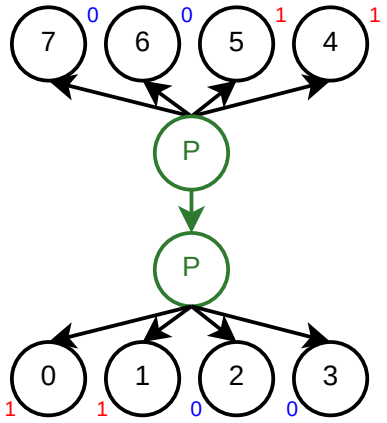
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7
Zeile 1:	1	1	0	0	1	1	0	0

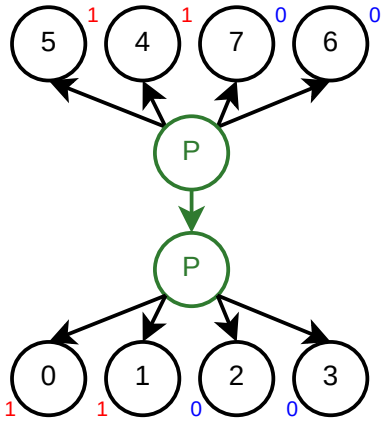
Wir markieren entsprechend der Zeile 1:



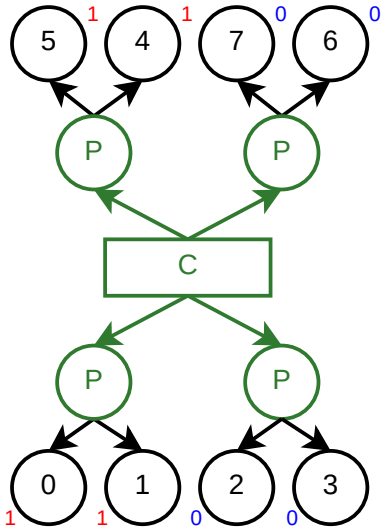
Wir bestimmen den Terminal-Pfad:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



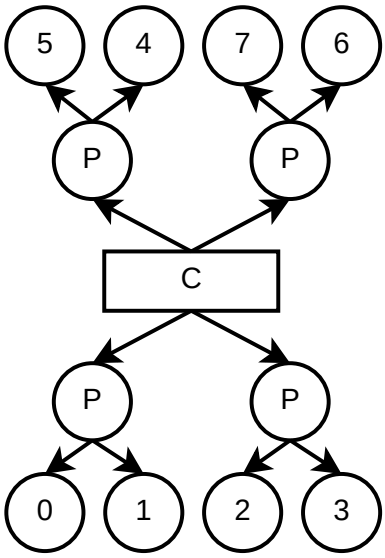
Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!

Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

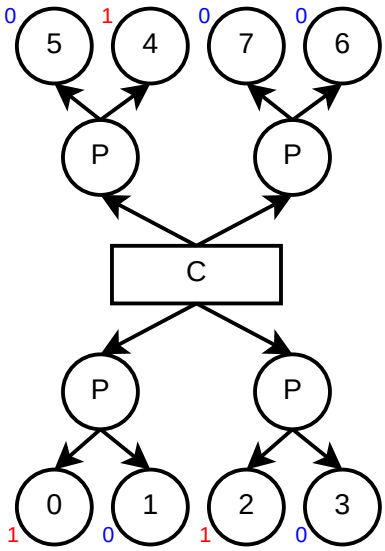
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen, und verbinden die Nachbarn direkt: Nichts zu tun!



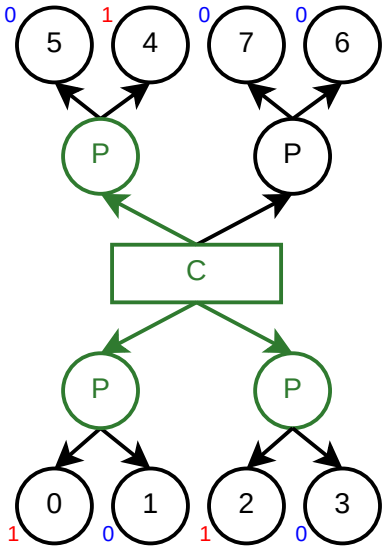
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7
Zeile 2:	1	0	1	0	1	0	0	0

Wir markieren entsprechend der Zeile 2:



Wir bestimmen den Terminal-Pfad:



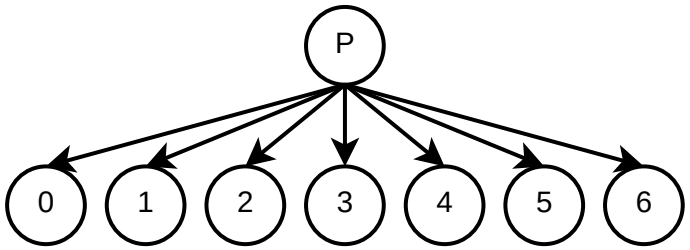
Der Terminal-Pfad ist kein Pfad (Fehler-Fall 1). Es ist somit nicht möglich, alle vollen Blätter auf die eine Seite des Pfades und alle leeren Blätter auf die andere Seite des Pfades zu bringen. Der PC-Tree kann nicht zerschnitten werden. Die Abarbeitung endet hier.

C.2. example2 Error2a

Gegeben sei ein Graph G durch seine Adjazenzmatrix:

	0	1	2	3	4	5	6
0	0	0	1	1	1	1	1
1	0	0	0	0	1	1	1
2	1	0	0	1	0	0	1
3	1	0	1	0	1	0	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

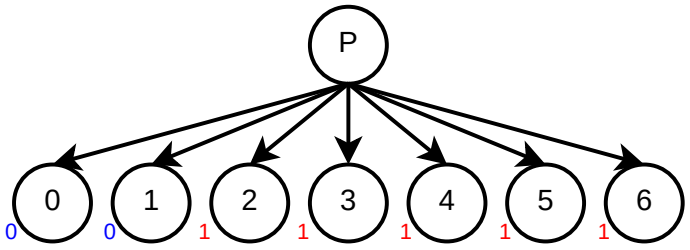
Wir initialisieren den PC-Tree:



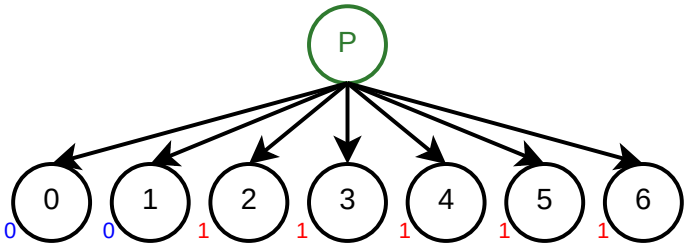
Wir arbeiten jede Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6
Zeile 0:	0	0	1	1	1	1	1

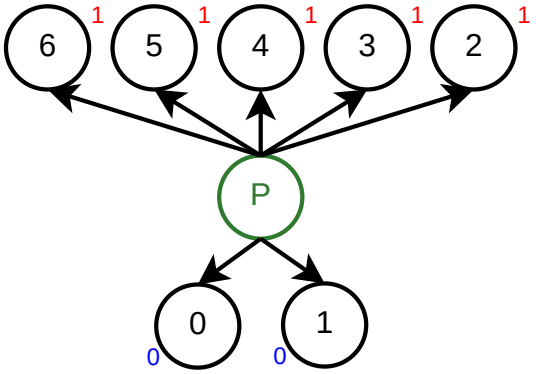
Wir markieren entsprechend der Zeile 0:



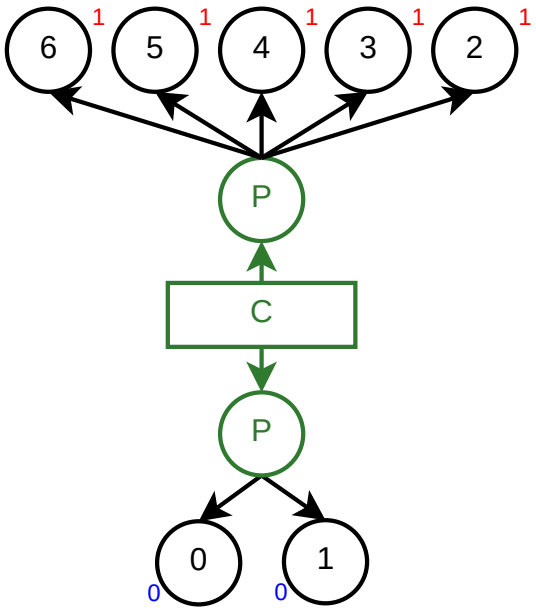
Wir bestimmen den Terminal-Pfad. Dieser besteht lediglich aus einem P-Knoten:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



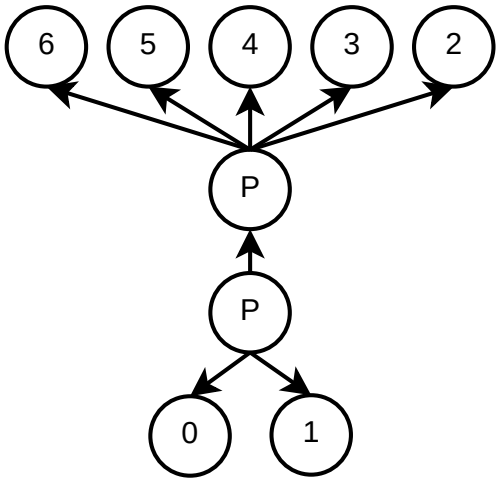
Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!

Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

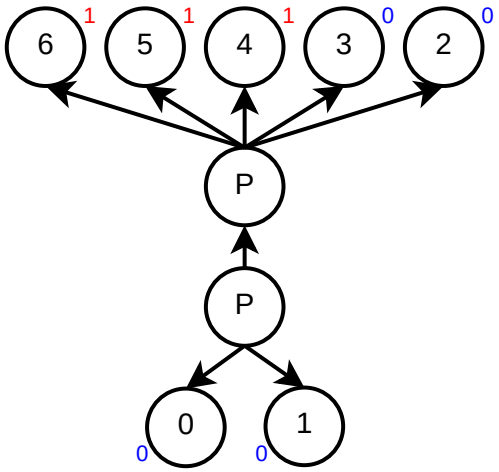
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen, und verbinden die Nachbarn direkt:



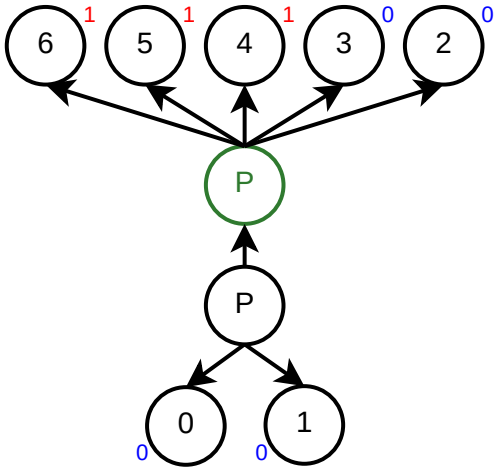
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6
Zeile 1:	0	0	0	0	1	1	1

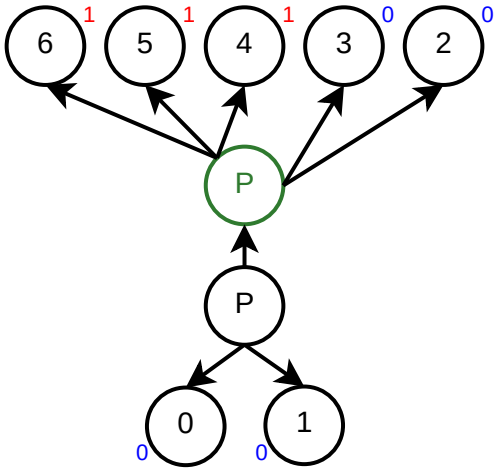
Wir markieren entsprechend der Zeile 1:



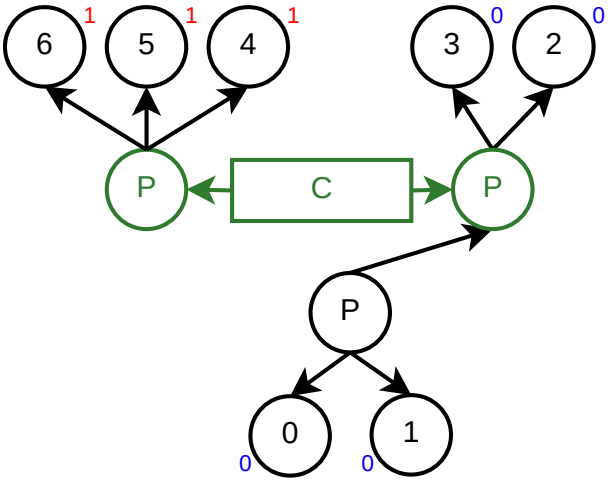
Wir bestimmen den Terminal-Pfad:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



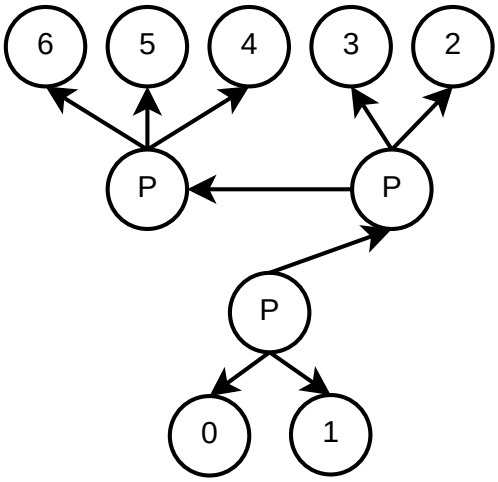
Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!

Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

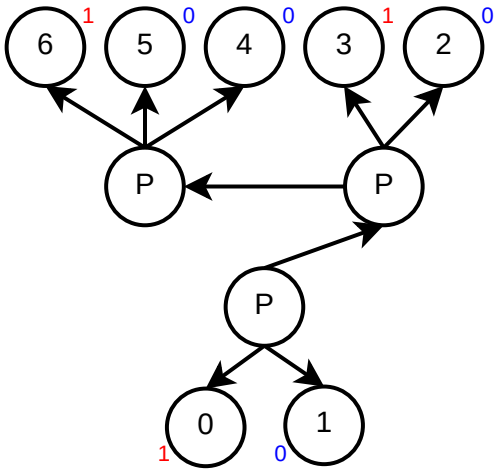
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt:



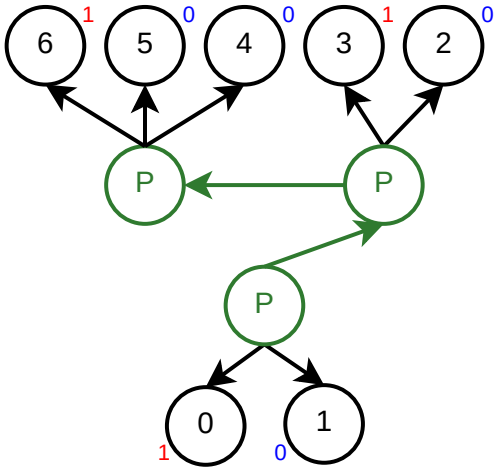
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6
Zeile 2:	1	0	0	1	0	0	1

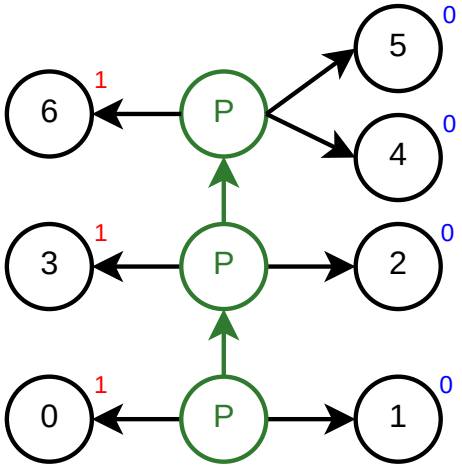
Wir markieren entsprechend der Zeile 2:



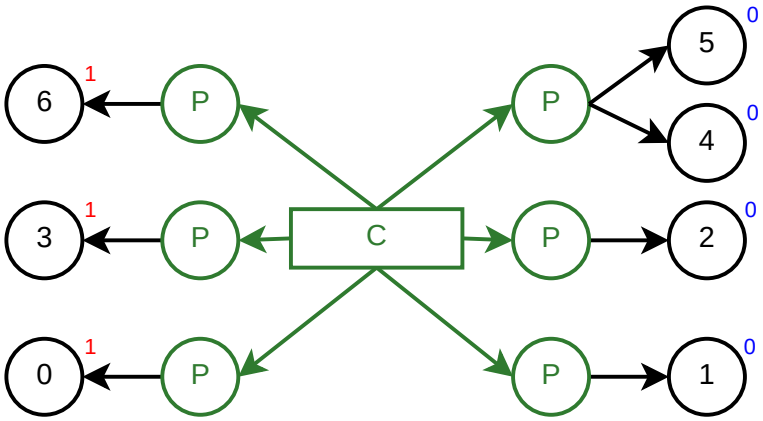
Wir bestimmen den Terminal-Pfad:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



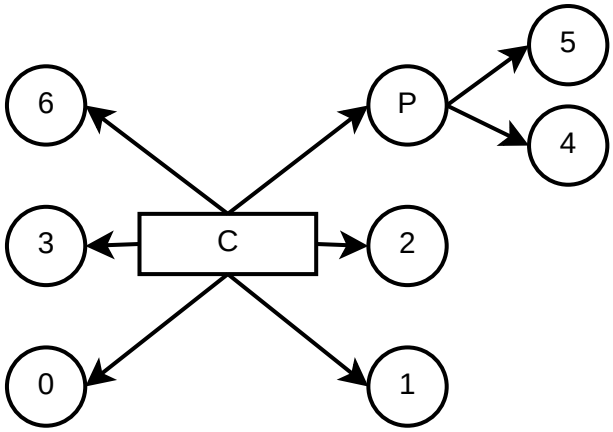
Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!

Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

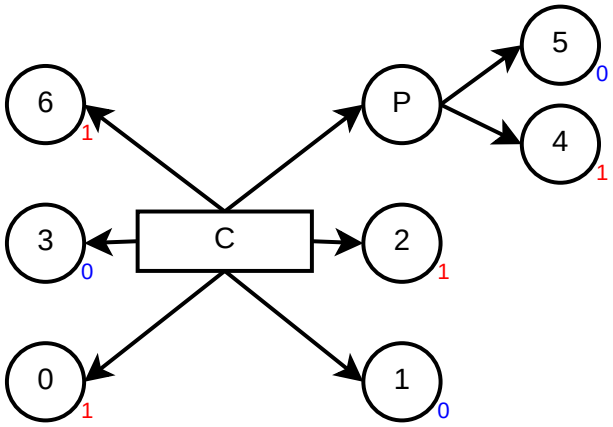
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt:



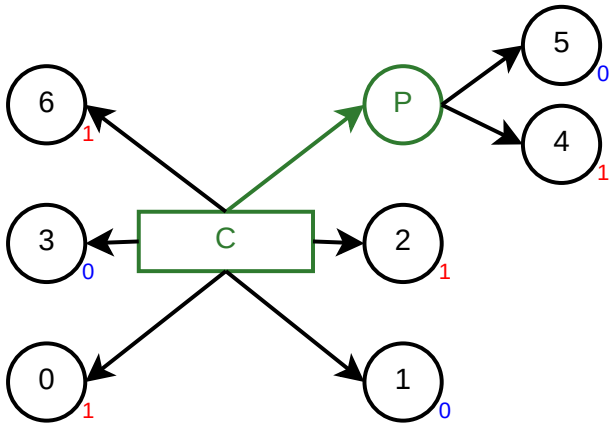
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6
Zeile 3:	1	0	1	0	1	0	1

Wir markieren entsprechend der Zeile 3:



Wir bestimmen den Terminal-Pfad:



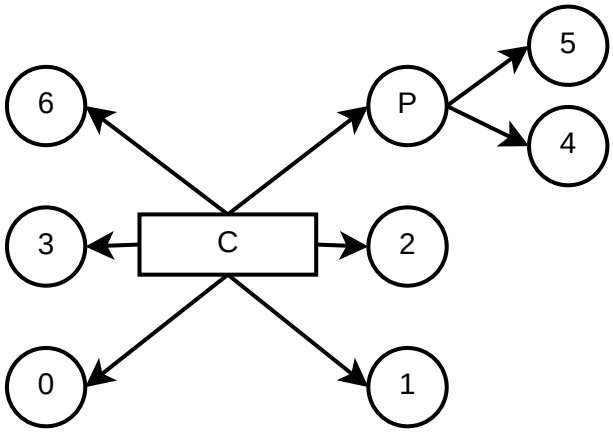
Der Terminal-Pfad ist bei diesem Beispiel zwar ein Pfad, jedoch sind die vollen Blätter des C-Knoten nicht konsekutiv angeordnet (Fehler-Fall 2a). Damit haben wir einen C-Knoten auf dem Pfad, der sich teilen lässt. Der PC-Tree kann nicht zerschnitten werden. Die Abarbeitung endet hier.

C.3. example3 Error2b

Gegeben sei ein Graph G durch seine Adjazenzmatrix:

	0	1	2	3	4	5	6
0	0	0	1	1	1	1	1
1	0	0	0	0	1	1	1
2	1	0	0	1	0	0	1
3	1	1	0	1	1	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

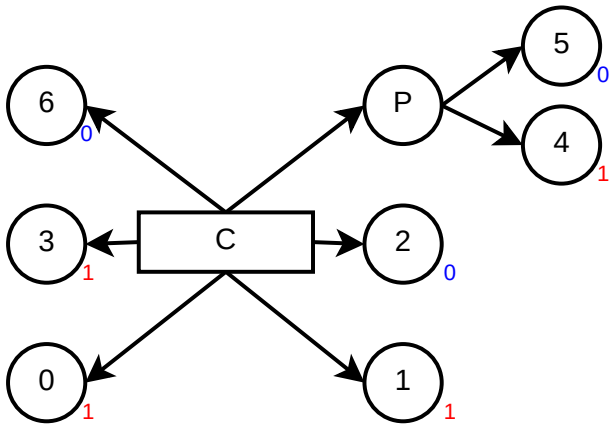
Die Zeilen 0 bis einschließlich Zeile 2 stimmen mit example7 überein.
Wir setzen daher einfach nach der Abarbeitung der Zeilen 0 bis einschließlich 3 fort:



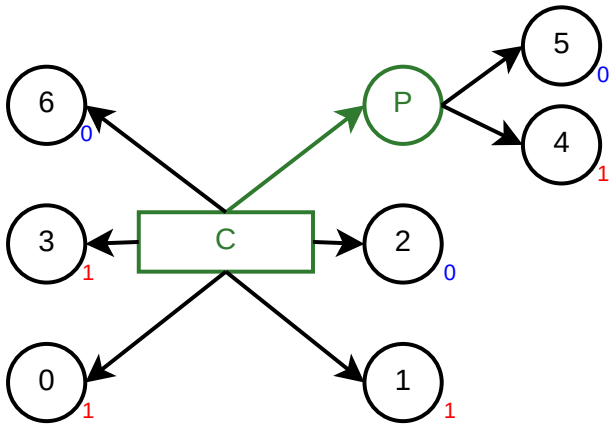
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6
Zeile 3:	1	1	0	1	1	0	0

Wir markieren entsprechend der Zeile 3:



Wir bestimmen den Terminal-Pfad:



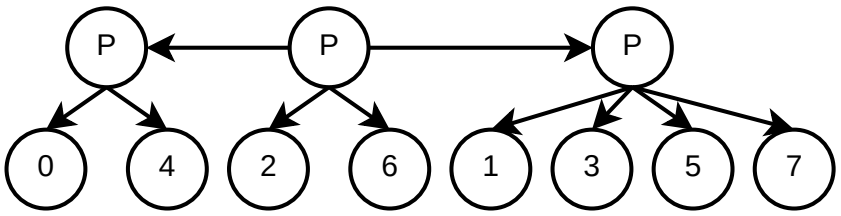
Der Terminal-Pfad ist bei diesem Beispiel zwar ein Pfad und auch die vollen Blätter des C–Knoten sind konsekutiv angeordnet, jedoch gilt dies nicht für die leeren (Fehler–Fall 2b). Damit haben wir einen C-Knoten auf dem Pfad, der sich teilen lässt. Der PC–Tree kann nicht zerschnitten werden. Die Abarbeitung endet hier.

C.4. example4

Gegeben sei ein Graph G durch seine Adjazenzmatrix:

	0	1	2	3	4	5	6	7
0	1	0	0	0	1	0	0	0
1	0	1	0	1	0	1	0	1
2	1	0	1	0	1	0	1	0
3	1	0	1	0	1	0	1	0
4	1	0	0	1	1	0	0	0
5	0	0	0	1	0	1	0	1
6	1	0	0	0	1	0	1	0
7	1	0	0	1	1	0	1	0

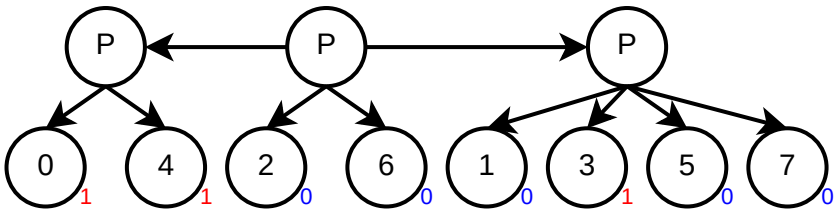
Die Initialisierung und die Abarbeitung der Zeilen 0 bis einschließlich 3 sparen wir und und fangen nach Zeile 3 an. Der PC-Tree nach Zeile 3:



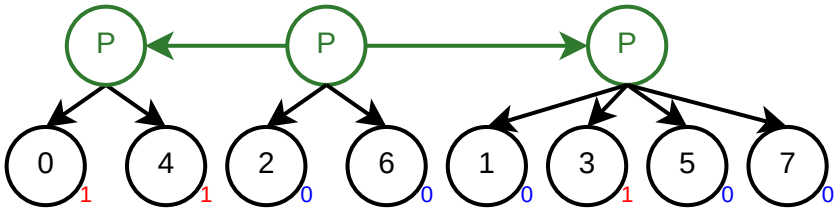
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7
Zeile 4:	1	0	0	1	1	0	0	0

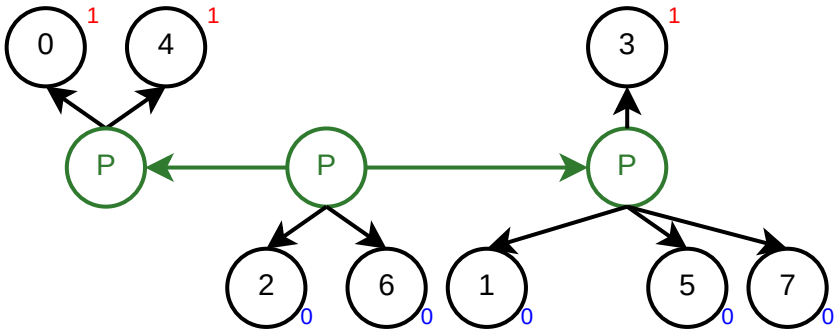
Wir markieren entsprechend der Zeile 4:



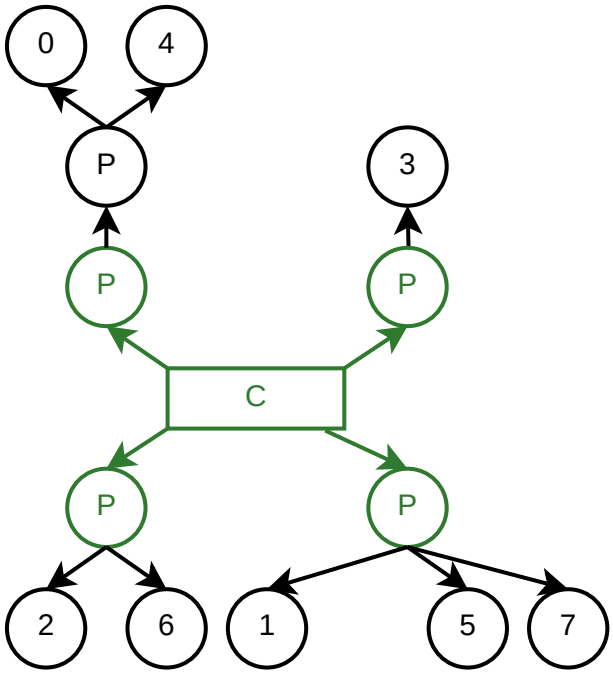
Wir bestimmen den Terminal-Pfad:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



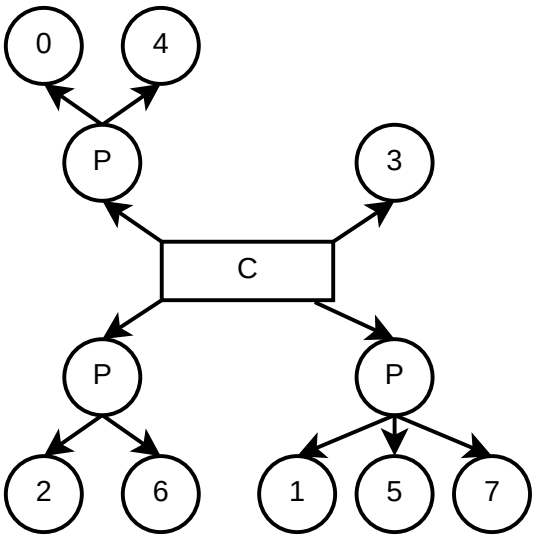
Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir verschmelzen alle benachbarten C-Knoten: Nichts zu tun!

Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

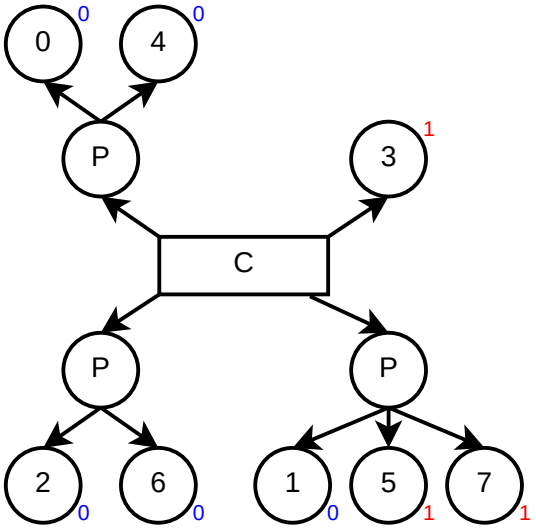
Wir entfernen alle Knoten, die nur zwei Nachbarn besitzen, und verbinden die Nachbarn direkt:



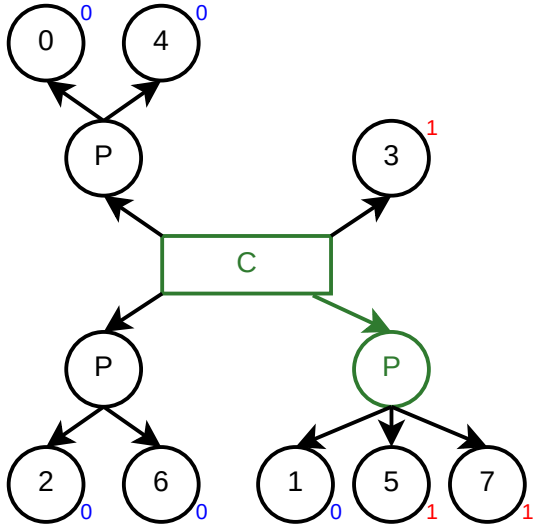
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7
Zeile 5:	0	0	0	1	0	1	0	1

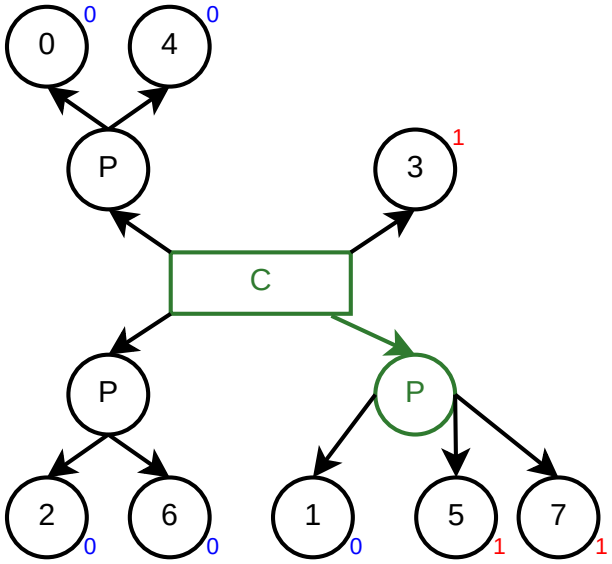
Wir markieren entsprechend der Zeile 5:



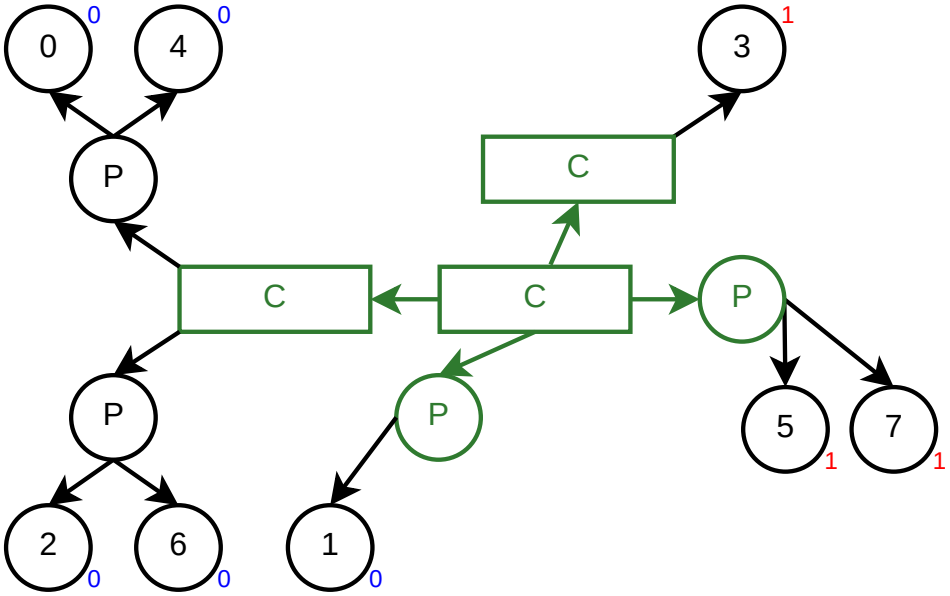
Wir bestimmen den Terminal-Pfad:



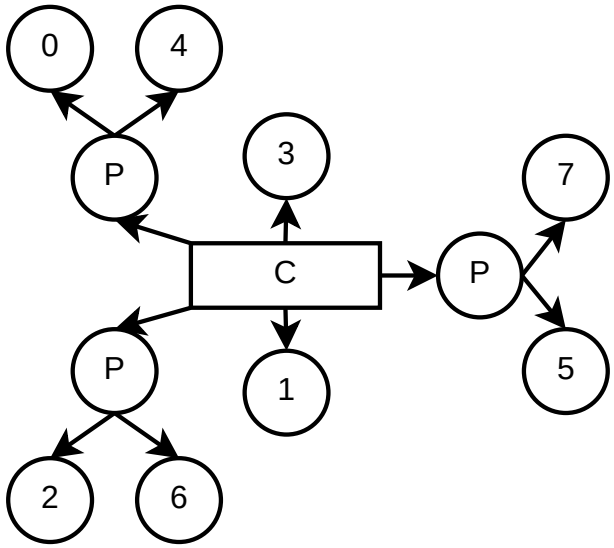
Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



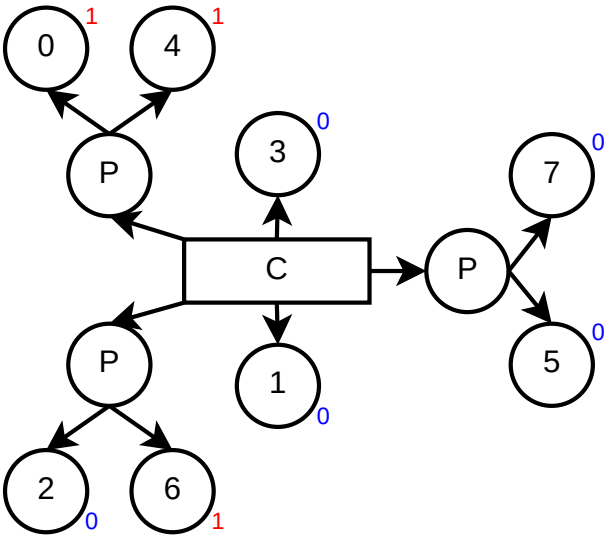
Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!
Wir verschmelzen alle benachbarten C-Knoten. Zudem entfernen wir alle Knoten, die nur zwei Nachbarn besitzen, und verbinden die Nachbarn direkt:



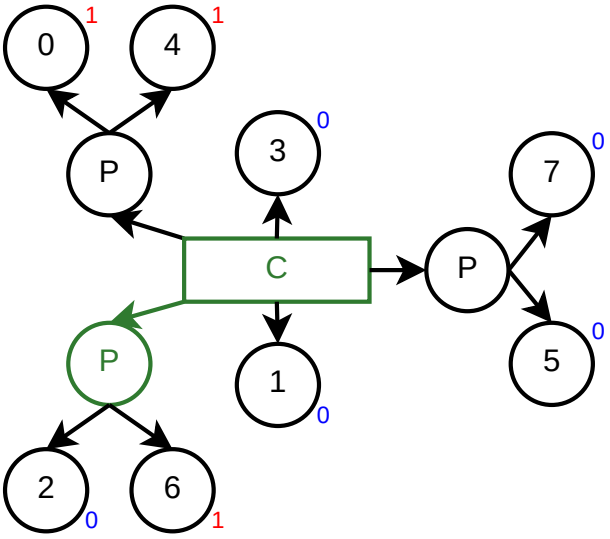
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7
Zeile 6:	1	0	0	0	1	0	1	0

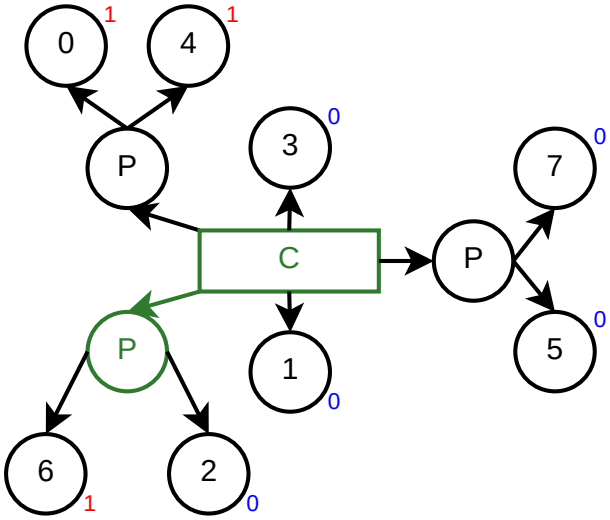
Wir markieren entsprechend der Zeile 6:



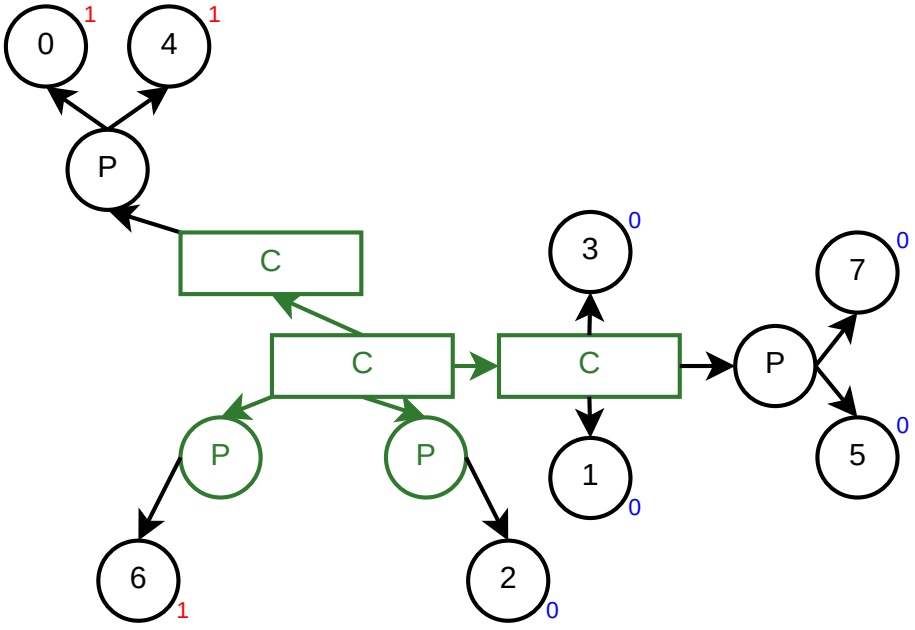
Wir bestimmen den Terminal-Pfad:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:

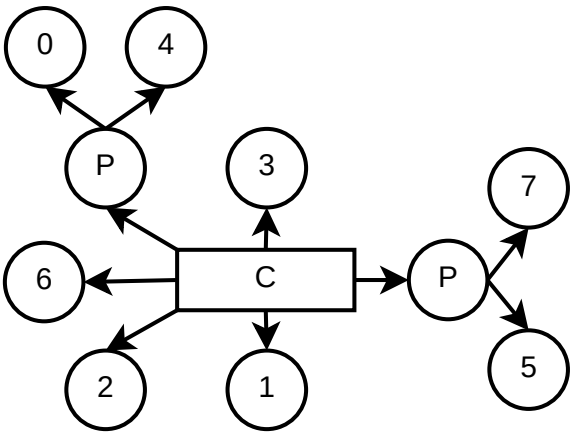


Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!

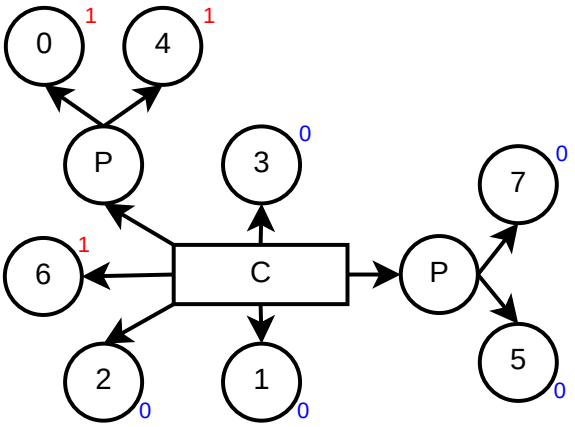
Wir verschmelzen alle benachbarten C-Knoten. Zudem entfernen wir alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt:



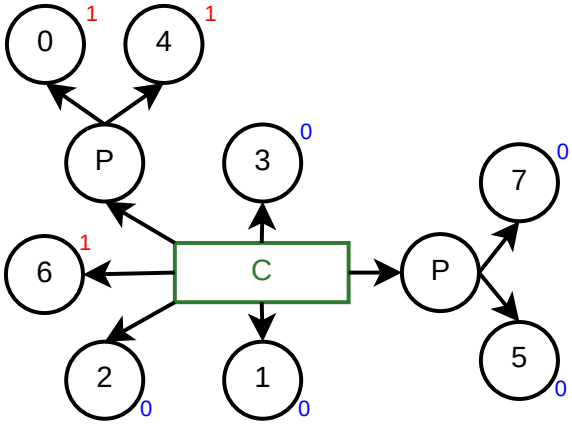
Wir arbeiten die nächste Zeile der Adjazenzmatrix des Graphen ab:

	0	1	2	3	4	5	6	7
Zeile 7:	1	0	0	0	1	0	1	0

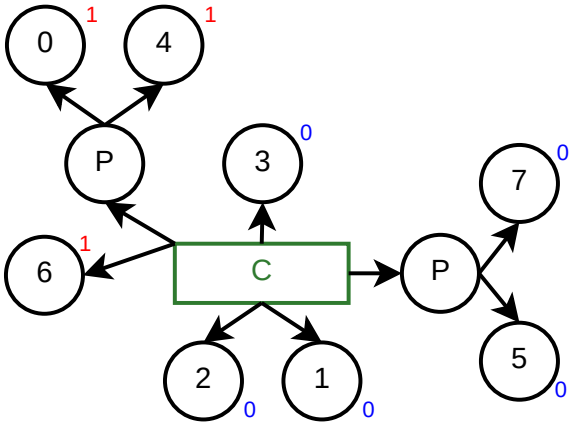
Wir markieren entsprechend der Zeile 7:



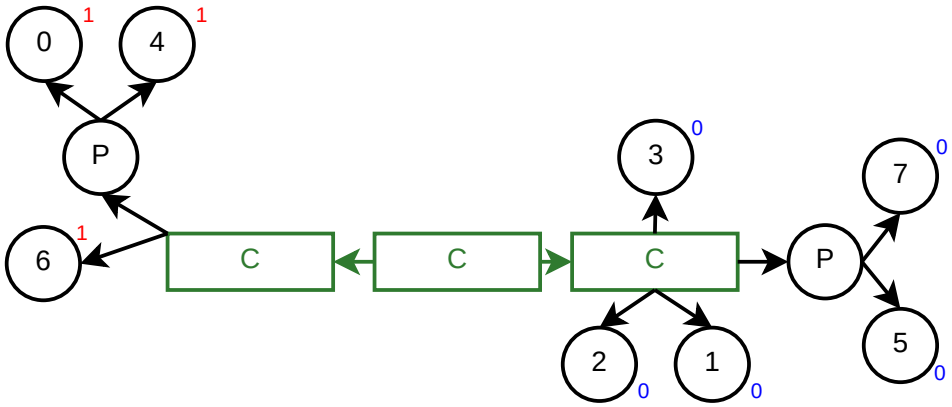
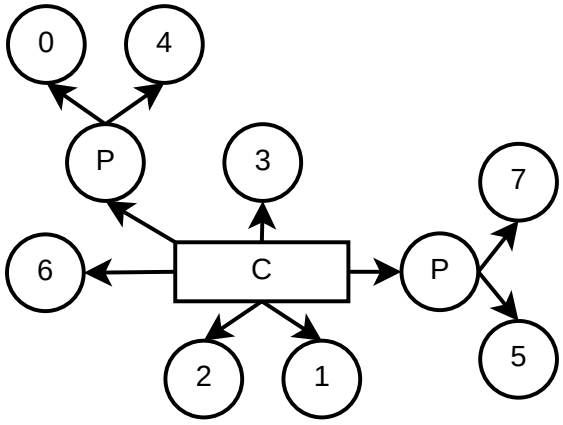
Wir bestimmen den Terminal-Pfad:



Wir modifizieren den Baum so, dass alle vollen Blätter auf der einen Seite des Terminal-Pfads liegen, und alle leeren Blätter auf der anderen:



Jeden Knoten auf dem Terminal-Pfad teilen wir in zwei Knoten gleichen Typs auf. An den einen Knoten hängen wir die Teilbäume mit den leeren Blättern, an den anderen Knoten hängen wir die Teilbäume mit den vollen Blättern. Wir ersetzen den Terminal-Pfad durch einen neuen C-Knoten, der die zyklische Ordnung der neuen Knoten erhält:



Wir entfernen alle P- oder C-Knoten, die nur einen Nachbarn besitzen: Nichts zu tun!
Wir verschmelzen alle benachbarten C-Knoten. Zudem entfernen wir alle Knoten, die nur zwei Nachbarn besitzen und verbinden die Nachbarn direkt: