Fachpraktikum "Erweiterbare Datenbanksysteme" WS 2010/2011 Aufgabe der Phase 2

Thomas Behr, Christian Düntgen,
Ralf Hartmut Güting, Simone Jandt

19. November 2010

Lehrgebiet Datenbanksysteme für neue Anwendungen
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen

1 Persistente Graphen

1.1 Motivation

In der Informatik werden Graphen häufig verwendet, um Beziehungen zwischen Objekten zu beschreiben und zu visualisieren. Die Anwendungsfelder reichen dabei von E/R-Diagrammen über Flussdiagramme bis hin zu Netzwerken. Auch viele Anwendungsprobleme außerhalb der Informatik lassen sich auf Graphprobleme abbilden.

Diese Aufgabe besteht darin, die `PersistentGraphAlgebra` zu implementieren. Diese Algebra soll innerhalb von SECONDO einen Datentyp `pgraph` für große, persistente, gerichtete, knoten- und kantenmarkierte Graphen mit Mehrfachkanten zur Verfügung stellen. Außerdem soll die Algebra einige interessante Algorithmen auf dieser Datenstruktur in Form von Operatoren zur Verfügung stellen.

Zwar gibt es bereits einen Datentypen `graph`, der durch die `GraphAlgebra` zur Verfügung gestellt wird, dieser wurde jedoch als Attributdatentyp realisiert und verfügt nur über begrenzte Anwendungsmöglichkeiten. So können Knoten des Graphen nur mit `point`-Objekten, Kanten nur mit `real`-Objekten markiert werden. Isolierte Knoten und Mehrfachkanten sind gar nicht darstellbar.

1.2 Implementierung des Typkonstruktors `pgraph`

Der erste Teil der Aufgabe besteht darin, den Typkonstruktor `pgraph` zu implementieren. Dabei müssen folgende Anforderungen berücksichtigt werden:

1. Der Graph ist persistent zu speichern.

2. Die Datenstrukturen müssen die Anforderungen der zu implementierenden Operatoren (vgl. Abschnitt 1.3) berücksichtigen. Insbesondere muss der Graph schnell traversiert werden können, d.h. der Zugriff auf mit einem Knoten inzidente Kanten muss effizient realisiert werden.

3. Alle Kanten sind gerichtet. Mehrfachkanten und Schleifen (Start- und Zielknoten sind identisch) sind möglich.

4. Es sind isolierte Knoten (also solche ohne inzidente Kanten) erlaubt. Kanten, die einen Start- oder Endknoten außerhalb der aktuellen Knotenmenge haben, sind nicht erlaubt und ggf. zu löschen.

5. Sowohl die Knoten, als auch die Kanten im Graph sollen mit beliebigen Tupeln markiert werden können. Die Markierungen für Knoten müssen jedoch ein Schlüsselattribut beinhalten, das den Knoten identifiziert. Die Kantentupel müssen zwei Fremdschlüssel für den Primärschlüssel der Knotenmenge beinhalten (Start- und Endknoten der Kante). Als Schlüsselattribute müssen alle Attributtypen akzeptiert werden. Sie müssen die Knoten und Kanten innerhalb geordneter Relationen (`orel`)speichern. Die Knoten werden entsprechend ihres Schlüssels sortiert gespeichert. Die Kanten verwenden als erstes Sortierkriterium ihren Startknoten und als zweites Sortierkriterium den Zielknoten. Das impliziert, dass Knotenschlüssel von jedem beliebigen Typ $T \in \{\texttt{int}, \texttt{real}, \texttt{string}, \texttt{tid}\} \cup INDEXABLE$ sein müssen.

6. Zusätzlich zu der vom Benutzer vorgegebenen Kantenmarkierung soll jeder einzelnen Kante intern ein generierter Schlüssel (vom Typ `tid`) hinzugefügt werden. Dieser Schlüssel wird bei Operatoren, die Kantenbeschreibungen ausgeben, jeweils mit exportiert. Über spezielle Operatoren soll mittels dieses Schlüssels effizient auf die Kanten zugegriffen werden können. Dies erlaubt es, externe Indizes über Kanten zu verwenden. (Die Verwendung externer Indizes für Knoten wird bereits aufgrund der extern vorgegebenen Schlüssel ermöglicht.) Der Kantenschlüssel wird in allen Ausgabetupeln bzw. -records den Namen EID (für "Edge IDentifier") tragen. Dieser Name wird bei der Erzeugung des Graphen mit angegeben. Es ist notwendig, dass über diesen Fremdschlüssel schnell die dazugehörige Kante gefunden werden kann. Dazu ist es erforderlich, innerhalb der Graph-Struktur eine zusätzliche Relation zu verwenden, die diese Fremdschlüssel auf die tatsächlichen Schlüssel der KantenORelation abbildet.

7. Graphen können definiert oder undefiniert sein. Leere Graphen (ohne Knoten und Kanten) sowie Graphen mit Knoten aber ohne Kanten sind erlaubt.

Zur Speicherung der Knotenmenge (V) und der Kantenmenge (E) sowie zur Darstellung von Indexstrukturen (etwa für den Zugriff auf Kanten über die intern generierten Kantenschlüssel) innerhalb Ihrer Datenstruktur können Sie bestehende Datentypen wie geordnete Relationen , ungeordnete Relationen (Typkonstruktor `orel` bzw. `rel`) sowie Indexstrukturen (z.B. `hash`, `btree`, `btree2`) verwenden. Alternativ können Sie auch eigene, direkt auf dem SecondoSMI aufsetzende Dateistrukturen definieren. Innerhalb dieser Strukturen müssen die

Markierungen verwaltet werden. Zur Realisierung der Algorithmen können dabei zusätzliche, automatisch generierte Attribute hinzugefügt werden.

Als *TypeExpression* für Objekte von Typ `pgraph` schlagen wir vor:
`pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID)

Hier und im Folgenden bezeichne V den Tupeltypen der Knotenmarkierung, E den Tupeltypen der Kantenmarkierung, IdentV den Attributnamen des Knotenschlüssels in V, IdentVS bzw. IdentVD den Attributnamen des Knotenfremdschlüssels für den Start- bzw. Zielknoten in E. EID entspricht dem Attributnamen der in Ausgabetupeln für Kantenschlüssel verwendet wird. T $\in$ DATA sei der Typ des Knotenschlüssels, also der Typ des Attributs IdentV in V und der Attribute IdentVS und IdentVD in E.

1.3 Implementierung von Operationen auf `pgraph`

1.3.1 Erzeugen von Graphen

createpgraph: `stream`(`tuple`(V)) × `stream`(`tuple`(E)) × IdentV × IdentVS × IdentVD × EID → `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID)

Dieser Operator erzeugt aus zwei Tupelströmen einen Graphen. Der erste Strom liefert die Knotenmarkierungen, der zweite die Kantenmarkierungen. IdentV ist der dem Primärknotenschlüssel im ersten Strom entsprechende Attributname, IdentVS und IdentVD sind die Attributnamen, die den Fremdschlüsseln für Start- und Endknoten aus dem zweiten Strom entsprechen. EID ist der Attributname, der in Ausgabetupeln als Name für den Kantenschlüssel verwendet wird.

Falls bei der Konstruktion des Graphen Fehler erkannt werden (etwa Kanten ohne gültige Knoten, undefinierte Knotenschlüssel), so soll eine Warnung ausgegeben und ein undefinierter Graph erzeugt werden.

1.3.2 Spezialoperator

Folgender Operator erlaubt es, für sämtliche `pgraph`-Algorithmen, die Anzeige von Warnmeldungen über das MessageCenter (siehe `include/Messages.h`) zu erlauben bzw. zu unterdrücken (Standard: keine Warnungen):

set_pgraph_warnings: `bool` → `bool`

1.3.3 Manipulation von Graphen

Die folgenden Operationen sollen einen persistenten Graphen persistent verändern (als Seiteneffekt).

insertedges: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`(E)) → `stream`(`tuple`(E @ EID:`tid`))

Dieser Operator fügt Kanten in einen Graphen ein. Falls eine Kante als gültig erkannt wird, so wird sie als Nebeneffekt in den Graphen eingefügt, ansonsten wird die Kante nicht eingefügt. Auf jeden Fall wird jedes Kanten-Tupel der Eingabe um ein Attribut EID erweitert und sofort in den Ausgabestrom weitergeleitet. Der Wert von EID ist undefiniert, wenn keine Kante eingefügt wurde, ansonsten ist er gleich dem generierten Kantenschlüssel. Ist der Graph undefiniert, so wird nichts eingefügt und der Graph bleibt undefiniert, die Kanten werden jedoch durchgereicht.

insertvertices: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`(V)) → `stream`(`tuple`(V))

Dieser Operator fügt Knoten in einen Graphen ein. Falls ein Knoten als gültig erkannt wird, so wird er als Nebeneffekt in den Graphen eingefügt, ansonsten wird eine Warnung ausgegeben und der Knoten nicht eingefügt. Auf jeden Fall wird jedes Knoten-Tupel der Eingabe sofort in den Ausgabestrom weitergeleitet. Ist der Graph undefiniert, so wird nichts eingefügt und der Graph bleibt undefiniert, die Knoten werden jedoch durchgereicht.

deletevertices: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`($a_1 : t_1, \ldots, a_k : T, \ldots, a_m : t_m$)) × a_k × *IdentDE* → `stream`(`tuple`(V) @ *IdentDE*: set(record(E @ EID:`tid`))),
wobei $type(V.IdentV) = T$

Dieser Operator löscht Knoten, deren Schlüssel im Attribut a_k des zweiten Arguments vorkommen, aus einem Graphen. Falls ein Knotenschlüssel als gültig erkannt wird, so wird er als Nebeneffekt aus dem Graphen gelöscht — ebenso werden alle mit diesem Knoten inzidenten Kanten aus dem Graphen gelöscht — ansonsten wird eine Warnung ausgegeben und der Knoten nicht gelöscht. Auf jeden Fall wird für jedes Element aus dem Stromargument der Eingabe genau ein Element in den Ausgabestrom weitergeleitet. Dieses besteht aus der Knotenmarkierung des gelöschten Knotens, die um ein Attribut *IdentDE* vom Typ `set`(`record`(E @ EID:`tid`)) erweitert wird (s.u.). Kann ein Knoten nicht gelöscht werden (weil dieser Knoten nicht vorhanden ist), so werden alle Attribute im Ergebnistupel auf 'undefined' gesetzt, nur der Knotenschlüssel wird aus dem Eingabeelement kopiert. Konnte der Knoten gelöscht werden, so enthält *IdentDE* die Menge der gelöschten Kanten. Jedes Kantentupel `tuple`(E@ EID:`tid`) ist dazu als ein `record`(E@ EID:`tid`) darzustellen und in den `set` aufzunehmen. Ist der Graph undefiniert, so wird dies so behandelt, als ob alle Knoten nicht gelöscht werden können, der Graph bleibt undefiniert.[1]

deleteedges: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`(E)) × `bool` → `stream`(`tuple`(E@EID:`tid`))

Dieser Operator löscht als Nebeneffekt Kanten aus einem Graphen. Ist der boolsche Parameter = FALSE, so wird für jedes Element des Argumentstroms maximal eine Kante gelöscht, ist er TRUE, so werden alle entsprechenden Kanten sofort gelöscht. Kann für eine Eingabekante keine Kante im Graphen gelöscht gelöscht werden, so ist der Wert von EID im Ausgabetupel undefiniert. Für jede Kante, die aus dem Graphen gelöscht wird, wird ein Ergebnistupel generiert, das aus der Kantenmarkierung und dem Kantenschlüssel der gelöschten Kante besteht. Eine Kante wird nur gelöscht, wenn ihre Kantenmarkierung mit der aus dem Eingabestrom übereinstimmt. Gibt es mehrere Kandidaten, so wird zunächst die gemäß interner Kantensortierung kleinste Kante entfernt. Ist der Graph undefiniert, so wird nichts gelöscht, der Graph bleibt undefiniert, alle Kanten werden durchgereicht.

deleteedges: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`($a_1 : t_1, \ldots, a_i : T, \ldots, a_j : T, \ldots, a_n : t_{n-2}$)) × a_i × a_j × `bool` → `stream`(`tuple`(E@EID:`tid`))

Diese Variante des Operators löscht entsprechend maximal eine (boolscher Parameter = FALSE) oder alle (boolscher Parameter = TRUE) Kanten, die

[1] Es ist noch nicht sicher, ob *IdentDE* mit ausgegeben werden muss: Die Stabilität der `RecordAlgebra` und der `CollectionAlgebra` ist fragwürdig.

vom durch das Attribut a_i bezeichneten Knoten zum durch das Attribut a_j bezeichneten Knoten führen.

deleteedges: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tid`) → `stream`(`tuple`(E@EID:`tid`))

Diese Variante des Operators löscht die Kanten, deren Kantenschlüssel den `tid`s aus dem Stromargument entsprechen. Die Ausgabe besteht aus der Kantenmarkierung der gelöschten Kante sowie dem Kantenschlüssel. Für jede `tid`, für die keine Kante mit entsprechendem Kantenschlüssel existiert, wird ein Ausgabetupel erzeugt, bei dem alle Attribute der Kantenmarkierung undefiniert sind. EID entspricht in jedem Fall dem Eingabeschlüssel.

updatevertices: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`(V)) → `stream`(`tuple`(V))

Dieser Operator aktualisiert Knoten in einem Graphen. Kann der Knotenschlüssel eines Tupels aus dem Stromparameter im Graphen aufgefunden werden, so wird dessen Markierung mit den Werten des Eingabetupels aktualisiert, ansonsten wird eine Warnung ausgegeben. Jedes Eingabetupel wird vom Operator direkt in den Ausgabestrom weitergeleitet. Ein undefinierter Graph wird dabei nicht aktualisiert und bleibt undefiniert.

updateedges: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`(E @ $E_{IdentSuffix}$ - {IdentVS, IdentVD})) × `bool` × *IdentSuffix* → `stream`(`tuple`(E @ $E_{IdentSuffix}$ - {IdentVS, IdentVD}, EID : `tid`))

Dieser Operator aktualisiert Kanten in einem Graphen. Jedes Tupel der Eingabe beinhaltet eine Beschreibung einer Originalkante E sowie eine Beschreibung der zu aktualisierenden Werte $E_{IdentSuffix}$. $E_{IdentSuffix}$ ist die Attributmenge von E ohne die Start-/Zielknoten, wobei jedem Attributnamen der Suffix *IdentSuffix* angehängt wird. Die gemäß interner Kantensortierung erste Kante, auf die die Beschreibung E genau zutrifft, wird mit den Werten aus $E_{IdentSuffix}$ aktualisiert (Mehrfachkanten!). Ist der `bool`-Parameter TRUE, so wird nicht nur die erste passende Kante aktualisiert, sondern gleich alle passenden Kanten. Für jede aktualisierte Kante wird ein Ausgabetupel erzeugt. Dieses besteht aus der originalen Kantenmarkierung, den aktualisierten Werten sowie dem Schlüssel der aktualisierten Kante. Wird zu einem Eingabetupel keine passende Kante gefunden, so besteht das Ausgabetupel aus dem Eingabetupel und einer undefinierten `tid`. Ein undefinierter Graph wird nicht aktualisiert und bleibt undefiniert.

updateedges: `pgraph`(`tuple`(V), `tuple`(E), IdentV, IdentVS, IdentVD, EID) × `stream`(`tuple`(EID : `tid` @ $E_{IdentSuffix}$ - {IdentVS, IdentVD})) × *IdentSuffix* → `stream`(`tuple`(E @ $E_{IdentSuffix}$ - {IdentVS, IdentVD}, EID : `tid`))

Diese Variante von **updateedges** aktualisiert die durch das Attribut EID spezifizierten Kanten des Graphen. Für eine aktualisierte Kante wird ein Ausgabetupel erzeugt, das aus der originalen Kantenmarkierung, den neuen Werten sowie dem Kantenschlüssel besteht. Wird zu einem Schlüssel keine passende Kante gefunden, werden die Werte der originalen Kantenmarkierung auf undefiniert gesetzt. Die restlichen Werte des Ausgabetupels werden aus der Eingabe übernommen.

1.3.4 An- und Abfragen

Folgende Operatoren erlauben es, Informationen aus dem Graphen zu extrahieren. Die Operatoren dürfen den Ausgangsgraphen dabei nicht verändern:

isdefined: `pgraph`(...) → `bool`

Überlädt den wohlbekannten Operator mit einer weiteren Signatur.

no_vertices, **no_edges** : `pgraph`(...) → `int`

Ermittelt die Anzahl der im Graphen enthaltenen Knoten (Kanten). Mehrfachkanten sind entsprechend mehrfach zu zählen.

indegree, **outdegree**: `pgraph`(...)× T → `int` (T sei der Typ von IdentV)

Ermittelt den Eingangsgrad (Ausgangsgrad) des Knotens, dessen Schlüssel dem zweiten Parameter entspricht. Das Ergebnis ist undefiniert, wenn der Graph undefiniert ist oder kein entsprechender Knoten im Graphen vorhanden ist.

maximum_indegree, **maximum_outdegree**, **minimum_indegree**, **minimum_outdegree**: `pgraph`(...) → `int`

Maximaler/minimaler Eingangsgrad (Ausgangsgrad) aller im Graphen vorhandenen Knoten. Undefiniert, falls der Graph undefiniert ist.

outedges : `pgraph`(...)× T → `stream`(`tuple`(E @ EID : `tid`))

Liefert einen Strom aller von einem Knoten ausgehenden Kanten in Reihenfolge der internen Kantensortierung. Diese sind durch ihre Kantenmarkierung und dazugehörigen Schlüssel bestimmt.

successors: `pgraph`(...)× T → `stream`(`tuple`(V))

Liefert einen Strom aller direkten Nachfolgerknoten eines Knotens in Reihenfolge der internen Kantensortierung,

get_vertices: `pgraph`(...)× T → `stream`(`tuple`(V))

Liefert die Markierung eines spezifizierten Knotens als einelementigen Strom. Ist der Graph undefiniert oder enthält keinen entsprechenden Knoten, so ist der Ergebnisstrom leer.

get_vertices: `pgraph`(...) → `stream`(`tuple`(V))

Liefert die Markierungen zu sämtlichen im Graphen enthaltenen Knoten als Strom. Die Reihenfolge entspricht der internen Knotensortierung. Der Strom ist leer, wenn der Graph keine Knoten enthält oder undefiniert ist.

get_edges: `pgraph`(...)× T × T → `stream`(`tuple`(E@EID : `tid`))

Liefert die Markierungen aller von einem Startknoten (Argument 2) zu einem Zielknoten (Argument 3) gerichteten Kanten in Reihenfolge der internen Kantensortierung als Strom. Ist der Graph undefiniert oder enthält keine entsprechende Kante, so ist der Ergebnisstrom leer.

get_edges: `pgraph`(...) → `stream`(`tuple`(E@EID : `tid`))

Liefert die Markierungen zu sämtlichen im Graphen enthaltenen Kanten als Strom. Die Reihenfolge entspricht der internen Kantensortierung. Der Strom ist leer, wenn der Graph keine Kanten enthält oder undefiniert ist.

get_edges: `pgraph`(...) × `stream`(`tid`) → `stream`(`tuple`(E@EID : `tid`))

Liefert die Kantenmarkierungen zu den Kantenschlüsseln aus dem Stromargument. Ist ein Schlüssel nicht mit einer Kante assoziiert, so werden sämtliche Markierungsattribute auf undefiniert gesetzt, nur EID wird dann aus der Eingabe übernommen. Der Strom ist insbesondere leer, wenn der Graph undefiniert ist.

getFileInfo: `pgraph`(...) → `text`

Liefert Informationen und Statistiken zu sämtlichen vom Graphobjekt verwendeten Dateien. Vergleiche gleichnamige Operatoren für rel, btree, rtree, hash.

1.3.5 Graphalgorithmen

Während die bisherigen Operatoren einfache Operationen auf Graphen gestatten, sollen nun einige interessante Algorithmen auf dieser Datenstruktur implementiert werden. Der Ausgangsgraph darf nicht verändert werden.

p_dijkstra: `pgraph`(...) × T × (`tuple`(E) → `real`) × *IdentSN* → `pgraph`(`tuple`(V), `tuple`(E @ *IdentSN* : T) ...)

Dieser Operator implementiert den Dijkstra-Algorithmus zur Suche kürzester Wege im Graphen. Das Ergebnis ist ein Baum kürzester, vom durch das 2. Argument bezeichneten Knoten ausgehender Wege zu allen Knoten. Die Parameterfunktion (3. Argument) berechnet die Kosten einer Kante aus deren Markierung, so dass hier nahezu beliebige Kostenmaße verwendet werden können. Ist der Ausgangsgraph undefiniert oder wird ein negatives oder undefiniertes Kantengewicht berechnet, so ist der Ergebnisgraph undefiniert und es erfolgt eine Warnung. Die Erweiterung der Kantenmarkierungen um das Attribut *IdentSN* (enthält immer den Wert des 2. Parameters) macht den Ergebnistyp mit dem des **p_floyd**-Operators (s.u.) kompatibel.

p_shortest_path: `pgraph`(...) × T × T × (`tuple`(E) → `real`) [× (`tuple`(V) × `tuple`(V) → `real`)] → `stream`(`tuple`(E@EID : `tid`))

Berechnet einen kürzesten Weg (als Strom traversierter Kanten in Besuchsreihenfolge) innerhalb des Graphen, ausgehend vom durch das 2. Argument bezeichneten Knoten zum durch das 3. Argument bezeichneten Knoten. Die obligatorische Parameterfunktion (Argument 4) berechnet dabei wieder das Gewicht einer Kante. Wird ein negatives oder undefiniertes Gewicht entdeckt, so ist das Ergebnis leer und es erfolgt eine Warnung. Die optionale Parameterfunktion (5. Argument) gibt eine Schätzfunktion für den Abstand zweier Knoten an und erlaubt es, den A*-Algorithmus zu verwenden. (Wird dieser Parameter weggelassen, ergibt sich der Dijkstra-Algorithmus.)

p_floyd: `pgraph`(...) × (`tuple`(E) → `real`) × *IdentSN* → `pgraph`(`tuple`(V), `tuple`(E @ *IdentSN* : T) ...)

Wendet den Floyd-Algorithmus auf einen Graphen an (kürzeste Wege zwischen allen Paaren von Knoten). Die obligatorische Parameterfunktion berechnet das Gewicht einer Kante. Wird ein negatives oder undefiniertes Gewicht entdeckt, so ist das Ergebnis undefiniert und eine Warnung wird ausgegeben. Das Ergebnis des Operators ist ein Graph, der für jeden enthaltenen Knoten einen Baum der kürzesten Wege zu allen Knoten enthält. Der Graph enthält alle Knoten des Eingabegraphen. Für jeden Knoten A im Graphen enthält die Ergebniskantenmenge eine Kopie aller Kanten des Originalgraphen, die auf dem Baum der kürzesten Wege von A zu allen anderen Knoten liegen. Jede dieser Kantenmarkierungen wird jeweils um das Attribut *IdentSN* mit dem Wert A erweitert. Das Ergebnis für einen undefinierten Graphen ist undefiniert.

p_kruskal: `pgraph`(...) × (`tuple`(E) → `real`) × *IdentCost* → `pgraph`(`tuple`(V), `tuple`(E @ *IdentCost*: `real`), ...)

Wendet den gleichnamigen Algorithmus zur Erzeugung des minimalen aufspannenden Waldes auf den Graphen an. Alle Kanten werden dabei als bidirektional (ungerichtet) aufgefasst. Das Ergebnis besteht aus allen Knoten des Graphen, die Kantenmenge enthält die (gerichteten!) Kanten des minimalen aufspannenden Waldes, deren Markierung um ein Attribut *IdentCost* mit dem Kostenwert der ausgewählten Kante erweitert ist.

components: `pgraph`(...) × `bool` × *IdentComponent* → `pgraph`(`tuple`(V

@ *IdentComponent* : `int`), `tuple`(E @ *IdentComponent* : `int`) ...)

Berechnet die (stark) verbundenen Komponenten eines Graphen. Das Ergebnis ist ein Graph, in dem die Knoten- und Kantenmarkierungen um ein Attribut *IdentComponent* erweitert sind, welches eine automatisch vergebene Nummer für die beinhaltende (starke) Zusammenhangskomponente enthält. Ist der `bool`-Parameter TRUE, so werden die starken Zusammenhangskomponenten berechnet, bei FALSE die (einfachen) Zusammenhangskomponenten.

p_bfs, **p_dfs** : `pgraph`(...) × (T × T → `bool`) → `stream`(`tuple`((Vertex: `record`(V), Edge : `record`(E@EID : `tid`)), EdgeClass : `string`))

Diese Operatoren bewirken Graphdurchläufe in Breitensuche (bfs) und Tiefensuche (dfs) mit Kantenklassifikation (Forward, Cross, Backward). Ergebnis ist ein Strom von Tupeln. Jedes Tupel beschreibt den erreichten Knoten, die verwendete Kante und im Attribut "EdgeClass" die Kantenklassifikation (einen der Werte {forward, cross, backward}). Die Tupel werden in Besuchsreihenfolge ausgegeben. Die Parameterfunktion stellt eine Ordnungsrelation (eine <-Relation) auf Knoten dar. Sie wird verwendet, um im Bedarfsfall einen Startknoten unter den noch unbesuchten Knoten zu ermitteln (nämlich das Minimum bezüglich <). Für besuchte isolierte Knoten bzw. Startknoten ist die Kantenbeschreibung "Edge" undefiniert zu setzen. Für leere oder undefinierte Graphen wird ein leerer Strom geliefert.

p_maxFlow : `pgraph`(...) × T × T × (`tuple`(E) → `real`) × *IdentFlow* → `pgraph`(... `tuple`(E @ *IdentFlow*: `real`) ...)

Dieser Operator berechnet einen maximalen Fluss von einer Quelle (2. Argument) zu einer Senke (3. Argument) auf dem Graphen. Die Kapazität einer Kante wird durch die Parameterfunktion berechnet. Der Operator liefert eine Kopie des ersten Arguments, in der alle Kanten zusätzlich mit dem Attribut *IdentFlow* markiert werden. Der Wert dieses Attribut entspricht dem Teilfluss über diese Kante. Wird ein unzulässiges Gewicht entdeckt, so ist eine Warnung auszugeben, das Ergebnis ist undefiniert. Es soll ein möglichst effizienter Algorithmus (etwa Push-Relabel-Algorithmus) verwendet werden.

p_cut_vertices: `pgraph`(...) × `bool` → `stream`(`tuple`(V))

Berechnet die Schnittecken des Graphen, also Knoten, bei deren Weglassen sich die Anzahl der (starken) Zusammenhangskomponenten im Graphen erhöht. Ist der `bool`-Parameter TRUE, so werden die starken Zusammenhangskomponenten berücksichtigt, bei FALSE einfache Zusammenhangskomponenten.

p_bridges: `pgraph`(...) × `bool` → `stream`(`tuple`(E@EID : `tid`))

Berechnet die Brücken des Graphen, also Kanten, bei deren Weglassen sich die Anzahl der (starken) Zusammenhangskomponenten im Graphen erhöht. Ist der `bool`-Parameter TRUE, so werden die starken Zusammenhangskomponenten berücksichtigt, bei FALSE (einfache) Zusammenhangskomponenten.

1.4 Tests

Tests zur Überprüfung der Korrektheit der Implementierung sollen in Form eines SECONDO-Testfiles verfügbar gemacht werden. Daneben sollen auch Performance-Tests durchgeführt werden. Dazu sollen insbesondere große Graphen verwendet werden (hohe Knoten- und Kantenzahlen, unterschiedlich dichte Graphen, viele Updates). Außerdem sollte — soweit möglich — ein Vergleich mit der `GraphAlgebra` vorgenommen werden.

A Richtlinien für die Teamarbeit der Phase 2

1. Die Systementwicklung sollte in mehrere Etappen aufgeteilt werden. Am Ende jeder Etappe (auch schon der ersten) sollte ein lauffähiges Teilsystem vorliegen. In Anbetracht der Gesamtzeit von etwa zwei Monaten sollten die Etappen eine Länge von 2-3 Wochen haben.

2. In der 2. Präsenzphase legen die Gruppen die Etappen und die Aufgabenverteilung fest. Aufgaben sollten möglichst so verteilt werden, dass jeder in jeder Etappe etwas beitragen kann. Die Etappen sind entsprechend zu planen.

3. Jedes Team sollte einen Gruppensprecher bzw. eine Gruppensprecherin auswählen, der jeweils zu den Etappenendterminen der Betreuerin bzw. dem Betreuer über die Erreichung der Ziele berichtet. Das Teilsystem der Etappe n liegt jeweils auf dem CVS-Server vor und kann somit auch vom Betreuer bzw. von der Betreuerin getestet werden.

4. Das Team ist gemeinsam für die Lösung der gestellten Aufgabe verantwortlich. Sollten Mitglieder des Teams nicht angemessen mitarbeiten, sollte zunächst versucht werden, das Problem innerhalb des Teams zu lösen. Falls das nicht gelingt, ist die Betreuerin bzw. der Betreuer zu informieren.

5. Die letzte Etappe endet spätestens 2 Wochen vor der 3. Präsenzphase. Zu diesem Zeitpunkt sollte die Softwareentwicklung abgeschlossen sein. In der 2. Woche vor der Präsenzphase werden die Betreuer und Betreuerinnen das entwickelte System ausprobieren. Ggf. können dabei noch entdeckte Fehler korrigiert werden.

6. Die letzte Woche sollte von den Teilnehmern und Teilnehmerinnen dazu genutzt werden, die Abschlusspräsentation vorzubereiten. In der Abschlusspräsentation sollten einerseits die Konzepte der Implementierung erklärt werden, andererseits natürlich das System vorgeführt werden. Für jede Präsentation steht etwa eine Stunde zur Verfügung. Bitte achten Sie darauf, dass genügend Zeit bleibt, das System tatsächlich vorzuführen.

B Wichtige CVS-Kommandos

Achtung: Lässt man ein bei einem CVS-Kommando aufgeführtes [$Dateiname(n)$] weg, so bezieht sich das Kommando auf sämtliche Dateien des Dateisystemteilbaums, dessen Wurzel das aktuelle Verzeichnis darstellt.

B.1 Einloggen

Tragen Sie in die Datei `/.bashrc` folgende Zeile ein:

```
export CVSROOT=":pserver:<user>@agnesi.fernuni-hagen.de:2401/home/cvsroot2"
```

Starten Sei dann eine neue Konsole und führen Sie das CVS-Kommando `cvs login` aus. Der erste Versuch schlägt fehl. Wiederholen Sie das Kommando – diesmal sollte es funktionieren.

B.2 Online-Hilfe

`cvs --help-commands` = Auflistung aller CVS-Kommandos
`cvs -H <command>` Hilfe zu bestimmten Kommando

B.3 Arbeitskopie erstellen und aktualisieren

cvs co secondo-graph Holt Version vom Server.

cvs update [*Dateinamen*] Aktualisiert Dateien unterhalb des aktuellen Verzeichnisses. Neue Verzeichnisse werden ignoriert.

cvs update -d [*Dateinamen*] Auch neue Verzeichnisse werden geholt.

cvs update -Pd [*Dateinamen*] Neue, nicht-leere Verzeichnisse werden berücksichtigt.

cvs update -PdA [*Dateinamen*] Entfernt zusätzlich Sticky Tags.

cvs update -r<version> [*Dateinamen*] Holt eine bestimmte Version vom Server; Setzt Sticky Tag.

B.4 Änderungen beobachten

cvs -n update [*Dateinamen*] Ohne Angabe von Dateinamen werden alle Dateien des aktuellen Verzeichnisses und seiner Unterverzeichnisse angezeigt. Es bedeuten:

- ? Datei nicht durch cvs verwaltet
- M Datei wurde lokal verändert
- U Es gibt ein Update auf dem Server
- C Konflikte zwischen eigenen Änderungen und Serverupdate

cvs stat [*Dateinamen*] zeigt Zustand der aktuellen Datei an (Versionsnummer, ...)

cvs log [*Dateinamen*] Zeigt Versionshistorie an.

B.5 Eigene Änderungen auf Server bringen

cvs ci -m "Kommentar" [*Dateinamen*] Bringt lokale Änderungen auf den Server.

cvs add *Dateinamen* Fügt eine Datei/Verzeichnis dem CVS hinzu. Für Dateien ist zusätzlich ein `cvs ci Dateiname` notwendig.

cvs delete *Dateinamen* Löscht eine Datei. Auf alte Versionen kann jedoch weiterhin zugegriffen werden.