1 Funktionen des bestehenden Systems

Beschreibung der aus dem GUI ableitbaren Funktionen

1.1 File

1.2 View

View stellt beim klicken auf den Menüreiter „View“ in der Navigationsleiste verschiedene Funktionen zur Optimierung der Ansicht zur Verfügung. Darunter fällt das Suchen nach einer Graphklasse, die Einstellung der Namensanzeige und das Markieren von „unproper“ (müsste eigentlich improper heißen) inclusions.

Search in drawing...
Beim Auswählen des Punktes „search in drawing“ durch einen linken Mausklick, öffnet sich das Fenster „Search for a graphclass“. Aus einer Liste kann in diesem Feld eine Graphklasse, aus allen Graphklassen ausgewählt werden und durch das Drücken von Search gesucht werden. Dabei wird stets nur die Graphklassen angezeigt, die im Drawing auffindbar sind. Durch „cancel“ kann der Suchvorgang abgebrochen werden. Sucht man durch Drücken auf „search“ nach einer aus der Liste gewählten Klasse, dann wird der Canvas so verschoben, dass die Graphklasse sichtbar ist.

Naming preference...
Beim Auswählen des Punktes „Naming preference..“, durch einen linken Mausklick, öffnet sich das Fenster „Naming preference“. Dieses stellt drei Möglichkeiten durch Radio-Buttons zur Auswahl. Dabei kann immer nur ein Radio-Button aktiv sein. (Eigenschaft der Radio-Buttons)

Basic e.g. threshold

Forbidden subgraphs e.g. $(P_4, 2K_2, C_4)$-free

Derived e.g. cograph $\cap$ split

Das Auswählen einer dieser drei Optionen und das bestätigen durch „OK“, hat eine Umbenennung aller Nodes im Drawing zur angegebenen Konvention zur Folge.

Mark unproper(improper) inclusions
Beim Auswählen des Punktes „Mark unproper inclusions“, durch einen linken Mausklick, wird dieser Menüpunkt mittels eines ein- / ausblendbaren Häkchen als aktiv / inaktiv gekennzeichnet. Wird dieser Menüpunkt als aktiv gekennzeichnet, so werden alle „unproper inclusions“ durch verblasste (geringere Opacity, Grauwert anstelle von Schwarz) Pfeile an einer Edge dargestellt. Wobei der Pfeil auf die Graphklasse mit unproper inclusions zeigt. Beim Deaktivieren dieser Funktion verschwindet dieser graue Pfeil.

1.3 Graph classes

1.4 Problems

1.5 Help

1.6 Rechtsklick auf gezeichneten Knoten

1.7 Information Fenster

2 wichtige Klassen des bestehenden Systems

2.1 ISGCI-Package JGraphT

Allgemein:
JGraphT wird benötigt, um die Graph-Struktur zu verwalten und Funktionen darauf auszuführen. Der Schwerpunkt liegt auf Funktionen zur logischen Reduzierung des Graphen und Filterung bestimmter Zusammenhänge (Nachbarn bestimmen etc.). Daneben werden noch nützliche Algorithmen zur Verarbeitung von Graphen bereitgestellt (z.B. Walkers).

Annotation.java
Zuständig für Informationen(Kommentare/Anmerkungen), die an Knoten oder Kanten angehängt werden (Benutzt z.B. von Walkern zur Markierung bereits besuchter Knoten).
Verschiedene Funktionen ermöglichen in einem Annotation-Objekt bestimmten Knoten/Kanten Informationen (Data) zuzuordnen und auszulesen.

AsWeightedDirectedGraph.java
Erweitert AsWeightedGraph und implementiert DirectedGraph
Notwendig zur Darstellung des Graphen als gewichteter, gerichteter Graph.
Mithilfe dieser Klasse können Algorithmen, die für ungewichtete Graphen ausgelegt sind auch auf gewichtete angewendet werden (Mapping von Kanten und Gewichten).

BFSWalker.java
Erweiterung von GraphWalker
Beinhaltet BFS Algorithmus. Durchläuft einen Graphen, ausgehend von einem gegebenen Startknoten (benutzt nur Kanten).
Umsetzung: Alle Knoten kommen in eine Warteschlange, werden nach und nach (via. BFS) besucht und aus der Queue gelöscht.

DFSWalker.java
Erweiterung von GraphWalker
Beinhaltet DFS Algorithmus. Durchläuft alle Knoten eines Graphen, ausgehend von einem gegebenen Startknoten.

Umsetzung: Über ein Annotation-Objekt können besuchte Knoten markiert werden. Durchlauf mittels DFS Algorithmus.

GraphWalker.java
Abstrakte Klasse
GraphWalkers werden benötigt, um einen Graphen zu durchlaufen - vgl. BFSWalker/DFSWalker.

CacheGraph.java
Erweiterung von ListenableDirectedGraph
Ermöglicht die Erstellung eines gerichteten Graphen mit cashing von Kanten und Knoten.
Umsetzung: HashMaps jeweils für Knoten und Kanten.
Überschreibt verschiedene Methoden/stellt zur Verfügung: Suchen von Knoten/Kanten, Feststellung ob Knoten/Kanten vorhanden sind und Konsistenz-Checks.

ClosingDFS.java
Erweiterung von DFSWalker
Definition von Closed Graph: In a directed graph G = (V, A), a set S of vertices is said to be closed if every successor of every vertex in S is also in S. Equivalently, S is closed if it has no outgoing edge.
Beinhaltet DFS Algorithmus zur rekursiven Bestimmung der transitiven Abgeschlossenheit eines Graphen (Menge an Knoten/Kanten, so dass der Graph abgeschlossen ist).

Deducer.java
Stellt einen Algorithmus zur Entfernung aller trivialen Inklusionen in einem Graphen zur
Verfügung.
Beinhaltet verschiedene Funktionen: Statistik - Auskunft über Relationen, Inklusionen und deren Beschaffenheiten; Suche und Entfernung Trivialer Inklusionen (Umformungen, teilweise Umstrukturierung des Graphen)

GAlg.java
Beinhaltet verschiedene Algorithmen und andere Funktionen für Graphen.
Funktionen: Teilen des Graphen in Zusammenhangskomponenten; Selektion aller Nachbarn eines Knoten; Bestimmung eines Pfades zwischen zwei Knoten; Angabe der Topologischen Ordnung eines Graphen(falls möglich); Transitive Reduzierung des Graphen.

Inclusion.java
extends org.jgrapht.graph.DefaultEdge implements Relation
Diese Klasse hält die Informationen einer Kante, nämlich Super- und Subklasse und ob es sich um eine unklare Inklusion handelt oder nicht.

ISGCIVertexFactory.java
implements VertexFactory<Set<GraphClass> >
Klasse um virtuelle Knoten zu erstellen

RevBFSWalker.java
extends BFSWalker<V,E>
Durchläuft einen Graphen mittels gerichteter Breitensuche.

TreeBFSWalker.java
extends UBFSWalker<V,E>
Durchläuft den Spannbaum eines Graphen mittels ungerichteter Beitensuche. Anmerkung: Markiert nicht selbständig den Spannbaum.

TreeDFSWalker.java
extends UDFSWalker<V,E>
Durchläuft die Knoten des Spannbaums eines Graphen mittels ungerichteter Teifensuche

UBFSWalker.java
extends BFSWalker<V,E>
Durchläuft den Graph mit einer ungerichteten Breitensuche.

UDFSWalker.java
extends DFSWalker<V,E>
Durchläuft den Graph mit einer ungerichteten Tiefensuche

WalkerInfo.java

Diese Klasse wird benutzt um Knoten Informationen zu geben, während der Graph mittels Tiefen- oder Breitensuche durchlaufen wird.

2.2 ISGCI-Package Layout

Alle Klassen, die in dem `./layout`-Ordner stehen sind für das Layout des letztendlich zu zeichnenden Graphen zuständig, dabei stehen diese stark mit den Klassen aus dem `./grapht`-Ordner in Zusammenhang. Die ganzen Klassen und deren Methoden werden über die Klassen des `./gui`-Ordners aufgerufen.

Im Folgenden werden alle Klassen aus dem `./layout` grob beschrieben und deren Funktionweise und Zuständigkeit näher erläutert. Konkrete Details, wie Methodendeklarationen, Methodenaufrufe und Variablen werden spärlich erläutert. Dies sollte aber der oberflächenden Verständlichkeit nicht im Wege stehen, wenn nicht sogar dienen.

GraphDrawInfo.java

Erweiterung von WalkerInfo.java

Diese Klasse erweitert die Klasse `WalkerInfo.java` aus dem Package `./grapht`, welche dafür zuständig ist, ob Kanten zu einem zu berechnenden/anzuzeigenden Graphen gehört oder nicht. Bei Knoten wird auf Adjazenz geprüft (näheres in der Beschreibung bei `grapht`), ob der Knoten zum dem zu zeichnenden Graphen gehört. Sie fügt weitere Variablen, die Werte für Knoten und Kanten, die zur Graphzeichnung benötigt werden, beinhalten. Dazugehören auch virtuelle Knoten, die keinen Inhalt haben und nur zur Kantenzeichnung benötigt werden. Außerdem werden Variablen zur Ranking-Berechnung deklariert, um die spätere richtige Position in der Zeichnung zu bestimmen. Alle benötigten Variablen werden im Konstruktor mit Default-Werten instanziiert und überschrieben, sobald diese benötigt werden. (i.e. Breite des Knotens, Koordinaten, Position im Ranking usw.)

HierachyLayout.java

Diese Klasse ist zur Berechnung der Hierarchie im Layout zuständig. Jedoch muss berücksichtigt werden, dass eine konsistente Hierarchie nur erreicht werden kann, wenn der Graph, zu dem die Hierarchie berechnet werden soll, ein gerichteter nicht-zyklischer Graph ist. Es muss beachtet werden, dass dieser transitiv reduziert ist. Erneut spielt die Berechnung des Rankings und der virtuellen Knoten eine entscheidende Rolle, wobei diese Berechnungen mit JGraphT stattfinden. Dazu gehört die Beschreibung des zum Knoten gehörenden Vertex (Weite, Rank..). Um das Layout der Hierarchie zu berechnen, wird die eben beschrieben Klasse `GraphDrawInfo.java` verwendet. Bei der Berechnung werden dabei die 4 Prinzipien berücksichtigt, die in der Dokumentation zur Zeichnung (`./doc/1993drawing.pdf`) näher erläutert und bei den 3 Durchläufen des "Network Simplex Algorithm"beachtet werden. Der konkrete Ablauf, welche Schritte getan werden müssen, werden dort näher erläutert. (Machbarkeitsbaumberechnung mit entsprechendem Ranking der den length-constraints genügen muss). Es werden verschiedene Algorithmen ausgeführt, welche mit einigen Hilfsmethoden u.a. Spannbäume mit festen Kanten, machbare Rankings berechnet [...]. Hierbei muss flüchtig erwähnt werden, dass die Berechnung in den Grundzügen auf Ersetzung von Graphkanten und -knoten durch Nichtgraphkanten und -knoten, um die Machbarkeit mit einem optimalem Ranking zu erreichen, basiert. Es wird versucht ein minimalen Graphen zu finden (optimales Ranking) der den 4 Prinzipien nachkommt. Dabei werden die Variablen der Klasse `GraphDrawInfo.java` benötigt.

LLWalker.java

Erweiterung von TreeDFSWalker.java

Traversiert die Knoten eines Baumes mithilfe von Tiefensuche, indem die Methoden der Oberklasse verwendet werden.

Ranks.java

In dieser Klasse werden die Knoten eines Graphen über Rankings verwaltet. Einer der Hauptaspekte für die Verwendung von Rankings ist, dass die Kantenkreuzungen soweit wie möglich minimiert werden. Ein initiales Ranking wird durch Zuweisung eines Rankings für je-

den über Breitensuche (durch Traversierung), wobei bei dem niedrigsten Ranking begonnen wird (in der Anwendung von oben nach unten), erstellt. Das ist notwendig, um sicherzugehen, dass dieses Ranking ein Baum ohne Kantenkreuzungen ist. Um dies zu erreichen wird `HierarchyLayout.java` verwendet, indem über sie virtuelle Knoten erstellt und gesetzt werden, um ein optimales Ranking für die Kantenkreuzungen zu erreichen. Um die Knoten innerhalb eines Rankings effektiv zu sortieren werden Heuristiken (siehe `./doc/1993drawing.pdf`) verwendet (z.B. Median, Barycenter, Minimum, Sifting..). Es werden iterative heuristische Berechnungen entsprechend der bereits genannten Dokumentation ausgeführt, um ein bestmögliches in einem akzeptablen Rahmen des Berechnung-Aufwandes Ergebnis zu erzielen (Vertauschung von Knoten, falls dadurch Kreuzungen vermieden werden).

TightTreeWalker.java
Erweiterung von UBFSWalker.java;
wird in `HierarchyLayout.java` *für den "Network Simplex Algorithm" verwendet*
Diese Klasse wird benötigt um einen gerichteten Graphen mithilfe von Breitensuche *ungerichtet* zu traversieren, dabei werden nur Kanten, die fest("tight"$\hat{=}$ slack $= 0$) sind, berücksichtigt. Es wird ein Startknoten definiert, mit dem diese Traversierung ausgeführt wird. Die zu traversierenden Kanten sind als Baumkanten markiert (Information aus `GraphDrawInfo.java`). Als Ergebnis erhält man einen Spannbaum für den Eingabe-Graphen.

TreeReranker.java
Erweiterung von TreeBFSWalker
Mithilfe dieser Klasse werden neue Rankings auf einem Layout-Graphen (als Baum übergeben) erstellt, um ein optimales Ranking zu erhalten (durch Breitensuche). Sie wird eigentlich nur verwendet, um eine semantische Trennung zum `TreeBFSWalker.java` zu erhalten, denn jede verwendete Methode ruft eine entsprechende Methode aus der Oberklasse auf.

2.3 ISGCI-Package db

Hier liegen viele Algorithmen z.b. beinhaltet die Klasse Algo.java Algorithmen um subklassen /superklassen einer Node zu finden und diese auszugeben. Diese werden benötigt um die Inklusionsinformationen zu speichern und abzurufen.

2.4 ISGCI-Package gc

GC steht für graphclass. In Dieses Package sind alle graphclass klassen enthalten. Jede enthaltene Java Klasse beschreibt eine GraphClass und deren besondere Eigenschaften.

2.5 ISGCI-Package iq

Hier finden ISGCI Queries statt. db und gc nutzen dieses package um Queries auszuführen.

2.6 ISGCI-Package problem

Store complexity information for graph problems, in diesem Package sind alle Funktionen enthalten die dazu genutzt werden ein Problem zu beschreiben.

2.7 ISGCI-Package util

In dem util Package liegen alle für spezielle Funktionalitäten benötigte Klassen. Dazu gehört eine Reihe von Klassen, die die Verwendung von "Latexïnnerhalb des ISGCI ermöglicht (d.h. das Verwenden einiger Zeichen).

2.8 ISGCI-Package xml

Das Package xml ist zum input/output von XML dateien, hier werden Informationen aus XML dateien gelesen und ausgegeben. Das Package xml greift dabei auf das Package sax zu und nutzt deren Filterfunktionalität. Die Daten aus einem XML input/output werden den Klassen grapht, ISG und Util zur Verfügung gestellt.

2.9 ISGCI-Package isq

in isq werden die Graph Familien, zusammenhänge aus einer XML gelesen und zum Zeichnen vorbereitet, also an JgraphT weitergereicht. Dabei nutzt isq funktionen des util packages.

2.10 ISGCI-Package sax

Das Package sax, beinhaltet nur die Klasse XMLWriter.java. Diese ist ein Filter um ein XML Dokument aus einem SAX (Simple API for XML) event stream zu erzeugen. Diese Klasse wird vom XML parser benötigt.