Random Graph Coloring Evaluation

Entwurfsdokument

Jonas Kasper, Bernard Hohmann, Thomas Fischer, Christian Jung,
Jonas Linßen

Inhaltsverzeichnis

1 Anmerkungen zum Pflichtenheft

1.1 Änderungen

1.1.1 GUI

Graphen-Vorschau In der Graphen-Preview-Ansicht in der GUI werden die einzelnen Graphen, seien sie generiert oder importiert, unter einem neuen Tab angezeigt. Dabei werden die Graphen eines jeweiligen Tabs in einer Grid-View dargestellt. Jeder der einzelnen Graphen sollte dabei auch gezeichnet werden.
Für den Benutzer hatte dies eine unübersichtliche Anzeige und Navigation und auch eine problematische Skalierung der Fenstergröße zur Folge. Für den Kunden bestand auch kein Interesse der sofortigen Betrachtung der Graphen.
Daher werden die Graphen nicht mehr sofort gezeichnet, sondern als Dropdown-Box, mit der der Benutzer sich bestimmte, von ihm auswählbare Graphen anzeigen lassen kann. Durch den verminderten Zeichen-Aufwand ist auch ein Performance-Gewinn zu erwarten.

Graphen-Generierung Um die Menge der Graphen, auf welchen die Heuristiken angewendet werden sollen, festzulegen, gibt es folgende Möglichkeiten:

1. Automatische zufällige Generierung mit zuvor getätigten Einstellungen.
2. Import bereits generierter Graphen.
3. Im Graph-Editor von Grund auf neue Graphen von Hand erstellen.

Bisher waren unter dem Tab „Generieren“ neben den Knöpfen zum Starten der zufälligen Generierung, des Imports und des Graphen-Editors auch Einstellmöglichkeiten für die zufällige Generierung. Diese machen aber nur im Zusammenhang mit der zufälligen Generierung Sinn, für den Import und das Editieren haben diese keine Bedeutung.
Daher werden diese Einstellmöglichkeiten erst angezeigt, wenn der Benutzer auch die zufällige Generierung von Graphen auswählt.

Graphen-Editor Beim Graphen-Editor kann man standardmäßig sowohl einen „Simple-Undirected-Graph“, als auch „Simple-Hyper-Graph“ editieren oder auch erstellen. Dabei wurden dem Benutzer verschiedene Funktionalitäten geboten.
Diese wurden bisher nicht auf spezielle Graphentypen eingeschränkt. Allerdings entsteht bei einigen der angebotenen Funktionen die Gefahr, dass der User den Graphentyp durch die gemachten Änderungen verändert, oder gar den gesamten Graphen ungültig für die weitere Bearbeitung macht.
Die von uns getroffene Anpassung war es die Funktionen auf den Graphen-Typ einzuschränken und den Graphen-Editor den Typ des editierten Graphen überprüfen zu lassen.

Speichern Es wird nicht möglich sein die Graphen und Heuristiken seperat zu speichern, stattdessen werden alle Informationen eines einzelnen Tabs gespeichert. Ein Tab enthält eine Menge von Graphen, eine Liste von Heuristiken und deren Einstellungen, Statistiken und Ergebnisse.

2 Übersicht

2.1 Architektur

Das System basiert auf dem Model-View-Controller(MVC)-Muster mit einer Drei-Schichten-Architektur und einer durch JavaFX realisierten graphischen Benutzerschnittstelle.
Beim MVC-Muster wird das System in die drei Komponenten Modell, Präsentation und Steuerung aufgeteilt. Das Modell (Model) enthält und verarbeitet die Daten, welche dann von der Präsentation (View) dargestellt werden. Die Steuerung (Controller) steuert den Ablauf und das Verhalten der Anwendung. Dafür werden Benutzereingaben auf Modeländerungen und Ausführung von Berechnungen abgebildet. Weiterhin informiert das Model direkt oder über den Controller die View über Änderungen am Model (z.B. Ergebnisse von Berechnungen) und sorgt damit für eine Anpassung bzw. Aktualisierung der View.
In der hier verwendeten Architektur erfolgt die Kommunikation zwischen View und Model immer über den Controller. Dabei gibt es `FXController`-Komponenten, welche die Steuerung der View übernehmen, und `LogicController`-Komponenten, welche als Schnittstelle zwischen den `FXControllern` und dem Model dienen.
Die View wird in FXML entwickelt, einer auf XML basierenden Sprache zur Definition der Benutzerschnittstelle.
Das Model besteht aus den für die Graphenspeicherung und die Anwendungen der Heuristiken benötigten Daten und den Heuristiken selbst.
Da die Kommunikation zwischen Model und View immer über den `LogicController` läuft, besteht keine direkte Abhängigkeit zwischen den beiden Teilen und es wird eine Modularität des Systems erreicht. Dadurch kann das System in drei logische Schichten unterteilt werden, eine Daten-, eine Logik- und eine Benutzerschicht.
Zwischen diesen Schichten besteht eine lose Kopplung und damit ist eine bessere Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit gegeben. Auch eine physische Trennung der einzelnen Schichten könnte mit der Einführung geeigneter Kommunikationsprotokolle erreicht werden. In Abbildung 1 ist eine schematische Darstellung der Schichtenarchitektur zusammen mit dem MVC-Muster zu sehen.

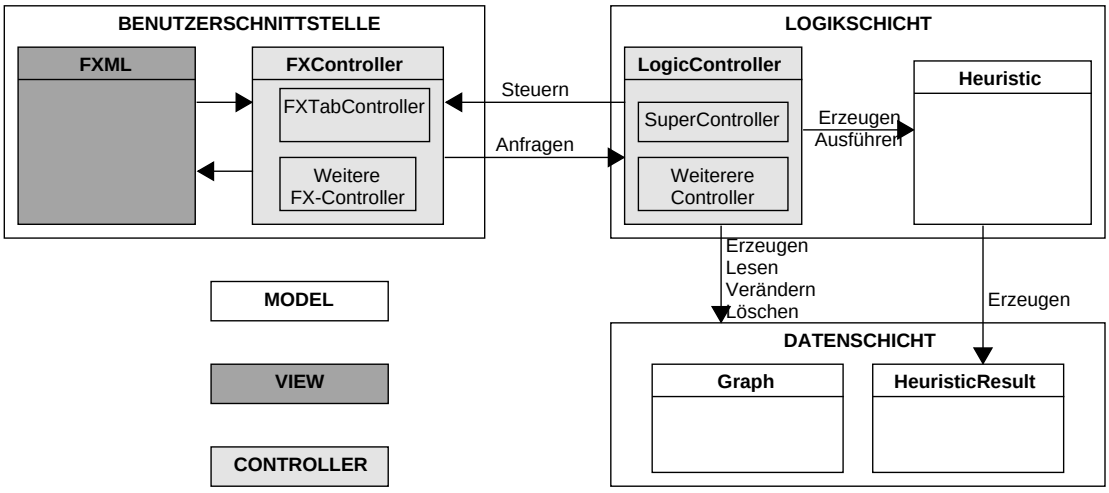


Abbildung 1: Die Systemarchitektur als Schichten-Model mit MVC-Muster. Pfeile stellen Assoziationen zwischen Komponenten dar.

3 Model

Package graph

This package contains the interfaces for the interaction with graphs. In the subpackages concrete graph-types are implemented.

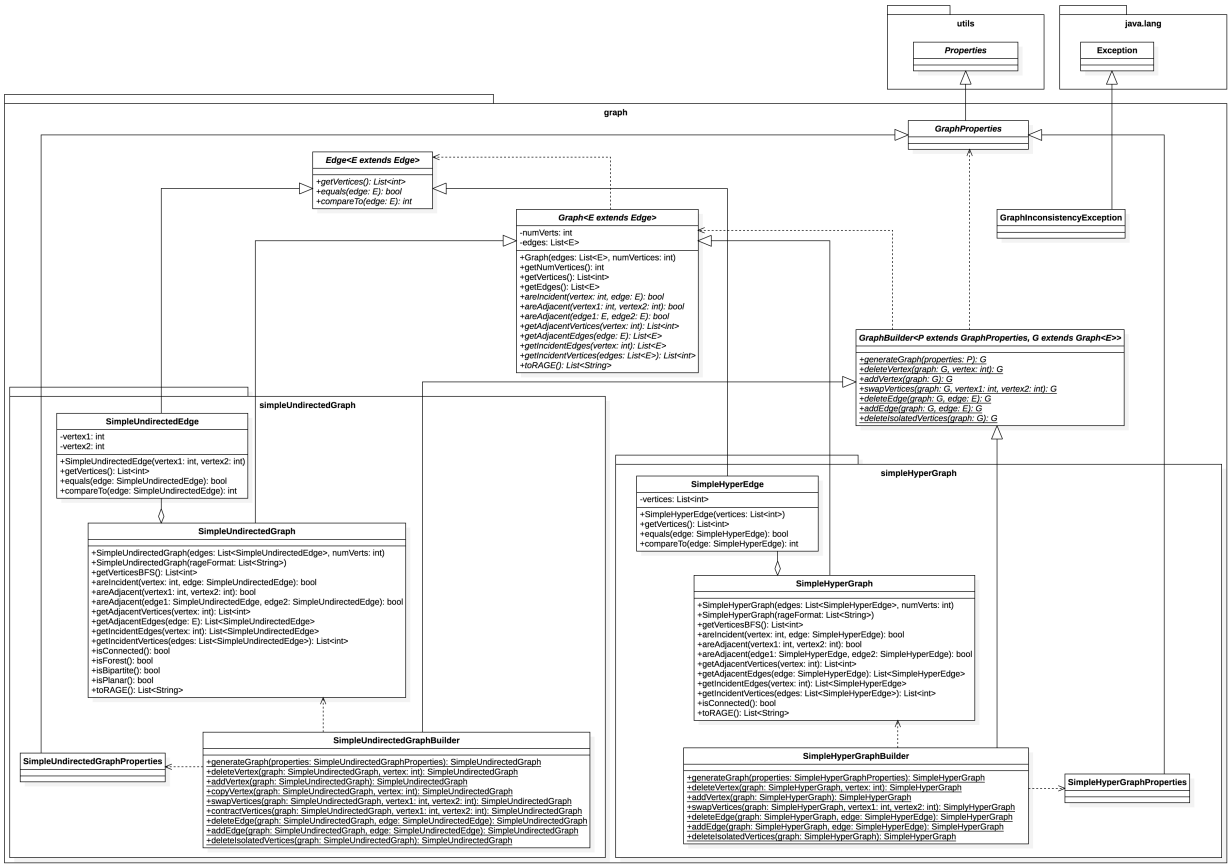


Abbildung 2: Das Paket graph

Class Graph

Description

This class describes the abstract structure of a graph. Each graph has (independent of its concrete type) a finite amount of vertices and edges, which define a relation of vertices. The type **E** of this edges defines the concrete graph type. The class has methods for retrieving the relations given by the edges. Vertices are identified with their unique index and thus are not saved explicitly.

Documentation

+ **Graph(edges: List<E>, numVertices: int)**
  the constructor of this class
  @param edges the edges belonging to this graph **@numVertices** the number of vertices this graph has

+ **getNumVertices(): int**
@return returns the number of vertices which the graph contains

+ **getVertices(): int**
convenience method for retrieving the list of vertex indices
@return returns the list [0 ... numVertices−1]

+ **getEdges(): List<E>**
@return returns the edges giving the graph its structure

+ ***areIncident(vertex: int, edge: E): bool***
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@param edge an edge of the graph
@return returns **true** iff the vertex is incident to the given edge
@throws GraphInconstistencyException if **vertex** is an invalid vertex index or **edge** is not an edge of the graph

+ ***areAdjacent(vertex1: int, vertex2: int): bool***
@param vertex1 the index of a vertex of the graph ie. in [0 ... numVertices−1]
@param vertex2 see **vertex1**
@return returns **true** iff there is an edge which is incident to both vertices
@throws GraphInconsistencyException if **vertex1** or **vertex2** is not a valid vertex index

+ areAdjacent(edge1: E, edge2: E): bool
@param edge1 an edge of the graph
@param edge2 another edge of the graph
@return returns **true** iff there is a vertex which is incident to both edges
@throws GraphInconsistencyException if **edge1** or **edge2** is not an edge of the graph

+ ***getAdjacentVertices(vertex: int): List<int>***
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all vertices which are adjacent to **vertex**
@throws GraphInconsistencyException if **vertex** is not a valid vertex index

+ ***getAdjacentEdges(edge: E): bool***
@param edge an edge of the graph
@return returns the list of all edges which are adjacent to **edge**
@throws GraphInconsistencyException if **edge** is not an edge of the graph

+ ***getIncidentEdges(vertex: int): List<E>***
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all edges incident to **vertex**
@throws GraphInconsistencyException if **vertex** is an invalid vertex index

+ ***getIncidentVertices(edges: List<E>): List<int>***
@param edges a list of edges of the graph
@return returns the list of all vertices which are incident to any of the edges in the list
@throws GraphInconsistencyException if there is an edge in **edges**, which is not an edge of the graph

+ ***toRAGE(): List<String>***
@return returns the line-by-line representation of the graph as specified in the RAGE-data format

Class Edge

Description

An edge always defines an adjacency-relation of the vertices incident to it. Moreover this class provides methods to compare edges.

Documentation

+ ***getVertices(): List<int>***
  @return returns the list of all indices of vertices incident to this edge
+ ***equals(edge: E): bool***
  @return returns **true** iff **edge** equals the edge this method is invoked upon. Note that the notion of equality depends on the concrete implementation.
+ ***compareTo(edge: E): int***
  @return returns **-1**/**0**/**1** if **edge** is greater/equal/smaller than the edge this method is invoked upon. Note that the notions of order and equality depend on the concrete implementation.

Class GraphProperties

Description

This class is required for exchanging data between controller and model, especially to signal the settings required to generate graphs. It assures that the following graph-properties can be retrieved and set at all times:

- "graphTypes" – a const list of strings, initialised with ["simpleUndirectedGraph", "simpleHyperGraph"]
- "type" – a string
- "numVertices" – a nonnegative integer

Class GraphBuilder

Description This class is a factory class to generate graphs of type **G** by given GraphProperties **G** as well as to modify graphs of this type.

Documentation

+ ***generateGraph(properties: P): G***
  @param properties the properties which the generated graphs will have
  @return returns a randomly generated graph satisfying the specified **properties**
+ ***deleteVertex(graph: G, vertex: int): G***
  @param graph the graph which is going to be modified
  @param vertex the index of a vertex of **graph**, which will be deleted
  @return returns a modified copy of **graph** in which the vertex with index **vertex** and all edges incident to it are deleted
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex**
+ ***addVertex(graph: G): G***
  @param graph the graph which is going to be modified
  @return returns a modified copy of **graph** which has precisely one isolated vertex more
+ ***swapVertices(graph: G, vertex1: int, vertex2: int): G***
  @param graph the graph which is going to be modified
  @param vertex1 the index of a vertex of **graph**
  @param vertex2 the index of another vertex of **graph**
  @return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** swap indices. Note this results in a different but isomorphic graph to **graph**
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**
+ ***deleteEdge(graph: G, edge: E): G***
  @param graph the graph which is going to be modified
  @param edge the edge which is going to be deleted
  @return returns a modified copy of **graph** in which **edge** is deleted, if it was an edge in **graph**. Otherwise it just returns **graph**

+ ***addEdge(graph: G, edge: E): G***
@param graph the graph which is going to be modified
@param edge the edge which is going to be inserted
@return returns a modified copy of **graph** in which **edge** is inserted if it wasnt already an edge in **graph** otherwise it returns just **graph**. Note that the edge may contain vertices which are not in **graph**, since missing vertices will automatically be added

+ ***deleteIsolatedVertices(graph: G): G***
@param graph the graph which is going to be modified
@return returns a modified copy of **graph** in which all isolated vertices are deleted

Class GraphInconsistencyException

Description

This class extends the usual Java Exception to an exception specifically thrown when graphs are treated wrong.

Package graph.simpleUndirectedGraph

In this package **simple undirected graphs** (ie. graphs where edges always connect two distinct vertices x and y, where there is no distinction between edges xy and yx and where there is at most one edge xy) are implemented. It offers methods to generate, modify and distinct them by some (for simple undirected graphs well defined) criterions.

Class SimpleUndirectedGraph

Description

This class concretizes the abstract Graph class in the sense of simple undirected graphs. As mentioned such a graph does not contain any loops or multiedges. Besides incidence relations, this class offers methods to identify properties of simple undirected graphs.

Documentation

+ **SimpleUndirectedGraph(edges: List<SimpleUndirectedEdge>, numVertices: int)**
  a constructor for this class
  @param edges the edges contained in this graph
  @param numVertices the amount of vertices this graph being strictly greater than zero
  @throws GraphInconsistencyException if numVertices ≤ 0 or if there is an edge with a vertex $\geq$ **numVertices** or of there exists an edge more than once

+ **SimpleUndirectedGraph(rageFormat: List<String>)**
  another constructor for this class
  @param rageFormat the lines of the line by line representation as specified in the RAGE data-format.
  @throws GraphInconsistencyException if rageFormat is not a valid representation of SimpleUndirectedGraph

+ **getVerticesBFS(): List<int>**
  @return returns the list of vertices of the graph in the order of a breadth first search

+ **areIncident(vertex: int, edge: SimpleUndirectedEdge): bool**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param edge an edge of the graph
  @return returns **true** iff the vertex is incident to the given edge
  @throws GraphInconstistencyException if **vertex** is an invalid vertex index or **edge** is not an edge of the graph

+ **areAdjacent(vertex1: int, vertex2: int): bool**
  @param vertex1 the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param vertex2 see **vertex1**
  @return returns **true** iff there is an edge which is incident to both vertices
  @throws GraphInconsistencyException if **vertex1** or **vertex2** is not a valid vertex index

+ **areAdjacent(edge1: SimpleUndirectedEdge, edge2: SimpleUndirectedEdge): bool**
  @param edge1 an edge of the graph
  @param edge2 another edge of the graph
  @return returns **true** iff there is a vertex which is incident to both edges
  @throws GraphInconsistencyException if **edge1** or **edge2** is not an edge of the graph

+ **getAdjacentVertices(vertex: int): List<int>**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @return returns the list of all vertices which are adjacent to **vertex**
  @throws GraphInconsistencyException if **vertex** is not a valid vertex index

+ **getAdjacentEdges(edge: SimpleUndirectedEdge): bool**
@param edge an edge of the graph
@return returns the list of all edges which are adjacent to **edge**
@throws GraphInconsistencyException if **edge** is not an edge of the graph

+ **getIncidentEdges(vertex: int): List<SimpleUndirectedEdge>**
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all edges incident to **vertex**
@throws GraphInconsistencyException if **vertex** is an invalid vertex index

+ **getIncidentVertices(edges: List<SimpleUndirectedEdge>): List<int>**
@param edges a list of edges of the graph
@return returns the list of all vertices which are incident to any of the edges in the list
@throws GraphInconsistencyException if there is an edge in **edges**, which is not an edge of the graph

+ **isConnected(): bool**
@return returns **true** iff the graph is connected ie. iff for any two vertices there is a sequence of edges where any two consecutive edges are adjacent

+ **isForest(): bool**
@return returns **true** iff the graph is a forest ie. acyclic

+ **isBipartite(): bool**
@return returns **true** iff the vertex set can be partitioned into two parts such that no two vertices from the same partition are adjacent

+ **isPlanar(): bool**
@return returns **true** iff the graph has an embedding into the plane such that no two edges intersect

+ **toRage(): List<String>**
@return returns the line-by-line representation of the graph as specified in the RAGE-data format

Class SimpleUndirectedEdge

Description

This class concretizes the class Edge in the sense of a simple undirected edge. It always relates two distinct vertices.

Documentation

+ **SimpleUndirectedEdge(vertex1: int, vertex2: int)**
a constructor for this class
@param vertex1 the index of the index of the first vertex this edge is incident to
@param vertex2 the index of the index of the second this edge is incident to
@throws GraphInconsistencyException if **vertex1** equals **vertex2**

+ **getVertices(): List<int>**
@return returns the list of all indices of vertices incident to this edge

+ **equals(edge: E): bool**
@return returns **true** iff both edges are adjacent to the same two vertices

+ **compareTo(edge: E): int**
The notion of order between edges (x, y) and (u, v) with $x \leq y$ and $u \leq v$ is defined by $(x, y) < (u, v)$ iff $x < u$ or $(x = u$ and $y < v)$
@return returns **-1**/**0**/**1** if **edge** is greater/equal/smaller than the edge this method is invoked upon

Class SimpleUndirectedGraphProperties

Description

This class is an extension of the GraphProperties class and serves as collection of data for exchange between controller and model, especially to signal the settings required for generating simple undirected graphs. It assures that the following properties can be retrieved and set at all times:

- "minDegree" – a nonnegative integer
- "maxDegree" – a nonnegative integer
- "connected" – a boolean
- "forest" – a boolean
- "bipartite" – a boolean
- "planar" – a boolean

Class SimpleUndirectedGraphBuilder

Description

This class concretizes the GraphBuilder class by offering methods for randomly generating simple undirected graphs after given SimpleUndirectedGraphProperties as well as modifying them.

Documentation

+ **generate(properties: SimpleUndirectedGraphProperties): SimpleUndirectedGraph**
@param properties the properties which the generated graphs will have
@return returns a randomly generated graph satisfying the specified **properties**

+ **deleteVertex(graph: SimpleUndirectedGraph, vertex: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex the index of a vertex of **graph**, which will be deleted
@return returns a modified copy of **graph** in which the vertex with index **vertex** and all edges incident to it are deleted
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex**

+ **addVertex(graph: SimpleUndirectedGraph): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@return returns a modified copy of **graph** which has precisely one isolated vertex more

+ **copyVertex(graph: SimpleUndirectedGraph, vertex: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex the index of a vertex of **graph**, which will be copied
@return returns a modified copy of **graph** in which the vertex with index **vertex** is duplicated i.e. there is a new vertex which has precisely the same neighborhood
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex**

+ **swapVertices(graph: SimpleUndirectedGraph, vertex1: int, vertex2: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex1 the index of a vertex of **graph**
@param vertex2 the index of another vertex of **graph**
@return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** swap indices. Note this results in a different but isomorphic graph to **graph**
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**

+ **contractVertices(graph: SimpleUndirectedGraph, vertex1: int, vertex2: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex1 the index of a vertex of **graph**
@param vertex2 the index of another vertex of **graph**
@return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** are contracted to a single vertex. Resulting loops will be deleted and multiedges will be reduced to one edge
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**

+ **deleteEdge(graph: SimpleUndirectedGraph, edge: SimpleUndirectedEdge): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param edge the edge which is going to be deleted
@return returns a modified copy of **graph** in which **edge** is deleted, if it was an edge in **graph**. Otherwise it just returns **graph**

+ **addEdge(graph: SimpleUndirectedGraph, edge: SimpleUndirectedEdge): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param edge the edge which is going to be inserted
@return returns a modified copy of **graph** in which **edge** is inserted if it wasnt already an edge in **graph** otherwise it returns just **graph**. Note that the edge being added may contain vertices which are not in **graph**, since missing vertices will automatically be added

+ **deleteIsolatedVertices(graph: SimpleUndirectedGraph): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@return returns a modified copy of **graph** in which all isolated vertices are deleted

Package graph.simpleHyperGraph

In this package simple hypergraphs (i.e. graphs whose edges are sets of at least two distinct vertices and whose edges dont overlap in more than one vertex) are implemented. It offers the functionality to generate, modify and distinct them by some for simple hypergraph welldefined criterions.

Class SimpleHyperGraph

Description

This class concretizes the graph class in the sense of a simple hypergraphs. Besides incidence relations this class offers methods to identify some of their properties.

Documentation

+ **SimpleHyperGraph(edges: List<SimpleHyperEdge>, numVertices: int)**
  A constructor for this class
  @param edges the edges this graph contains
  @param numVertices the amount of vertices this graph has
  @throws GraphInconsistencyException if **numVertices** ≤ 0, if there is a hyperedge with a vertex $\geq$ **numVertices** or if the resulting hypergraph is not simple

+ **SimpleHyperGraph(rageFormat: List<String>)**
  A constructor of this class, assuring that this graph type can be loaded from harddrive
  @param rageFormat the line by line representation of the graph as specified in the RAGE data format

+ **getVerticesBFS(): List<int>**
  @return returns the list of vertices of the graph in the order of a breadth first search

+ **areIncident(vertex: int, edge: SimpleHyperEdge): bool**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param edge an edge of the graph
  @return returns **true** iff the vertex is incident to the given edge
  @throws GraphInconstistencyException if **vertex** is an invalid vertex index or **edge** is not an edge of the graph

+ **areAdjacent(vertex1: int, vertex2: int): bool**
  @param vertex1 the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param vertex2 see **vertex1**
  @return returns **true** iff there is an edge which is incident to both vertices
  @throws GraphInconsistencyException if **vertex1** or **vertex2** is not a valid vertex index

+ **areAdjacent(edge1: SimpleHyperEdge, edge2: SimpleHyperEdge): bool**
  @param edge1 an edge of the graph
  @param edge2 another edge of the graph
  @return returns **true** iff there is a vertex which is incident to both edges
  @throws GraphInconsistencyException if **edge1** or **edge2** is not an edge of the graph

+ **getAdjacentVertices(vertex: int): List<int>**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @return returns the list of all vertices which are adjacent to **vertex**
  @throws GraphInconsistencyException if **vertex** is not a valid vertex index

+ **getAdjacentEdges(edge: SimpleHyperEdge): bool**
  @param edge an edge of the graph
  @return returns the list of all edges which are adjacent to **edge**
  @throws GraphInconsistencyException if **edge** is not an edge of the graph

+ **getIncidentEdges(vertex: int): List<SimpleHyperEdge>**
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all edges incident to **vertex**
@throws GraphInconsistencyException if **vertex** is an invalid vertex index

+ **getIncidentVertices(edges: List<SimpleHyperEdge>): List<int>**
@param edges a list of edges of the graph
@return returns the list of all vertices which are incident to any of the edges in the list
@throws GraphInconsistencyException if there is an edge in **edges**, which is not an edge of the graph

+ **isConnected(): bool**
@return returns **true** iff the graph is connected ie. iff for any two vertices there is a sequence of edges where any two consecutive edges are adjacent

+ **toRage(): List<String>**
@return returns the line-by-line representation of the graph as specified in the RAGE-data format

Class SimpleHyperEdge

Description

This class concretizes the class edge in the sense of a hyperedge. It always relates at least two distinct vertices.

Documentation

+ **SimpleHyperEdge(vertices: List<int>)**
A constructor for this class
@param vertices the vertices this edge sets in relation
@throws GraphInconsistencyException if the list is empty, contains just one vertex or any vertex twice

+ **getVertices(): List<int>**
@return returns the list of all indices of vertices incident to this edge

+ **equals(edge: E): bool**
@return returns **true** both edges are adjacent to the same vertices

+ **compareTo(edge: E): int**
The notion of order between edges $(x_1, ..., x_n)$ and $(y_1, ..., y_m)$ with $x_1 < ... < x_n$, $y_1 < ... < y_m$ and $n \leq m$ is defined by $(x1, ..., x_n) < (y_1, ..., y_n)$ iff $x_1 < y_1$ or ($x_1 = y_1$ and $x_2 < y_2$) or ... or ($x_1 = y_1$ and ... and $x_n = y_n$ and $n < m$)
@return returns **-1**/**0**/**1** if **edge** is greater/equal/smaller than the edge this method is invoked upon

Class SimpleHyperGraphProperties

Description

This class is an extension of the GraphProperties class and is likely meant for the exchange of data between controller and model, especially for transferring the settings required for generating simple hyper graphs. It assures that the following graph properties can be retrieved and set at all times:

- "uniform" – a nonnegative integer
- "minDegree" – a nonnegative integer
- "maxDegree" – a nonnegative integer
- "connected" – a boolean

Class SimpleHyperGraphBuilder

Description

This class concretizes the GraphBuilder class by offering methods for randomly generating simple hypergraphs after given SimpleHyperGraphProperties as well as modifying them.

Documentation

+ **generate(properties: SimpleHyperGraphProperties): SimpleHyperGraph**
  @param properties the properties which the generated graphs will have
  @return returns a randomly generated graph satisfying the specified **properties**
+ **deleteVertex(graph: SimpleHyperGraph, vertex: int): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @param vertex the index of a vertex of **graph**, which will be deleted
  @return returns a modified copy of **graph** in which the vertex with index **vertex** and all edges incident to it are deleted
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex**
+ **addVertex(graph: SimpleHyperGraph): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @return returns a modified copy of **graph** which has precisely one isolated vertex more
+ **swapVertices(graph: SimpleHyperGraph, vertex1: int, vertex2: int): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @param vertex1 the index of a vertex of **graph**
  @param vertex2 the index of another vertex of **graph**
  @return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** swap indices. Note this results in a different but isomorphic graph to **graph**
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**
+ **deleteEdge(graph: SimpleHyperGraph, edge: SimpleHyperEdge): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @param edge the edge which is going to be deleted
  @return returns a modified copy of **graph** in which **edge** is deleted, if it was an edge in **graph**. Otherwise it just returns **graph**

+ **addEdge(graph: SimpleHyperGraph, edge: SimpleHyperEdge): SimpleHyperGraph**
@param graph the graph which is going to be modified
@param edge the edge which is going to be inserted
@return returns a modified copy of **graph** in which **edge** is inserted if it wasnt already an edge in **graph** otherwise it returns just **graph**. Note that the edge being added may contain vertices which are not in **graph**, since missing vertices will automatically be added

+ **deleteIsolatedVertices(graph: SimpleHyperGraph): SimpleHyperGraph**
@param graph the graph which is going to be modified
@return returns a modified copy of **graph** in which all isolated vertices are deleted

Package heuristic

The package contains the interface for implementing heuristics. In the subpackages some heuristics for the total coloring conjecture as well as for the Erdös-Faber-Lovasz conjecture are implemented.

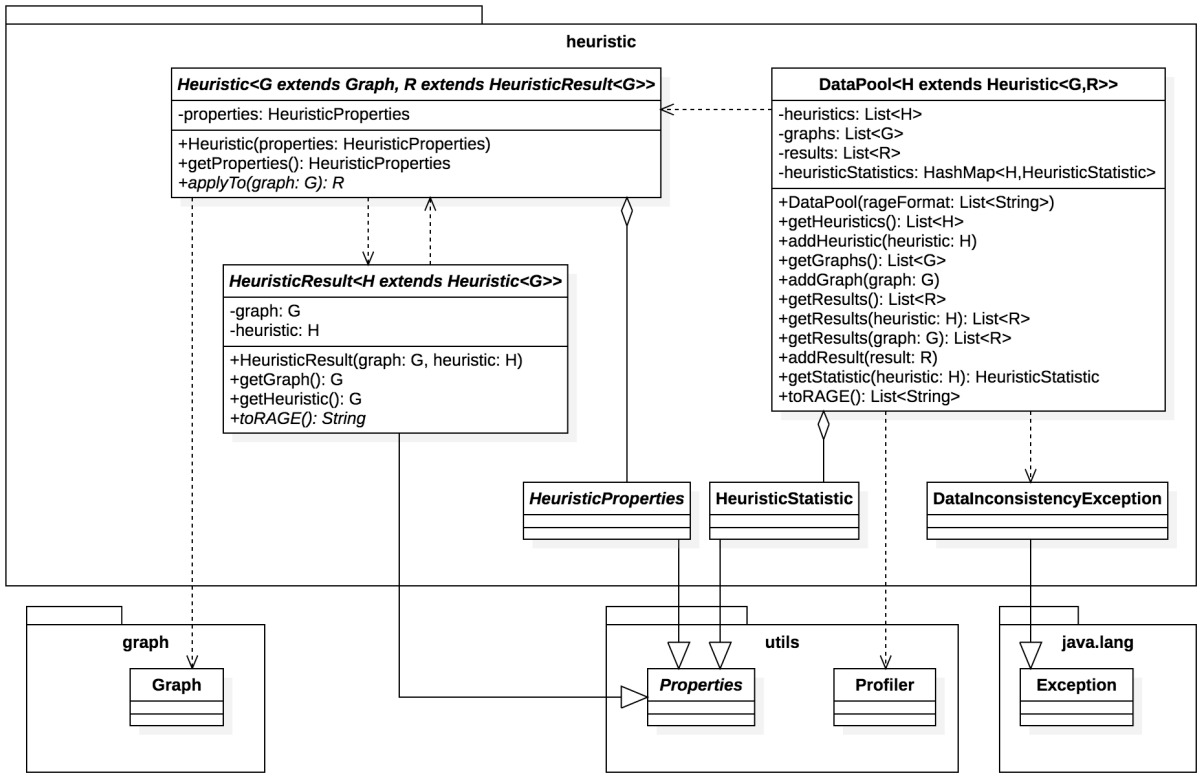


Abbildung 3: Das Paket heuristic

Class Heuristic

Description

The class is the abstract interface of a heuristic which is applied to a graph of typ **G** which has a result of type **R**.

Documentation

+ **Heuristic(properties: HeuristicProperties)**
  A constructor for this class
  @param properties the properties defining this heuristic
+ **getProperties(): HeuristicProperties**
  @return returns the properties of this heuristic
+ ***applyTo(graph: G): R***
  @param graph the graph of type **G** on which the heuristic will be applied
  @return returns the result of the heuristic application

Class HeuristicResult

Description

This class is the abstract interface of the result of a specific calculation of an heuristic **H** on a specific graph of type **G**.

Documentation

+ **HeuristicResult(graph: G, heuristic: H)**
  The constructor of this class
  @param graph the graph this heuristic was calculated upon
  @param heuristic the heuristic by which the result was calculated
+ **getGraph(): G**
  @return returns the graph this result was calculated upon
+ **getHeuristic(): H**
  @return returns the heuristic by which this result was calculated
+ ***toRAGE(): List<String>***
  @return returns the line-by-line representation of this heuristic result as specified in the RAGE data format

Class HeuristicProperties

Description

This class serves as collection of data for exchange between controller and model, especially to transfer properties of heuristics. It assures that the following properties may be retrieved and set at any time:

- ”name” – ein String
- "valid- ein Boolean

Class DataPool

Description

The class manages the application of heuristics of type **H** on graphs of type **G** which results have type **R**. It assures that every heuristic stored in the pool is applied to every graph stored in the pool. Moreover it gathers statistics over this applications.

Documentation

+ **DataPool(rageFormat: List<String>)**
  A constructor for this class, assuring that the datapool can be loaded from harddrive
  @param rageFormat the line by line representation of a datapool as specified in the RAGE data format.
+ **getHeuristics(): List<H>**
  @return returns the list of heuristics currently in this data pool
+ **addHeuristic(heuristic: H)**
  @param heuristic the heuristic to be added to data pool, which then will be applied to every graph in the data pool
  @throws DataInconsistencyException if heuristic may not be applied on graphs of type **G** or does not has results of type **R**
+ **getGraphs(): List<G>**
  @return returns the list of graphs currently in this data pool
+ **addGraph(graph: G)**
  @param graph the graph to be added to the data pool, on which then all heuristics in the data pool will be applied
  @throws DataInconsistencyException if heuristics of type **H** may not be applied on this graph
+ **getResults(): List<R>**
  @return returns the list of all results calculated on graphs by heuristics in this data pool
+ **getResults(heuristic: H): List<R>**
  @param heuristic the heuristic the results were calculated by
  @return returns all results calculated by **heuristic** on graphs in this data pool
+ **getResults(graph: G)**
  @param graph the graph the results were calculated upon
  @return returns all results calculated on **graph** by heuristics in this data pool

+ **getStatistics(heuristic: H): HeuristicStatistic**
@param heuristic the heuristic whose statistics are requested
@return returns the statistic gathered for **heuristic**
@throws DataInconsistencyException if **heuristic** is not a heuristic of this data pool

+ **toRAGE(): List<String>**
@return returns the line by line representation of this data pool as specified in the RAGE data format

Class HeuristicStatistic

This class collects some statistics overt the applications of a specific heuristic within a data pool. It assures that the following properties may be retrieved at any time:

- "minRuntime" – a floating point number
- "avgRuntime" – a floating point number
- maxRuntime- a floating point number
- numApplications- a nonnegative integer
- numSuccesses- a nonnegative integer

Class DataInconsistencyException

Description

This class extends the usual Java Exception to an exception specifically thrown when data pools are treated wrong.

Package heuristic.totalColoring

In this package and its subpackages some heuristics for the **total coloring conjecture** (ie. any simple undirected graph with maximal degree Δ has a total coloring with $\Delta + 2$ colors) are implemented.

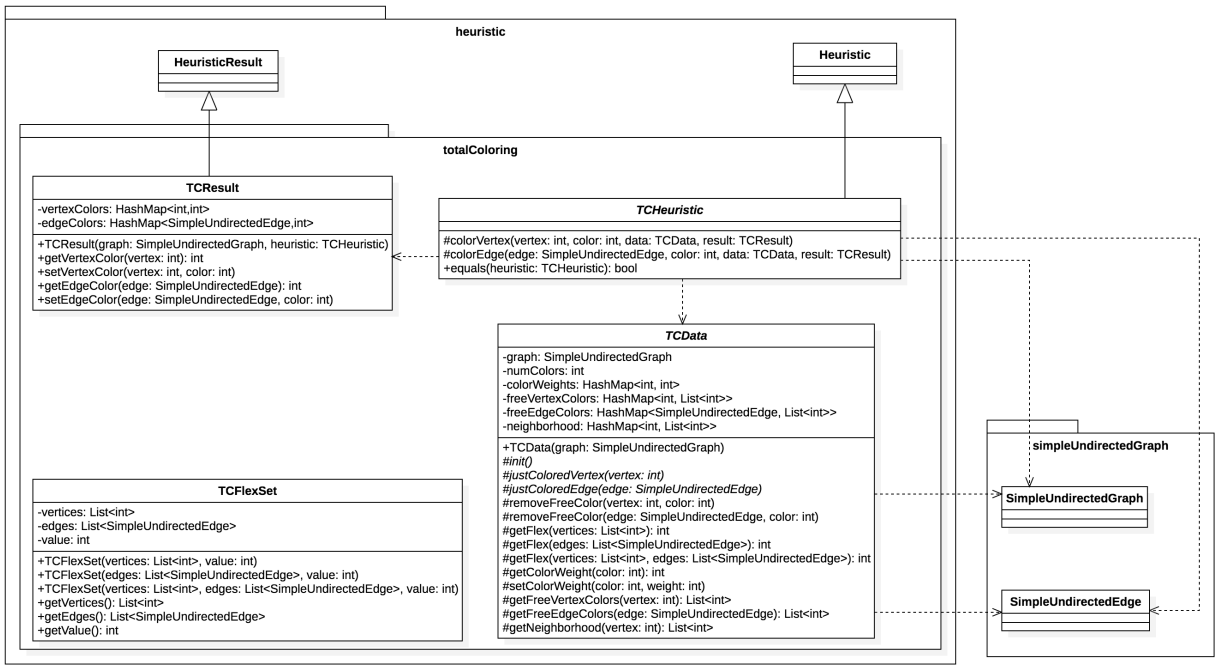


Abbildung 4: totalColoring

Class TCHeuristic

Description

This abstract class is the abstract interface for a total coloring heuristic. It assures that any total coloring heuristic is calculated on SimpleUndirectedGraphs and returns a TCResult as result. It provides some methods, which any total coloring heuristic needs, such as coloring vertices and edges.

Documentation

\# **colorVertex(vertex: int, color: int, data: TCData, result: TCResult)**
@param vertex the vertex to be colored
@param color the color which will be assigned to the vertex
@param data the data required for the calculation of a total coloring
@param result the resulting total coloring

\# **colorEdge(edge: SimpleUndirectedEdge, color: int, data: TCData, result: TCResult)**
@param edge the edge to be colored
@param color the color which will be assigned to the edge
@param data the data required for the calculation of a total coloring
@param result the resulting total coloring

\+ **equals(heuristic: TCHeuristic): bool**
@param heuristic another TCHeuristic this will be compared to
@return returns **true** iff the other TCHeuristic of of the same type and has exactly the same properties

Class TCResult

Description

This class represents a total coloring of a simple undirected graph ie. a coloring of vertices and edges, such that no two adjacent or incident objects share the same color. Colors are represented as integers.

Documentation

+ **TCResult(graph: SimpleUndirectedGraph, heuristic: TCHeuristic)**
  A constructor for this class **@param graph** the graph this result was calculated upon
  @param heuristic the heuristic this result was calculated by

+ **getVertexColor(vertex: int): int**
  @param vertex the vertex whose color is requested
  @return returns the color of **vertex**
  @throws DataInconsistencyException if vertex has no color

+ **setVertexColor(vertex: int, color: int)**
  @param vertex the vertex to be colored
  @param color the color to color **vertex** with

+ **getEdgeColor(edge: SimpleUndirectedEdge): int**
  @param edge the edge whose color is requested
  @return returns the color of **edge**
  @throws DataInconsistencyException if edge has no color

+ **setEdgeColor(edge: SimpleUndirectedEdge, color: int)**
  @param edge the edge to be colored
  @param color the color to color **edge** with

Class TCData

Description

This abstract class encapsulates the data required temporarly to calculate a total coloring, such as the lists of **free colors** of uncolored vertices and edges (ie. the colors which are not used by other objects adjacent / incident to them). Moreover it stores the weighted (vertex vs. edges) sum of how often colors are used.

Documentation

TCData(graph: SimpleUndirectedGraph)
  A constructor of this class
  @param graph the graph the heuristic is running at

init()
  May be implemented to (re-)initialize the data at any time within the running heuristic

justColoredVertex(vertex: int)
May be implemented to update data anytime when a vertex was colored
@param vertex the vertex which was just colored

justColoredEdge(edge: SimpleUndirectedEdge)
May be implemented to update data anytime when an edge was colored
@param edge the edge which was just colored

removeFreeColor(vertex: int, color: int)
@param vertex the vertex which will have one free color less
@param color the color which **vertex** mustnt use

removeFreeColor(edge: SimpleUndirectedEdge, color: int)
@param edge the edge which will have one free color less
@param color the color which **edge** mustnt use

getFlex(vertices: List<int>): int
@param vertices the set of vertices whose flexibility should be calculated
@return returns the flexibility of these vertices ie. # of colors free for all vertices – # of **vertices**

getFlex(edges: List<SimpleUndirectedEdge>): int
@param vertices the set of vertices whose flexibility should be calculated
@return returns the flexibility of these vertices ie. # of colors which are free for all edges – # of **edges**

getFlex(vertices: List<int>, edges: List<SimpleUndirectedEdge>): int
@param vertices a set of vertices
@param edges a set of edges
@return returns the flexibility of these objects ie. # of colors which are free for all objects – # of **objects**

getColorWeight(color: int): int
@param color the color whose weight is requested
@return returns the weight of this color ie. how often it was used weighted differently by vertices and edges

setColorWeight(color: int, weight: int)
@param color the color whose weight will be updated
@param weight the new weight of **color**

getFreeVertexColors(vertex: int): List<int>
@param vertex the vertex whose free colors are requested
@return returns the list of free colors of **vertex**

getFreeVertexColors(edge: SimpleUndirectedEdge): List<int>
@param edge the edge whose free colors are requested
@return returns the list of free colors of **edge**

Class TCFlexSet

Description

This class represents a subset of vertices and edges of a graph with a given **flexibility value** (ie. # colors free for all objects – # objects) used heavily in some TCHeuristics.

Documentation

TCFlexSet(vertices: List<int>, value: int)
A constructor of this class
@param vertices some vertices
@param value the flexibility value of **vertices**

TCFlexSet(edges: List<SimpleUndirectedEdge>, value: int)
A constructor of this class
@param edges some edges
@param value the flexibility value of **edges**

TCFlexSet(vertices: List<int>, edges: List<SimpleUndirectedEdge>, value: int)
A constructor of this class
@param vertices some vertices
@param edges some edges
@param value the flexibility value of the set of objects in **vertices** and **edges**

getVertices(): List<int>
@return returns the vertices in this flex set

getEdges(): List<SimpleUndirectedEdge>
@return returns the edges in this flex set

getValue(): int
@return returns the flexibility value of this set of objects

Package heuristic.totalColoring.greedy

In this package some greedy heuristics for the total coloring conjecture are implemented. They all have in common, that the vertices are colored first and the edges are colored afterwards. The heuristics differ in the way the edges are colored.

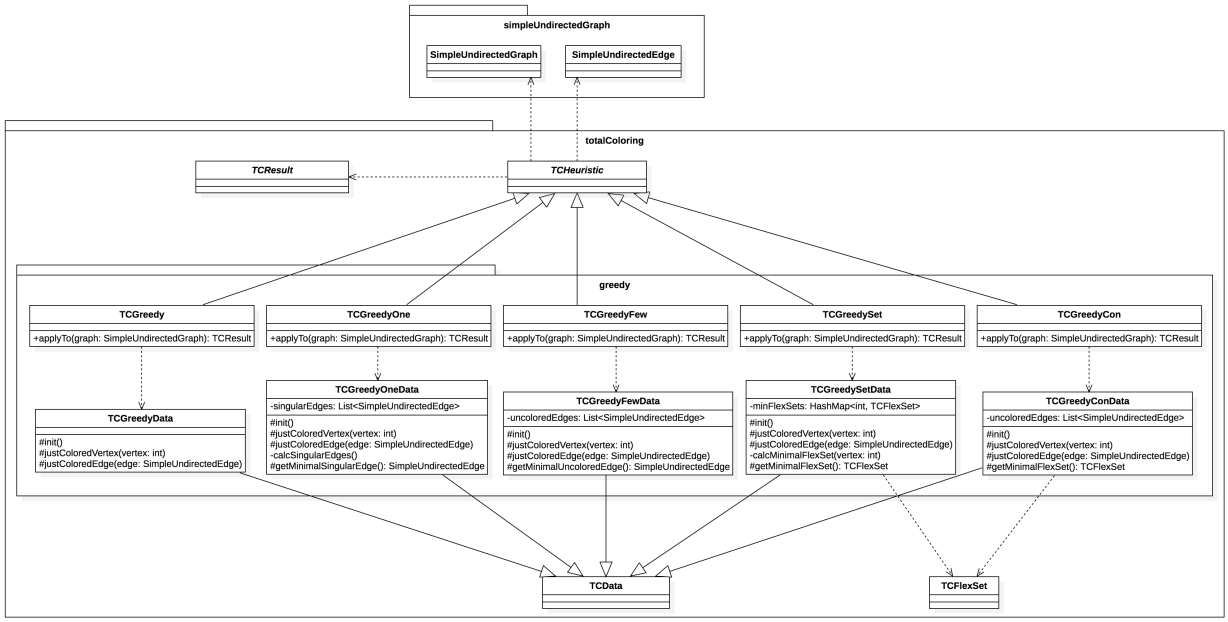


Abbildung 5: TCGreedy

Class TCGreedyData

Description

Since TCData is abstract this class is required such that the TCGreedy heuristic has its own data class, even if with respect to TCData no additional attributes or methods are added.

Class TCGreedy

Description

This class implements the TCGreedy heuristic which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
  @param graph the graph this heuristic will be applied on
  @return returns the calculated coloring

Class TCGreedyOneData

Description

This class stores all uncolored edges with exactly one free color temporarily.

Documentation

init()
initializes the list of all uncolored edges with exactly one free color

justColoredEdge(edge: SimpleUndirectedEdge)
updates the list of edges with exactly one free color
@param edge the edge which was just colored

- **calcSingularEdges()**
updates the list of edges with exactly one free color

getMinimalSingularEdge(): SimpleUndirectedEdge
@return returns the minimal edge with exactly one free color with respect to the order defined on edges

Class TCGreedyOne

Description

This class implements the TCGreedyOne heuristic which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class TCGreedyFewData

Description

This class stores all uncolored edges sorted first by their amount of free colors and then by the order defined on edges.

Documentation

init()
initializes the list of uncolored edges

justColoredEdge(edge: SimpleUndirectedEdge)
updates the list of uncolored edges

getMinimalUncoloredEdge(): SimpleUndirectedEdge
@return returns the minimal uncolored edge with respect to the number of free colors and the order defined on edges

Class TCGreedyFew

Description

This class implements the TCGreedyFew heuristic, which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class TCGreedySetData

Description

This class stores for any vertex v the subset of all uncolored edges incident to v which has the lowest flexibility value (ie. # of colors which are free for every edge in this set − # of edges in the set) and is the lowest with respect to lexicographic ordering using the order defined on edges. These sets are from now on referred to as minimal flex sets

Documentation

init()
initializes the minimal flex sets

justColoredEdge(edge: SimpleUndirectedEdge)
updates the minimal flex sets of the vertices incident to edge

- **calcMinimalFlexSet(vertex: int)**
calculates the minimal flex set of **vertex**
@param vertex the vertex whose minimal flex set is calculated

getMinimalFlexSet(): TCFlexSet
@return returns the minimal flex set belonging to the vertex with minimal index

Class TCGreedySet

Description

This class implements the TCGreedySet heuristic, which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class TCGreedyConData

Description

This class stores the list of uncolored edges temporarily to compute connected subsets of uncolored edges up to a specific size.

Documentation

- # **init()**
  initializes the list of uncolored edges
- # **justColoredEdge(edge: SimpleUndirectedEdge)**
  updates the list of uncolored edges
- # **getMinimalFlexSet(): TCFlexSet**
  @return returns the connected set of uncolored edges with minimal flexibility value (# of colors which are free for all edges – # of edges) and minimal lexicographic order using the order defined on edges.

Class TCGreedyCon

This class implements the TCGreedyCon heuristic, which tries to calculate a total coloring as specified in the addendum.

Documentation

- + **applyTo(graph: SimpleUndirectedGraph): TCResult**
  @param graph the graph this heuristic will be calculated on
  @return returns the calculated coloring

Package heuristic.totalColoring.mixedGreedy

In this package some heuristics for the total coloring conjecture are implemented. In comparison to the greedy heuristics, these heuristics do not seperate the coloring of vertices and edges strictly but rather alternate between them. Nethertheless they work in a similiarly greedy fashion.

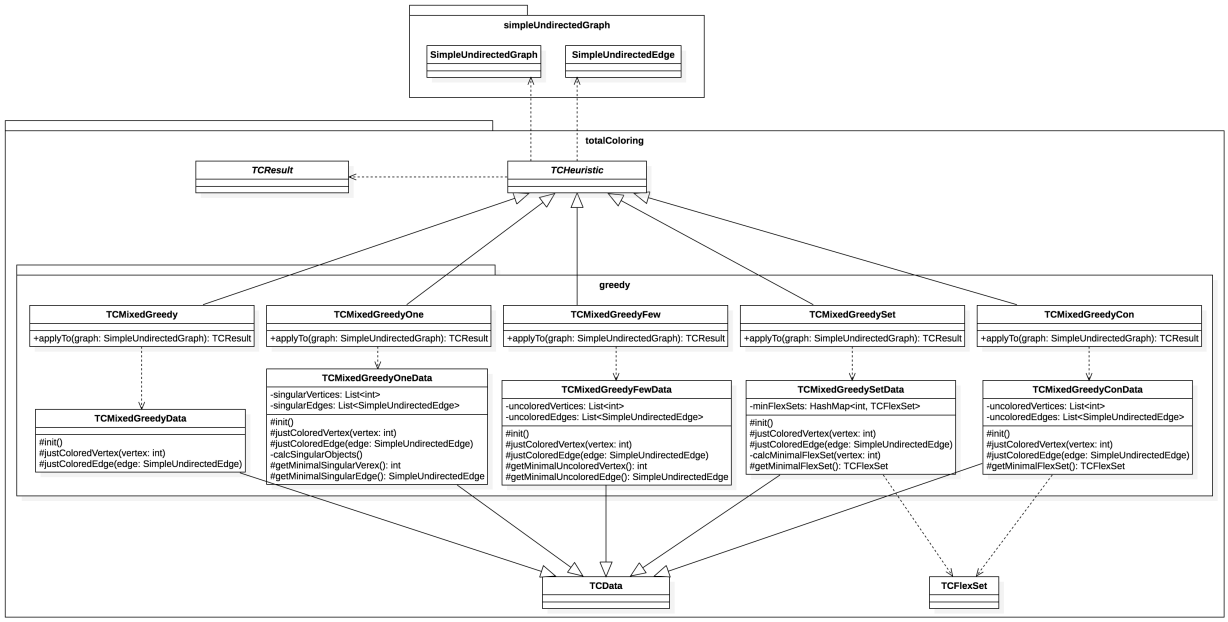


Abbildung 6: TCMixedGreedy

Class TCMixedGreedyData

Description

Since TCData is abstract this class is required such that the TCGreedy heuristic has its own data class, even if with respect to TCData no additional attributes or methods are added.

Class TCMixedGreedy

Description

This class implements the TCMixedGreedy heuristic which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
  @param graph the graph this heuristic will be applied on
  @return returns the calculated coloring

Class TCMixedGreedyOneData

Description

This class stores all uncolored vertices and all uncolored edges with exactly one free color temporarily.

Documentation

init()
initializes the list of all uncolored edges with exactly one free color

justColoredVertex(vertex: int)
updates the lists of objects with exactly one free color
@param vertex the vertex which was just colored

justColoredEdge(edge: SimpleUndirectedEdge)
updates the lists of objects with exactly one free color
@param edge the edge which was just colored

- **calcSingularObjects()**
updates the lists of objects with exactly one free color

getMinimalSingularVertex(): int
@return returns the vertex with minimal index and exactly one free color

getMinimalSingularEdge(): SimpleUndirectedEdge
@return returns the minimal edge with exactly one free color with respect to the order defined on edges

Class TCMixedGreedyOne

Description

This class implements the TCMixedGreedyOne heuristic which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
@param graph the graph this heuristic will be applied on
@return returns the calculated coloring

Class TCMixedGreedyFewData

Description

This class stores all uncolored vertices and all uncolored edges, both sorted by their number of free colors and inherent order.

Documentation

init()
initializes the list of all uncolored objects

justColoredVertex(vertex: int)
updates the lists of uncolored objects
@param vertex the vertex which was just colored

justColoredEdge(edge: SimpleUndirectedEdge)
updates the lists of uncolored objects
@param edge the edge which was just colored

getMinimalUncoloredVertex(): int
@return returns the vertex with minimal amount of free colors and minimal index

getMinimalUncoloredEdge(): SimpleUndirectedEdge
@return returns the edge with the minimal amount of free colors and minimal order

Class TCMixedGreedyFew

Description

This class implements the TCMixedGreedyFew heuristic which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
@param graph the graph this heuristic will be applied on
@return returns the calculated coloring

Class TCMixedGreedySetData

Description

This class stores for every vertex the subset of edges incident to this vertex which has minimal flexibility and minimal lexicographic order using the order defined on the edges. If the vertex is uncolored, versions with the vertex added to the set and without it are considered. These sets will from now on be reffered to as flexibility sets.

Documentation

init()
initializes the list of flexibility sets

justColoredVertex(vertex: int)
updates the minimal flexibility set of **vertex**
@param vertex the vertex which was just colored

justColoredEdge(edge: SimpleUndirectedEdge)
updates the minimal flexibility sets of both vertices incident to **edge**
@param edge the edge which was just colored

- **calcMinimalFlexSet(vertex: int)**
@param vertex the vertex whose minimal flexibility set will be updated

getMinimalFlexSet(): int
@return returns the vertex with minimal amount of free colors and minimal index

Class TCMixedGreedySet

Description

This class implements the TCMixedGreedySet heuristic which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
@param graph the graph this heuristic will be applied on
@return returns the calculated coloring

Class TCMixedGreedyConData

Description

This class stores uncolored vertices and edges to compute connected subsets of uncolored vertices and edges up to a specific size.

Documentation

init()
initializes the list of flexibility sets

justColoredVertex(vertex: int)
updates the list of uncolored vertices
@param vertex the vertex which was just colored

justColoredEdge(edge: SimpleUndirectedEdge)
updates the list of uncolored edges
@param edge the edge which was just colored

getMinimalFlexSet(): int
@return returns the connected set of uncolored vertices and edges with minimal flexibility value (# of colors which are free for all objects − # of objects) and minimal lexicographic order using the indices of vertices and the order defined on edges

Class TCMixedGreedyCon

Description

This class implements the TCMixedGreedyCon heuristic which tries to calculate a total coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
@param graph the graph this heuristic will be applied on
@return returns the calculated coloring

Package heuristic.erdosFaberLovasz

In this package and its subpackages some heuristics for the Erdös-Faber-Lovasz conjecture (ie. any simple hypergraph on n vertices has a proper edge coloring with n colors) are implemented.

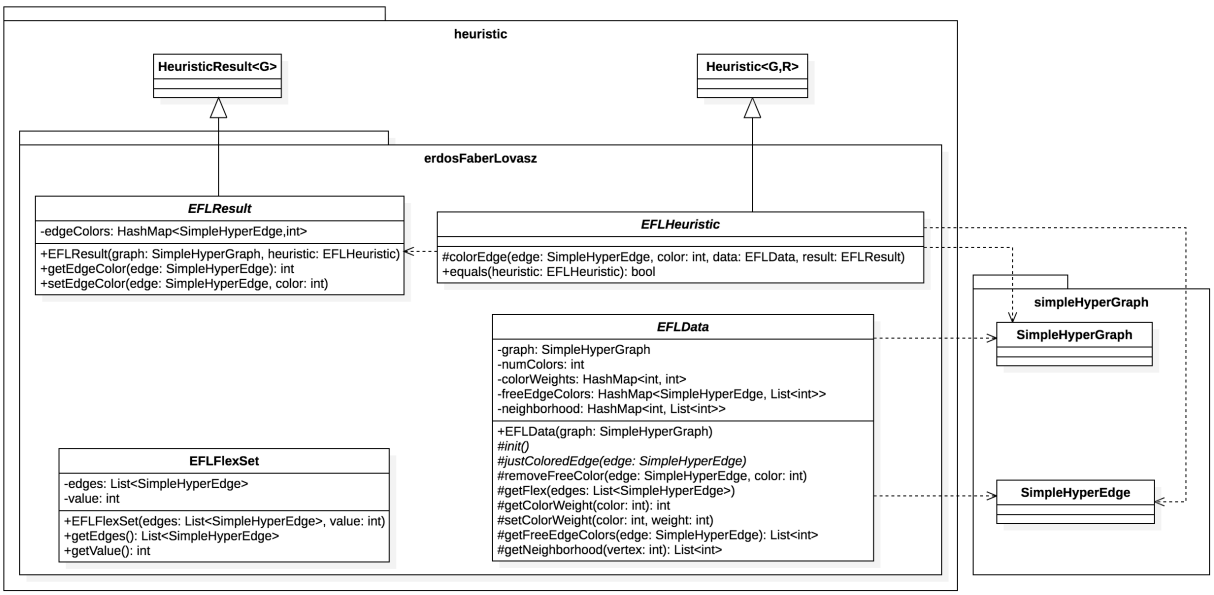


Abbildung 7: Erdös-Faber-Lovasz

Class EFLHeuristic

Description

This abstract class is the abstrac interface for an Erdös-Faber-Lovasz heuristic. It assures that every EFL-heuristic is calculated on SimpleHyperGraphs and returns an EFLResult as result. It provides some methods which any EFL-heuristic needs, such as coloring edges.

Documentation

\# **colorEdge(edge: SimpleHyperEdge, color: int, data: EFLData, result: EFLResult)**
@param edge the hyperedge to be colored
@param color the color which will be assigned to the edge
@param data the data required for the calculation of a proper coloring
@param result the resulting edge coloring

\+ **equals(heuristic: EFLHeuristic): bool**
@param heuristic another EFLHeuristic this will be compared to
@return returns **true** iff the other EFLHeuristic is of the same type and has exactly the same properties

Class EFLResult

Description

This class represents an edge coloring of a simple hypergraph ie. a coloring of the hyperedges of a simple hypergraph, such that no two adjacent hyperedges share the same color. Colors are as always represented as integers.

Documentation

+ **EFLResult(graph: SimpleHyperGraph, heuristic: EFLHeuristic)**
  A constructor for this class
  @param graph the graph this result was calculated upon
  @param heuristic the heuristic this result was calculated by

+ **getEdgeColor(edge: SimpleHyperEdge): int**
  @param edge the edge whose color is requested
  @return returns the color of **edge**
  @throws DataInconsistencyException if edge has no color

+ **setEdgeColor(edge: SimpleHyperEdge, color: int)**
  @param edge the edge to be colored
  @param color the color to color **edge** with

Class EFLData

Description

This abstract class encapsulates the data required temporarly to calculate a EFL-coloring, such as the lists of **free colors** of uncolored edges (ie. the colors which are not used by other objects adjacent / incident to them). Moreover it stores how often colors are used.

Documentation

- # **EFLData(graph: SimpleHyperGraph)**
  A constructor of this class
  @param graph the graph the heuristic is running at
- # ***init()***
  May be implemented to (re-)initialize the data at any time within the running heuristic
- # ***justColoredEdge(edge: SimpleHyperEdge)***
  May be implemented to update data anytime when an edge was colored
  @param edge the edge which was just colored
- # **removeFreeColor(edge: SimpleUndirectedEdge, color: int)**
  @param edge the edge which will have one free color less
  @param color the color which **edge** mustnt use
- # **getFlex(edges: List<SimpleUndirectedEdge>): int**
  @param vertices the set of vertices whose flexibility should be calculated
  @return returns the flexibility of these vertices ie. # of colors which are free for all edges – # of **edges**
- # **getColorWeight(color: int): int**
  @param color the color whose weight is requested
  @return returns the weight of this color ie. how often it was used weighted differently by vertices and edges
- # **setColorWeight(color: int, weight: int)**
  @param color the color whose weight will be updated
  @param weight the new weight of **color**
- # **getFreeVertexColors(edge: SimpleUndirectedEdge): List<int>**
  @param edge the edge whose free colors are requested
  @return returns the list of free colors of **edge**

Class EFLFlexSet

Description

This class represents a subset of edges of a graph with a given **flexibility value** (ie. # colors free for all edges – # edges) used heavily in some EFLHeuristics.

Documentation

- # **EFLFlexSet(edges: List<SimpleHyperEdge>, value: int)**
  A constructor of this class
  @param edges some edges
  @param value the flexibility value of **edges**
- # **getEdges(): List<SimpleHyperEdge>**
  @return returns the edges in this flex set
- # **getValue(): int**
  @return returns the flexibility value of this set of objects

Package heuristic.erdosFaberLovasz.greedy

In this package some heuristics for the Erdös Faber Lovasz conjecture are implemented. They behave like the edge coloring part of the TCGreedy- heuristics.

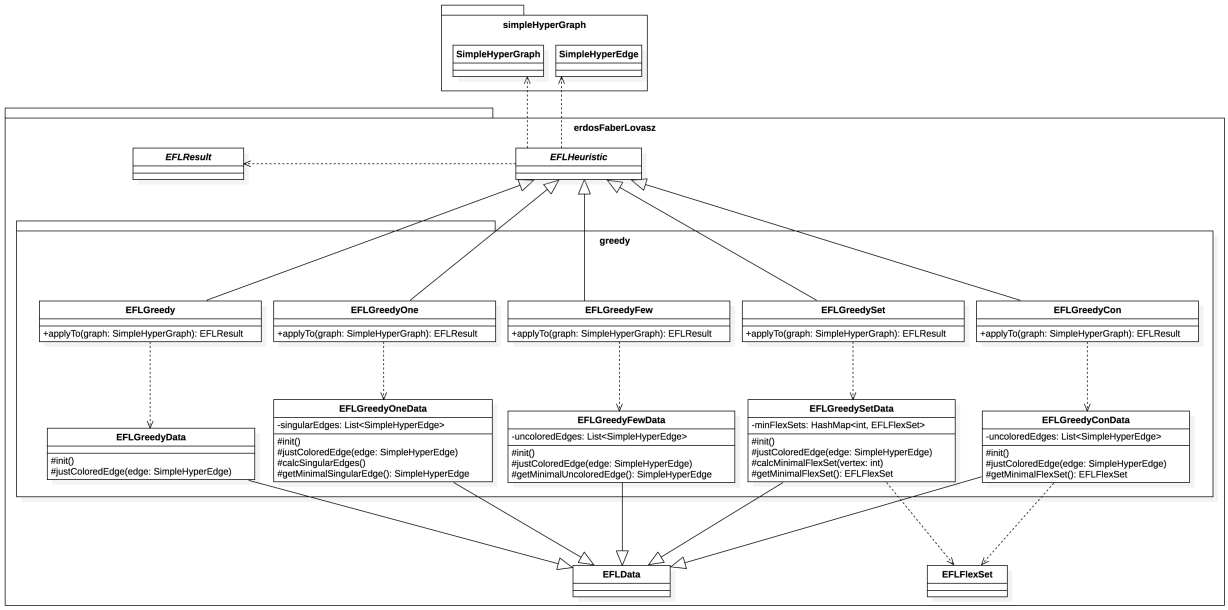


Abbildung 8: EFLGreedy

Class EFLGreedyData

Description

Since EFLData is abstract this class is required such that the EFLGreedy heuristic has its own data class, even if with respect to EFLData no additional attributes or methods are added.

Class EFLGreedy

Description

This class implements the EFLGreedy heuristic which tries to calculate a proper hyperedge coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleHyperGraph): EFLResult**
  @param graph the graph this heuristic will be applied on
  @return returns the calculated coloring

Class EFLGreedyOneData

Description

This class stores all uncolored edges with exactly one free color temporarily.

Documentation

init()
initializes the list of all uncolored edges

justColoredEdge(edge: SimpleHyperEdge)
updates the list of edges with exactly one free color
@param edge the edge which was just colored

- **calcSingularEdges()**
updates the list of edges with exactly one free color

getMinimalSingularEdge(): SimpleUndirectedEdge
@return returns the minimal edge with exactly one free color with respect to the order defined on edges

Class EFLGreedyOne

Description

This class implements the EFLGreedyOne heuristic which tries to calculate a hyperedge coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleHyperGraph): EFLResult**
@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class EFLGreedyFewData

Description

This class stores all uncolored edges sorted first by their amount of free colors and then by the order defined on edges.

Documentation

init()
initializes the list of uncolored edges

justColoredEdge(edge: SimpleHyperEdge)
updates the list of uncolored edges

getMinimalUncoloredEdge(): SimpleUndirectedEdge
@return returns the minimal uncolored edge with respect to the number of free colors and the order defined on edges

Class EFLGreedyFew

Description

This class implements the EFLGreedyFew heuristic, which tries to calculate a hyperedge coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleHyperGraph): EFLResult**
@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class EFLGreedySetData

Description

This class stores for any vertex v the subset of all uncolored hyperedges incident to v which has the lowest flexibility value (ie. # of colors which are free for every edge in this set – # of edges in the set) and is the lowest with respect to lexicographic ordering using the order defined on edges. These sets are from now on referred to as minimal flex sets

Documentation

init()
initializes the minimal flex sets

justColoredEdge(edge: SimpleHyperEdge)
updates the minimal flex sets of the vertices incident to edge

- **calcMinimalFlexSet(vertex: int)**
calculates the minimal flex set of **vertex**
@param vertex the vertex whose minimal flex set is calculated

getMinimalFlexSet(): EFLFlexSet
@return returns the minimal flex set belonging to the vertex with minimal index

Class EFLGreedySet

Description

This class implements the EFLGreedySet heuristic, which tries to calculate a hyperedge coloring as specified in the addendum.

Documentation

+ **applyTo(graph: SimpleHyperGraph): EFLResult**
@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class EFLGreedyConData

Description

This class stores the list of uncolored edges temporarily to compute connected subsets of uncolored edges up to a specific size.

Documentation

- # **init()**
  initializes the list of uncolored edges
- # **justColoredEdge(edge: SimpleHyperEdge)**
  updates the list of uncolored edges
- # **getMinimalFlexSet(): EFLFlexSet**
  @return returns the connected set of uncolored edges with minimal flexibility value (# of colors which are free for all edges – # of edges) and minimal lexicographic order using the order defined on edges.

Class EFLGreedyCon

This class implements the EFLGreedyCon heuristic, which tries to calculate a hyperedge coloring as specified in the addendum.

Documentation

- + **applyTo(graph: SimpleHyperGraph): EFLResult**
  @param graph the graph this heuristic will be calculated on
  @return returns the calculated coloring

4 View

4.1 Allgemein

Die View steuert die Benutzerschnittstelle und damit die graphische Repräsentation der Daten und kommuniziert über Controller mit den restlichen Komponenten des Systems.
Für die Entwicklung einer graphischen Benutzerschnittstelle unter Java stehen unter anderem folgende Frameworks zur Verfügung:

1. Standart Widget Toolkit (SWT)
2. Abstract Widget Toolkit (AWT)
3. Swing
4. JavaFX

Aufgrund bisheriger Entwicklungs-Erfahrung mit JavaFX und FXML wurde in diesem System das JavaFX-Framework verwendet.

4.1.1 JavaFX

- JavaFX ist eine Abkürzung für Java Graphics.
- JavaFX ist eine Möglichkeit unter Java eine graphische Oberfläche zu erstellen.
- JavaFX ist eine komplette Neuentwicklung von Oracle.
- Es ist unabhängig von den bisherigen Methoden AWT und Swing.
- JavaFX wurde 2014 veröffentlicht.
- Es ist seit Version 7.6 in x86 Java Standard Edition (JavaSE) Runtime Installation enthalten.
- Da wir mit Java 8 arbeiten werden ist dies somit kein Problem.

JavaFX arbeitet mit einem Szenengraphen (engl. scene graph), der die einzelnen Bestandteile einer GUI verwaltet. Auf diesen werden dann alle weiteren Bestandteile gesetzt.

4.1.2 FXML

Die FX-Markup-Language (FXML) ist eine deklarative Beschreibung der grafischen Oberflächen auf XML-Basis. Dies bietet einige Vorteile gegenüber der konventionellen GUI-Entwicklung.
Zum einen ist durch diese Technologie die Trennung des Designs der GUI und deren Funktionalität strikt getrennt. Zum anderen ist das Einfügen von GUI-Bestandteilen, die an mehreren Stellen der Benutzeroberfläche zum Einsatz kommen sehr einfach möglich.
Dies Ermöglicht, dass der mehrfachverwendbare Code nur einmal in einem Separatem FXML-Dokument abgespeichert werden muss und dann über den „include-Tag“ an allen Stellen verwendet werden kann. Darüber hinaus können für die Gestaltung auch Web-Technologien wie CSS eingesetzt werden. Dies sorgt zusätzlich für eine Trennung von Layout auf der einen und Style und Design auf der anderen Seite, da separate CSS-Dateien erstellt werden können. Diese können dann in den FXML-Code eingebettet werden, sodass die GUI das Design übernehmen kann.
Die Entwicklung der FXML-Dateien erfolgt zuerst über den SceneBuilder. Dieser ist ein grafisches Tool, das die Erstellung von FXML-Dateien vereinfacht. Der daraus generierte Code wird bei Bedarf dann nochmals per Hand nachbearbeitet. Zur Nachbereitung zählen unter anderem auch das Einfügen der „include-Tags“ (wie oben beschrieben).

4.2 Entwurf

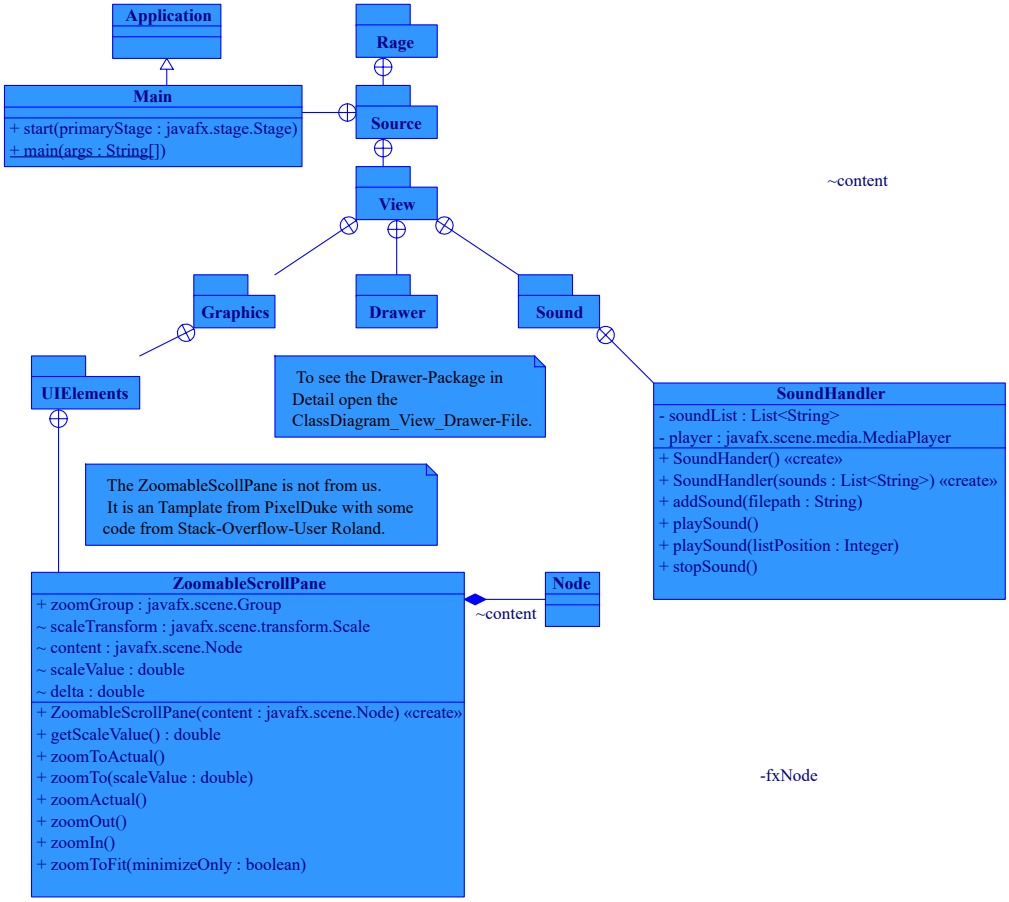


Abbildung 9: Das Paket View

Der Entwurf der View gliedert sich prinzipiell in folgende Pakete auf:

1. Graphic
2. Drawer
3. Sound

Diese sind Unterpakete des „View-Packages“ und werden im folgenden genauer betrachtet.

Package Graphic

The Graphic-Package is a Package for some adaptations and expansions with the JavaFx Stuff.

Package Graphic.UIElements

The UI-Elements-Package contains new created UI-Elements that expand the JavaFx-UI.

Class ZoomableScrollPane

Beschreibung
This is an expansion to the JavaFx-ScrollPane. This adds the ScrollPane that it can be zoomed.

This is used so that the drawn Graph could be zoomed in/out so that the user can easily look for some Edges.

This Class is not made by ourself. @author https://www.pixelduke.com/2012/09/16/zooming-inside-a-scrollpane/

Dokumentation
Because this Class is already fully implemented by the creator, there will be no Documentation from our side.

Package Drawer

The Drawer-Package. This Package contains everything that belongs to the Drawing of the Graphs. It is a upper-Package, therefore no further Documentation for this.

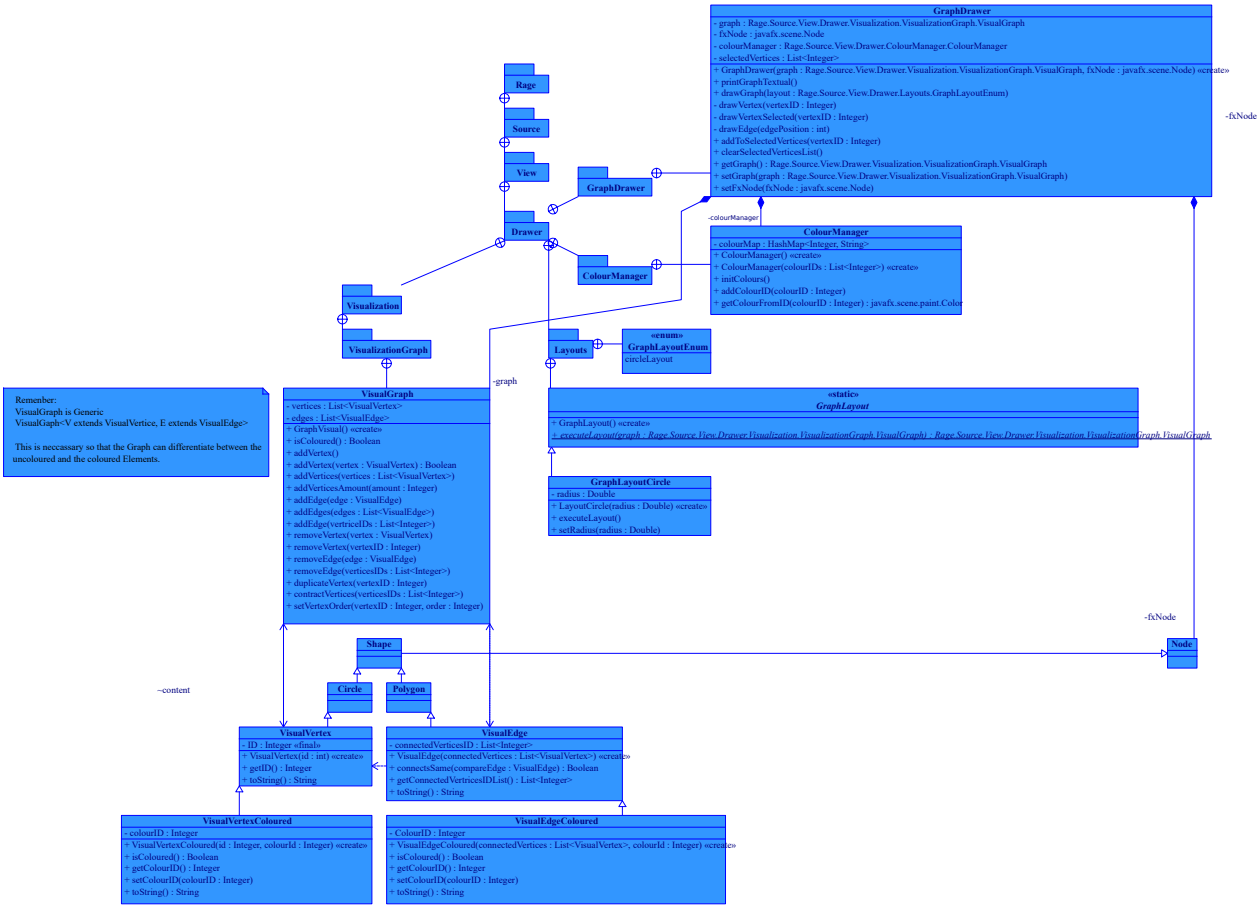


Abbildung 10: ViewDrawer

Package Drawer.GraphDrawer

The GraphDrawer-Package contains like the Name suggested the GraphDrawer that visualizes the Graph and "drawsït to a JavaFx-Node for the User.

Class GraphDrawer

Description
The Drawer that draws the given Graph to the given JavaFx-Node.

Documentation

- **graph : VisualGraph**
  The Graph that should be drawn.
- **fxNode : javafx.scene.Node**
  The JavaFx-Node where the Graph should be drawn on.
- **colourManager : ColourManager**
  The ColourManager of this Drawer to Map the ColourID's to the actual Colours of the to drawn Objects.

  This Object is created at the Constructor as new ColourManager and before the Drawing the ColourID's are added.
- **selectedVertices : List<Integer>**
  The List of Vertices-ID's that the user selected the Vertices at the GUI.

+ **GraphDrawer(graph : VisualGraph, fxNode : javafx.scene.Node)**
  The Constructor of this Class.

  Sets the given Graph and fxNode. Also initializes the ColourManager.
  @param graph The Graph that should be set as the Graph of this Drawer.
  @param fxNode The JavaFx-Node that should be set as the fxNode of this Class.

+ **printGraphTextual()**
  This Method prints the textual Representation onto the given JavaFx-Node.

+ **drawGraph(layout : GraphLayoutEnum)**
  This Method draws the Graph to the given JavaFx-Node by using the given Layout to position it's Vertices.
  @param layout The Enum that indicates which Layout the Drawer should use. If it is null the Drawer will use the Circle Layout.

- **drawVertex(vertexID : Integer)**
  Draw the given Vertex.

  Get the Vertex by searching for the given VertexID at the vertices-List of the given Graph. Use the GraphicLayout to get the correct Position of this Vertex.
  @param vertexID The ID of the to drawn Vertex.

- **drawVertexSelected(vertexID : Integer)**
  Draw the given Vertex as a selected Vertex.

  This Method is called if the to drawn Vertex of the drawVertex-Method is in the selectedVertices-List.

  The Vertex is drawn as selected by adding the corresponding Picture into the Vertex-JavaFx-Shape. Then the standard draw-Method is used to do the rest.
  @param vertexID The ID of the Vertex that should be drawn as a selected Vertex.

- **drawEdge(edgePosition : Integer)**
  Draw the Edge that is on the given Position at the Edge-List of the Graph of this Drawer.

  This Method only draws one Edge so that the Editor can show specific Edges. This Method is also called multiple times to draw all Edges.
  @param edgePosition The Position of the to drawn Edge at the List of Edges of the Graph.

+ **addToSelectedVertices(vertexID : Integer)**
  Add the given VertexID to the List of selected ones.
  @param vertexID The Vertex-ID that should be added to the List of selected Vertices.

+ **clearSelectedVerticesList()**
  Clear the List of selected Vertices-ID's.

+ **getGraph() : VisualGraph**
  Get the VisualGraph of this Drawer.
  @return returns The VisualGraph of this Drawer.

+ **setGraph(graph : VisualGraph)**
  Set the VisualGraph of this Drawer.
  @param graph The VisualGraph that should be set.

+ **setFxNode(fxNode : javafx.scene.Node)**
  Set the JavaFx-Node where the Graph should be drawn on.
  @param fxNode The Node that should be set as the JavaFx-Node to draw on.

Package Drawer.ColourManager

The ColourManager-Package. This Package only contains the ColourManager which maps the abstract ColourID's that are given by the calculation into a real Colour-Value that could be drawn. This Class is separately because it provides a relatively general task, that easily can be (re)used elsewhere.

Class ColourManager

Description
The ColourManager manages the different Colours by Mapping the ColourID's to an actual Colour-Value, so that the Drawer can draw the coloured Graph by these ColourID's.

Documentation

- **colourMap : Hashmap<IntegerString>**
  The HashMap of every ColourID to the actual Colour-Value that is represented as a String.

+ **ColourManager()**
  The Empty-Constructor of this Class. The Colours are added step by step at a later point.

+ **ColourManager(colourIDs : List<Integer>)**
  The Constructor of this Class. It adds the given ColourID's of the List and puts them into the Hashmap. Then the initColours-Method is called so that the mapping is completed for the given ColourId's.
  @param colourIDs The List of colourID's that should be mapped to real Colour-Values.

+ **initColours()**
  This Method has to be called when every ColourID is put into the HashMap. Then this Method calculates a Assignment of real Colours to the ColourID's and writes them into the HashMap, where it can be read out at a later Time.

+ **addColourID(colourIDs : Integer)**
  Add a new ColourID to the HashMap, where later the real Colour is mapped to.

  It is checked if the given ColourID is already at the HashMap.
  @param colourIDs The ColourID that should be added.

+ **getColourFromID(colourID : Integer) : javafx.scene.paint.Color**
  Get the real Colour-Object from the given ColourID. This Colour is then used to draw the Vertex/Edge to the screen to represent the Colouring-Solution.

  The initColour-Method has to be called first so that the ColourManager has already mapped the Colour-Values at the HashMap.
  @param colourID The colourID from what the colour should be.
  @return returns The actual Colour of the Object.

Package Drawer.Layouts

This Package contains the implemented Layouts for the GraphDrawer and the Enum that Lists all of them.

Class GraphLayoutEnum

Description
This Enum Contains all implemented GraphLayout's that can be used by the graphDrawer to position the VisualVertices. This Enum is needed because the Drawer needs to know whitch Layout to use for the drawing of the Graph and this is done via this Enumeration. In our case there is only one Layout, because we well always draw the Graphs in a Circle. If someone wants to Expand this Drawer by adding a new Layout he/she/it has to update this Enum as well. This is not against ObjectOrienting Programming because the Programmer that would add this new Layout already needs to recompile the Program and therefore can expand the Enum as well.

Documentation

+ **circleLayout**
  The Enum for the possible Layouts.

  There will be only one Value in it because we will only use the Circle-Layout. But this is needed for possible extensions by other Programmers.

Class GraphLayout

Description
This is the Layout of the Drawing of the Graph. It is an abstract class so that there could be multiple Layouts for the Representation that implements this.

Documentation

+ **GraphLayout()**
  The Constructor of this abstract Class. This is used at the Childs if they do not have an separate

Constructor because they do not need parameters to set as well.

+ **executeLayout(graph : VisualGraph) : VisualGraph**
  This is an abstract Method and has to be implemented at the Sub-Classes.

  This Method set's the given Graph to the implemented Layout of the particular Child-Class. Therefore it sets the Positions of the Vertices of the given Graph.
  @param graph The Graph that gets the layout set on it. Therfore all Elements of this given Graph will be relocated to the calculated Position this Method calculates.
  @return returns The given Graph with the calculated Layout.

Class GraphLayoutCircle

Description
This is the Circle Layout of the Graph. Therefore this Layout orders the Graph-Nodes into a Circle.

It is an Child-Class of the abstract GraphLayout-Class.

Documentation

- **radius : Double**
  The Radius of the Circle where the Elements should be positioned at.

+ **GraphLayoutCircle(radius : Double)**
  The Constructor of this Class.

  Sets the given Radius as radius of this Layout.
  @param NAME The Radius to set.

+ **executeLayout()**
  This is the overwritten Method from the abstract-Parent-Class.

+ **setRadius(radius : Double)**
  The Setter for the Radius.
  @param radius The Radius to set.

Package Drawer.Visualization.VisualizationGraph

Class VisualVertex

Description
The Vertex of an Visual-Graph. It is the Child of the JavaFx-Circle Object so this Vertex can be drawn.

Documentation

- **ID : Integer**
  The Identification-Number (ID) of this Node. This Variable is Final.

+ **VisualVertex(id : Integer)**
  The Constructor of this Class.

  It contains only the final-ID as Parameter to set. The Parameters of the JavaFx-Node will be set by the Layout if it calculates the Position of this Vertex.

@param id The ID that will be set to this Vertex.

+ **getID() : Integer**
  Get the ID of this Vertex.
  @return returns The Integer-Value of the ID of this Vertex.

+ **toString() : String**
  This Method overwrites the standard toString-Method.
  @return returns It returns a String-Representation of this VisualVertex. «ID>"

Class VisualVertexColoured

Description
Extends the VisualVertex Class.

This Vertex also contains a Colour-ID so that the Vertex can be coloured.

Documentation

- **colourID : Integer**
  The ID of the Colour used by the Heuristic. This is like a Foreign-Key of the Colour.

  Remember: The actual colour of the specific Elements are not important because the User wants to see if the calculation of the Heuristic found a solution not what colour the Elements have. The Colour-ID can be associated with different drawing-colours for different draws without changing the statement of the Program.

+ **VisualVertexColoured(id : Integer, colourId : Integer)**
  The Constructor of this Class.

  It contains only the final-ID as Parameter to set. The Parameters of the JavaFx-Node will be set by the Layout if it calculates the Position of this Vertex.
  @param id The ID that will be set to this Vertex.
  @param coulorID The ID that will be set to this Vertex.

  If this Vertex is not coloured jet set the colour to null or use the other constructor.

+ **isColoured() : Boolean**
  Checks if this Vertex is Coloured.

  Therefore this Method checks if the ColourID is null or an actual Integer-Value.
  @return returns True if the ColourID-Varialbe is set and false if not.

+ **getColourID() : Integer**
  Get the ColourID of this Vertex.
  @return returns The Integer-Value of the ColourID of this Vertex.

+ **setColourID(colourID : Integer)**
  Set the ColourID of this Vertex.
  @param The Colour-ID this Vertex should be coloured with.

+ **toString() : String**
  This Method overwrites the standard toString-Method.

@return returns It returns a String-Representation of this VisualVertexColoured. «ID>:<ColourID>"

Class VisualEdge

Description
The Edge of an Visual-Graph. It is the Child of the JavaFx-Polygon Object so this Edge can be drawn.

Documentation

- **connectedVerticesID : List<Integer>**
  This List contains all Vertices-ID's from the Vertices this Edge connects.

+ **VisualEdge(connectedVertices : List<VisualVertex>)**
  The Constructor of this Class.

  Set's the given List of by this Edge connected Vertices to the List of this Object.
  @param connectedVertices The List of by this Edge connected Vertices. This given List will be set to the List of this Edge-Object.

+ **connectsSame(compareEdge : VisualEdge) : Boolean**
  Checks if the given VisualEdge is an edge between the Same Vertices as this Edge.
  @param compareEdge The Edge of which the connected-Vertices should be checked with.
  @return returns If the two Edges are conections between the same Vertices it returns true, else false.

+ **getConnectedVertricesIDList() : List<Integer>**
  Get the List of the connected VerticesIDs.
  @return returns The List of the Vertices-ID's that this Edge connects.

+ **toString() : String**
  This Method overwrites the standard toString-Method.
  @return returns It returns a String-Representation of this VisualEdge. "<VertexID1>, ..."

Class VisualEdgeColour

Description
Extends the VisualEdge Class. This Edge also contains a Colour-ID so that the Edge can be coloured.

Documentation

- **colourID : Integer**
  The ID of the Colour used by the Heuristic. This is like a Foreign-Key of the Colour.

  Remember: The actual colour of the specific Elements are not important because the User wants to see if the calculation of the Heuristic found a solution not what colour the Elements have. The Colour-ID can be associated with different drawing-colours for different draws without changing the statement of the Program.

+ **VisualEdgeColoured(connectedVertices : List<VisualVertex>, colourId : Integer)**
  The Constructor of this Class.

  Set's the given List of by this Edge connected Vertices to the List of this Object.
  @param connectedVertices The List of by this Edge connected Vertices. This given List will be set to the List of this Edge-Object.
  @param coulorID The ColourID that will be set to this Edge.

If this Edge is not coloured jet set the colour to null or use the other constructor.

+ **isColoured() : Boolean**
Checks if this Vertex is Coloured.

Therefore this Method checks if the ColourID is null or an actual Integer-Value.
@return returns True if the ColourID-Varialbe is set and false if not.

+ **getColourID() : Integer**
Get the ColourID of this Edge.
@return returns The Integer-Value of the ColourID of this Edge.

+ **setColourID(colourID : Integer)**
Set the ColourID of this Edge.
@param The Colour-ID this Edge should be coloured with.

+ **toString() : String**
This Method overwrites the standard toString-Method.
@return returns It returns a String-Representation of this VisualEdge-Coloured. "<VertexID1>, ...:<ColourID>"

Class VisualGraph

Description
This is the VisualGraph. It is the Graph-Construct that is used for the Drawing.

Remember: VisualGraph is Generic VisualGaph<V extends VisualVertex, E extends VisualEdge> This is necessary so that the Graph can differentiate between the uncoloured and the coloured Elements.

This separate Graph-Representation for the View is necessary because the Model and the View of the Rage-Program should be strictly separated and therefore the View could not use the same Graph-Object. As well this Graph-Representation uses special Nodes and Edges as Elements that could be drawn.

Documentation

- **vertices : List<VisualVertex>**
This is a List of all Vertices (=Node's) of this Graph.

Remember: At any further Point the Nodes"will be named Vertex/Vertices because of the confusion with JavaFx-Nodes that would otherwise occur.

- **edges : List<VisualEdge>** This is a List of all Edge's of this Graph.

+ **VisualGraph()**
The Empty-Constructor of this Class.

+ **isColoured()**
Checks if the Graph is made out of VisualVertexColoured and VisualEdgeColoured and if so if the ColouredID's of all Objects are set.
@return returns If they are set it returns true, and if not false.

+ **addVertex()**
Add a new Vertex to the List of Vertices of this Graph.

If the List is not instanciated yet this will be done.

To add a new Vertex this Method searches for the next unused Integer-ID that could be used for a new Node and created the VisualVertex-Object with this Parameter. This created Object will be added to the List.

+ **addVertex(vertex : VisualVertex) : Boolean**
Add the given Vertex to the List of Vertices.

If the List is not instanciated yet this will be done.

Also it is checked that the Vertex-ID is not already used by another Vertex. If so the given Vertex will not be added.
@param vertex The Vertex that should be added to this Graph.
@return returns If the Vertex-ID was added this Method returns true, otherwise false.

+ **addVertex(vertices : List<VisualVertex>)**
Add a whole List of Vertices to this Graph.

This is done by calling the addVertex-Method multiple times.
@param vertices The List of Vertices that should be added to the List.

+ **addVertex(amount : Integer)**
Add the given amount of Vertices to the Graph.

This is done by calling the addVertex-Method multiple times.
@param amount The amount of Vertices the user wants to add to this Graph.

+ **addEdge(edges : VisualEdges)**
Add the given Edge to the Graph.

If the List is not instanciated yet this will be done.

Also it is checked if this Edge has the exact same connected Vertices as any other Edge of this Graph. This is done by calling the connectSame-Method of the given Edge.

Also it is checked that the given Edge is valid. That means that this method checks if all connected-Vertices that are given by ID are Vertices of this Graph. If there is an unexisting Vertex this Vertex will be created and added to the Graph by calling the addVertex(VisualVertex)-Method.
@param edge The Edge that should be added to this Graph.

Check if this Edge contains valid VertexID's and if it only connects Vertices that are not currently connected.

+ **addEdge(vertices : List<VisualEdges>)**
Add a whole List of Edges to this Graph.

This is done by calling the addEdge-Method multiple times.
@param edges A List of Edges that should be added.

+ **addEdge(verticeIDs : List<Integer>)**
Add the Edge, that is given by the List of Vertice-ID's, to the graph.

This is done by creating an new VisualEdge-Object with the given List as Parameter and then calling the addEdge-Method.
@param verticeIDs The List of Vertice-ID's that should be connected by Edge that should be added.

+ **removeVertex(vertex : VisualVertex)**
Remove the given Vertex from the Graph.

If an Edge was connected to this Vertex and it only contains one other Vertex after the deletion, the Edge will be removed too.
@param vertex The Vertex that should be removed.

+ **removeVertex(vertexID : Integer)**
Remove the Vertex, by the given ID, from the Graph.

This is done by calling the removeVertex-Method. (The Vertex that should be deleted can be found at the Vertices-List by the given ID).
@param vertexID The Vertex-ID from the Vertex that should be removed from the Graph.

+ **removeEdge(edge : VisualEdge)**
Remove the given Edge from the Graph.
@param edge The Edge of the VisualGraph that should be removed.

+ **removeEdge(verticesIDs : List<Integer>)**
Remove the Edge between the given Vertrice.
@param verticesIDs The List of the Vertices-ID's that the Edge is between, that should be removed.

+ **duplicateVertex(vertexID : Integer)**
Duplicate the given Vertex so that a new Vertex is at the Graph with exactly the same neighbourhood.
@param vertexID The Vertex-ID of the Vertex that should be duplicated.

+ **contractVertices(verticesIDs : Integer)**
Contract the given Vertices to one Vertex.

Multiple Edges between the same destinations will be removed, so that only one of these Edges is in the Graph. Edge-Loops will be removed.
@param verticesIDs The List of the given VertricesID's.

+ **setVertexOrder(vertexID : Integer, order : Integer)**
Set the Vertex to the given Order.

The Vertex that was at this Position of the List earlier will be put behind the set Vertex.
@param vertexID The ID of the Vertex that should be moved to a different Order.
@param vertexID The order the Vertex should be set to.

Package Sound

The Sound-Package contains everything that has to do with the Sounds. It separates the SoundHandler from the other parts.

Class SoundHandler

Description
The Sound Handler that manages the different Sounds the Program can make. Including the Error and finish Sound.

Documentation

- **soundList : List<String>**
  The List of all paths to the Audio-Files.

- **player : javafx.scene.media.MediaPlayer**
  The MediaPlayer that plays the given Music.

+ **SoundHandler()**
  The Constructor of this Class. Has no parameters so it only sets the List to an Empty List so that the User can add File-paths to the playable Sounds later.

+ **SoundHandler(sounds : List<String>)**
  The Constructor of this Class. The List of Strings should contain path to the Sound-Files the Player should play. The given List will be set at the soundList of this Class.
  @param sounds The Path-List that the SoundHandler should use as soundList.

+ **addSound(filepath : String)**
  Add a new Sound-Filepath to the soundList.
  @param filepath The Filepath that should be added.

+ **playSound()**
  Starts the MediaPlayer with a random Sound of the given List.

  Therefore it calls the playSound(listPosition)-Method with an randomly choosen Value.

+ **playSound(listPosition : Integer)**
  Starts the MediaPlayer with the Sound at the given position of the soundList of this Class.

  Therefore it checs the given position if it is valid. Then it loads the File from the path that is stored at the soundList at the given Position. If the File could not be loaded the Method stops. Else the loaded File will be passed on to the MediaPlayer of this class. The MediaPlayer will be started, so that the Sound is played.
  @param listPosition The position of the Sound at the soundList that should be played.

+ **stopSound()**
  Stops the playing of the MediaPlayer.

5 Controller

Der Controller des verwendeten MVC-Musters lässt sich wie in Abschnitt 2.1 beschrieben in zwei verschiedene Komponenten, den `FXController` und den `LogicController` aufteilen.
Der `FXController` verarbeitet die Benutzereingaben und steuert die Darstellung auf der View. Der `LogicController` ist für die Kommunikation zwischen `FXController` und dem Model zuständig. Der `FXController` ist Teil des JavaFX-Programms und ist direkt mit der FXML-Datei der Benutzeroberfläche verknüpft.

5.1 LogicController

Package Controller

Manages interactions with the user and asks the model to execute tasks.

Class SuperController

Description

The SuperController has one or more instances of theGrapgGeneratorController, List of TabController, GraphEditorController.
Documentation

+ **SuperController**
  Constructor: Creates a SuperController and gives him immedeately a GraphGeneratorController instance.
  @param GGC The Param GGC is the instance of a GraphGeneratorController.

+ **getGGC**
  @return GraphGeneratorController.

  createGEC
  Creates a new GraphEditorController with(out) a graph to display.
  @param pool The DataPool, where the created graph from the user will be added.
  @param graphl The Graphl, that should be modified.

+ **getTabList**
  @return List <TabController>.

+ **getTabController**
  @param name name is the PreviewTab identifier, with it, the SuperController can identify the current TabController, the User is working on.
  @return TabController.

+ **getGEC**
  @return GraphEditorController.
  createTabController
  Creates a new preview tab, with a graph liast and a heuristics list.
  @param graphList List of graphs that should be taken to the new tab.
  @param heurList List of heuristics that should be taken to the new tab.
  @return TabController.

createTabController
Creates a new preview tab with its own DataPool, and calls the GrapgGeneratorController to generate graphs for the DataPool and it will show the graphs in the preview Tab.
@param GgenPropertiest The properties, that dictates how the random graph generation generates graphsö.
@return TabController.

Class StatisticController

Description

Reads the statistics for a heuristic out of the Model and collects them to show it to the View.

Documentation

+ **StatisticController**
  Constructor: Creates a StatisticController and gets himself a DataPool. **@param pool** The DataPool, that the StatisticController belongs to.

+ **getAllStatistics**
  @return List <Statistic >.

+ **getStatistic**
  @param heur heur is the Name of the Heuristic, that you want the statistics from.
  @return Statistic.

Class TabController

Description

The Controller of exactly one Preview Tab in the View, that manages the DataPool of this Preview Tab.

Documentation

+ **TabController**
  Constructor: Creates a new TabController and connects it with an own DataPool, it also creates an own StatisticController.
  @param tabnamel The name of this TabController.
  @param pool The DataPool, that belongs to the TabController.

+ **getDVCList**
  @return List <DetailViewController>.

+ **getDVC**
  @param name The name is the DetailViewController identifier, with it, theTabController can identify the current DetailViewController, the User is working on.
  @return DetailViewController.

+ **getDataPool**
  @return DataPool

+ **addGraphToDataPool**
Adds one Graph to the DataPool, that belongs to the TabController instance.
@param graph The graph that should be added to the DataPool.
@throws EXCEPTION if the type of the Graph is not of the same graph type in the DataPool.

+ **mergeDataPool**
Merges two DataPools under one of the two TabController. The other TabController with its DataPool remains untouched.
@param pool The DataPool, that should be copied.
@throws EXCEPTION if the graph type of both DataPools is not equal.

+ **getStatisticController**
@return StatisticController

+ **getHeuristicController**
@return HeuristicController

+ **getFilterController**
@return FilterController

+ **heuristicApplyToDataPool**
Calls the HeuristicController to collor the graphs.

+ **createHeuristicController**
Instanciates a HeuristicController and gives him a DataPool. **@param pool** The given DataPool.

+ **createDetailViewController**
Instanciates a DetailViewController and gives him a graph to display with all heuristics, that tried to collor it. **@param graphPositionl** The position of the graph in the graph list in the given DataPool.

+ **createFilterController**
Instanciates a FilterController and gives him a DataPool. **@param pool** The given DataPool.

Class GraphGeneratorController

Description

The controller for the graph generation communication between the view and the GraphBuilder in the Model.

Documentation

+ **GraphGeneratorController**
Constructor: Creates a GraphGeneratorController.

+ **generate**
Commands the GraphBuilder to create random graphs with specific properties.
@param genProperties The properties, that restrict the randomnes of the GraphBuilder.

+ **createManuallyGraph**
Creates an empty GraphEditorController, that adds that manually generated graph from a user.

It calls the SuperController to start the method createGEC without a DataPool and without a graph.

Class GraphEditorController

Description

Manages the manipulated or created graph by the user and adds it to the right DataPool.

Documentation

+ **GraphEditorController**
  Constructor: Creates a GraphEditorController and it will get a DataPool instance. When it was created by the GraphGeneratorController, it will create an GraphEditorController without a graph.
  If it was created by the DetailViesController, it will get a graph to the new instance.

+ **setGraph**
  Sets the graph of this instance.
  @param g The graph, that belongs to this instance.

+ **addGraph**
  Adapts the created visualGraph to a Graph and adds the created graph to the DataPool. If there is no DataPool, it will create a new one.
  @param vGraph The created visualGraph.

+ **getVisualGraph**
  Returns the Graph of this instance as a visualGraph.

Class FilterController

Description

Controlls the filter set by the user and manages the filtered graph pool and the graph pool in the DataPool.

Documentation

+ **createFilterController**
  Constructor: Creates a FilterrController and it will get a DataPool instance.

+ **filter**
  Filters the graph list from the DataPool.
  @param List <Heuristic, value: int> It determines how the list will be filtered.
  @param sort It determines how list will be sorted (decending, ascending ...).
  @return returns List <VisualGraph>, the DataPool remains untouched.
  @throws EXCEPTION if filter and sort are contradictory.

+ **getAllHeuristics**
  Returns all properties of the used heuristics and the heuristic name is also a heuristic property.
  @return returns List <HeuristicProperties>

Class DetailViewController

Description

Manages the chosen graph and the heuristics that only apply to this graph, the so called local heuristics.

Documentation

+ **startCalculation**
  Applies the local Heuristics to the one graph in the DetailView.
  @param graph The graph that get colloerd by the local heuristics.
  @param localHeuristicList The list of local heuristics, that collors the one graph.

+ **modifyGraph**
  Creates a GraphEditorController instance with the one graph.
  @param graph The graph that should be modified.

+ **loadModifiedGraph**
  Loads the modified graph into the DataPool and in to the DetailViewController.
  @param modgraph The modified graph.

+ **addLocalHeuristics**
  Adds the local heuristics chosen by the user to the localHeuristics.
  @param List <Heuristic> The local Heuristics.

+ **DetailViewController**
  Creates a new Detail View Controller and creates an empty localHeuristicsList.
  @param graph The graph, thatshouldbe loaded in to the DetailViewController.

Class HeuristicController

Description

Manages the chosen graph and the heuristics that only apply to this graph, the so called local heuristics.

Documentation

+ **HeuristicController**
  Creates a HeuristicController and gives him a DataPool.
  @param pool The DataPool to give.

+ **addToHeuristics**
  Applies the chosen heuristics to the heuristic pool in the DataPool. Calls the createHeuristics method.
  @param hName The name of the heuristic.
  @param hProp The properties of the heuristic.
  @param pool The DataPool, where the heuristics belong.

+ **startCalculation**
  Commands the Heuristics to calculate their results on the graph pool.
  @param pool The graph list.

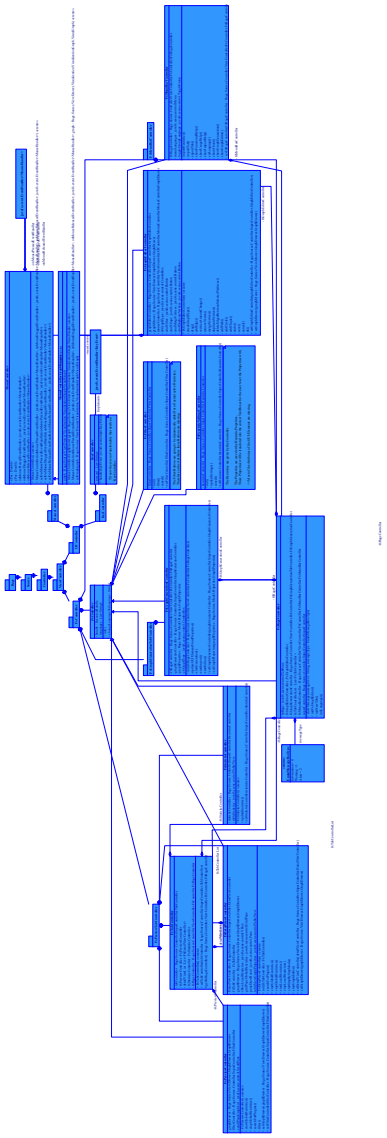
@param hpool The heuristicList of the DataPool, that should calculate the result on the graph list.

\+ **getAllHeuristics**
Returns all possible Heuristics.
@return returns List <HeuristicsProperties>.

\# **createHeuristics**
Get invoked by addToHeuristics and commands the model to create a new heuristic.
@param hName The heuristic name.
@param hProp The heuristic properties.

5.2 View-Controller

5.2.1 Entwurf



Package ViewController

This Package contains the View Controller. It contains the FxController and the User-Input-Controller for the different GUIs. It works like a Fassade (Interface-like) between the actual Controller of the Program and the GUI.

Package ViewController.UIController

The UI-Controller-Package contains the Input-Controller/Handlers. They catches the Mouse or Key Events and managing the things to perform the correct actions.

Package ViewController.UIController.MouseController

The MouseController is the Part where the different Mouse-Events are catched, so that the Program can react to them.

Class MouseController

Description The MouseController so that the Program could detect Mouse Clicks, Drag and Drop.

Documentation

- **xPos : Double**
  The X-Position of the Mouse at the time of the Event.
- **yPos : Double**
  The Y-Position of the Mouse at the time of the Event.
- **onMousePressedEventHandler : javafx.event.EventHandler<MouseHandler>**
  This is the MouseHandler for the onPressed-Action. This is an Implementation of the EventHandler<MouseHandler>, that overwrites the handle-Method.
- **onMouseDraggedEventHandler : javafx.event.EventHandler<MouseHandler>**
  This is the MouseHandler for the onDragged-Action. This is an Implementation of the EventHandler<MouseHandler>, that overwrites the handle-Method.
- **onMouseReleasedEventHandler : javafx.event.EventHandler<MouseHandler>**
  This is the MouseHandler for the onRelease-Action. This is an Implementation of the EventHandler<MouseHandler>, that overwrites the handle-Method.

+ **MouseController()**
  The Empty-Constructor of this Class. The MouseHandler of this Class had to be set at a later time.

+ **MouseController(onMousePressedEventHandler : javafx.event.EventHandler<MouseHandler>, onMouseDraggedEventHandler : javafx.event.EventHandler<MouseHandler>, onMouseReleasedEventHandler : javafx.event.EventHandler<MouseHandler>)**
  The Constructor of this Class. It gets the differentEventHandlers as Parameters and sets them.
  @param onMousePressedEventHandler The MouseHandler for the onPressed-Action that should be set.
  @param onMouseDraggedEventHandler The MouseHandler for the onDragged-Action that should be set.
  @param onMouseReleasedEventHandler The MouseHandler for the onReleased-Action that should be set.

+ **setOnMousePressedEventHandler(onMousePressedEventHandler : javafx.event.EventHandler<MouseH**
Set's the onMousePressedEventHandler of the Class.
@param onMousePressedEventHandler The EventHandler for the onMousePressed-Event to set.

+ **setOnMouseDraggedEventHandler(onMouseDraggedEventHandler : javafx.event.EventHandler<Mouse**
Set's the onMouseDraggedEventHandler of the Class.
@param onMouseDraggedEventHandler The EventHandler for the onMouseDragged-Event to set.

+ **setOnMouseReleasedEventHandler(onMouseReleasedEventHandler : javafx.event.EventHandler<Mouse**
Set's the onMouseReleasedEventHandler of the Class.
@param onMouseReleasedEventHandler The EventHandler for the onMouseReleased-Event to set.

Class MouseControllerGraphDrawer

Description

This is a Child-Class of the MouseController. It is used for the Mouse-Controlling at the Graph-Drawer, to notify the modifying.

For Example: It registers when a Vertex is pressed so that it can be added to the selected Vertices. Or if the Vertex-Order should be changed so the User draggs the Vertex to the new Position.

Documentation

- **graph : VisualGraph**
The Graph the User wants to modify by using the Mouse.

+ **MouseControllerGraphDrawer(graph : VisualGraph)**
This is the Constructor of this Class. It uses the Empty-SuperConstructor. It get's an Graph as a Parameter and sets them.
@param graph The VisualGraph that should be set to modified with the Mouse.

+ **MouseControllerGraphDrawer(onMousePressedEventHandler : javafx.event.EventHandler<MouseHand onMouseDraggedEventHandler : javafx.event.EventHandler<MouseHandler>, onMouse-ReleasedEventHandler : javafx.event.EventHandler<MouseHandler>, graph : VisualGraph)**
The Constructor of this Class. It gets the different EventHandlers as Parameters and sets them. Also it sets the given Graph.
@param onMousePressedEventHandler The MouseHandler for the onPressed-Action that should be set.
@param onMouseDraggedEventHandler The MouseHandler for the onDragged-Action that should be set.
@param onMouseReleasedEventHandler The MouseHandler for the onReleased-Action that should be set.
@param graph The VisualGraph that should be set to modified with the Mouse.

+ **setGraph(graph : VisualGraph)**
Set the VisualGraph of this MouseControllerGraphDrawer.
@param graph The VisualGraph that should be set.

Package ViewController.UIController.KeyController

The KeyController is the Part where the different Keyboard-Events are catched, so that the Program can react to them.

Class KeyController

Description

The Abstract KeyController that should be implemented at it's childs.

Because for our purpose it is not important if the KeyEvent was a KeyPress or a KeyRelease this Class does not contain other Handler that Handle these different Events. (unlike at the MouseController) Instead this Class is the only KeyEventHandler.

Documentation

- **pressed : List<String>**
  The List of pressed Keys. This is used if multiple Keys are pressed. Then the Program puffers them in here.

  This List will be managed by this Class completely and does not need to be seen by outer Classes. Therefore there is not getter/setter-Method.

+ **handle(keyEvent : KeyEvent)**
  This is the actual Handler of the Key-Event wich is overwritten at the Child's to react to the different Events.
  @param keyEvent The actual Event that was triggered by the KeyBoard

Package ViewController.FxController

This Package Fontains all the FxController for the Program. This is the Interface Between the actual FXML-Scene GUI and the Controller where the work will be passed on to the Model-Part.

Class FxController

Description

This is the FxController. It is the abstract Class from which every other FxController of this Program gets passed on.

Documentation

- **bundle : ResourceBundle**
  This is the Resource-Bundle of the Object. This is a kind of Key-Value storage-Unit.

  It contains the Text, ToolTip, ... of a specific Language for every GUI-Element that is identified by its fx-id.

  The ResourceBundle will be changed to the correct Language if needed.

- **languages : list<String>**
  A List of all possible Languages. This list will be dynamically filled with the correct String-ID's of the implemented Languages.

  This is needed for every controller-Class because of the ResourceBundle switching for different Languages. Every time the GUI-Language is switched a new ReourceBundle has to be loaded. Therefore

the Program loads a new File from the Disk where the Language-ID will be the Ending-String of the Filename by which the program identifies the correct ResourceBundle for the wished Language.

+ **init()**
Initializes the whole Controller by calling all necessary Methods.

Remember: The Constructor is called first. Then all Fields annotated with @FXML will be loaded. After that the init-Method will run. That means that the Constructor does not have access to the FXML-GUI-Elements but the init-method does.

This is an abstract Method so this will be Overwriten by the Childs. Then the Child can also overload this Method with different Parameters to give, and so on.

+ **changeLanguageTo(language : String)**
Set the currently active Language of the GUI of this Controller to the given Language. Sets the correct ResourceBundle.
@param language The Language that should be set.

Da es zu jedem FXML-Dokument und somit zu jeder GUI einen eingenen FxController gibt, benötigt man eine zentrale Einheit für das Programm, das als Manager für die einzelnen FxController fungiert. Diese Aufgabe übernimmt die Folgende Klasse FxRageController".

Class FxRageController

Description

This is the FxController for the general Program-Window. It functions as the Manager of the calls of the other FxController. It is the only FxController that communicates with any SuperController and passes on the needed Values to the other FxController.

Documentation

- **tpRage : javafx.scene.control.TabPane**
The Main-TabPane of the Program.
- **fxGraphEditorController : FxGraphEditorController**
The FxGraphEditorController of this Class. There should be only one active Editor at a time.
- **fxGraphGenerationController : FxGraphGenerationController**
The GraphGenerationController of this Class. It is no list because there can only be one Generation-Tab per Rage-Program.
- **fxTabControllerList : List<FxTabController>**
The List of all FxTabController that are needed. Remember: Every Super-Preview-Tab needs it's own FxTabController. Therefore there are multiple FxTabController possible.
- **fxMenuBarController : FxMenuBarController**
The FxController for the MenuBar. This Attribute will be set at this Main-Controller and passed on to the other Tab-Controller where the MenuBar is needed as well.
- **superController : SuperController**
This is the SuperController of the Program. It contains all other Controller. These other Controllers will be read out by this Class (the FxRageController) and then passed on to the Sup-FxController so that the whole Program can operate like it should.

+ **showMessage(messageText : String, messageType : EnumMessageBoxType)**
Show a MessageBox at the Screen. This Method is Used at the ExceptionHandler to show the Error Log.

@param messageText The text that should be shown at the MessageBox.
@param messageType The Type of Message you want to Show. (Information, Warning, Error, ...)

+ **initFxGraphEditor()**
  This Method starts the GraphEditor by calling the FXMLLoader and passing the correct FxGraphEditorController.

+ **initNewTab()**
  This Method adds a new FxTabController to the List and by calling the FXMLLoader and then passing it to the FXML. This Tab will be added to the GUI.

  Also it calls the SuperController so that it can a new Tab for its own.

+ **initLanguages()**
  Is called at the init-Method.

  Calls the IOHandler to search for ResourceBundles. Adds all found Languages to the List so that the user can select them.

Nach dieser Betrachtung der oberen Struktur des View-Controllers benötigt man natürlich noch die einzelnen dort aufgeführten FxController. Diese werden im Anschluss beschrieben.

Class FxFilterController

Description

This is the FxController for the Filter-Window. This is a separate Window that will pop up if the User wants to Filter the shown Graphs at the Preview-Tab.

Documentation

- **filterController : FilterController**
  The FilterController for the passing of the User-Filter to the Model.
- **init()**
  Create the Needed GUI-Components. Therefore the FilterController is called to get the List of all Heuristics. Then for every Heuristic the ToggleButtonGroup to let the User select if the Heuristic should be correctly or not or if it does not matter will be added.

- **filter()**
  Parse the selected Filter into the correct form and give them to the FilterController by calling the corresponding Method. Everything further will be done by the FilterController.

- **cancel()**
  Cancel the Filtration and close the Filter-Window.

+ **setFilterController(filterController : FilterController)**
  Set the FilterController of this Class.
  @param filterController The FilterController to set the attribute of this Class to.

Class FxHeuristicSettingsController

Description

The FxController for the HeurisitcSetting-Screen. This is a separate Window that will pop up if the User wants to start a Heuristic that needs some Settings. The GUI-Elements are dynamically added using the HeuristicController to get the HeuristicProperties to read the possible settings the user can/have to take.

Documentation

- **heuristicController : HeuristicController**
  This is the HeuristicController of this Class. It will be passed on from the FxTabController via the TabController that has the HeuristicController instance.

- **init()**
  Create the Needed GUI-Components. The Elements are created from the HeuristicProperties-List that it gets from the HeuristicController of this Class. These Elements are used so that the User can set his/her Properties of the Heuristic. These Values will be passed on the the HeuristicController so that the DataPool can be updated and the Model calculates the correct stuff.

- **updateSettings()**
  This Method calls the HeuristicController and calls the addToHeuristicsMethod with the set Properties, that are given by the dynamically added TextBoxes.

- **cancel()**
  Cancel the HeuristicSetting and close the Settings-Window.

+ **setHeuristicController(heuristicController : HeuristicController)**
  Set the HeuristiCController to the given.
  @param heuristicController The HeuristicController to set.

Class FxGraphEditorController

Description

The FxController of the GraphEditor Window.

Documentation

- **graphController : GraphController**
  This is the GraphEditorController of this Class.

- **graphDrawer : GraphDrawer**
  The GraphDrawer of this Editor. This contains the Graph that will be modified and drawn.

- **mouseController : MouseControllerGraphDrawer**
  This is the MouseController for this Editor. It will be created dynamically whenever the graph is changed and it was not jet created, so that an simple update of the graph via the Setter would be enough. So there is no need for a Setter-Method.

  The Graph of the Drawer of this Class will be passed on as the Graph of this MouseController. Then the MouseEvent's can be managed as needed and the Vertices can be dragged.

  The MouseHandler does not need to be accessed by any other Object so there is no need of Getter Methods.

- **cbGraphType : javafx.scene.control.ComboBox**
  This is the ComboBox where the GraphType will be set.

  The Value of this Box will be read-out and then the function of the Editor will be adapted to the Graph-Type.

  If the Editor is opened for an already existing Graph the ComboBox will be set to the type of the given Graph. If the Editor is opened for creating a new Graph the User has to set this ComboBox to the wished Type.

- **cmdCirlce : javafx.scene.control.Button**
  The Button for the AddCircle-Operation. This Buttons is needed because this Function has to be hidden if the Editor works on an VisualGraph of an SimpleHyperGraph.

- **cmdCirlce : javafx.scene.control.Button**
  The Button for the AddCircle-Operation. This Buttons is needed because this Function has to be hidden if the Editor works on an VisualGraph of an SimpleHyperGraph.

- **cmdClique : javafx.scene.control.Button**
  The Button for the addClique-Operation. This Buttons is needed because this Function has to be hidden if the Editor works on an VisualGraph of an SimpleHyperGraph.

- **cmdMergeVertices : javafx.scene.control.Button**
  The Button for the Merge-Vertices-Operation. This Buttons is needed because this Function has to be hidden if the Editor works on an VisualGraph of an SimpleHyperGraph.

- **cmdDuplicateVertices : javafx.scene.control.Button**
  The Button for the Duplicate-Vertices-Operation. This Buttons is needed because this Function has to be hidden if the Editor works on an VisualGraph of an SimpleHyperGraph.

+ **FxGraphEditorController()**
  This is the Empty-Constructor of this Class.

- **checkGraphType()**
  This Method is called at the init-Method and whenever a new GraphDrawer is set.

  Checks the Properties of the currently edited Graph and checks the Type.

  Then the corresponding ComboBox for the Graph-Type will be set to the correct Value.

  If the Editor works on an SimpleHyperGraph some Buttons will be hidden.

- **drag()**
  This Method is accessed if the cmdDrag-Button is pressed.

  Therefore the user wants to relocate an selected Node. Then this Method calls the corresponding Method at the GraphDrawer.

- **addEdge()**
  This Method is accessed if the cmdAddEdge-Button is pressed.

  Therefore the user wants to add an Edge between the selected Vertices. It checks the graph Type of the given Graph and if the Edge is between only two Vertices for an SimpleGraph. Then this Method calls the corresponding Method at the GraphDrawer.

- **vertriceAmountChange()**
  This Method is accessed if the VerticesAmount-Spinner changes it's Value.

If this happens this can mean one of two different things the User wants to Delete or Add Vertices from the Graph. Therefore check if the Amount was changed to a smaller or bigger Value and then call the corresponding Method at the Graph from the GraphDrawer. In case the User wants to delete Vertices this is only possible while there are unconnected Nodes at the End of the List. If there are none left, the User has to delete the Vertices manually.

- **removeVertex()**
  This Method is accessed if the cmdEraser-Button is pressed.

  Therefore the user wants to delete the selected Vertex. Then this Method calls the corresponding Method at the GraphDrawer.

- **mergeVertices()**
  This Method is accessed if the cmdMergeVertices-Button is pressed.

  Therefore the user wants to merge the selected Vertices. Then this Method calls the corresponding Method at the GraphDrawer.

- **duplicateVertices()**
  This Method is accessed if the cmdDuplicateVertices-Button is pressed.

  Therefore the user wants to duplicate the selected Vertices. Then this Method calls the corresponding Method at the GraphDrawer.

- **deleteEdgesBetweenSelectedVertrices()**
  This Method is accessed if the cmdDeleteEdgesBetweenSelectedVertrices-Button is pressed.

  Therefore the user wants to delete the Edge between the selected Vertices. Then this Method calls the corresponding Method at the GraphDrawer.

- **addPath()**
  This Method is accessed if the cmdAddPath-Button is pressed.

  Therefore the user wants to add an Path between the selected Vertices. Then this Method calls the corresponding Method at the GraphDrawer.

- **addCircle()**
  This Method is accessed if the cmdAddCircle-Button is pressed.

  Therefore the user wants to add an Circle between the selected Vertices. Then this Method calls the corresponding Method at the GraphDrawer.

- **addClique()**
  This Method is accessed if the cmdAddClique-Button is pressed.

  Therefore the user wants to add an Clique. Then this Method calls the corresponding Method at the GraphDrawer.

- **undo()**
  This Method is accessed if the cmdUndo-Button is pressed.

  Therefore the user wants to revert the last done Change. Then this Method calls the corresponding Method at the GraphDrawer.

- **save()**
  This Method is accessed if the cmdSave-Button is pressed.

  Therefore the user wants to save the modified Graph. Then this Method calls the corresponding Method so that the VisualGraph of the Drawer can be converted to an Graph of the Model-Structure by the Adapter Class and be passed on to the Model by the general Controller.

- **cancel()**
  This Method is accessed if the cmdCancel-Button is pressed.

  Therefore the user wants to cancel the Modification of the Graph. So the Editor closes and the Modifications are deleted.

- **ok()**
  This Method is accessed if the cmdOk-Button is pressed.

  Therefore the user wants to save the modified Graph and go on.

+ **setGraphEditorController(graphEditorController : GraphEditorController)**
  Set the GraphEditorController.
  @param graphEditorController The GraphEditorController so set.

+ **setGraphDrawer(graphDrawer : GraphDrawer)**
  Set the GraphDrawer of this Class. Also calls the checkGraphType-Method.
  @param graphDrawer The GraphDrawer to set.

Package ViewController.FxController.FxPreviewTabController

This Package orders the FxController a little bit so that the FxController for the Preview Tab are inside this Package.

Class FxTabController

Description

This is the FxControler for the upper Preview-Tab. It manages the whole thing including the Sub Tabs like: Preview, DetailView and Statistics

Documentation

- **tabController : TabController**
  This is the TabController which Contains the DataPool of this Tab as well as the other sub-Controller like Statistic, Heuristic and DetailView.
- **fxPreviewController : FxPreviewController**
  The FxController for this Preview-Tab.
- **detailViewList : List<FxDetailViewController>**
  A List of all FxDetailViews that are currently needed because these DetailViews are opened.
- **fxStatisticController : FxStatisticController**
  The FxStatisticController for this whole Preview-Tab.

- **fxRageController : FxRageController**
  This is the Upper-Controller. It is needed in this Class so that this Controller can access the needed Super-Controller parts. (Because the FxRageController is the only direct connection between the ViewController and the SuperController)

+ **FxTabController()**
  The Empty-Constructor of this Class.

+ **setTabController(tabController : TabController)**
  This Method sets the TabController of this Class. It is called from FxRageController that gets the needed TabController from the SuperController.
  @param tabController The given TabController to set the Attribute to.

getFxRageController() : FxRageController
  Get the FxRageController. This Method is protected. This is used so that the startEditor-Method can do what it should and starts the Editor.
  @return returns The FxRageController of this Class.

Class FxPreviewController

Description

This is the FxController for the lower Preview-Tab, where all Graphs are shown as a List of Tabs.

Documentation

- **graphDrawer : GraphDrawer**
  The GraphDrawer of this Tab. The Graph of this Tab will be passed on to the Drawer whenever the User opens the DropDown-Box of an Graph, so it can be drawn.

- **filterController : FilterController**
  The FilterController for the passing of the Filter, selected by the User at the GUI via the FxFilterController, to the Model.

- **scrollPaneGraphList : javafx.scene.control.ScrollPane**
  The ScrollPane where all the DropDowns of the Graphs are in.

+ **FxPreviewController()**
  The Empty-Constructor of this Class. It creates a new GraphDrawer, where the needed Graph will be put in if the User opens the Tab of a Graph.

+ **startGlobalHeuristics()**
  This Method will start the calculation of the GlobalHeuristics. This will be done by calling the given TabController with the Method "heuristicApplyToDatapool()".

+ **stopGlobalHeuristics()**
  This Method will start the calculation of the GlobalHeuristics. This will be done by calling the given DetailViewController.

+ **searchForPlugins()**
  Try to add new Plugins to the List. This Method opens an FileChooser which will return a Folder that will be passed on to the IOController to perform the PlugIn-Loading.

+ **filter()**
This Method will show the Filter-Window, so that the User can select the Filters. This is done by Loading the Filter-FXML-File which creates the new Window. This already creates the needed FxFilterController which is read out, so that the FilterController at this Class can be set. Then the User-Input will be registers by this passes FxFilterController and will be passed on the given FilterController via the FxFilterController directly.

+ **setGraphDrawer(graphDrawer : graphDrawer)**
Set the Graph Drawer of this Class.
@param graphDrawer Set the Graph Drawer of this Class.

+ **setFilterController(filterController : FxFilterController)**
Set the FilterController of this Class.
@param graphDrawer The FilterController to set the attribute of this Class to.

Class FxDetailViewController

Description

The FxController for the DetailView.

Documentation

- **detailViewController : DetailViewController**
The DetailVewController of this Class. It Contains the Lists of all Local and Global Heuristics and the Graph of this DetailView.
- **fxTabController : FxTabController**
The FxTabController that contains this DetailView in the List of DetailViews. This is passed on by the FxTabController itself.

+ **FxDetailViewController()**
The Empty-Constructor of this Class.

+ **init(fxTabController : FxTabController)**
This method overrides the init-Method from the abstract-Class FxController. It sets the FxRageController by the given Parameter. This is possible because the FxRageController calls this Method and gives itselv as parameter with it.
@param fxTabController The FxTabController that will be set.

+ **searchForPlugins()**
Try to add new Plugins to the List. This Method opens an FileCooser which will return a Folder that will be passed on to the IOController to perform the PlugIn-Loading.

+ **startGlobalHeuristics()**
This Method will start the calculation of the GlobalHeuristics. This will be done by calling the given DetailViewController.

+ **stopGlobalHeuristics()**
This Method will stop the calculation of the GlobalHeuristics. This will be done by calling the given DetailViewController.

+ **startLocalHeuristics()**
This Method will start the calculation of the Local Heuristics for the Graph of this Detail View. This will be done by calling the given DetailViewController.

+ **stopLocalHeuristics()**
This Method will stop the calculation of the Local Heuristics for the Graph of this Detail View. This will be done by calling the given DetailViewController.

+ **stertStepByStepMode()**
This Method will start the Step-By-Step-Mode of the Heuristic. This will be done by calling the given SuperDetailViewController.

+ **startGraphEditor()**
Beschreibung

+ **setDetailViewController(detailViewController : DetailViewController)**
Set the DetailViewController of this Class.
@param detailViewController The DetailViewController to set.

+ **setGraphDrawer(graphDrawer : GraphDrawer)**
Set the Graph Drawer of this Class.
@param graphDrawer The GraphDrawer to set.

Class FxStatisticController

Description

The FxConntroller for the Statistic-Tab.

Only fills in the Table with the Collected Data from the DataPool provided from the Model via the given HeristicController.

Documentation

- **statisticController : StatisticController**
This is the Controller that manages the DataPool which contains all the Statistics.
- **tableStatistics : javafx.scene.control.TableView**
This is the Table view where the Statistics will be showed in.

+ **FxStatisticController()**
The Empty-Constructor of this Class.

+ **updateStatistics()**
This Method updates the Table-View with new Statistics, that should be shown to the User. It gets the Statistics from the HeuristicController.

+ **setStatisticController(statisticController : StatisticController)**
This is the setter-Method for the StatisticController. It only sets the StatisticCotroller of this Class to the given Controller.

This Method is called form the FxTabController which gets the StatisticController via the TabController.
@param statisticController The StatisticController to set.

Package ViewController.FxController.FxGraphGenerationController

This Package orders the FxController a little bit so that the FxController for the Graph-Generation Tab is inside this Package.

Class FxGraphGenerationController

Description

The FxController for the Graph-Generation-Tab.

Documentation

- **fxRageController : FxRageController**
  This is the FxRageController. This is needed so that the GraphGenerationTab can start an Editor by calling this Super-Class. It is passed on by the init-Method.
- **graphGeneratorController : GraphGeneratorController**
  The GraphGeneratorController for this Class.
- **graphProperties : GraphProperties**
  The GraphProperties of this Class. It among other things contains the differentGraphTypes that are implemented in this Program and the User can choose.

  This is only a copy that is get via the GeneratorController, so that the access to this Object is easier for the FxGraphGeneratorController and the GraphGenerationController is not flooded with requests. Therefore there is no need for a getter-Method.
- **cbGraphType : javafx.scene.control.ComboBox**
  The Combobox where the User can choose the Type of the generated Graph.

  The Possible Choices are given via ...??

  The other Settings will be based on this Choice, to provide the user with the correct Settings for the different Graph-Types.

+ **init(fxRageController : FxRageController)**
  This method overrides the init-Method from the abstract-Class FxController. It sets the FxRageController by the given Parameter. This is possible because the FxRageController calls this Method and gives itself as parameter with it.
  @param fxRageController The FxRageController that will be set.

- **updateGUIElementFromProperties()**
  This Method is called by the GraphProperties-Setter. It changes the GUI-Elements to the needed ones. They will be dynamically created.

- **startGeneration()**
  This Method starts the generation by calling the correct Method at the GraphGenerationController.

- **startImport()**
  This Method starts the import of an previously saved Tab by calling the correct Method at the GraphGenerationController.

- **startEditor()**
  This Method starts the GraphEditor by calling the correct Method at the GraphGenerationController. Also it init's the GraphEditor by calling the FxRageController.

+ **setGraphGeneratorController(graphGeneratorController : GraphGeneratorController)**
  Set the GraphGeneratorController of this Class. Then set the graphProperties of this Class from the Value out of the GraphGenerationController. Then call the updateGUIElementFromProperties-Method to update the GUI to the given Properties.
  @param graphGeneratorController The GraphGeneratorController to set.

+ **setGraphGeneratorController(graphProperties : GraphProperties)**
  The Setter for the GraphProperties of this Class. It also calls the updateGUIElement-Method.
  @param graphProperties The GraphProperties to set.

Package ViewController.FxController.FxMenuBarController

This Package orders the FxController a little bit, so that the FxController for the MenuBar is inside this Package.

Class FxMenuBarController

Description

This is the FxController for the MenuBar.

Documentation

- **fxRageController : FxRageController**
  This is the FxRageController of this Class.

  This is needed for several Reasons: - So that the MenuBar can change some Properties of all Tabs,Windows,... (like the Language). - For the Tab-Switching. For example: at the showGraphGeneration-Method.

- **menuLanguages : javafx.scene.control.Menu**
  The Menu for the Language-Choosing.

  This is the Menu where where the MenuItems for the available Languages will be put into. By these added MenuItems the User can change the Language of the GUI.

  The MenuItems for the different Languages will be added dynamically. That's why this JavaFx-Element is an Object at this Controller. (unlike the others of this Scene).

- **toggleGroupLanguage : javafx.scene.control.ToggleGroup**
  The ToggleGroup for the different MenuItems for the Language-Choosing.

  The MenuItems for the different Languages will be added dynamically to the Menu. Therefore the dynamically generated Objects will be set into this ToggleGroup. So that the User only can select one of the available Languages to set the GUI to.

- **importHeuristics()**
  This Method calls the corresponding Controller to perform the Plugin-Import.

Therefore a new FileChooser will be opened so that the User can choose the Folder where the Plugin is stored in.

- **importTab()**
  This Method calls the corresponding Controller to perform the Tab-Import.

  Therefore a new FileChooser will be opened so that the User can choose the Folder where the Graph is stored in.

- **exportTab()**
  This Method calls the corresponding Controller to perform the Tab-Export. Therefore a new FileChooser will be opened so that the User can choose the Folder where the Tab ahould be exported to.

- **showFunctionalHelp()**
  This Method opens the Help Dialogue for the Functional-Things of this Program.

  This Dialogue is shown as a new Window over the current Screen.

- **showUsageHelp()**
  This Method opens the Help Dialogue for the Usage-Things of this Program. It explains the GUI and the steps the User has to do to accomplish the different functional things the Program provides him/her with. This Dialogue is shown as a new Window over the current Screen.

- **showImportHelp()**
  This Method opens the Help Dialogue for the Import. It explains what can be Imported and how the user has to do it. This Dialogue is shown as a new Window over the current Screen.

- **setLanguage()**
  This Method sets the Language of the GUI. Therefore this class calls the changeLanguageMethod of the FxRageController so that all GUI-Elements that contain Text will be shown correct Language.

- **showAbout()**
  This Method opens the About Dialogue from this Program. This Dialogue is shown as a new Window over the current Screen.

- **showGraphGeneration()**
  This Method will switch the GUI to the GraphGeneration-Tab, so that the User can start set the properties and so on. Therefore uses the FxRageController.

- **showGraphEditor()**
  This Method will open the Graph-Editor, so that the User can start working on his/her own Graph. Therefore uses the FxGraphEditorController.

+ **setFxRageController(fxRageController : FxRageController)**
  Set the FxRageController. **@param fxRageController** The FxRageController to set.

6 Resources

6.1 Allgemein

Die Ressourcen sind alle Dateien, die nicht in direktem Zusammenhang mit der Funktionalität und des Programms stehen und keinen Einfluss auf den Ablauf haben. Hierunter fallen meist Bilder, wie Icons, oder auch andere Mediendateien und vieles mehr. Diese Dateien muss unser Programm aus externen Stellen ziehen.

6.2 Entwurf

Diese Daten werden getrennt vom Programmcode abgelegt und dann bei Bedarf aus der vordefinierten Stelle vom Programm eingeladen.

Package Resources

This contains all the Resources that are needed for the Project.

* **FXML**
  This contains all the FXML files for the GUI. They are arranged in different Sub-Folders to separate.

  Main

  StartTab

  Preview

  GraphGeneration

  MenuBar

  Editor

  Popups

* **Pictrues**
  This contains all the Pictures used at the GUI organized by sub-Folders.

  Icons
  This contains all Icons for the Buttons, ... of the GUI.

  Logo
  This contains all Logos used at the GUI.

* **Sound**
  This contains all the Sounds that can be played by default.

* **StyleSheets**
  This Contains all the CSS-Files for the GUI.

* **Plugins**
  This Contains all the Plugins the User could add to the Rage-Program. By Default, there are the Plugins for the TC and EFL that we should implement.

* **Log**
  Contains the Log-Files.

7 Input-Output

Package IO

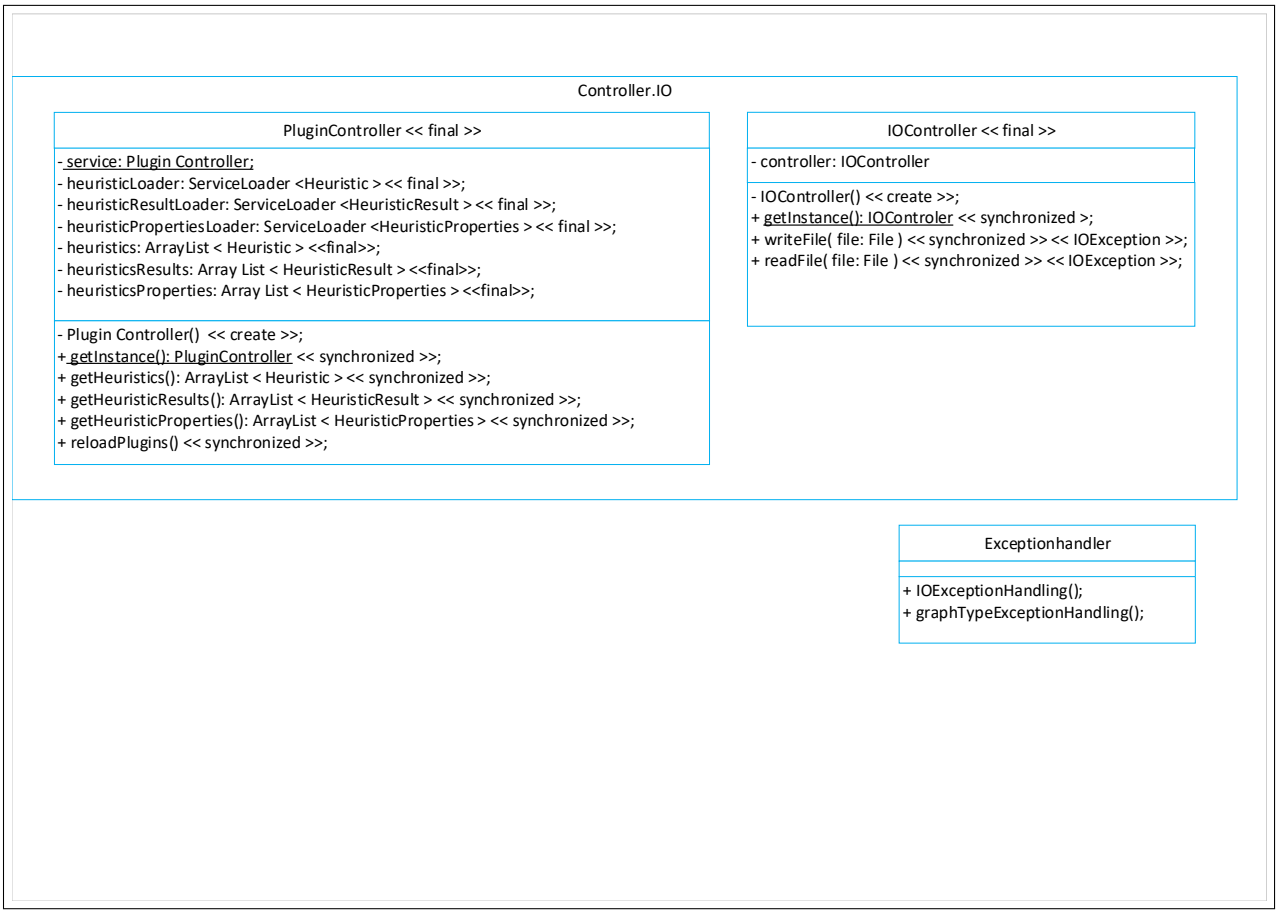


Abbildung 11: Das Paket IO

This package contains classes for input, output and plugin loading.

Class PluginController

Description

Loads all Heuristic, HeuristicResult and HeuristicProperties classes using the ServiceLoader class. It uses the singelton design pattern.

Documentation

+ **getInstance():PluginController**
  This method is the only way to access the PluginController. Creates a new PluginController if it does not exist.
  @return returns the PluginController itself.

+ **getHeuristics():ArrayList<Heuristic>**
  Loads all Heuristic classes if they are not already loaded.

@return returns a list with all Heuristics.

+ **getHeuristicResults():ArrayList<HeuristicResult>**
  Loads all HeuristicResult classes if they are not already loaded.
  @return returns a list with all HeuristicResult classes.

+ **getHeuristicProperties():ArrayList<HeuristicProperties>**
  Loads all HeuristicProperties classes if they are not already loaded.
  @return returns a list with all HeuristicProperties classes.

+ **reloadPlugins()**
  Clears the pluginlists and then loads all plugins.

Class IOController

Description

Saves and loads the data of a single view tab. The file has the extension „.RAGE“. It uses the singelton design pattern.

Documentation

+ **getInstance():IOController**
  This method is the only way to access the IOController. Creates a new IOController if it does not exist.
  @return returns the IOController itself.

+ **writeFile(File file)**
  Writes a RAGE file to the disk.
  @param file Information about the file.
  @throws IOException if saving fails print: „Error while saving the file.“.

+ **readFile(File file)**
  Reads a RAGE file from the disk and sends the content to the model.
  @param file Information about the file.
  @throws IOException if loading fails print: „Error while loading the file.“.

8 Sequenzdiagramme

Durch Benutzereingaben initiierte Aktionen werden durch das JavaFX-Framework an die entsprechenden `FXController` weitergeleitet. Dieses wird in den folgenden Diagrammen deshalb nicht dargestellt, sondern nur die anschließenden Interaktionen.

8.1 Heuristiken ausführen

Eine Benutzereingabe zum Ausführen der ausgewählten Heuristiken wird durch den `FXTabController` verarbeitet und and den `TabController` weitergeleitet. Zu jeder ausgewählten Heuristik wird die Methode `addToHeuristic` aufgerufen. Diese erzeugt zuerst ein Objekt der entsprechenden Heuristik, welches anschliessend zum `DataPool` hinzugefügt wird. Der `DataPool` enthält alle Graphen eines Tabs mit den darauf auszuführenden Heuristiken. Beim Hinzufügen wird die Heuristik über die `applyTo`-Methode auf alle Graphen im `DataPool` angewandt.
Ist die Anwendung aller Heuristiken abgeschlossen, wird eine Liste von Ergebnissen der Berechnungen vom Typ `HeuristicResult` vom `DataPool` zurückgegeben und über den `TabController` an den `FXTabController` weitergeleitet.

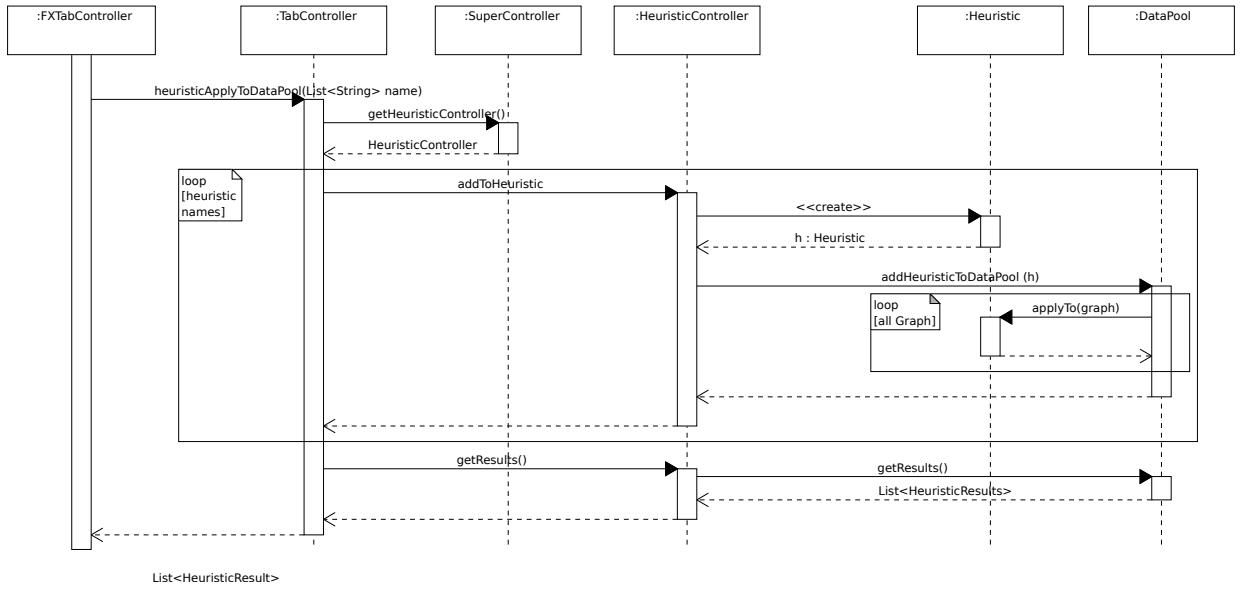


Abbildung 12: Sequenzdiagramm zum Ausführen von Heuristiken

8.2 Graphen generieren

Eine Benutzereingabe zum Generieren von Graphen wird vom `FXGraphGeneratorController` verarbeitet. Dieser delegiert durch den Aufruf der Methode `generate` an den `GraphGeneratorController`. Hier wird für alle n zu generierenden Graphen durch die Klasse `GraphBuilder` jeweils ein Objekt vom Typ `Graph` erzeugt und zu einem `DataPool` hinzugefügt. Jeder dieser Graphen wird anschließend durch die Klasse `GraphAdapter` in eine für die View benötigte Graphenstruktur umgewandelt.
Nach Generierung der Graphen wird ein neuer `TabController` erzeugt. Diesem wird der `DataPool` der neu generierten Graphen übergeben. Zum Schluss wird noch ein neuer `FXTabController` erzeugt, welcher für die Anzeige des Tabs für die generierten Graphen zuständig ist.

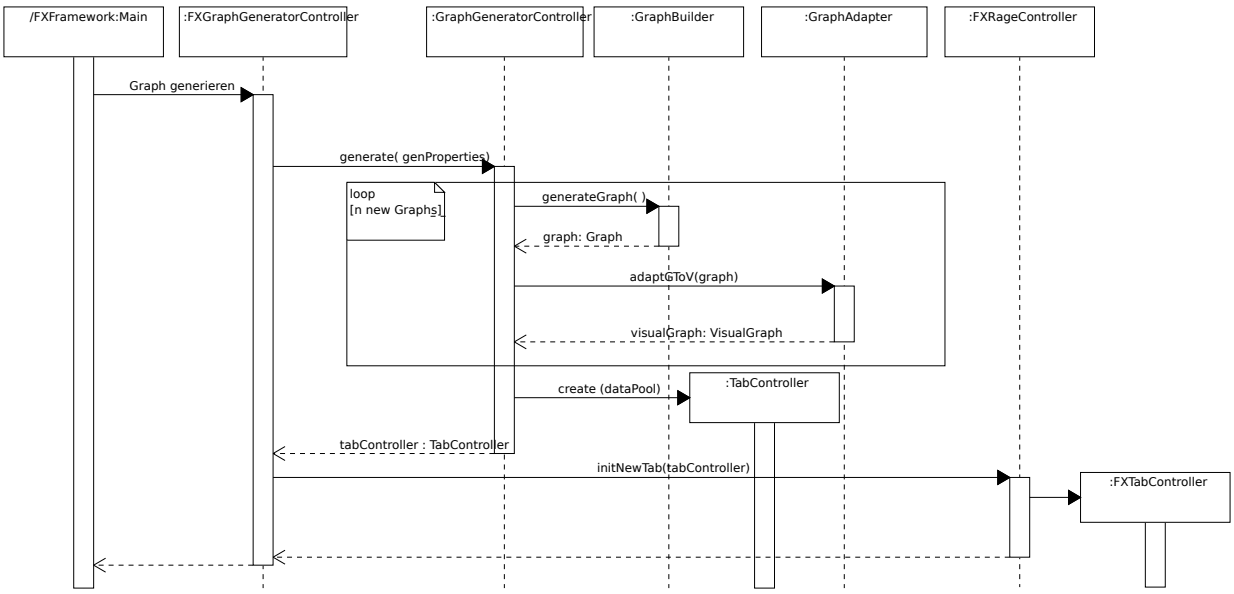


Abbildung 13: Sequenzdiagramm zum Generieren von Graphen

8.3 Graphen modifizieren

Das Modifizieren von Graphen erfolgt über die Detailansicht des Graphen. Dafür wird zuerst ein `FXDetailViewController` erzeugt. Bekommt dieser anschließend eine Benutzereingabe zum Modifizieren, dann wird ein `FXGraphEditorController`-Objekt erzeugt. Den dafür benötigten `GraphEditorConroller` erhält er über den Aufruf der Methode `getGEC` des `SuperControllers`.
Anschließende Editierbefehle werden direkt an den `FXGraphEditorController` weitergeleitet und von diesem verarbeitet. Bei Abschluss des Editierens durch den Benutzer wird der modifizierte Graph als neuer Graph durch den `GraphEditorController` zum `DataPool` hinzugefügt.

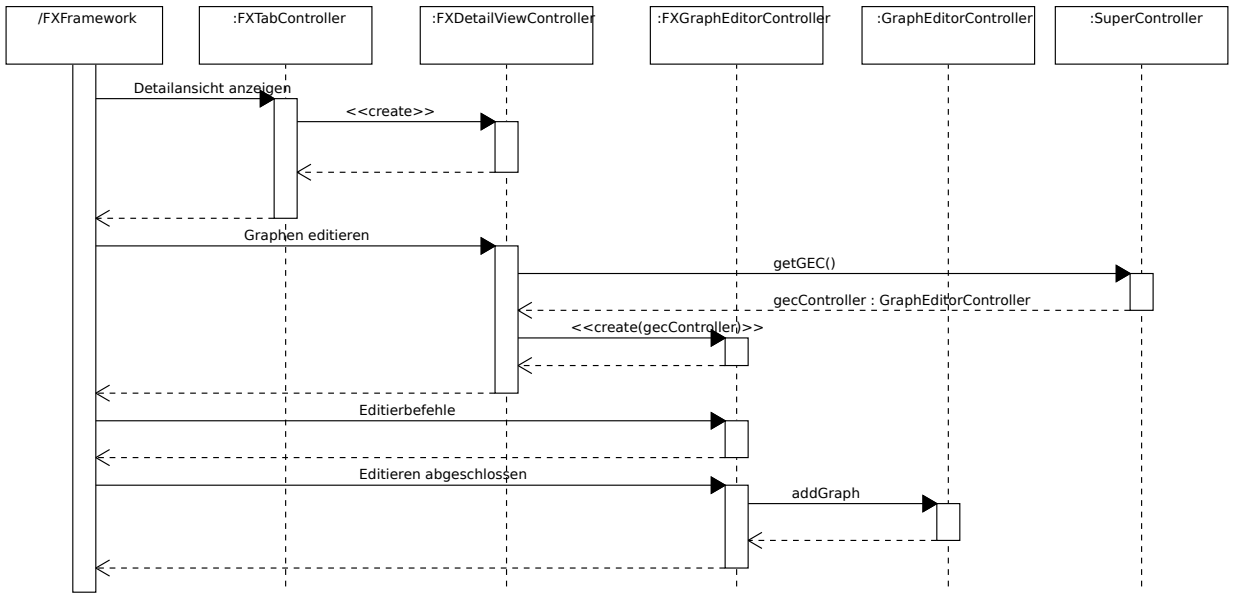


Abbildung 14: Sequenzdiagramm zum Modifizieren von Graphen

9 Utils

In this package some helpful classes are implemented.

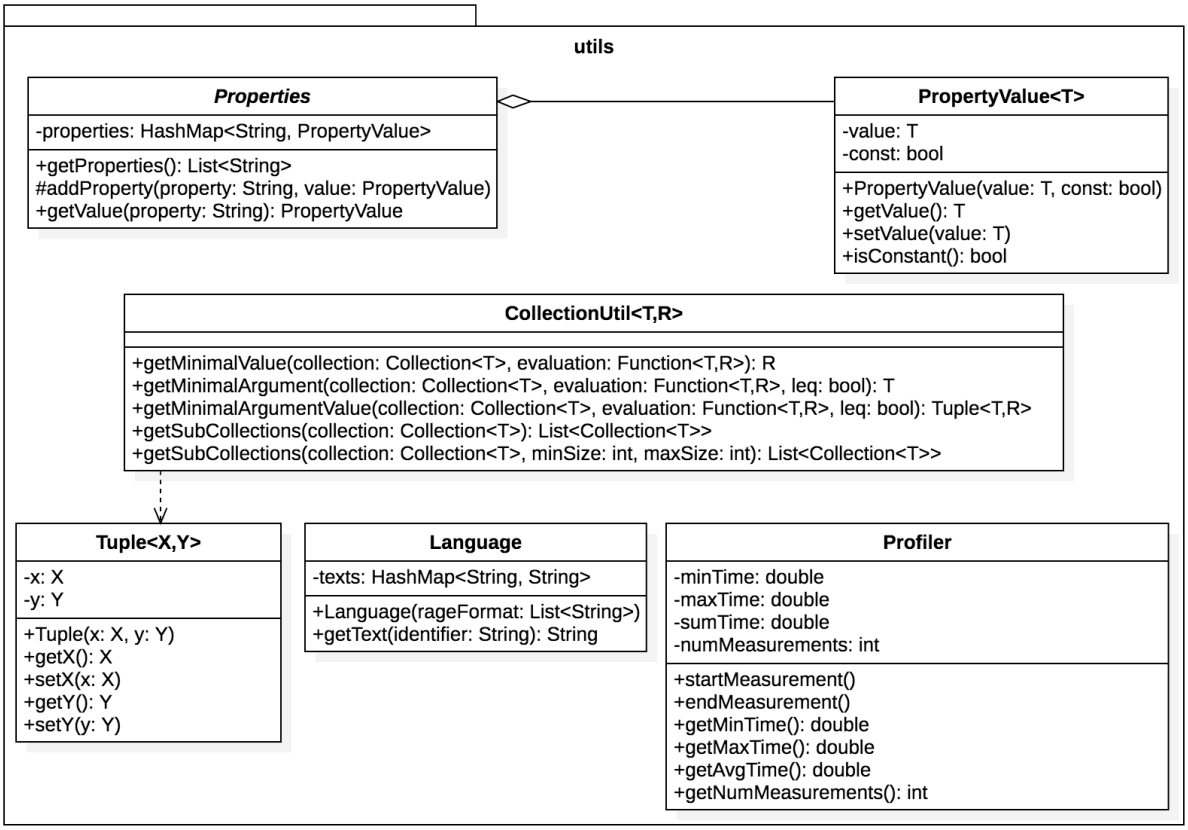


Abbildung 15: Das Paket Utils

Class Properties

Description

This abstract class implements a key value storage used for exchanging data between the different layers of the application. It is used to adapt the user interface to varying settings of heuristics. The keys may be used to translate the GUI to different languages as well. New key-value pairs can only be added by subclasses, thus guaranteeing their presence at any time.

Documentation

+ **getProperties(): List<String>**
  @return returns the list of all keys ie. properties

addProperty(property: String, value: PropertyValue)
  @param property the property to be added
  @param value the value(-type) the property will have
  @throws DataInconsistencyException if the property is already set

+ **getProperty(property: String): PropertyValue**
  @param property the property whose value is requested

@return returns the value of the property
@throws DataInconsistencyException if there is no key **property**

Class PropertyValue

Description

This class stores data of the type **T**, which might be constant.

Documentation

+ **PropertyValue(value: T, const: bool)**
  A constructor of this class
  @param value the value of this PropertyValue
  @param const whether the value is constant ie. might not be altered after its initialization
+ **getValue(): T**
  @return returns the value stored in this PropertyValue
+ **setValue(value: T)**
  @param value the value which will be stored in this PropertyValue
  @throws DataInconsistencyException if the const flag is set
+ **isConstant(): bool**
  @return returns **true** iff this PropertyValue is constant ie. the const flag is set

Class Language

Description

This class stores language data ie. maps tokens to translations.

Documentation

+ **Language(rageFormat: List<String>)**
  @param rageFormat the line by line representation of a language / translation as specified in the RAGE data format
+ **getText(identifier: String): String**
  @param identifier the identifier of the translation
  @return returns the translation corresponding to **identifier** or "MISSING:**identifier**" if there is no translation

Class Profiler

Description

This class is used to gather statistics about timing.

Documentation

+ **startMeasurement()**
  starts measuring the time
+ **endMeasurement()**
  ends the running measurement and adds it to the collected data
+ **getMinTime(): double**
  @return returns the minimal time measured by this Profiler
+ **getMaxTime(): double**
  @return returns the maximal time measured by this Profiler
+ **getAvgTime(): double**
  @return returns the average time measured by this Profiler
+ **getNumMeasurements(): int**
  @return returns the number of measurements taken by this Profiler

Class Tuple

Description

This class represents a tuple of two variables x and y of types **X** and **Y** respectively.

Documentation

+ **Tuple(x: X, y: Y)**
  A constructor of this class
  @param x the first entry of this tuple
  @param y the second entry of this tuple
+ **getX():X**
  @return returns the first component of this tuple
+ **setX(x: X)**
  @param x the value to be assigned to the first component of this tuple
+ **getY():Y**
  @return returns the second component of this tuple
+ **setY(y: Y)**
  @param y the value to be assigned to the second component of this tuple

Class CollectionUtil

Description

This class provides some helper methods for collections of elements of type **T** as finding minimal elements (of a cost function returning values of type **R**) or subsets.

Documentation

+ **getMinimalValue(collection: Collection<T>, evaluation: Function<T,R>): R**
  @param collection the collection on which **evaluation** will be minimized
  @param evaluation the evaluation / cost function which will be minimized
  @return returns the minimal value of **evaluation**
+ **getMinimalArgument(collection: Collection<T>, evaluation: Function<T,R>, leq: bool): T**
  @param collection the collection on which **evaluation** will be minimized
  @param evaluation the evaluation / cost function which will be minimized
  @param leq if **false** the first entry in the collection minimizing **evaluation** will be returned, otherwise the last one will be returned
  @return returns the argument minimalizing **evaluation**
+ **getMinimalArgumentValue(collection: Collection<T>, evaluation: Function<T,R>, leq: bool): Tuple<T,R>**
  @param collection the collection on which **evaluation** will be minimized
  @param evaluation the evaluation / cost function which will be minimized
  @param leq if **false** the first entry in the collection minimizing **evaluation** will be returned, otherwise the last one will be returned
  @return returns the tuple (argument, value) minimalizing **evaluation**

+ **getSubCollections(collection: Collection<T>): List<Collection<T»**
@param collection the collection whose subcollections shall be retrieved
@return returns the list of subcollections of **collection** (in lexicographic order, if **collection** is sorted)

+ **getSubCollections(collection: Collection<T>, minSize: int, maxSize: int): List<Collection<T»**
@param collection the collection whose subcollections shall be retrieved
@param minSize the minimal size of the subcollections returned
@param maxSize the maximal size of the subcollections returned
@return returns the list of subcollections of **collection** (in lexicographic order, if **collection** is sorted) of size between **minSize** and **maxSize**

10 Addendum: Heuristiken

10.1 Einleitung

Im folgenden werden die implementierten Heuristiken erklärt. Im Wesentlichen gibt es zwei Familien von Heuristiken: die TCHeuristiken und die EFLHeuristiken.

Erstere behandeln das **Total Coloring Conjecture**. Dieses besagt, dass jeder einfache ungerichtete Graph mit Maximalgrad Δ eine valide Totalfärbung mit $\Delta+2$ Farben besitzt. Dabei ist ein **einfacher ungerichteter Graph** ein Graph, in dem Kanten immer genau zwei verschiedene Knoten verbinden, die Reihenfolge dieser Knoten nicht relevant ist und es nicht mehrere Kanten zwischen zwei Knoten gibt. Der **Grad** eines Knoten eines solchen Graphen ist die Anzahl der zu ihm inzidenten Kanten. Eine valide **Totalfärbung** ist eine Färbung der Knoten und Kanten eines solchen Graphen, sodass keine zwei zueinander adjazente Knoten, adjazente Kanten oder inzidente Knoten und Kanten dieselbe Farbe besitzen.

Zweitere behandeln das **Erdös Faber Lovasz Conjecture**. Es besagt, dass jeder einfache Hypergraph auf n Knoten eine valide Kantenfärbung mit n Farben besitzt. Dabei ist ein **Hypergraph** ein Graph, dessen (Hyper-)Kanten beliebige Teilmengen der Knotenmenge darstellen. Ein solcher Graph heißt **einfach**, wenn alle Kanten mindestens zwei Knoten umfassen und es keine zwei Kanten gibt, die mehr als einen Knoten gemeinsam haben. Eine **Kantenfärbung** ist eine Färbung der Hyperkanten dergestalt, dass keine zwei adjazente Hyperkanten dieselbe Farbe zugewiesen haben.

Die **freien Farben** $\mathcal{F}(v)$ eines Knotens bzw. $\mathcal{F}(e)$ einer Kante sind all die Farben, die nicht von einem dazu adjazenten oder inzidenten Objekt belegt sind. Die **Flexibilität** einer Menge X von Knoten und Kanten ist die Anzahl der Farben, die für jedes Objekt frei sind, minus der Anzahl der Objekte, d.h. $\#\bigcap_{x\in X}\mathcal{F}(x) - \#X$. Negative Flexibilität bedeutet insbesondere, dass die Menge X nicht gefärbt werden kann.

Grundlegend für die Arbeitsweise der Heuristiken ist die inhärente Ordnung der Knoten gegeben über ihre Indizes. Auf einfachen ungerichteten Kanten und Hyperkanten wird dadurch eine **lexikographische Ordnung** induziert, indem man sie als geordnete Tupel von Knotenindizes auffasst. Es gilt dann z.B. $(1,2) < (1,4) < (2,3) < (2,3,4)$.

Um eine möglichst gleichmäßige Färbung zu erzeugen zählen die Heuristiken mit, wie oft einzelne Farben benutzt worden sind, gewichtet nach der Anzahl der gefärbten Knoten und der gefärbten Kanten. Konkret bedeutet das, dass die Anzahl der mit einer bestimmten Farbe c gefärbten Knoten $V(c)$ eine andere Gewichtung ω_v haben kann als die Gewichtung ω_e der mit dieser Farbe gefärbten Kanten $E(c)$. Das Gewicht einer Farbe wird demnach durch $\#V(c)\cdot\omega_v + \#E(c)\cdot\omega_e$ bestimmt. Da Farben ebenso wie Knoten mit einer natürlichen Zahl identifiziert werden, gibt es stets eine eindeutige Farbe minimaler Gewichtung und minimalem Index. Diese wird im Folgenden nur mit **minimal benutzter Farbe** bezeichnet.

Die in den folgenden Heuristiken auftretende **Breitensuche** startet stets im Knoten mit minimalem Index. Für jeden Knoten werden wie bei einer Breitensuche üblich alle noch unberührten Nachbarn nach ihren Indizes sortiert inspiziert. Sollte der Graph nicht zusammenhängend sein, wird immer der kleinste unberührte Knoten als neuer Startpunkt genommen.

10.2 TCGreedy

Beschreibung

Diese Heuristik färbt zuerst die Knoten mithilfe einer Breitensuche. Anschließend färbt sie (ebenfalls in der Reihenfolge einer Breitensuche) der Reihe nach alle Kanten inzident zu einem gemeinsamen Knoten.

Pseudocode

```
for every vertex v in order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

for every vertex v in order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.3 TCGreedyOne

Beschreibung

Diese Heuristik färbt zuerst die Knoten mithilfe einer Breitensuche. Anschließend färbt sie (ebenfalls in der Reihenfolge einer Breitensuche) der Reihe nach alle Kanten inzident zu einem gemeinsamen Knoten, wenn es grad keine Kanten gibt, die genau eine freie Farbe haben. Letztere werden bevorzugt koloriert.

Pseudocode

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

for every vertex v in the order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        while there are uncolored edges with exactly one free color
            get minimal uncolored edge with exactly one free color f
            color f with its unique free color

        if e is colored already
            continue
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.4 TCGreedyFew

Beschreibung

Diese Heuristik färbt zuerst die Knoten mithilfe einer Breitensuche. Anschließend färbt sie der Reihe nach alle Kanten inzident zu einem gemeinsamen Knoten. Die Knoten werden ebenfalls in der Reihenfolge einer Breitensuche ausgewählt. Sollte es Kanten in diesem Graphen geben, die weniger freie Farben haben als die grade zu kolorierende, so werden sie bevorzugt gefärbt.

Pseudocode

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

for every vertex v in the order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        while there are uncolored edges with less free colors than e and lower order than e
            get minimal uncolored edge f
            if f cannot be colored
                return incomplete coloring
            get minimally used free color c of f with respect to the color weights
            color f with color c

        if e is colored already
            continue
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.5 TCGreedySet

Beschreibung

Diese Heuristik färbt zuerst die Knoten mithilfe einer Breitensuche. Anschließend färbt sie die Kanten durch auffinden derjenigen Teilmenge von Kanten inzident zu einem gemeinsamen Knoten, die minimale Flexibilität besitzt und bezüglich lexikographischer Ordnung minimal ist.

Pseudocode

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

while there is a set with minimal flexibility
    find the set X of minimal flexibility belonging to one vertex and with lowest lexicographic
    order
    if X has negative flexibility
        return incomplete coloring

    for every edge e of X in the order defined on edges
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.6 TCGreedyCon

Beschreibung

Diese Heuristik färbt zuerst die Knoten mithilfe einer Breitensuche. Anschließend färbt sie die Kanten durch auffinden derjenigen Teilmenge von zusammenhängenden Kanten, die minimale Flexibilität besitzt und bezüglich lexikographischer Ordnung minimal ist. Die Größe dieser zusammenhängenden Teilmengen kann dabei nach oben beschränkt sein.

Pseudocode

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

while there is a set with minimal flexibility
    find the set X of minimal flexibility which has the lowest lexicographic order
        if X has negative flexibility
            return incomplete coloring

        for every edge e of X in the order defined on edges
            if e cannot be colored
                return incomplete coloring
            get minimally used free color c of e with respect to the color weights
            color e with color c

return complete coloring
```

10.7 TCMixedGreedy

Beschreibung

Diese Heuristik nutzt eine Breitensuche, um von Knoten zu Knoten zu kommen. Sie färbt stets zunächst den Knoten und anschließend alle dazu inzidenten Kanten, bevor sie sich dem nächsten Knoten annimmt.

Pseudocode

```
for every vertex v in order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

    for every uncolored edge e incident to v in the order defined on edges
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.8 TCMixedGreedyOne

Beschreibung

Diese Heuristik nutzt eine Breitensuche, um von Knoten zu Knoten zu kommen. Sie färbt stets den Knoten und anschließend alle dazu inzidententen Kanten, es sei denn es gibt unkolorierte Knoten oder Kanten, die genau eine freie Farbe haben. Diese werden stets bevorzugt koloriert, wobei Knoten den Kanten vorgezogen werden.

Pseudocode

```
for every vertex v in order of a breadth first search
    while there are objects with exactly one free color
        if there are vertices with exactly one free color
            get the vertex w with exactly one free color and minimal index
            color w with its unique free color
        else
            get the minimal edge f with exactly one free color with respect to the order
            defined on edges
            color f with its unique free color

    if v is not colored yet
        if v cannot be colored
            return incomplete coloring
        get minimally used free color c of v with respect to the color weights
        color v with color c

    for every uncolored edge e incident to v in the order defined on edges
        while there are objects with exactly one free color
            if there are vertices with exactly one free color
                get the vertex w with exactly one free color and minimal index
                color w with its unique free color
            else
                get the minimal edge f with exactly one free color with respect to the order
                defined on edges
                color f with its unique free color

        if e is not colored yet
            if e cannot be colored
                return incomplete coloring
            get minimally used free color c of e with respect to the color weights
            color e with color c

return complete coloring
```

10.9 TCMixedGreedyFew

Beschreibung

Diese Heuristik nutzt eine Breitensuche, um von Knoten zu Knoten zu kommen. Sie färbt stets zunächst den Knoten, falls es keine anderen unkolorierten Knoten mit weniger freien Farben und geringerem Index gibt, die bevorzugt zu behandeln sind. Anschließend färbt sie alle Kanten inzident zu dem Knoten, falls es keine unkolorierten Kanten mit weniger freien Farben und geringerer Ordnung gibt, die zu bevorzugen sind.

Pseudocode

```
for every vertex v in order of a breath first search
    while there is a vertex w with less free colors than v and lower index
        if w cannot be colored
            return incomplete coloring
        get minimally used free color c of w with respect to the color weights
        color w with color c

    if v is not colored yet
        if v cannot be colored
            return incomplete coloring
        get minimally used free color c of v with respect to the color weights
        color v with color c

    for every uncolored edge e incident to v in the order defined on edges
        while there is an edge f with less free colors than e and lower order
            if f cannot be colored
                return incomplete coloring
            get minimally used free color c of f with respect to the color weights
            color f with color c

        if e is not colored yet
            if e cannot be colored
                return incomplete coloring
            get minimally used free color c of e with respect to the color weights
            color e with color c

return complete coloring
```

10.10 TCMixedGreedySet

Beschreibung

Diese Heuristik sucht Teilmengen von Kanten inzident zu einem gemeinsamen Knoten, die minimale Flexibilität hat und bezüglich lexikographischer Ordnung minimal ist. Sollte der gemeinsame Knoten unkoloriert sein, werden Varianten mit dem Knoten und ohne den Knoten betrachtet, wobei letztere bevorzugt wird. Besagte Teilmenge minimaler Flexibilität wird dann koloriert.

Pseudocode

```
while there are uncolored vertices or edges
    find set X of minimal flexibility with lowest lexicographic order incident to a vertex v
        if X has negative flexibility
            return incomplete coloring

        if v is uncolored and contained in X
            if v cannot be colored
                return incomplete coloring
            get minimally used free color c of v with respect to the color weights
            color v with color c

        for every edge e in X
            if e cannot be colored
                return incomplete coloring
            get minimally used free color c of v with respect to the color weights
            color e with color c

return complete coloring
```

10.11 TCMixedGreedyCon

Beschreibung

Diese Heuristik sucht unter denjenigen Menge zusammenhängender Kanten, und zu ihnen inzidenten Knoten, die minimale Flexibilität und minimale lexikographische Ordnung hat. Dabei werden zunächst die Kanten verglichen und erst am Ende die Knotenmengen, aufgefasst als Hyperkanten. Die Anzahl der Kanten in einer solchen Menge kann dabei nach oben beschränkt sein.

Pseudocode

```
while there are uncolored vertices or edges
    find connected set X of minimal flexibility and minimal lexicographic order
    if X has negative flexibility
        return incomplete coloring

    for every vertex v in X
        if v cannot be colored
            return incomplete coloring
        get minimally used free color c of v with respect to the color weights
        color v with color c

    for every edge e in X
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.12 EFLGreedy

Beschreibung

Diese Heuristik färbt (in der Reihenfolge einer Breitensuche) der Reihe nach alle Hyperkanten inzident zu einem gemeinsamen Knoten.

Pseudocode

```
for every vertex v in order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.13 EFLGreedyOne

Beschreibung

Diese Heuristik färbt (in der Reihenfolge einer Breitensuche) der Reihe nach alle Hyperkanten inzident zu einem gemeinsamen Knoten, wenn es grad keine Hyperkanten gibt, die genau eine freie Farbe haben. Letztere werden bevorzugt koloriert.

Pseudocode

```
for every vertex v in the order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        while there are uncolored edges with exactly one free color
            get minimal uncolored edge with exactly one free color f
            color f with its unique free color

        if e is colored already
            continue
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.14 EFLGreedyFew

Beschreibung

Diese Heuristik färbt (in der Reihenfolge einer Breitensuche) der Reihe nach alle Hyperkanten inzident zu einem gemeinsamen Knoten, wenn es grad keine Hyperkanten gibt, die weniger freie Farben haben. Letztere werden bevorzugt koloriert.

Pseudocode

```
for every vertex v in the order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        while there are uncolored edges with less free colors than e and lower order than e
            get minimal uncolored edge f
            if f cannot be colored
                return incomplete coloring
            get minimally used free color c of f with respect to the color weights
            color f with color c

        if e is colored already
            continue
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.15 EFLGreedySet

Beschreibung

Diese Heuristik färbt die Hyperkanten durch auffinden derjenigen Teilmenge von Hyperkanten inzident zu einem gemeinsamen Knoten, die minimale Flexibilität besitzt und bezüglich lexikographischer Ordnung minimal ist.

Pseudocode

```
while there is a set with minimal flexibility
    find the set X of minimal flexibility belonging to one vertex and with lowest lexicographic
    order
    if X has negative flexibility
        return incomplete coloring

    for every edge e of X in the order defined on edges
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

10.16 EFLGreedyCon

Beschreibung

Diese Heuristik färbt die Hyperkanten durch auffinden derjenigen Teilmenge von zusammenhängenden Hyperkanten, die minimale Flexibilität besitzt und bezüglich lexikographischer Ordnung minimal ist. Die Größe dieser zusammenhängenden Teilmengen kann dabei nach oben beschränkt sein.

Pseudocode

```
while there is a set with minimal flexibility
    find the set X of minimal flexibility which has the lowest lexicographic order
        if X has negative flexibility
            return incomplete coloring

        for every edge e of X in the order defined on edges
            if e cannot be colored
                return incomplete coloring
            get minimally used free color c of e with respect to the color weights
            color e with color c

return complete coloring
```

11 Addendum: RAGE-Datenformate

11.1 Graphen

Graphen werden wie folgt im Textformat gespeichert:
Die erste Zeile enthält den Typ des Graphen und die zweite Zeile lediglich eine Ganzzahl, die die Anzahl der Knoten speichert. Es folgen Zeile für Zeile die Kanten, dargestellt als aufeinanderfolgende, aufsteigend sortierte Ganzzahlen.

11.2 Properties

Properties werden wie folgt im Textformat gespeichert:
Die erste Zeile enthält den konkreten Typ der Properties-Klasse. Ab der zweiten Zeile werden zeilenweise die einzelnen Properties im Format `property value:type` gespeichert.

11.3 Heuristiken

Die Heuristiken speichern neben ihren Properties nur ihren Namen.