Random Graph Coloring Evaluation

Entwurfsdokument

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

Inhaltsverzeichnis

1 Anmerkungen zum Pflichtenheft

1.1 Klarstellungen

1.2 Änderungen

2 Übersicht

3 Model

Package graph

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

TODO graph.png UML einfügen

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.

TODO heuristic.png UML

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.

TODO tc.png UML

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.

TODO tcgreedy.png UML

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.

TODO tcmixedgreedy.png UML

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.

TODO efl.png UML

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.

TODO eflgreedy.png UML

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 with exactly one free color

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

5 Controller

6 Input-Output

7 Utils

In this package some helpful classes are implemented. TODO utils.png UML

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**

8 Addendum: Heuristiken

8.1 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.

8.2 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
```

8.3 TCGreedyOne

Beschreibung

Diese Heuristik färbt zuerst die Knoten mithilfe einer Breitensuche. Anschließend färbt sie der Reihe nach alle Kanten inzident zu einem gemeinsamen Knoten (ebenfalls in der Reihenfolge einer Breitensuche). Wenn es jedoch Kanten gibt, die genau eine freie Farbe haben, werden diese 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 uncolored edge with exactly one free color f which is minimal with respect to
            the order given on edges
            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
```

8.4 TCGreedyFew

Beschreibung

Diese Heuristik färbt zuerst die Knoten mithilfe einer Breitensuche. Anschließend färbt sie der Reihe nach alle Kanten inzident zu einem gemeinsamen Knoten(ebenfalls in der Reihenfolge einer Breitensuche). Wenn es jedoch Kanten gibt, die weniger freie Farben haben, als die aktuell zu kolorierende, werden diese 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 such uncolored edge f which is minimal with respect to the order given on edges
            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
```

8.5 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
```

8.6 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
```

8.7 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
```

8.8 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 edge f with exactly one free color which is minimal 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 edge f with exactly one free color which is minimal 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
```

8.9 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
```

8.10 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
```

8.11 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
```

8.12 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
```

8.13 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 uncolored edge with exactly one free color f which is minimal with respect to
            the order given on edges
            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
```

8.14 EFLGreedyFew

Beschreibung

Diese Heuristik färbt der Reihe nach alle Hyperkanten inzident zu einem gemeinsamen Knoten (in der Reihenfolge einer Breitensuche). Wenn es Hyperkanten gibt, die weniger freie Farben haben, werden diese 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 such uncolored edge f which is minimal with respect to the order given on edges
            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
```

8.15 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
```

8.16 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
```

9 Addendum: RAGE-Datenformate

9.1 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.

9.2 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.

9.3 Heuristiken

Die Heuristiken speichern neben ihren Properties nur ihren Namen.