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

Paket graph

Das Paket enthält die Schnittstellen für die Interaktion mit Graphen. In den Unterpaketen sind diese für konkrete Graphtypen implementiert.

Klasse Graph

Beschreibung

Diese Klasse beschreibt die abstrakte Struktur von Graphen. Jeder Graph hat (unabhängig von seinem konkreten Typ) eine endliche Anzahl von Knoten und Kanten, die diese Knoten in Relation setzen. Die Art **E** dieser Kanten definiert dann den konkreten Graphentyp. Die Klasse stellt Methoden zur Abfrage der durch die Kanten gegebenen Relationen. Knoten werden dabei mit einem eindeutigen Index identifiziert und werden daher nicht explizit gespeichert.

Dokumentation

+ **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

Klasse Edge

Beschreibung

Eine Kante definiert stets eine Adjazenzrelation der zu ihr inzidenten Knoten. Zudem stellt die Klasse sicher, dass Kanten miteinander verglichen werden können.

Dokumentation

+ ***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.

Klasse GraphProperties

Beschreibung

Die Klasse dient als Datensammlung zum Austausch zwischen Controller und Model, speziell zum Übermitteln der bei der Graphgenerierung benötigten Einstellungen. Sie stellt sicher, dass die folgenden Graph-Eigenschaften stets abgefragt und gesetzt werden können:

– ”graphTypes” – eine unveränderbare Liste von Strings, initialisiert mit [”simpleUndirectedGraph”, ”simpleHyperGraph”]

– ”type” – ein String

– ”numVertices” – ein nichtnegativer int

Klasse GraphBuilder

Beschreibung

Die Klasse bietet die Funktionalität zum zufälligen Generieren eines Graphen des Graphentyps **G** (nach gegebenen GraphProperties **P**) sowie zum Modifizieren von Graphen diesen Typs.

Dokumentation

+ ***generate(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

Paket graph.simpleUndirectedGraph

In diesem Paket sind einfache ungerichtete Graphen implementiert. Es bietet die Funktionalität diese zu generieren, zu modifizieren und nach einigen für einfache ungerichtete Graphen wohldefinierten Kriterien zu unterscheiden.

Klasse SimpleUndirectedGraph

Beschreibung

Die Klasse stellt eine Konkretisierung der Graph-Klasse im Sinne einfacher ungerichteter Graphen dar. Ein solcher Graph enthält weder Schleifen noch Multikanten. Die Klasse bietet Methoden zur Erkennung diverser Eigenschaften einfacher ungerichteter Graphen an.

Dokumentation

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

Klasse SimpleUndirectedEdge

Beschreibung

Die Klasse konkretisiert die Klasse Edge im Sinne einer einfachen ungerichteten Kante. Sie setzt stets zwei verschiedene Knoten in Beziehung, stellt also niemals eine Schleife dar.

Dokumentation

+ **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

Klasse SimpleUndirectedGraphProperties

Beschreibung

Die Klasse ist eine Erweiterung der GraphProperties Klasse und dient ebenso als Datensammlung zum Austausch zwischen Controller und Model, speziell zum Übermitteln der bei der Graphgenerierung benötigten Einstellungen. Sie stellt sicher, dass die folgenden Graph-Eigenschaften stets abgefragt und gesetzt werden können:

- "minDegree" – ein nichtnegativer int
- "maxDegree" – ein nichtnegativer int
- "connected" – ein bool
- "forest" – ein bool
- "bipartite" – ein bool
- "planar" – ein bool

Klasse SimpleUndirectedGraphBuilder

Beschreibung

Die Klasse bietet die Funktionalität zum zufälligen Generieren (nach gegebenen SimpleUndirectedGraphProperties) sowie zum Modifizieren einfacher ungerichteter Graphen.

Dokumentation

+ **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

Paket graph.simpleHyperGraph

In diesem Paket sind einfache Hypergraphen implementiert. Es bietet die Funktionalität diese zu generieren, zu modifizieren und nach einigen für einfache Hypergraphen wohldefinierten Kriterien zu unterscheiden.

Klasse SimpleHyperGraph

Beschreibung

Die Klasse stellt eine Konkretisierung der Graph-Klasse im Sinne einfacher Hypergraphen dar. Ein solcher Graph enthält keine Hyperkanten der Kardinalität eins und keine zwei verschiedenen Hyperkanten, die mehr als einen Knoten gemein haben. Die Klasse bietet Methoden zur Erkennung diverser Eigenschaften einfacher Hypergraphen an.

Dokumentation

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

Klasse SimpleHyperEdge

Beschreibung

Die Klasse konkretisiert die Klasse Edge im Sinne einer einfachen Hyperkante. Sie setzt stets mindestens zwei verschiedene Knoten in Beziehung.

Dokumentation

+ **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

Klasse SimpleHyperGraphProperties

Beschreibung

Die Klasse ist eine Erweiterung der GraphProperties Klasse und dient ebenso als Datensammlung zum Austausch zwischen Controller und Model, speziell zum Übermitteln der bei der Graphgenerierung benötigten Einstellungen. Sie stellt sicher, dass die folgenden Graph-Eigenschaften stets abgefragt und gesetzt werden können:

- "uniform" – ein nichtnegativer int
- "minDegree" – ein nichtnegativer int
- "maxDegree" – ein nichtnegativer int
- "connected" – ein bool

Klasse SimpleHyperGraphBuilder

Beschreibung

Die Klasse bietet die Funktionalität zum zufälligen Generieren (nach gegebenen SimpleUndirectedGraphProperties) sowie zum Modifizieren einfacher ungerichteter Graphen.

Dokumentation

+ **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

Paket heuristic

Das Paket beinhaltet das Interface für die Implementierung von Heuristiken. In den Unterpaketen sind einige Heuristiken für das Total-Coloring-Conjecture sowie für das Erdös-Faber-Lovasz-Conjecture implementiert.

Klasse Heuristic

Beschreibung

Die Klasse stellt die abstrakte Schnittstelle einer Heuristik zur Anwendung auf Graphen des Typs **G** mit Ergebnis vom Typ **R** dar.

Dokumentation

+ **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

Klasse HeuristicResult

Beschreibung

Die Klasse ist die abstrakte Schnittstelle für das Ergebnis der Berechnung einer Heuristik **H** auf einem Graphen des Typs **G**.

Dokumentation

+ **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

Klasse HeuristicProperties

Beschreibung

Die Klasse dient als Datensammlung zum Austausch zwischen Controller und Model, speziell zum Übermitteln der Einstellungen für Heuristiken. Sie stellt sicher, dass die Eigenschaften stets abgefragt und gesetzt werden können:

– ”name” – ein String

Klasse DataPool

Paket heuristic.totalColoring

Klasse TCHeuristic

Klasse TCResult

Paket heuristic.totalColoring.greedy

Klasse TCGreedyData

Klasse TCGreedy

Klasse TCGreedyOneData

Klasse TCGreedyOne

Klasse TCGreedyFewData

Klasse TCGreedyFew

Klasse TCGreedySetData

Klasse TCGreedySet

Klasse TCGreedyConData

Klasse TCGreedyCon

Paket heuristic.totalColoring.mixedGreedy

Klasse TCMixedGreedyData

Klasse TCMixedGreedy

Klasse TCMixedGreedyOneData

Klasse TCMixedGreedyOne

Klasse TCMixedGreedyFewData

Klasse TCMixedGreedyFew

Klasse TCMixedGreedySetData

Klasse TCMixedGreedySet

Klasse TCMixedGreedyConData

Klasse TCMixedGreedyCon

Paket heuristic.erdosFaberLovasz

Klasse EFLHeuristic

Klasse EFLResult

Paket heuristic.erdosFaberLovasz.greedy