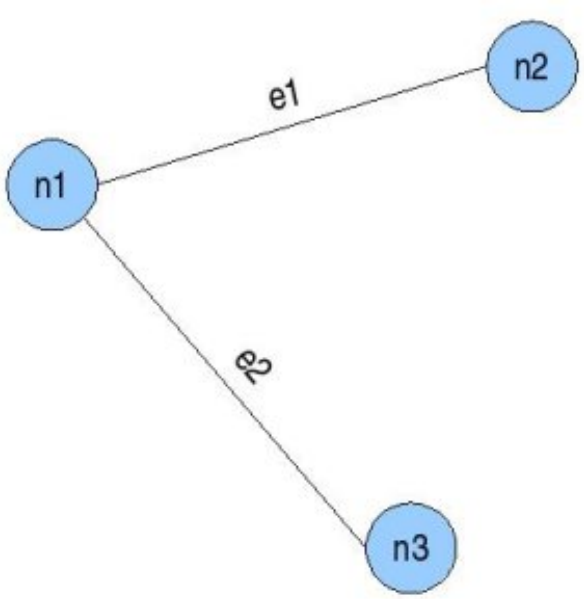
Graphen =

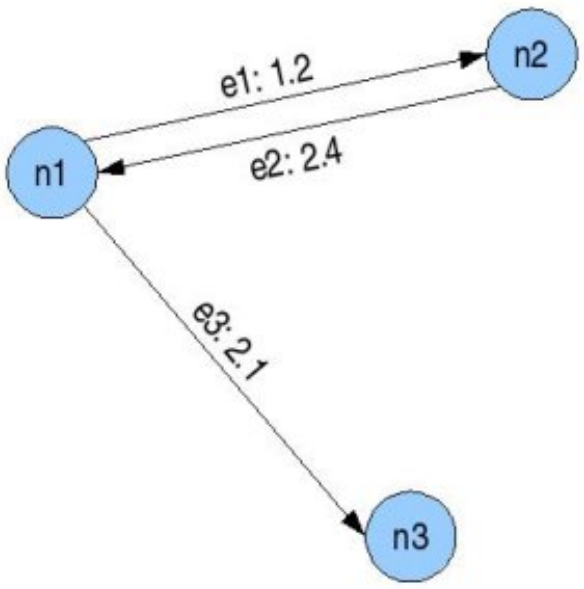
Grundidee

Es soll ein Programm in mehreren Schritten entwickelt werden mit dem in einem Graphen nach kürzesten Wegen gesucht, und die Suche (schrittweise) visualisiert werden kann.

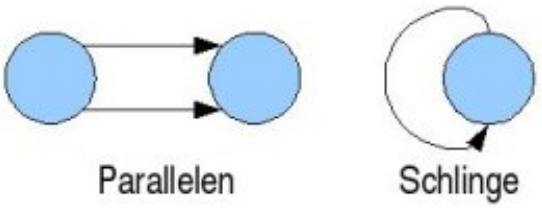
Bei einem Graphen (http://de.wikipedia.org/wiki/Graph_(Graphentheorie)) handelt es sich hierbei um eine Menge von Knoten (z. B. Städte), welche über Kanten (z. B. Straßen) verbunden sind. In folgender Abbildung sieht man einen Graphen mit den Knoten n1, n2 und n3, sowie den Kanten e1 und e2.



In der Praxis werden werden bei Graphen oft gerichtete Kanten verwendet und man ordnet den Kanten einen Wert bzw. Gewichten zu (z.B. Länge der Straße).

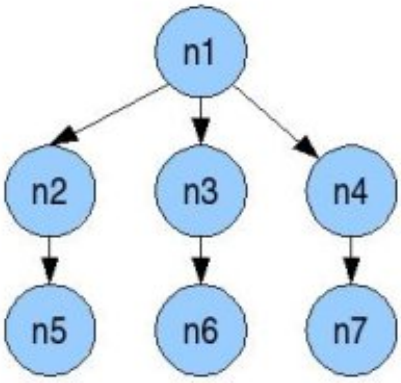


Weiter treten auch häufig Schlingen und Parallelen auf.



Suche in Graphen

Eine in der Praxis sehr verbreitete Anwendung auf Graphen ist die Suche nach (kürzesten) Wegen, wie sie z. B. von Routenplanern bekannt ist. Hierfür sind in der Graphentheorie (http://de.wikipedia.org/wiki/Graphentheorie) viele verschiedene Algorithmenansätze bekannt, von welchen wir im Verlauf dieser Aufgabe eine kleine Auswahl betrachten wollen.



Breitensuche

Bei der Breitensuche (http://de.wikipedia.org/wiki/Breitensuche) (BFS, breadth-first search) handelt es sich um einen uninformierten Suchalgorithmus, der einen Weg von einem Startknoten zu einem Zielknoten im Graphen findet. Voraussetzung ist natürlich, dass dieser Weg auch im Graphen existiert. Hierbei wird der Graph bei der Expansion der einzelnen Ebenen in der Breite durchsucht.

Beispiel: Wird in obigen Graphen eine Weg von n1 nach n6 gesucht, so werden beispielsweise die Knoten n1, n2, n3, n4, n5 und n6 in dieser Reihenfolge expandiert.

Pseudocode:

```
bfs(source, target)
    queue = {}
    queue.enqueue(source)

    while(not queue.isEmpty()) {
       c = queue.dequeue()
        if (c == target)
            return getWay(min)
```

```
        for each successor s of c {
            if s not visited
                queue.enqueue(s)
        }
    }
    return fail
```

Tiefensuche

Bei der Tiefensuche (http://de.wikipedia.org/wiki/Tiefensuche) (DFS, depth-first search) handelt es sich um einen uninformierten Suchalgorithmus,
der einen Weg von einem Startknoten zu einem Zielknoten im Graphen findet. Voraussetzung ist ebenfalls, dass dieser Weg auch im Graphen existiert. Hierbei werden jeweils die ersten Nachfolgerknoten expandiert.

Beispiel: Wird im obigen Graphen eine Weg von n1 nach n6 gesucht, so werden beispielsweise die Knoten n1, n2, n5, n3 und n6 expandiert.

Pseudocode:

```
dfs(source, target)
    stack = {}
    stack.push(source)

    while(not stack.isEmpty()) {
        c = stack.pop()
        if (c == target)
            return getWay(min)

        for each successor s of c {
            if s not visited
                stack.push(s)
        }
    }
return fail
```

A*-Suche

Der A-Stern-Algorithmus (http://de.wikipedia.org/wiki/A-Stern-Algorithmus) ist ein informierter Suchalgorithmus auf Graphen, welcher einen (kürzesten) Weg von einem Startknoten zum Zielknoten berechnet. Voraussetzung ist natürlich, dass ein Weg existiert.

Wichtig ist beim A-Stern-Algorithmus, eine der informierten Suche zur Verfügung stehenden Schätzfunktion h(s). Diese Funktion h(s) schätzt von einem Knoten s die wahrscheinlichen Kosten zum Zielknoten der Suche. Von Vorteil ist eine monotone Schätzfunktion h(s), welche sowohl die Kosten zum Ziel und von Knoten zu Knoten niemals überschätzt. Durch die Verwendung einer monotonen Schätzfunktion wird sichergestellt, dass ein kürzester Weg gefunden wird. Überschätzt h(s) lediglich die Kosten zum Ziel niemals, so handelt es sich um eine zulässige Schätzfunktion. Für zulässige

Schätzfunktionen kann aber nicht garantiert werden, dass der gefundene Weg auch der kürzeste Weg ist. Wendet man z. B. den A-Stern-Algorithmus in einem Routenplaner an, so kann man h(s) als Luftline zwischen der Stadt s und dem Ziel definieren und erhält somit einen monotonen Schätzer.

In der Suche wird dann immer der Knoten s expandiert, für welchen die bis jetzt berechneten Kosten g(s), also die Kosten vom Startknoten nach s, zusammen mit den geschätzten noch verbleibenden Kosten h(s) am geringsten sind. Man betrachtet also immer den Knoten s für den f(s):= g(s) + h(s) minimal ist.

Weitere Details über den Algorithmus entnehmen Sie bitte Wikipedia (http://de.wikipedia .org/wiki/A-Stern-Algorithmus).

Pseudocode:

```
astar(source, target)
    g(source) = 0
    open = {}
    close = {}
    open.insert(source)

    while(not open.isEmpty()) {
        remove min with the smallest f-value from open
        if (min == target)
            return getWay(min)

        close.insert(min)
        for each successor s of min {
            if (s not in open and s not in close) {
                open.insert(s)
                g(s) = g(min) + cost(min, s)
                saveWay(s)
            }
            else if (g(min) + cost(min, s) < g(s)) {
                g(s) = g(min) + cost(min, s)
                saveWay(s)
                if (s in close) {
                    open.insert(s)
                    close.remove(s)
                }
            }
        }
    }
return fail
```

Dijkstra

Der Dijkstra-Algorithmus ist ein in der Graphentheorie häufig verwendeter Algorithmus zur Suche von (kürzesten) Wegen im Graphen. Es handelt sich hierbei lediglich um einen A-Stern-Algorithmus mit einer konstanten Schätzfunktion h(s):= 0. Somit ist die Schätzfunktion, unter der Voraussetzung dass alle Kantengewichte positiv sind, offensichtlich monoton und die Suche liefert einen kürzesten Weg.

Teilaufgabe 1: Datenstruktur

Hinweis: In dieser Aufgabe müssen relativ viele Klassen immplementiert werden. Die die JUnit-Tests in PABS erst dann kompilieren, wenn alle diese Klassen vorhanden sind, können sie auch erst dann sinnvoll ausgeführt werden. Testen Sie Ihre Klassen daher ausgiebig lokal. Verwenden Sie dazu u. a. sinnvoll implemetierte `toString()` Methoden der Klassen für Knoten und Kanten. Auch einen Graphen können Sie sich sinnvoll durch die Ausgabe seiner Knoten und Kanten darstellen lassen.

Hinweis: Sind in einzelnen Aufgaben nicht alle Funktionen, die durch ein Interface vorgegeben sind gefordert, implementieren Sie diese Funktionen auf sinnvolle Weise.

Nun soll eine flexible Datenstruktur für Graphen entworfen werden. .

Im Paket `jpp.digraph.graph` sind die folgenden Interfaces vorgegeben:

- `INode`: Interface eines Knotens
- `IXYNode`: Interface eines Knotens mit X und Y Koordinaten.
- `IEdge<N>`: Interface für eine Kante zwischen zwei Knoten vom Typ N.
- `ICostEdge<N>`: Interface für eine gewichtete Kante
- `IDiGraph<N, E>`: Interface eines gerichteten Graphen.

Für die Implementierung dieser Klassen werden verschiedene Exceptions benötigt, die im Paket `jpp.digraph.exceptions` zu implementieren sind:

- **Klasse `DiGraphException`** vom Typ `Exception`
- **Klasse `EdgeNotExistsException`** vom Typ `DiGraphException`
  - Konstruktoren:
    - `public EdgeNotExistsException(IEdge<? extends INode> edge)`
    - `public EdgeNotExistsException(String msg, IEdge<? extends INode> edge)`
  - Methode:
    - `public IEdge<? extends INode> getEdge()`
      Gibt die Kante zurück.
- **Klasse `InvalidEdgeException`** vom Typ `DiGraphException`
  - Konstruktoren:
    - `public InvalidEdgeException(IEdge<? extends INode> edge)`
    - `public InvalidEdgeException(String msg, IEdge<? extends INode> edge)`
  - Methode:
    - `public IEdge<? extends INode> getEdge()`
      Gibt die Kante zurück.
- **Klasse `NodeNotExistsException`** vom Typ `DiGraphException`
  - Konstruktoren:
    - `public NodeNotExistsException(INode node)`
    - `public NodeNotExistsException(String msg, INode node)`
  - Methode:
    - `public INode getNode()`

Gibt den Knoten zurück.

Implementieren Sie die oben genannten Schnittstellen nun wie folgt:

- Implementieren Sie das Interface `IXYNode` in einer **Klasse `XYNode`** im Paket `jpp.digraph.graph` mit folgenden
  - Konstruktoren
    - `public XYNode(String id, int x, int y)`
      Erzeugt einen Knoten mit ID `id` und den Koordinaten `x` und `y`. Als Beschreibung des Knoten wird ein leerer String verwendet. Die ID des Knotens soll nicht mehr veränderbar sein. Ist die übergebene ID `null` oder ein leerer String, soll eine `IllegalArgumentException` geworfen werden.
    - `public XYNode(String id, int x, int y, String description`)
      Erzeugt einen Knoten mit ID `id`, den Koordinaten `x` und `y` sowie der Beschreibung `description`. Die ID des Knotens soll nicht mehr veränderbar sein. Ist die übergebene ID `null` oder ein leerer String, soll eine `IllegalArgumentException` geworfen werden.
  - und Methoden
    - `public String getId()`
      Gibt die ID eines Knotens zurück.
    - `public String getDescription()`
      Gibt die Beschreibung eines Knoten zurück.
    - Implementieren Sie `hashCode()` und `equals()` entsprechend der Java-Konventionen. Zwei Knoten sollen dann gleich sein, wenn ihre ID gleich ist.

- Implementieren Sie das Interface `ICostEdge<N>` in einer **Klasse `CostEdge<N extends INode>`** im Paket `jpp.digraph.graph` mit folgendem
  - Konstruktoren
    - `public` CostEdge(String id, N source, N target, double cost)
      Erzeugt eine gewichtete Kante zwischen den übergebenen Knoten. Als Beschreibung der Kante wird ein leerer String verwendet. Die ID der Kante soll nicht mehr veränderbar sein. Ist die übergebene ID `null` oder ein leerer String, soll eine `IllegalArgumentException` geworfen werden.
    - `public` CostEdge(String id, N source, N target, double cost, String description)
      Erzeugt eine gewichtete Kante zwischen den übergebenen Knoten mit ID id. Die ID der Kante soll nicht mehr veränderbar sein. Ist die übergebene ID `null` oder ein leerer String, soll eine `IllegalArgumentException` geworfen werden.
  - und Methoden
    - `public String getDescription()`
      Gibt die Beschreibung einer Kante zurueck.
    - `public N getSource()`
      Gibt den Quellknoten zurück.
    - `public N getTarget()`
      Gibt den Zielknoten zurück.
    - `public double getCost()`
      Gibt die Kosten der Kante zurück
    - Implementieren Sie `hashCode()` und `equals()` entsprechend der Java-

Konventionen. Zwei Kanten sollen dann gleich sein, wenn ihre ID gleich ist.

- Implementieren Sie das Interface `IDiGraph<N, E>` in einer **Klasse `DiGraph<N extends INode, E extends IEdge<N>>`** im Paket `jpp.digraph.graph` mit folgenden
  - Konstruktoren
    - `public DiGraph()`
      Erzeugt einen leeren Graphen.
    - `public DiGraph(Collection<N> nodes)`
      Erzeugt einen Graphen mit den übergebenen Knoten. Hierbei soll eine *Kopie* der Collection erstellt werden
    - `public DiGraph(Collection<N> nodes, Collection<E> edges) throws InvalidEdgeException`
      Erzeugt einen Graphen mit den übergebenen Knoten und Kanten. Existieren die Knoten, die durch eine Kante verbunden sein sollten, nicht, so wird eine `InvalidEdgeException` geworfen. Hierbei soll jeweils eine *Kopie* der beiden Collections erstellt werden.
  - und Methoden:
    - `public int getNodeCount()`
      Gibt die Anzahl der Knoten im Graphen zurück.
    - `public int getEdgeCount()`
      Gibt die Anzahl der Kanten im Graphen zurück.
    - `public Collection<N> getNodes()`
      Gibt die Knoten des Graphen zurück.
    - `public Collection<E> getEdges()`
      Gibt die Kanten des Graphen zurück.
    - `public void setNodes(Collection<N> nodes)`
      Setzt die Knoten des Graphen. Hierzu werden erst alle Knoten und Kanten aus dem Graphen gelöscht, dann werden die neuen Knoten aus der übergebenen Collection gesetzt.
    - `public void setEdges(Collection<E> edges) throws InvalidEdgeException`
      Setzt die Kanten des Graphen. Hierzu werden erst alle Kanten aus dem Graphen gelöscht, und dann die neuen Kanten aus der übergebenen Collection gesetzt. Wirft eine `InvalidEdgeException`, wenn eine Kante nicht zu den Knoten im Graphen passt.
    - `public void setGraph(Collection<N> nodes, Collection<E> edges) throws InvalidEdgeException`
      Setzten der Knoten und Kanten des Graphen. Hierzu werden erst alle Knoten und Kanten aus dem Graphen gelöscht, und dann die neuen Knoten und Kanten aus den Collections gesetzt. Wirft eine `InvalidEdgeException`, wenn eine Kante nicht zu den Knoten passt.
    - `public void removeNodes()`
      Löscht alle Knoten aus dem Graphen, und somit auch alle Kanten.
    - `public void removeEdges()`
      Löscht alle Kanten aus dem Graphen.
    - `public boolean containsNode(N node)`

Prüft, ob ein Knoten im Graph enthalten ist.

- `public boolean containsEdge(E edge)`
  Prüft, ob eine Kante im Graphen enthalten ist.
- `public boolean containsEdge(N source, N target)`
  Prüft ob zwischen zwei Knoten mindestens eine Kante im Graph existiert.
- `public void addNode(N node)`
  Fügt einen Knoten zum Graphen hinzu. Existiert der Knoten bereits im Graph, soll er nicht erneut hinzugefügt werden.
- `public void addEdge(E edge) throws InvalidEdgeException`
  Fügt eine Kante zum Graphen hinzu. Wirft eine `InvalidEdgeException`, wenn die Kante nicht zu den Knoten im Graphen passt. Existiert die Kante bereits im Graph, soll sie nicht erneut hinzugefügt werden.
- `public void removeNode(N node) throws NodeNotExistsException`
  Löscht einen Knoten und die mit ihm verbundenen Kanten aus dem Graphen. Wirft eine `NodeNotExistsException`, wenn der Knoten nicht existiert.
- `public void removeEdge(E edge) throws InvalidEdgeException, EdgeNotExistsException`
  Löscht eine Kante aus dem Graphen. Wirft eine `EdgeNotExistsException`, wenn die Kante nicht im Graph existiert. Wirft eine `InvalidEdgeException`, wenn die Kante nicht zu den Knoten im Graphen passt.
- `public Collection<N> getPredecessors(N node) throws NodeNotExistsException`
  Gibt alle Vorgängerknoten eines Knotens zurück. Wirft eine `NodeNotExistsException`, wenn der Knoten nicht im Graph existiert.
- `public Collection<E> getPredecessorEdges(N node) throws NodeNotExistsException`
  Gibt alle Kanten von Vorgängerknoten zum Knoten zurück. Wirft eine `NodeNotExistsException`, wenn der Knoten nicht im Graph existiert.
- `public Collection<N> getSuccessors(N node) throws NodeNotExistsException`
  Gibt alle Nachfolgerknoten eines Knoten zurück. Wirft eine `NodeNotExistsException`, wenn der Knoten nicht im Graph existiert.
- `public Collection<E> getSuccessorEdges(N node) throws NodeNotExistsException`
  Gibt alle Kanten vom Knoten zu Nachfolgerknoten zurück.Wirft eine NodeNotExistsException, wenn der Knoten nicht im Graph existiert.
- `public Collection<E> getEdgesBetween(N source, N target) throws NodeNotExistsException`
  Gibt alle direkten Kanten von einem Knoten zum Nächsten zurück. Wirft eine `NodeNotExistsException`, wenn einer der Knoten nicht im Graph existiert.

Teilaufgabe 2: Ein-/Ausgabe (Persistenz)

Nun soll die Funktionalität zur Ein- und Ausgabe bzw. zum Speichern und Laden dieser Datenstrukturen (Persistenz (https://de.wikipedia.org/wiki/Persistenz_%28Informatik%29)) implementiert werden. Eine Möglichkeit dazu ist, Java-Objekte in XML-Dokumente (marshalling (https://de.wikipedia.org/wiki/Marshalling)) bzw. XML-Dokumente in Java-Objekte (unmarshalling) umzuwandeln. Es bietet sich hier die auf XML (http://www.w3.org/XML) aufbauende Graph eXchange Language (http://www.gupro.de/GXL) an. Wir werden uns aber nur auf die folgende Teilmenge der GXL - Document Type Definition (DTD) (http://www.gupro.de/GXL/dtd/dtd .html) beschränken:

DTD

```
<!ELEMENT gxl  (graph)>

<!ELEMENT graph ((node | edge)*) >
<!ATTLIST graph
    id          ID       #REQUIRED
>

<!ELEMENT node (attr*) >
<!ATTLIST node
    id          ID       #REQUIRED
>

<!ELEMENT edge (attr*) >
<!ATTLIST edge
    id            ID       #IMPLIED
    from          IDREF    #REQUIRED
    to            IDREF    #REQUIRED
>

<!ELEMENT bool    (#PCDATA) >
<!ELEMENT int     (#PCDATA) >
<!ELEMENT float   (#PCDATA) >
<!ELEMENT string  (#PCDATA) >

<!ELEMENT attr (bool|int|float|string) >
<!ATTLIST attr
    id     ID       #IMPLIED
    name   NMTOKEN  #REQUIRED
>
```

Im Rahmen dieser Aufgabe können Sie davon ausgehen, dass das bei Elementen des typs `edge` optionale Attribute `id` immer gesetzt ist.

Um die Ein-/Ausgabe für die Datenstruktur zu realisieren verwenden wir das Entwurfsmuster (http://de.wikipedia .org/wiki/Entwurfsmuster) Schablone (http://de.wikipedia.org/wiki/Schablonenmethode). Dazu ist im Paket `jpp.digraph.io` das Interface `IGXLSupport<G, N, E>` gegeben

Implementieren Sie das Interface `IGXLSupport<G, N, E>` in einer abstrakten **Klasse `GXLSupport<G extends IDiGraph<N,E>, N extends INode, E extends IEdge<N>>`** im Paket `jpp.digraph.io` mit folgenden Methoden:

- `public G read(InputStream is) throws ParserConfigurationException, IOException, SAXException, InvalidEdgeException`
  Liest einen mittels Graph eXchange Language (http://www.gupro.de/GXL/) definierten Graphen über DOM (http://de.wikipedia.org/wiki/Document_Object_Model) von einen InputStream ein und gibt eine Graph-Datenstruktur vom Typ `G` zurück. Hierbei sollen die abstrakten Methoden `createGraph()`, `createNode(org.w3c.dom.Node element)` und `createEdge(org.w3c.dom.Node element)` verwendet werden.
- `public void write(G graph, OutputStream os) throws ParserConfigurationException, IOException, TransformerConfigurationException, TransformerException`
  Schreibt eine Graph-Datenstruktur vom Typ `G` im Format der Graph eXchange Language (http://www.gupro.de/GXL/) auf einen OutputStream. Hierbei sollen die abstrakten Methoden `createElement(org.w3c.dom.Document doc, N node)` und `createElement(org.w3c.dom.Document doc, E edge)` verwendet werden.
- `public abstract G createGraph()`
  Abstrakte Methode welche eine neue leere Graph-Datenstruktur vom Typ `G` zurückgeben soll.
- `public abstract N createNode(org.w3c.dom.Node element)`
  Abstrakte Methode welche aus einem `org.w3c.dom.Node` einen Knoten vom Typ `N` erzeugen soll.
- `public abstract E createEdge(org.w3c.dom.Node element)`
  Abstrakte Methode welche aus einem `org.w3c.dom.Node` eine Kante vom Typ `E` erzeugen soll.
- `public abstract org.w3c.dom.Element createElement(org.w3c.dom.Document doc, N node)`
  Abstrakte Methode welche aus einem Knoten vom Typ `N` mittels `doc` ein `org.w3c.dom.Element` erzeugen soll.
- `public abstract org.w3c.dom.Element createElement(org.w3c.dom.Document doc, E edge)`
  Abstrakte Methode welche aus eine Kante vom Typ `E` mittels `doc` ein `org.w3c.dom.Element` erzeugen soll.

Hinweis:

Verwenden Sie für das Einlesen des Graphen im Format Graph eXchange Language den `javax.xml.parsers.DocumentBuilder` wie folgt:

- Dokument vom `InputStream is` lesen:
  ```
  DocumentBuilder builder =
  DocumentBuilderFactory.newInstance().newDocumentBuilder();
  Document doc = builder.parse(is);
  ```
- Wurzelelement aus dem Dokument lesen:
  ```
  Node root = doc.getDocumentElement();
  ...
  ```

Verwenden Sie zum Schreiben des Graphen den `javax.xml.transform.Transformer` wie folgt:

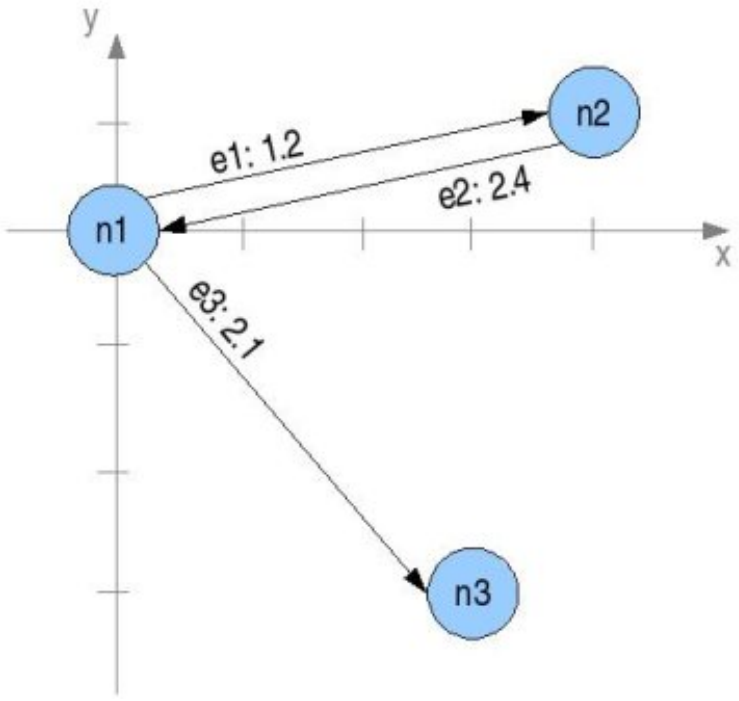
- neues Dokument erzeugen:
  ```
  DocumentBuilder docBuild =
  DocumentBuilderFactory.newInstance().newDocumentBuilder();
  ```

```
Document doc = docBuild.newDocument();
```

- Wurzelelement erzeugen und einfügen:
```
Element root = doc.createElement("gxl");
doc.appendChild(root);
...
```
- Dokument auf den OutputStream os schreiben
```
DOMSource domSource = new DOMSource(doc);
TransformerFactory tf = TransformerFactory.newInstance();
Transformer transformer = tf.newTransformer();
transformer.setOutputProperty(OutputKeys.METHOD, "xml");
transformer.setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
transformer.transform(domSource, new StreamResult(os));
```

Nun erweitern wir unser Grundgerüst für die Ein-/Ausgabe. Schreiben Sie hierfür eine **Klasse** **`XYGXLSupport`** im Paket `jpp.digraph.io`, welche von der abstrakten Klasse `GXLSupport<DiGraph<XYNode, CostEdge<XYNode>>, XYNode, CostEdge<XYNode>>` erbt. An Konstruktoren wird in dieser Klasse lediglich der Default-Konstruktor benötigt.

Nachfolgend ein Beispiel für eine Speicherung von `XYNode` und `CostEdge<XYNode>` im Format der Graph eXchange Language.



Beispiel: XML-Export

```
<gxl>
    <graph id="id1">
        <node id="id2">
            <attr name="description">
                <string>n1</string>
            </attr>
```

```
        <attr name="x">
            <int>0</int>
        </attr>
        <attr name="y">
            <int>0</int>
        </attr>
    </node>
    <node id="id3">
        <attr name="description">
            <string>n2</string>
        </attr>
        <attr name="x">
            <int>4</int>
        </attr>
        <attr name="y">
            <int>1</int>
        </attr>
    </node>
    <node id="id4">
        <attr name="description">
            <string>n3</string>
        </attr>
        <attr name="x">
            <int>3</int>
        </attr>
        <attr name="y">
            <int>-3</int>
        </attr>
     </node>
    <edge from="id2" id="id5" to="id3">
        <attr name="description">
            <string>e1</string>
        </attr>
        <attr name="cost">
            <float>1.2</float>
        </attr>
    </edge>
    <edge from="id2" id="id6" to="id4">
        <attr name="description">
            <string>e3</string>
        </attr>
        <attr name="cost">
            <float>2.1</float>
        </attr>
    </edge>
    <edge from="id3" id="id7" to="id2">
```

```
            <attr name="description">
                <string>e2</string>
            </attr>
            <attr name="cost">
                <float>2.4</float>
            </attr>
        </edge>
    </graph>
</gxl>
```

Teilaufgabe 3 - Suche

In diesem Aufgabenblock wollen wir nun die Suchalgorithmen auf der Datenstruktur aus Teilaufgabe 2 implementieren. Auch hier verwenden wir für die Implementierung wieder das Schablonenmuster (http://de.wikipedia.org/wiki/Schablonenmethode).

Dazu ist das Interface `IDiGraphSearch<G, N, E>` im Paket `jpp.digraph.search` gegeben:

- `public List<E> search(G graph, N source, N target) throws NodeNotExistsException`
  Gibt einen Weg vom Quellknoten zum Zielknoten zurück. Wirf eine `NodeNotExistsException`, wenn einer der übergebenen Knoten nicht im Graph existiert.
- `public void addListener(IDiGraphSearchListener<N, E> listener)`
  Fügt einen IDiGraphSearchListener hinzu.
- `public void removeListener(IDiGraphSearchListener<N, E> listener)`
  Entfernt einen Listener.
- `public Collection<IDiGraphSearchListener<N, E>> getListeners()`
  Gibt die Listener zurück.

Für die Visualisierung des Suchalgorithmus in Teilaufgabe 4 notwendig, dem Algorithmus sogenannte Callback-Methoden (http://de.wikipedia.org/wiki/R%C3%BCckruffunktion) zu übergeben, welche dann beim Expandieren eines Knotens aufgerufen werden. Diese Callback-Methoden werden über das Interfaces `IDiGraphSearchListener<N, E>` im gleichen Paket realisiert.

Hinweis: Ein Knoten wird genau dann expandiert, wenn er im entsprechenden Pseudocode der Suchalgorithmen aus der jeweiligen Datenstruktur herausgenommen und untersucht wird. Dies ist in den angegebenen Algorithmen jeweils die Zeile:

- Breitensuche:
  - `c = queue.dequeue()`
- Tiefensuche:
  - `c = stack.pop()`
- A*-Suche**:**
  - `remove min with the smallest f-value from open`

Somit werden auch, wie man in den Beispielen bei der Algorithmenbeschreibung sieht, sowohl der Start- als auch der Endknoten expandiert.

Implementieren Sie nun das Interface `IDiGraphSearch<G, N, E>` in folgenden Klassen:

- **Klasse `BFS<G extends IDiGraph<N, E>, N extends INode, E extends ICostEdge<N>>`** im Paket `jpp.digraph.search`.
  Dabei soll die Methode `search(G graph, N source, N target)` aus dem Interface `IDiGraphSearch<G, N, E>` die **Breitensuche** implementieren.
- **Klasse `DFS<G extends IDiGraph<N, E>, N extends INode, E extends ICostEdge<N>>`** im Paket `jpp.digraph.search`.
  Dabei soll die Methode `search(G graph, N source, N target)` aus dem Interface `IDiGraphSearch<G, N, E>` die **Tiefensuche** implementieren.
- Abstrakte **Klasse `AbstractAStar<G extends IDiGraph<N, E>, N extends INode, E extends ICostEdge<N>>`** im Paket `jpp.digraph.search`.
  Dabei soll die Methode `search(G graph, N source, N target)` aus dem Interface `IDiGraphSearch<G, N, E>` die **A*-Suche** implementieren, wobei hierfür die Schätzfunktion der Heuristik des Algorithmus die abstrakte Methode `public abstract double h(N node, N target)` der Klasse ist.

Hinweis: Stellen Sie in allen zu implementierenden Suchalgorithmen mindestens den Default-Konstruktor zur Verfügung.

Hinweis: Berücksichtigen Sie auch die durch das Interface `IDiGraphSearchListener<N, E>` bei den Algorithmen registrierten Callback-Methoden.

Hinweis: Eine Implementierung mit bestmöglicher Laufzeit von O(n) = nlog(n) der A*-Suche ist z.B. mithilfe der Klasse java.util.TreeSet<E> möglich. Beachten Sie, dass für eine korrekte Arbeitsweise des TreeSets insbesondere die Java-Konventionen von compareTo() beachtet werden müssen. Insbesondere dürfen 2 Objekte nur genau dann bei Aufruf der compareTo-Methode 0 zurückliefern, wenn der Aufruf von equals() ein true zurückgeben würde. Es reicht also u.U. nicht aus, Punkte nur nach ihren Koordinaten zu sortieren, ggf. müssen weiteren Faktoren wie die ID beachtet werden. Das gilt insbesondere auch für selbstgeschriebene (Hilfs-)Klassen.

Erweitern Sie die abstrakte Klasse `AbstractAStar<G, N, E>` in den **Klassen `Dijkstra<G extends IDiGraph<N, E>, N extends INode, E extends ICostEdge<N>>`** und **`XYAStar<G extends IDiGraph<N, E>, N extends IXYNode, E extends ICostEdge<N>>`** im Paket `jpp.digraph.search`. Implementieren Sie hierbei die Schätzfunktion `h(N node, N target)` für die Klasse `XYAStar` als Abstand von node zu target im Koordinatensystem.

Beispiel Graph

Hier (/courses/23100/assignments/9793700/user/s344360/instructions/resources/base/sample.gxl) können Sie eine Beispiel-GXL-Datei herunterladen.

Teilaufgabe 4 - GUI

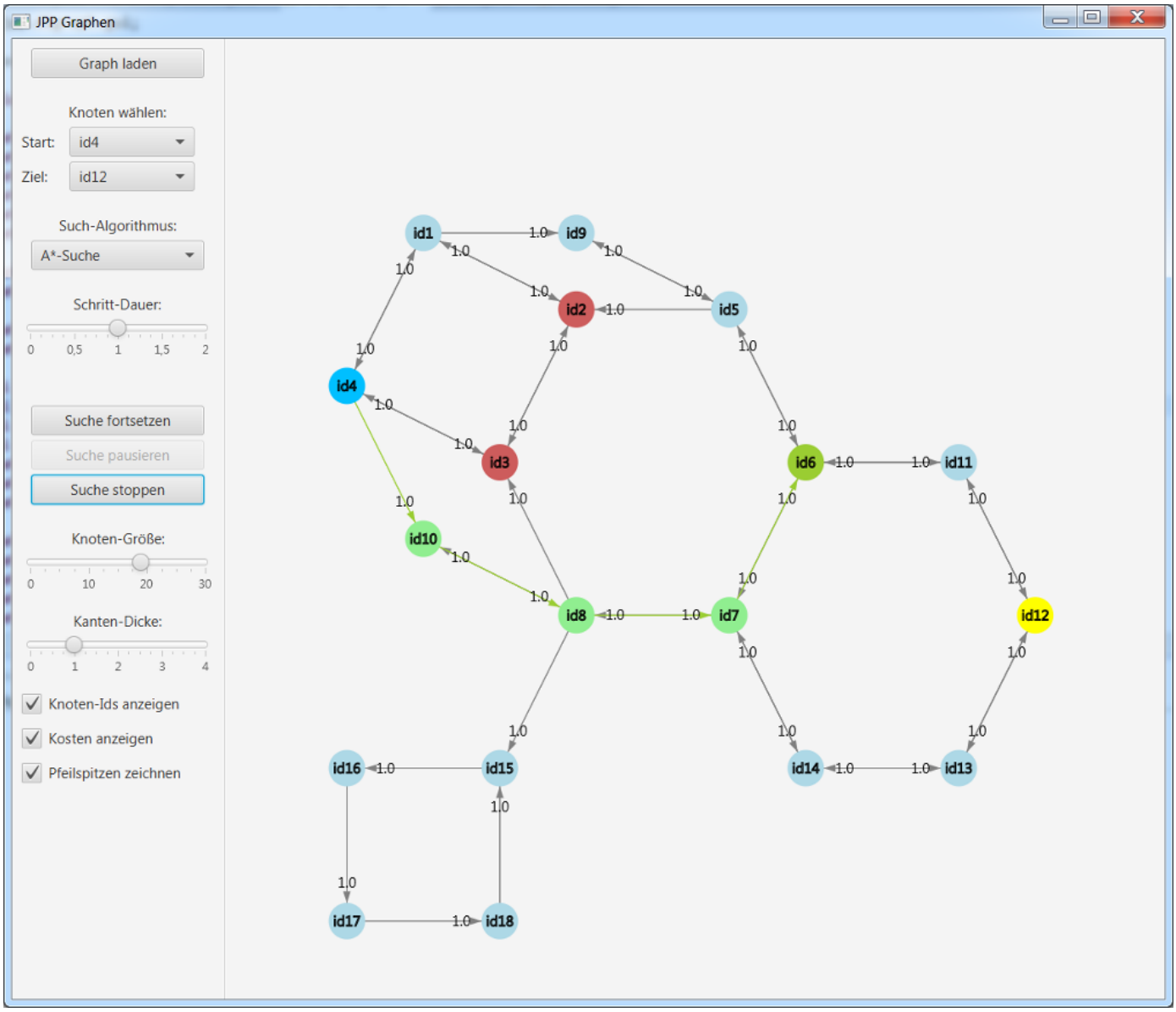
Im letzten Aufgabenblock wollen wir nun für die Datenstruktur aus Teilaufgabe 1 und die Suchalgorithmen aus Teilaufgabe 3 eine grafische Benutzeroberfläche mit JavaFX entwickeln. Schreiben Sie hierfür eine **Klasse `DiGraphGUI`** im Paket `jpp.digraph.gui` mit folgenden Eigenschaften:

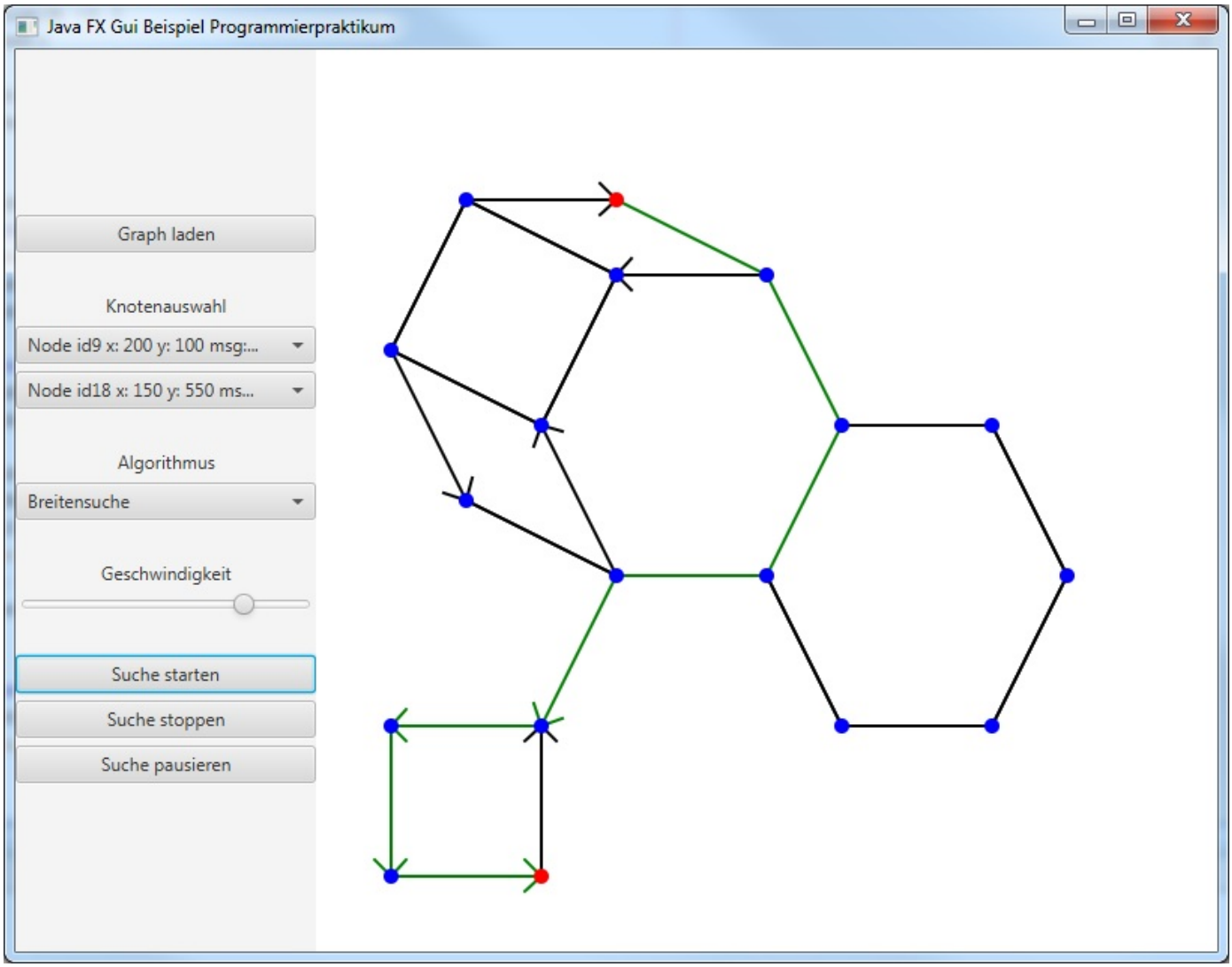
- Es ist möglich, Graphen mittels `XYGXLSupport` aus Teilaufgabe 2 zu laden.
- Die Knoten und Kanten (inklusive Richtung) werden auf der Oberfläche angezeigt.
- Der Graph soll auf die initiale Größe der Oberfläche skaliert werden. Die Koordinaten eines `XYNode`

müssen also als relativ betrachtet werden. (Die Größe des Graphen muss sich dabei nicht zwingend auf eine nachträgliche Veränderung der Größe der Oberfläche anpassen.)
- Es ist möglich, Start- und Zielknoten für die Wegsuche auszuwählen.
- Es ist möglich, aus den vier in Teilaufgabe 3 implementierten Suchalgorithmen Breitensuche, Tiefensuche, A-Stern und Dijkstra auswählen und damit den Graphen durchsuchen.
- Die Suche im Graphen wird schrittweise mittels `IDiGraphSearchListener<N extends INode, E extends IEdge<N>>` grafisch dargestellt. Hierfür soll die Verzögerung einstellbar sein.
- Der gefundene Weg wird grafisch dargestellt.

Hinweis: Für die Implementierung der Verzögerung kann es nützlich sein, die IDiGraphSearchListener nicht in einem ausgelagertem Thread aufzurufen, damit die gesamte Suche warten kann. Dabei muss jedoch beachtet werden, dass der Thread der Suche nicht durch einen Fehler in der GUI zum Absturz gebracht wird.
Um busy waiting (https://de.wikipedia.org/wiki/Aktives_Warten) zu reduzieren, empfiehlt sich die Benutzung der Thread.yield() Methode. Erläuterungen zu den deprecated Methoden Thread.suspend() und Thread.resume() finden sich hier (https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html).

Für die weitere Gestaltung der Oberfläche gibt es keine festen Vorgaben. Die hier angezeigten Lösungen soll nur einen Anhaltspunkt liefern.


JPP Graphen
Graph laden
Knoten wählen:
Start:
id4
Ziel:
id12
Such-Algorithmus:
A*-Suche
Schritt-Dauer:
0
0,5
1
1,5
2
Suche fortsetzen
Suche pausieren
Suche stoppen
Knoten-Größe:
0
10
20
30
Kanten-Dicke:
0
1
2
3
4
Knoten-Ids anzeigen
Kosten anzeigen
Pfeilspitzen zeichnen
id1
id9
id2
id5
id4
id3
id6
id11
id10
id8
id7
id12
id16
id15
id14
id13
id17
id18
1.0



Viel Erfolg und viel Spaß!

PABS 3.1 - University of Würzburg