JAGAL Tutorial

Documentation & Code Examples for the Java Graph Library

Adrian Lange

1 Introduction

The Java Graph Library (JAGAL) is a Java library for modelling directed graphs. It comes with implementations of various types of graphs and transition systems, as well as utilities for their modification and traversal.

Its key features include among others:

- Implementation of directed graphs
- Implementation of directed weighted graphs
- Graph algorithms (Tarjan for SCCs)
- Graph visualization (Circle layout, topological layout)
- Graph traversal (depth-first-search, breadth-first-search)
- Traversal utilities (Predecessors, Siblings, Cycles, ...)
- Implementation of Transition Systems
- Implementation of Labelled Transition Systems

This document is a mix of a programming tutorial and a library documentation, where the features get demonstrated with some visualisations and minimal programming code examples.

The examples in this document have been tested against the JAGAL release version 1.0.0, which can be found at `https://github.com/iig-uni-freiburg/JAGAL`. This document was last updated on July 29, 2015.

1.1 Library Dependencies

JAGAL builds upon the Java library TOVAL, which is a set of Java classes for common programming issues. It is located under `https://github.com/GerdHolz/TOVAL` and must be added to the class path to be able to use JAGAL.

For the visualization of graphs, JAGAL uses the JGraphX library, which is a Java Swing diagramming library specialized on node-edge graphs. JGraphX can be downloaded under `https://github.com/jgraph/jgraphx`.

1.2 Package Structure

The packages in `de.uni.freiburg.iig.telematik.jagal` are logically divided into the following sub-packages. Methods of data structure defining classes like graphs and transition systems are always defined in abstract classes with generics to keep constraints between data types and to allow users to easily define their own sub-classes. Through using generics, the types of the vertices and edges can be set as needed. Their concrete subclasses like `Graph` or `TransitionSystem` only contain their constructors and need only to specify the vertex data type over generics. This way, clarity in the data structure and an intuitive depth of inheritance is ensured.

- The package `graph` contains all classes which are needed to define a graph structure. Weighted graphs can be found in the sub-package `weighted`.
- The `traverse` package contains both classes and interfaces for traversals through data structures and algorithms using traversal classes.
- Classes in the package `visualization` can be used to visualize graph structures. For each structure, a corresponding component class is defined.
- Transition systems are defined in the package `ts`. For labelled transition systems there is another sub-package named `labeled`. Two more packages contain classes for serializing and parsing transition systems.

2 Basic Graphs

Although JAGAL has some visualization capabilities, we first consider underlying classes and interfaces.

All graph classes build upon the abstract class `AbstractGraph<V,E,U>`, where `V` is the vertex type, `E` the type of edges, and `U` the type of the element enclosed by the vertex. The class defines basic operations, which can be performed on graphs, like adding and removing vertices and edges, getting meta information about a graph and its components like the in- and out-degree of a vertex, its successors etc., and getting topological information about a graph and its components.

The basic classes for vertices and edges are called `Vertex<U>` and `Edge<V>`, which depend on the vertex type. Based on these classes, two graph types are predefined:

1. A *directed unweighted graph* only allows directed edges, which aren't weighted. Its main class is `Graph<U>`, where `U` defines the type of the vertex labels.
2. A *directed weighted graph* complements the unweighted graph by edge weights, which are floating number values. The class `WeightedGraph<U>` therefore uses the class `WeightedEdge<V>` for the edges.

In the following sections some simple examples for the visualization of the graphs and use cases with graph algorithms using these classes are presented.

2.1 Creating a Directed Graph

As an example a simple directed graph is created. As type of the vertex elements we choose `Integer`.

```
// Graph<U> where U is the type of the vertex elements
Graph<Integer> g = new Graph<Integer>();
// Add some vertices
g.addVertex("A", 4);
g.addVertex("B", 3);
g.addVertex("C", 1);
g.addVertex("D");
g.getVertex("D").setElement(2);
g.addVertex("E", 8);
// Add some edges
g.addEdge("A", "B");
g.addEdge("B", "C");
g.addEdge("B", "D");
g.addEdge("C", "A");
g.addEdge("C", "D");
g.addEdge("D", "E");
g.addEdge("E", "B");
// Let's take a look at the result
System.out.println(g);
System.out.println(g.getVertex("E").getElement());
```

Printing out the graph results in the following output:

```
Graph: V=[A, B, C, D, E]
       E=[(A -> B), (B -> C), (B -> D), (C -> A),
          (C -> D), (D -> E), (E -> B)]
```

Note that vertices are created with their name. When creating edges, they also refer to the vertex names instead of the vertex objects. If a vertex with a given name already exists, the `addVertex()`-method does not add a new vertex and returns `false`. If an unknown vertex name is referred to when creating an edge, the `addEdge`-method throws a `VertexNotFoundException`.

Now it is possible to request some meta information from the graph. For example the in-degree of the vertex `D` can be retrieved using `g.inDegreeOf("D")` (which would result in 2) and the method `g.isDrain("A")` returns `false`, since the vertex `A` has a successor.

2.2 Creating a Directed Weighted Graph

JAGAL already comes with a class for directed weighted graphs. In this section, we create a simple graph with weighted edges. Weights are floating number values. By default all edges have a weight of 1.0.

```
// WeightedGraph<U> where U is the vertex element type
WeightedGraph<Integer> gw = new WeightedGraph<Integer>();
// Add some vertices
gw.addVertex("A");
gw.addVertex("B");
gw.addVertex("C");
gw.addVertex("D");
gw.addVertex("E");
```

```
// Add some edges
gw.addEdge("A", "B");
gw.addEdge("B", "C");
gw.addEdge("B", "D", 2.0);
gw.addEdge("C", "A", 3.4);
gw.addEdge("C", "D");
gw.addEdge("D", "E");
gw.addEdge("E", "B");
// Let's take a look at the result
System.out.println(gw);
```

The output of the weighted graph is the following:

```
Graph: V=[D, E, A, B, C]
       E=[(A-1.0->B), (B-1.0->C), (B-2.0->D), (C-3.4->A),
          (C-1.0->D), (D-1.0->E), (E-1.0->B)]
```

2.3 Visualizing Graphs

The JAGAL library also comes with visualization capabilities. With the following command a new JFrame gets created, where the class `DisplayFrame` is part of the TOVAL library (see figure 1a):

```
new DisplayFrame(new GraphComponent(g), true);
```

For the visualization of weighted directed graphs, JAGAL comes with another component class named `WeightedGraphComponent`. It complements the standard graph representation by edge labels with the specified weights. The resulting graph visualization can be seen in figure 1b.

```
new DisplayFrame(new WeightedGraphComponent(gw), true);
```

3 Graph Traversal

Graphs implement the interface `Traversable`, which has methods to get the predecessors and successors of a vertex and thus gives the opportunity to traverse a graph.

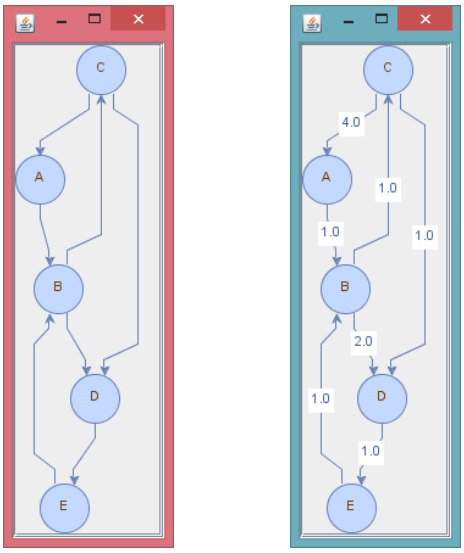
This section will show some examples of graph algorithms using traversal.

3.1 Tarjan's Strongly Connected Components

As a use case, the traversal package contains an implementation of Tarjan's SCC algorithm, which divides a graph in its strongly connected components. A graph component is called strongly connected, if all pairs of vertices inside a component are reachable by each other.

As an example we create a new graph and run Tarjan's algorithm on it:

```
Graph<Integer> gt = new Graph<Integer>();
gt.addVertices(Arrays.asList("A", "B", "C", "D", "E", "F"));
gt.addEdge("A", "B"); gt.addEdge("B", "C"); gt.addEdge("C", "A");
gt.addEdge("D", "B"); gt.addEdge("D", "C"); gt.addEdge("D", "E");
```



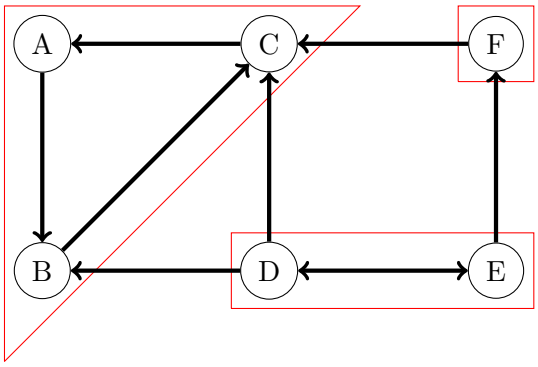
(a) (b)

Figure 1: Visualization of an instance of the `Graph` class and an instance of the `WeightedGraph` class.

```
gt.addEdge("E", "D"); gt.addEdge("E", "F"); gt.addEdge("F", "C");

SCCTarjan tarjan = new SCCTarjan();
System.out.println(tarjan.execute(g));
```

The method `getStronglyConnectedComponents()` in the class `TraversalUtils` also uses this algorithm. It results in the following three components:

```
[[A, B, C], [D, E], [F]]
```



3.2 Custom Algorithm Using Traversal: Weakly Connected Graph

Besides using existing algorithms, it is also possible to use the `Traversal` interface to define custom algorithms. In this example, we want to know whether a given graph is weakly connected. A directed graph is called weakly connected if all pairs of edges are connected ignoring the direction of an edge.

The property can be retrieved in linear time by checking if all vertices are reachable over a weak connection from one random vertex. For this, we need a method, which recursively traverses all neighbours of a node. Neighbours can be retrieved by merging children and parents of a vertex. Vertices which are reachable from the starting vertex are stored in a set `nodes`.

```
private static <V extends Object> void weakConnectivityRec(
        Traversable<V> graph, V v, Set<V> nodes) {
    Set<V> neighbours = new HashSet<V>();
    neighbours.addAll(graph.getChildren(v));
    neighbours.addAll(graph.getParents(v));
    for (V n : neighbours) {
        if (false == nodes.contains(n)) {
            nodes.add(n);
            weakConnectivityRec(graph, n, nodes);
        }
    }
}
```

The algorithm collects all weakly connected vertices from an arbitrary vertex. A graph is weakly connected, if the set of weakly connected vertices is as large as the set of vertices in the graph. Since the set of connected nodes also contains the start node, we just test if this set is smaller than the set of all vertices.

```
public static <V extends Object> boolean isWeaklyConnected(
        Traversable<V> graph) {
    for (V node : graph.getNodes()) {
        Set<V> nodes = new HashSet<V>();
        weakConnectivityRec(graph, node, nodes);
        if (nodes.size() < graph.getNodes().size())
            return false;
        break;
    }
    return true;
}
```

Though this code snippet looks unnecessarily confusing, the outer loop, which gets interrupted after the first iteration, is needed to retrieve an element from a `Collection` data type.

The class `TraversalUtils` also contains methods to retrieve meta information of a graph and its components. To check if a graph is strongly connected, we can use the static method `isStronglyConnected`. Furthermore it's worth to take a look at the method implementations in that class, since they are using the `Traverser` class, which already implements the depth-first-search and the breadth-first-search. As a consequence the implemented methods stay very compact.

3.3 Coloring

A coloring of graphs assigns a color to each vertex, such that no neighboured vertices have the same color. Graph coloring algorithms try to get along with a minimum number of

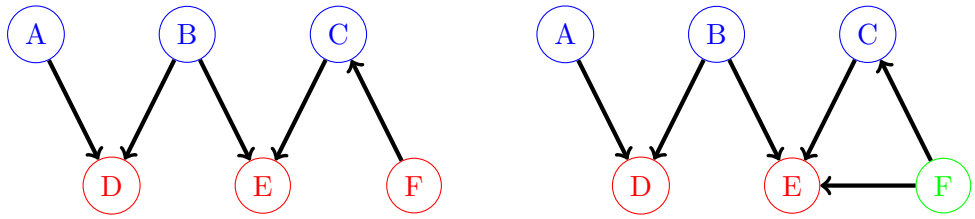
different colors. If a graph can be colored using k different colors, it is called a k-coloring graph. For example it is possible to color a world map with four different colors, such that no neighboured countries have the same color (*Four color theorem*).

The `Coloring` class represents a coloring of vertices. The class `GraphColoringFactory` has a method `exactGreedyColoring` to create a coloring for a graph.

Graph coloring can be used to determine whether a graph is bipartite i.e. it can be separated in two disjoint groups of vertices, which are not adjacent among themselves. In this case, there must be a coloring for the graph with only two different colors:

```
public static <V extends Object> boolean isBipartite(Graph<V> g) {
    Coloring c = GraphColoringFactory.exactGreedyColoring(g);
    return c.getColorGroups().size() == 2;
}
```

The following two graphs show examples of the algorithm. The left one is bipartite, since it can be colored using two colors, where The right graph needs three colors and therefore is not bipartite.



4 Transition Systems

Besides graph representation and traversal, JAGAL also considers transition systems. Transition systems are used in the computation and automata theory and describe possible states in a state-based system. Relations between the states are called transitions.

A transition system has a non-empty set of start states and a set of final states.

Transition systems can be represented by directed graphs, where the vertices are states connected by transitions. The edges can either have labels or not.

In the following sections we will show the different types of transition systems as well as JAGAL's serialization and parsing functionalities.

4.1 Unlabelled Transition Systems

A good example for transition system without transition labels is a finite-state machine representing a traffic light. It has the states $S = \{r, ry, g, y\}$, where r, y, and g stand for the colors red, yellow, and green. While operating the traffic lights change the states in the following order, where all of the states can be start states:

$$r \to ry \to g \to y \to r$$

In the JAGAL framework, this transition system can be built in the following way:

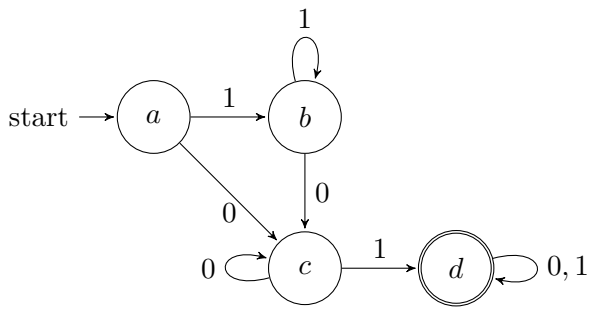
```
TransitionSystem ts = new TransitionSystem();

ts.addState("r");
ts.addState("ry");
ts.addState("y");
ts.addState("g");
ts.addStartState("r");

ts.addRelation("r", "ry");
ts.addRelation("ry", "g");
ts.addRelation("g", "y");
ts.addRelation("y", "r");
```

Since the class `TransitionSystem` inherits from `AbstractGraph` and thus implements the `Traversable` interface, we can traverse through a transition system and also use all the methods we can use on graphs.

4.2 Labelled Transition Systems

Transition systems can be complemented by transition labels. In the graph representation, transition labels are edge labels. Labels can represent different things depending on the user's interest. Often, they are seen as conditions that must hold true or events that must be triggered to reach the next state.

As an example we want to build a deterministic finite automata (DFA) with the alphabet $\Sigma = \{0, 1\}$, that only accepts words containing the symbol sequence 01. The corresponding transition system looks like the following:



The transition system can also be represented in JAGAL:

```
LabeledTransitionSystem l = new LabeledTransitionSystem();

l.addState("a");
l.addState("b");
l.addState("c");
l.addState("d");

l.addStartState("a");
l.addEndState("d");

l.addEvent("0");
```

```
l.addEvent("1");

l.addRelation("a", "b", "1");
l.addRelation("a", "c", "0");
l.addRelation("b", "b", "1");
l.addRelation("b", "c", "0");
l.addRelation("c", "c", "0");
l.addRelation("c", "d", "1");
l.addRelation("d", "d", "0");
l.addRelation("d", "d", "1");
```

Since we defined exactly one start state and the transition system is deterministic, the finite-state machine is a DFA. This property can also be checked by using the `isDFA()` method. With the method `acceptsSequence(String... s)` we can check if an input sequence gets accepted by the DFA or not:

```
System.out.println(l.acceptsSequence("0", "0", "1", "0"));
// true

System.out.println(l.acceptsSequence("1", "1", "0", "0"));
// false
```

Once again we can use the visualization classes to generate a graphical output of the transition system (see figure 2):

```
new DisplayFrame(new LabeledTransitionSystemComponent(l), true);
```

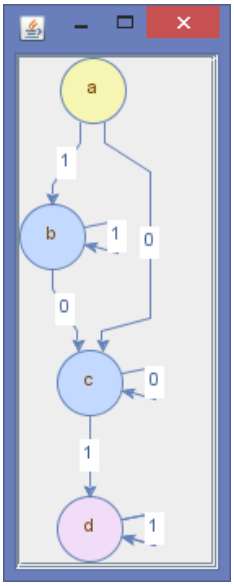


Figure 2: Visualization of an instance of the `LabeledTransitionSystem` class.

4.3 Serializing Transition Systems

To store transition systems and make them available for other users and tools, the JAGAL framework comes with a serialization functionality. Although the serializing classes are prepared for supporting different output formats, currently only the Petrify file format is supported. The serializer takes a transition system, the target format and the path as argument and serializes the transition system.

```
TSSerialization.serialize(l, TSSerializationFormat.PETRIFY,
                          "test-ts", "/arbitrary/path");
```

The labelled transition system gets stored under the file name `test-ts.sg`. Since `l` refers to the transition system from the last subsection, the generated file contains the following information (a `#` indicates the beginning of a comment):

```
.outputs 1 0  # Labels
.state graph
a 1 b         # Transitions
a 0 c
b 1 b
b 0 c
c 0 c
c 1 d
d 0 d
d 1 d
.marking {a}  # Start marking
.final {d}    # Final states
.end
```

4.4 Parsing Transition System

Transition systems in the Petrify file format can also be parsed using JAGAL's built in parser. It recognizes the file type by the file extension and returns an instance of `AbstractLabeledTransitionSystem`.

```
AbstractLabeledTransitionSystem lts = TSParser.parse(
        new File("/arbitrary/path/test-ts.sg"));
```

The resulting transition system can be used just like any other transition system object we defined by hand.