Visualization of Dijkstra's algorithm

Algorithmics Lab, winter term 2007

Uwe Schmidt

April 27, 2008

1 Task

We were given the task to develop a program which interactively visualizes Dijkstra's algorithm for beginners. We were not supposed to focus on rigorous teaching of the algorithm. First of all, the program should be appealing to people without prior knowledge of computer science. It should keep their attention and convey the message that computers actually make our daily live easier (at least in some regards) by putting Dijkstra's algorithm into context.

Some of the technical requirements were to provide a graph editor with additional random graph generation and the possibility to save and load graphs. The user should have the possibility to step through the algorithm (forward and backward). Each step the progress made by the algorithm should be reflected by coloring the graph and presenting detail information (on demand).

2 Rationale behind my solution

The program starts up and shows an introduction window that shows an entry to the topic for the novice user. The text tries to have an informal dialog with the reader. The intention is to motivate to continue reading while keeping the complexity low, taking away the fear of computers the user might have. It is of course not mandatory to read the whole introduction, as it can be brought up at any later time form the menu or toolbar.

The main window is fully visible after the introduction has been closed. It initially shows two main areas (panels): One for the graph and another for all things related to the algorithm. The program is basically always in one of two modes (*graph edit* or *algorithm*).

The algorithm panel is always grayed out when the program is in graph edit mode. Hiding the panel avoids distracting the user with things not relevant to

making a graph. The legend of node and edge colors could for instance cause confusion since they have nothing to do with the colors when editing the graph. The user can switch to algorithm mode when done with the graph, revealing the algorithm panel with all UI widgets related to the algorithm.

2.1 Editing the graph

The user can load or randomly generate a graph, or start from scratch. The graph panel shows two contextual semi-transparent information boxes at the bottom. Their content changes when the mouse is moved over nodes and edges and tells the user which operations can be performed with the left and right mouse buttons.

The whole process of graph editing is also *not modal* like in many similar applications. That means the user cannot switch between modes like "editing", "picking", or "translating" (these are actually the modes that are used by the JUNG2 graph library by default). Modeless editing might slow down the experienced user but it also protects the novice user from making mistakes due to mode confusion. The latter case is clearly more relevant for this program, thus my decision to avoid modes.

2.2 Running the algorithm

The user can switch to "algorithm mode" when the graph is ready. This causes the graph panel to become locked against changes that alter the position or distance between nodes. Only the distance-invariant graph operations (rotation, panning and zooming) are allowed in this mode.

The algorithm panels contains a legend explaining the colors used to visualize the state of nodes and edges during the course of the algorithm.

Below the legend, a textbox provides information about the current step's activity, trying to explain what's going on in plain language, omitting pseudo-code or algorithmic slang (like the word "relaxed"). The idea for the specific text is borrowed from Carla Laffra's applet[1].

A progress bar visually and textually conveys the algorithm's progress. The total number of steps is pre-calculated by counting the number of reachable nodes from the start.

Detail information for nodes and edges are shown when mouse is moved over them in the graph panel. Node's path length and the path's current state (e.g. "just found" or "improved") is shown among other information. The current path from the start to the highlighted node is also displayed if one has already been found.

[1] `http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html`

This is done in a "text-marker style", not violating the legend by preserving edge colors and shapes along the path.

Changes between algorithm steps can either be displayed all at once or by animation. Animation makes it much easier to follow the algorithm, especially in the beginning and in smaller graphs. The user can of course turn this feature off, which can be useful in bigger graphs and for performance reasons.

2.3 Help

The program includes help that can be displayed on the left side of the main window. The help is meant to be read on demand. It is hopefully not necessary for most users to read it in order to get started. Additional information on Dijkstra's algorithm are also provided for the interested user.

2.4 Language

The whole program, including help and introduction, is available in English and German. The language can be changed at any time without the need to restart the program. It would also be easy to add other languages through new property files and having a list of languages to choose from instead of a toggle button.

2.5 System requirements

I decided to use Java 5 for compatibility reasons. For instance, Mac OS X currently only offers Java 6 as a developer preview. I also used *Swing* since it is truly platform-independent without the need to provide a separate version for each platform.

2.6 Appearance

I wanted to have the same program appearance on all platforms to assure the equal program experience. I decided not to use Java's default "Metal Look & Feel" because it isn't very good in my opinion. I thus used the third-party JGoodies Look & Feel but also added the possibility to change the look & feel to any other installed on the user's computer. This could lead to unexpected behavior of some UI widgets though.

3 Software Design

I decided to assign an *attribute* to each node and edge. This attribute is changed while the algorithm progresses and used by the GUI to render nodes and edges accordingly. The algorithm assigns one of following attributes to nodes and edges at each step.

Nodes: `START_NODE`, `VISITED`, `NOT_VISITED`, `SETTLED`, `CURRENTLY_SETTLED`, `VISITED_NEXT_SETTLED`, `PATH_FOUND`, `PATH_FOUND_NEXT_SETTLED`, `PATH_IMPROVED`, `PATH_IMPROVED_NEXT_SETTLED`

Edges: `ADDED_TO_SHORTEST_PATH`, `REMOVED_FROM_SHORTEST_PATH`, `NOT_VISITED`, `VISITED`, `ON_SHORTEST_PATH`

Although the software can distinguish all those attributes, I decided against choosing a different color for each attribute to avoid color proliferation.

There are three observable parts of the system:

The graph notifies its observer when nodes or edges are added and deleted, or the entire graph is replaced (e.g. loaded).

The algorithm publishes 3 types of events to its subscribers:

- `Initialized`, providing the number of steps the algorithm will needs to finish.
- `Step Changed`, passing the current step number and all changes made from the last step.
- `Reset`, sent when the algorithm is stopped.

The language choice is observable too. It is observed by all GUI elements that need to output localized content.

The software can be divided into six logical subsystems which are explained in the following and depicted in figure 1.

3.1 Graphical User Interface

3.1.1 Swing components

This includes the panels for graph, algorithm and help. It also contains the introduction window and the menu-, tool-, and status- bar.

All components listen for language changes to update their content accordingly. They use resource injection (provided by AppFramework) from localized property files to configure subcomponents such as labels and buttons.

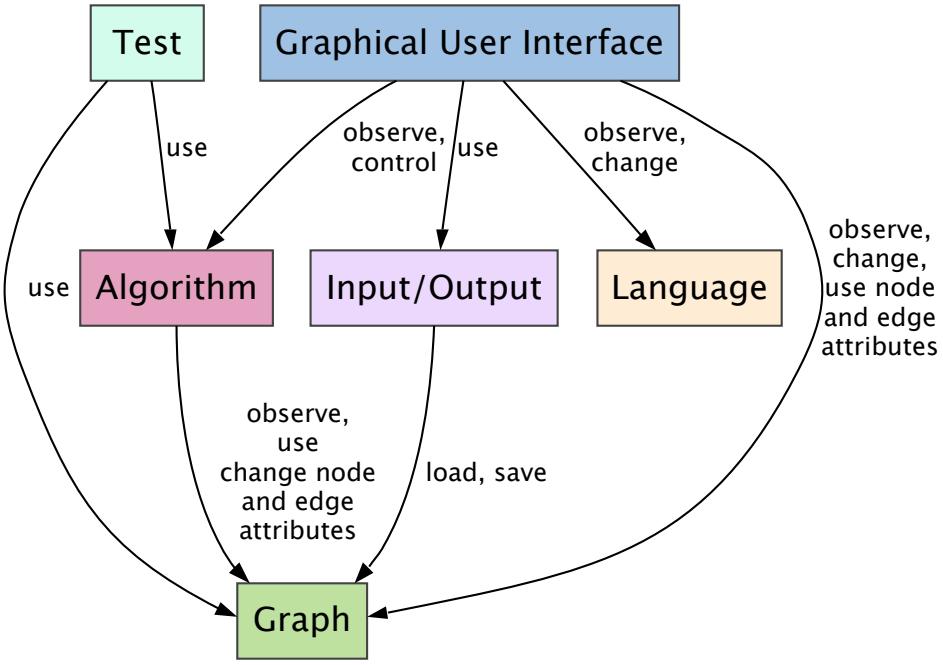

Figure 1: Main components.

3.1.2 Graph

All parts that tie into to the JUNG2 graph library:

Transformer take nodes and edges as input and return graphical properties mostly based on their attributes (e.g. edge paint, edge stroke, node shape).

Graph Mouse controls the mouse interaction with the graph. It is based on a plug-in architecture that passes mouse events along a hierarchy until they are consumed.

Animation adds animation capabilities to the graph library by replacing the renderers for the graph, nodes and edges. Animations are created from changes, published by the algorithm in each step. Edges are additionally sorted into several sets based on their attributes and painted in order of their current importance. Unused edges are for instance painted first while edges that are just added or removed to a path are painted on top of them.

3.2 Graph

The graph model consists of the abstract definitions for graphs, nodes and edges and a graph manager that allows graph listeners to subscribe and be notified of changes. The graph implementation realizes the graph model and is used by the graph library. It also contains factories for graphs, nodes and edges and a pseudo-planar graph generator.

3.3 Algorithm

Includes the implementation of Dijkstra's algorithm, that encapsulates step changes using the *Command pattern*[2] to offer *undo*-functionality. It also contains an algorithm manager that decouples the algorithm from its listener. Note that the algorithm only uses the abstract graph definition and isn't coupled with the GUI nor the graph library.

3.4 Language

Offers a locale manager that many GUI components observe to be notified of language changes. Also includes generic translation for all classes that do not use property files themselves.

3.5 Input/Output

Consists of a single class that is called by the GUI to load and save graphs from and to files. Graph files are ordinary ZIP-archives containing XML-files that primarily specify the nodes with their connections and locations. Other properties like the selected start node and graph background image are saved too when available.

Using XML has the advantage of being human-readable and makes it easy to change or generate files with other applications. Their disadvantage is the rather big size in comparison to binary data. This is made up by using ZIP to compress them. A ZIP file also has the advantage of bundling several files, only having one file per graph that can even be extended later on. Adding a graph background is as simple as unzipping with a standard archive program, adding a PNG background image and zipping all files again.

[2] `http://en.wikipedia.org/wiki/Command_pattern`

3.6 Test

Contains a JUnit test for the algorithm with two test cases. The first is testing the generation of a big graph and running the various phases of the algorithm with an eye towards speed and memory consumption. The second and more important test checks the algorithm for correctness. Steps are executed forward and backward while checking node and edge attributes.

4 Libraries

The following libraries are used by the program. Some of my extensions and customizations to JUNG2 are based on their source code.

Swing Application Framework (AppFramework) Simplifies Swing applications.
`https://appframework.dev.java.net/`

2nd Generation Java Universal Network Graph (JUNG2) Graph library.
`https://jung2.dev.java.net/`

Swing Worker Used by JUNG2 and AppFramework.
`https://swingworker.dev.java.net/`

Jakarta Commons-Collections (Generics-enabled version) Used by JUNG2.
`http://sourceforge.net/projects/collections/`

Colt Used by JUNG2.
`http://dsd.lbl.gov/~hoschek/colt/`

JGoodies Looks (Plastic Look & Feel) Java Look & Feel.
`http://www.jgoodies.com/freeware/looks/`

MiGLayout Swing Layout Manager.
`http://www.miglayout.com/`

Timing Framework Used for animations.
`https://timingframework.dev.java.net/`

XStream Used to serialize objects to XML and back again.
`http://xstream.codehaus.org/`

SwingX Swing Component Extensions.
`https://swingx.dev.java.net/`

Bare Bones Browser Launch Launch web browser.
`http://www.centerkey.com/java/browser/`