How-To: Develop plugins for LAVES

In this article we want to show you how you can develop plugins for the algorithm visualization software LAVES[1] using the open source LAVESDK[2].

If you are already familiar with developing plugins for LAVES, nevertheless have a look at the index of this article to step into specific points that you possibly missed, that may optimize your plugin, or where you can learn more about things that you may not have considered before.

Author: Jan Dornseifer

Contact: jan.dornseifer@student.uni-siegen.de or jan.dornseifer@t-online.de

Index

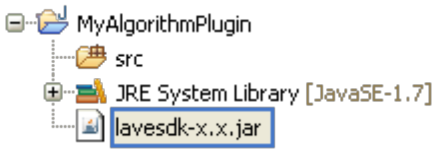
[1] http://www.wiwi.uni-siegen.de/mis/software/laves.html

[2] http://www.wiwi.uni-siegen.de/mis/software/lavesdk.html

Getting started

First download the SDK from http://www.wiwi.uni-siegen.de/mis/software/lavesdk.html, and open your Java development environment. The following description refers to the Eclipse IDE[3]:

1. Go to File → New → Java Project and create a new plugin project in your workspace.
2. Extract the LAVESDK zip file that you previously downloaded and copy one of the SDK JAR files into your project folder that you have created under 1. *It is recommended that you use the release version but if you need debug support for the LAVESDK you have to use the debug version.*
3. Go back to Eclipse, right-click onto the project root in the Package Explorer and select Refresh.
4. Afterwards you have to add the SDK (marked blue) to your Java Build Path by right-clicking onto the JAR file in the Package Explorer and select Build Path → Add to Build Path in the pop-up menu.



5. Finally we have to activate UTF-8 encoding because LAVESDK works with language files that may contain languages that do not use ASCII characters. Therefore select the project root again, go to the properties (Project → Properties), update the text file encoding (see Resources) from "*Inherited from container*" to "*Other*" and choose "*UTF-8*" from the list.

That's all to prepare Eclipse for developing plugins with the LAVESDK. Now you can create your plugin class (File → New → Class) and implement the interface *AlgorithmPlugin*. In the next section we'll show you the methods that *AlgorithmPlugin* comes with.

Implement AlgorithmPlugin

The interface *AlgorithmPlugin* is the entrypoint for LAVES to communicate with a plugin, and each plugin must provide just one class that extends *AlgorithmPlugin*.

This interface contains methods to depict the algorithm, for customization, to open and save files, and to process events. The basic implementation without overriding any method is shown below.

```
import lavesdk.algorithm.plugin.AlgorithmPlugin;


public class MyAlgorithmPlugin implements AlgorithmPlugin {

}
```

[3] http://www.eclipse.org/

We now want to describe the major functionality step by step. To begin with we look at the method initialize.

Method initialize

This method is invoked once the plugin is loaded in LAVES and should set the initial values of the instance.

```
@Override
public void initialize(PluginHost host, ResourceLoader resLoader,
                       Configuration config) {
  // (1) load the language file of the plugin
  try {
    langFile = new LanguageFile(
         resLoader.getResourceAsStream("main/resources/myLangFile.txt"));
    langFile.include(host.getLanguageFile());
  } catch (IOException e) {
    langFile = null;
  }
  langID = host.getLanguageID();

  // (2) save the parameters
  this.host = host;
  this.config = (config != null) ? config : new Configuration();
  // ...

  // (3) load configuration data
  colorA = config.getColor(CFGKEY_COLOR_A, new Color(255, 255, 255));
  // ...

  // ...
}
```

The snippet above contains some possible initializations for the plugin. But first we should look at the parameters. The variable host contains the LAVES instance that loads the plugin and provides communication functionality, resLoader is only necessary when you want to load a resource file, and with config it is feasible to get access to the plugin's configuration data that is stored persistently and managed by the host system.

In the example beyond we create a language file (**1**). With a language file you can display your plugin in different languages. In a later section you will see how a language file is constructed. For now we create just a new one using the relative path to the file in the project that refers to the package structure in which the file is deposited. In our sample there is *myLangFile.txt* in a sub-package named *resources* that is contained in the base package *main*. It is recommended to separate implementation and resources (like files, images, etc.) using own packages. After the creation we integrate the language file of the host. Doing this, we can forward our language file to basic components of the LAVESDK (like an algorithm text view, a table view, etc.) without rewriting all the language labels that are required by these elements. Finally we save the language identifier, which is a short tag like "en", that indicates the active language in LAVES. If your plugin should not support different languages you must not provide an own language file.
Under (**2**), we initialize the member variables of the plugin, and in the last step (**3**) we load some configuration data. The configuration is a key-value storage. colorA represents the coloration of a variable "a" that is used in the algorithm for visualization, and we use a static class member CFGKEY_COLOR_A that contains the key as a string. If there is no value for the key in the configuration you can specify a default value as the second parameter.

This is only a fraction of points you can or have to do in the initialization method. Your own implementation should normally contain much more aspects (as the ellipsis should illustrate).

Methods to describe the algorithm

Beside the initialization method there are several properties to describe the algorithm. These include the name, a description, the type, the assumptions, the problem affiliation, and the subject of the algorithm as well as the author, its contact information, and some instructions of the plugin. All follow the same approach, thus we only look at one of them:

```java
@Override
public String getName() {
  return LanguageFile.getLabel(langFile, "ALGO_NAME",
                               langID, "My algorithm");
}
```

The getName property returns the name of the algorithm we'll implement. But instead of returning the string "My algorithm", where My algorithm should be the real name of your algorithm, we use a construct that queries the previously mentioned language file. This enables us to depict the name in certain languages. We use the static method getLabel of the class LanguageFile that simplifies the usage of the langFile instance. Thus it is not necessary to check whether the instance, we have created in the initialization method, is valid (not null). We just pass our language file instance, the language label, the current language identifier, and a default value to the method. If langFile is invalid or there is no "ALGO_NAME" label in the file it is returned the specified default value. To get to know how a language file is structured, please see the section Construct a language file. At the moment we accept that there is a label (or placeholder) in a file and with the presented method it is possible to load the correct name dependent on the determined language identifier.

You can apply the same approach equally for the other properties getDescription, getType, and so on. See the corresponding Javadoc for more information.

Versioning

The interface AlgorithmPlugin contains two getters that include the name "Version". They could be implemented as follows:

```java
@Override
public String getVersion() {
  return "1.0";
}

@Override
public LAVESDKV getUsedSDKVersion() {
  return new LAVESDKV(1, 4);

}
```

The first returns the version of your plugin. This information is helpful for the user, if he wants to check whether he has installed the newest version of your plugin in LAVES. The format of the version is not predefined but should be something like "x.y" or "x.y.z".

The second one is a very important property to reduce the error-proneness of your plugin. To decrease the size of a plugin and to increase the loading time the LAVESDK is not included in a plugin's JAR file, but provided by LAVES. This requires a mechanism to avoid installing plugins in LAVES that are developed using a newer version of the SDK than the one that is shipped with LAVES. Furthermore it is possible that a signature of a class, interface, or method that comes with LAVESDK has changed in a newer version, which breaks compatibility. To validate whether a plugin is runnable, LAVES requests the SDK version that is used at the plugin's development time and checks if this version can be used with the SDK version that is integrated in LAVES. **So you must always update this property when you bring your plugin up to date with another LAVESDK version.**

In our case we use the latest version for development. You can retrieve the version of your LAVESDK either from the filename or if you have changed it, you can output LAVESDK.CURRENT. But be aware of never return something like LAVESDK.CURRENT in the property getUsedSDKVersion. Just do it as shown above.

Creator preferences and customization

Up to that point we have discussed the initialization method, the getters to describe the algorithm, and the versioning. In this chapter we take a look at creator preferences and customization.

It might be necessary to give the user the opportunity to change the initial behavior of the plugin. You can achieve this by setting up creator preferences.

```java
@Override
public boolean hasCreatorPreferences() {
  return true;
}

@Override
public void loadCreatorPreferences(PropertiesListModel plm) {
  final BooleanPropertyGroup group = new BooleanPropertyGroup(plm);
  plm.add(new BooleanProperty(LanguageFile.getLabel(langFile,
                                "CREATORPREFS_DIRECTED", langID,
                                "directed"),
                              LanguageFile.getLabel(langFile,
                                "CREATORPREFS_DIRECTED_DESC", langID,
                                "Apply algorithm to a directed graph"),
                              creatorPrefsDirectedValue, group));
  plm.add(new BooleanProperty(LanguageFile.getLabel(langFile,
                                "CREATORPREFS_UNDIRECTED", langID,
                                "undirected"),
                              LanguageFile.getLabel(langFile,
                                "CREATORPREFS_UNDIRECTED_DESC", langID,
                                "Apply algorithm to an undirected graph"),
                              !creatorPrefsDirectedValue, group));
  // ...
}
```

The first step is to override hasCreatorPreferences and return true. This tells LAVES that the user can configure the plugin before it is opened. The next step is to load the options that the user can define. This is done in loadCreatorPreferences. The method provides a PropertiesListModel where you can add the options using the add method. Possible properties might be TextProperty, NumericProperty, BooleanProperty, ListProperty, and ColorProperty. For instance, in the code above we create two boolean options and connect them with a BooleanPropertyGroup. Thereby it is only possible to select one of the options. We use the boolean property to give the user the possibility to indicate whether the algorithm should be applied on a directed or an undirected graph.

To be language independent we use our language file again like in Methods to describe the algorithm.

The customization of a plugin is similar to the creator preferences, but in contrast to the preferences the customizable options are long-term. That means they should only be defined once and not changed each time the plugin is opened. Therefore the host application (LAVES) provides a customization dialog. To enable customization you have to override hasCustomization with a return value of true. Afterwards you have to implement loading and applying the options. Loading is pretty equal to what is done in loadCreatorPreferences from above.

```
@Override
public void loadCustomization(PropertiesListModel plm) {
  plm.add(new ColorProperty(CFGKEY_COLOR_A,
                            LanguageFile.getLabel(langFile,
                              "CUSTOMIZE_COLOR_A", langID,
                              "Color of the variable a"),
                            colorA));
  // ...
}

@Override
public void applyCustomization(PropertiesListModel plm) {
  Color cA = plm.getColorProperty(CFGKEY_COLOR_A).getValue();
  colorA = config.addColor(CFGKEY_COLOR_A, cA);
  // ...
}
```

In applyCustomization you obtain a PropertiesListModel with the values of the customizable options. We retrieve these values and store it directly in the configuration so that they can be loaded again the next time LAVES, respectively the plugin, is started. During runtime these values are kept alive in member variables too. In this example colorA is a class member that saves the current value of the color (the initial value is set in the initialization method, see Method initialize).

Create the plugin view (major events)

By now our plugin can be initialized, customized and prepared by the user using creator preferences. In this section we want to look at the view concept of LAVES.

Each plugin consists of one or more views. These views have different functions. For instance, the AlgorithmTextView, which comes with the LAVESDK, presents the pseudocode of the algorithm to the user. Another view might be a graph or a table that depicts algorithm variables. By default these views can be closed by the user, which means that the user can made the steps by hand instead of being visualized in the specific view.
Each view inherits from View which is the abstract base class. You can develop your own views by extending View or you can use views that are provided by the LAVESDK. The SDK includes an AlgorithmTextView, as mentioned above, a GraphView, a TextView, an ExecutionTableView, and a LegendView. Feel free to inform you about these views using the Javadoc.

In this article we present you at short the AlgorithmTextView and the GraphView. Views should be declared as private class members and initially be instantiated in the initialize method (see Method initialize).

The build-in views that come with LAVESDK accept some parameters in their constructors especially a language file. As mentioned in the section Method initialize we have integrated the host's language file into our own one, which gives us the ability to utilize the default language labels. Thus we can just use our langFile instance when we create those build-in views.
Let's assume we have already created our views. Then we come to the point at which these views should be displayed to the user. This happens in the onCreate method of the AlgorithmPlugin. That method is invoked when a user opens the plugin in LAVES. All we have to do is to add the views, our plugin is composed of, to the ViewContainer that is obtained by the method.

```
@Override
public void onCreate(ViewContainer container,
                     PropertiesListModel creatorProperties) {
  // (1) get the chosen creator preference
  String creatorPrefsDirected = LanguageFile.getLabel(langFile,
              "CREATORPREFS_DIRECTED", langID, "directed");
  creatorPrefsDirectedValue = (creatorProperties != null) ?
   creatorProperties.getBooleanProperty(creatorPrefsDirected).getValue()
   : false;

  // (2) update the configuration
  config.addBoolean(CFGKEY_CREATORPROP_DIRECTED,
                    creatorPrefsDirectedValue);

  // (3) change the graph in the view
  graphView.setGraph(new SimpleGraph<Vertex, Edge>(
                                     creatorPrefsDirectedValue));
  graphView.repaint();

  // (4) create a view group for the views
  vg = new ViewGroup(ViewGroup.HORIZONTAL);
  vg.add(algoTextView);
  vg.add(graphView);
  vg.restoreWeights(config, "weights_vg", new float[] { 0.4f, 0.6f });

  container.setLayout(new BorderLayout());
  container.add(vg, BorderLayout.CENTER);

}
```

Before we add the views to the container we do some configuration. At first (**1**) we request the state of one of the creator preferences we have defined earlier (see section Creator preferences and customization). The value indicates whether the user wants to apply the algorithm to a directed or an undirected graph. Additionally (**2**), we save this value in the configuration, so the next time the user wants to open the plugin this value is pre-selected. Then we initialize our graphView instance with a new graph (**3**), otherwise the user would see the one he has created before because a plugin is stateful. Since the graph should conform the chosen preference, we pass in the value from (1). Finally we add the views to the container (**4**). In our example we use a view group that contains the algoTextView and the graphView. With view groups we can create an adjustable user interface whereby users can use the sashs inside such groups to change the sizes of the components. Instead of using static sizes (resp. weights) we restore the last settings of our configuration by making use of the restoreWeights method.

Afterwards we create a layout in the container and add the view group. You can use the provided layout managers by Swing[4] or custom ones.

The counterpart of the onCreate method is onClose. This method is invoked each time a user closes the plugin, meaning he exits LAVES or opens another algorithm plugin. Therefore we store the configuration of our views (and groups) and reset them to avoid obsolete representations. This is done in the snippet below:

```
@Override
public void onClose() {
  // save view configurations
  graphView.saveConfiguration(config, "graphView");
  algoTextView.saveConfiguration(config, "algoTextView");

  // save weights
  if(vg != null)
    vg.storeWeights(config, "weights_vg");

  // reset view content where it is necessary
  graphView.reset();

}
```

Load the algorithm text

Now it is time to visualize something in our plugin. If you have followed the steps in the previous sections it should be possible to start the plugin, for instance using the sandbox (see section Test your plugin). As a result you get two empty views. You may edit the graph but what's missing is the pseudocode of our algorithm. That's what we want to change now.

We have to separate between the view AlgorithmTextView that is responsible for visualizing the text, and the data structure that manages the pseudocode, which can be found in the class AlgorithmText and its elements AlgorithmParagraph and AlgorithmStep. Beginning with the data structure, it is obvious that an algorithm text contains paragraphs which consist of steps. For example an algorithm may look like the following:

```
1. Initialization:   Let $a$ be an arbitrary edge and $c:=0$.
2. Iteration:        For each $e \in E$
                       If $w(e) == w(a)$ then
                         $c:=c+1$
```

Whether this is a reasonable algorithm is not up for debate. It should only illustrate the structure of a pseudocode that can be visualized in an AlgorithmTextView just as shown above.
A paragraph is highlighted in bold. That what's executed in a paragraph should be separated into steps in a meaningful fashion. In our example it would be exaggerating if we split the first statement. Regarding the second paragraph, it is suitable to divide this paragraph into three steps, one for the head and two for the body of the loop.

[4] https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

We now implement a method that loads the text. For simplicity, we don't take account of multilingualism, although it is easily realizable the same way we did it before.

```
private AlgorithmText loadAlgorithmText() {
  AlgorithmStep step;

  final AlgorithmText text = new AlgorithmText();

  // create paragraphs
  final AlgorithmParagraph initParagraph = new AlgorithmParagraph(text,
                                                  "1. Initialization:", 1);
  final AlgorithmParagraph itParagraph = new AlgorithmParagraph(text,
                                                  "2. Iteration:", 2);

  // 1. initialization
  step = new AlgorithmStep(initParagraph,
              "Let _latex{a} be an arbitrary edge and _latex{c := 0}.", 1);

  // 2. iteration
  step = new AlgorithmStep(itParagraph, "For each _latex{e \\in E}", 2);
  step = new AlgorithmStep(itParagraph,
                           "If _latex{w(e) == w(a)} then", 3, 1);
  step = new AlgorithmStep(itParagraph, "_latex{c := c + 1}", 4, 2);

  return text;
}
```

First of all we create the two paragraphs. Afterwards we add steps to the previously created paragraphs. That's all we have to do here, mapping the algorithm pseudocode to the object structure as presented above.
Lastly, we have to clarify the sequence "_latex" in the description of the steps. If you surround a sequence with the prefix "_latex{" and the suffix "}" you indicate that you want to display this sequence in $\LaTeX$. This is useful for formulas or mathematical expressions. The most of $\LaTeX$ commands are available, and some macros and environments too[5].

Our loadAlgorithmText method should be invoked in the initialize method of our plugin, before the AlgorithmTextView is created. Remember that this text should also be returned by the getText property.

Create exercises

We can extend the algorithm text, respectively the steps, by exercises that the user has to solve during the execution of the algorithm. Therefore, we have to do two things. On the one hand we have to say the host application that our plugin supports an exercise mode (using the property hasExerciseMode) and on the other hand we have to add exercises to all or some of the steps. The first point is very simple as shown below.

```
@Override
public boolean hasExerciseMode() {
  return true;
}
```

The second one depends on the complexity of your exercise. As an example, we implement an exercise for the step with the id 3 ("If w(e) == w(a) then"). A detailed description follows directly below the source code.

[5] https://forge.scilab.org/index.php/p/jlatexmath/

```
private AlgorithmText loadAlgorithmText() {
  // ...
  step = new AlgorithmStep(itParagraph,
                           "If _latex{w(e) == w(a)} then", 3, 1);
  step.setExercise(new AlgorithmExercise<Boolean>(
                           "Does <i>w(e)</i> equals <i>w(a)</i>?", 1.0f) {
    @Override
    protected Boolean[] requestSolution() {
      // (1) create two grouped radio buttons for the answers yes and no
      final ButtonGroup group = new ButtonGroup();
      final JRadioButton rdobtn1 = new JRadioButton("Yes");
      final JRadioButton rdobtn2 = new JRadioButton("No");

      group.add(rdobtn1);
      group.add(rdobtn2);

      // (2) create two entries for the options
      final SolutionEntry<JRadioButton> entryYes = new SolutionEntry<>(
                                                        "", rdobtn1);
      final SolutionEntry<JRadioButton> entryNo = new SolutionEntry<>(
                                                        "", rdobtn2);
      // (3) and show these entries in a dialog
      if(!SolveExercisePane.showDialog(host, this,
          new SolutionEntry<?>[] { entryYes,  entryNo }, langFile, langID))
        return null;

      return new Boolean[] {
        (!rdobtn1.isSelected() && !rdobtn2.isSelected())
        ? null : rdobtn1.isSelected()
      };
    }

    @Override
    protected String getResultAsString(Boolean result, int index) {
      if(result == null)
        return super.getResultAsString(result, index);
      else
        return (result.booleanValue() == true) ? "Yes" : "No";
    }

    @Override
    protected boolean examine(Boolean[] results, AlgorithmState state) {
      final int idOfe = state.getInt("e");
      final int idOfa = state.getInt("a");
      final Edge e = graphView.getGraph().getEdgeByID(idOfe);
      final Edge a = graphView.getGraph().getEdgeByID(idOfa);

      return (results[0] != null &&
              results[0] == (e.getWeight() == a.getWeight()));
    }
  });
  // ...
}
```

The implementation of the exercise is done using a technique called anonymous classes. If you aren't familiar with this concept, please take a look at the online tutorials[6].
Beginning with requestSolution, this method queries the solution by presenting the user a dialog with two options, yes and no. These are the possible answers that are

[6] https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

available to the user. Therefore, we create two Swing based radio button components in (**1**) and add them to a group. Thus the user can only select one of them at the same time. Then we use a utility class, named SolveExerciseDialog, to display these options to the user. In this generic dialog you can display multiple solution entries, equipped with a text label and a component, for each possible solution in a consistent way. Hence, we wrap the radio buttons using SolutionEntry in (**2**) and present the dialog to the user in (**3**). If the user cancels the dialog we return a null value that indicates the cancelation. Otherwise we check whether there is one option selected and if so, we return the user's solution. It is enough to return rdobtn1. isSelected() because if the second one is selected then the first one returns false which matches the answer no.
In getResultAsString we make a cosmetic change. Overriding this method, we can determine the representation of the result. By default the method returns the toString of the result object, meaning in our case it returns "true" or "false". But we want to return "yes" respectively "no" as the user's answer. So dependent on the result we convert true to "yes" and false to "no".
Finally we must verify whether the user has answered correctly. This must be done in examine. Therefore, we request the values of the two variables *e* and *a*, and check whether the solution of the user (note that an exercise can have multiple solutions) corresponds with the real solution. The values of the variables are stored in the AlgorithmState, which is discussed in chapter The runtime environment.

AlgorithmExercise provides a host of other capabilities in addition to those shown above. Please read the Javdoc to expand your knowledge.

Save and open files

The AlgorithmPlugin interface provides methods to open and save files which enables to load and store data from views. For example it is possible to store a constructed graph of a GraphView or to load a persistently saved graph instance. Beside the methods open and save there are two properties getOpenFileFilters and getSaveFileFilters that can be used to restrict the choosable files in the file chooser dialog of LAVES. They return an array of filters that describe the file extensions of the files which can be opened or saved.

```
@Override
public void save(File file) {
  final FileNameExtensionFilter vgfFilter = new FileNameExtensionFilter(
                                    "Visual Graph File (*.vgf)", "vgf");
  final FileNameExtensionFilter pngFilter = new FileNameExtensionFilter(
                                 "Portable Network Graphic (*.png)", "png");

  try {
    if(vgfFilter.accept(file))
      graphView.save(file);
    else if(pngFilter.accept(file))
      graphView.saveAsPNG(file);
  }
  catch(IOException e) {
    host.showMessage(this, "File could not be saved!\n\n" + e.getMessage(),
                     "Save File", MessageIcon.ERROR);
  }
}

@Override
public void open(File file) {
  final FileNameExtensionFilter vgfFilter = new FileNameExtensionFilter(
                                       "Visual Graph File (*.vgf)", "vgf");

  try {
    if(vgfFilter.accept(file))
      graphView.load(file);
  }
  catch(IOException e) {
    host.showMessage(this, "File could not be opened!\n\n" +
                     e.getMessage(), "Open File", MessageIcon.ERROR);
  }
}

@Override
public FileNameExtensionFilter[] getSaveFileFilters() {
  return new FileNameExtensionFilter[] {
    new FileNameExtensionFilter("Visual Graph File (*.vgf)", "vgf"),
    new FileNameExtensionFilter("Portable Network Graphic (*.png)", "png")
  };
}

@Override
public FileNameExtensionFilter[] getOpenFileFilters() {
  return new FileNameExtensionFilter[] {
    new FileNameExtensionFilter("Visual Graph File (*.vgf)", "vgf")
  };
}
```

At first we want to take a look at the file filters properties in the snippet above. There we return two extension filters, one for the save operation and one for the open operation. Thereby we restrict LAVES to only load and save .vgf files and generate .png files. These filters are also used in the save and open method to identify which operation should be executed. In our plugin we offer the user to load and save visual graph files, meaning to load and save the created graphs, and to save the graph in the related view as an image. In contrast to the snippet above, it is recommended to instantiate these filters once in the initialize method and only return, respectively use, the references.

If you don't want to offer this service you can simply return null in the file filters methods, which disables loading and saving files from LAVES.

Extend the toolbar

Another feature of a plugin is to extend the toolbar in LAVES with additional functionality. ToolBarExtension is the responsible class that allows implementing your own extensions. The LAVESDK provides several extensions. Build-in toolbar extensions can be found in the following classes.

- RandomGraphToolBarExtension (*since 1.4*): supports creation of randomly generated graphs by specifying the number of vertices, the minimum weight and the maximum weight of edges.
- MatrixToGraphToolBarExtension: supports creation of graphs by use of an adjacency matrix.
- CompleteGraphToolBarExtension: supports creation of complete graphs by specifying the number of vertices, the minimum weight and the maximum weight of edges, and validation of whether a graph is complete.
- BipartiteGraphToolBarExtension: validation whether a graph is bipartite.

- CompleteBipartiteGraphToolBarExtension: creation of complete bipartite graphs (similar to above), and validation of whether a graph is complete bipartite.
- CircleLayoutToolBarExtension: layout vertices of a graph in a circle.
- BipartiteLayoutToolBarExtension: layout vertices of a graph bipartite.

To extend the toolbar you have to override getToolBarExtensions and return an array of extension objects that your plugin supports. In the example below we only use one extension for simplicity. Notice that the extensions should be created in the initialize method as implied.

```
// ...
private MatrixToGraphToolBarExtension<Vertex, Edge> matrixToGraphExt;


@Override
public void initialize(PluginHost host, ResourceLoader resLoader,
                       Configuration config) {
  // ...

  // create the toolbar extensions
  matrixToGraphExt = new MatrixToGraphToolBarExtension<Vertex, Edge>(
           host, graphView, AllowedGraphType.BOTH, langFile, langID, true);

  // ...
}

@Override
public ToolBarExtension[] getToolBarExtensions() {
  return new ToolBarExtension[] { matrixToGraphExt };
}
```

As mentioned at the beginning of this section, you can easily make your own toolbar extensions by inherit from ToolBarExtension. Follow the Javadoc to get detailed information.

Runtime events (minor events)

In the last practical section of this chapter we want to dedicate ourselves to the runtime events.

As the name suggests, these events have something to do with the execution of our algorithm, which is discussed later on in chapter The runtime environment.
There are six events available, namely beforeStart, beforeResume, beforePause, onStop, onRunning, and onPause. These events should be self-explaining, otherwise please have a look at the documentation of the methods. Here, we want to study beforeStart and onStop, and what we can do when we receive such an event.

In beforeStart we may check whether the created graph is conform to the assumptions of the algorithm and we can prepare the views for execution. Therefore, a GraphView should be set to non-edit mode for instance, like in the snippet below. As a consequence we have to enable the edit mode when the algorithm has finished. That's the task of the onStop method (resp. event).

```
@Override
public void beforeStart(RTEvent e) {
  // we have to check whether the created graph has at least one edge,
  // which is an assumption of our algorithm
  if(graphView.getGraph().getSize() < 1) {
    // cancel the start
    e.doit = false;
    // inform the user
    host.showMessage(this, "The graph must have at least one edge!",
                     "Invalid graph", MessageIcon.INFO);
  }

  if(e.doit) {
    // clear any selection and disable edit mode of the graph view
    graphView.deselectAll();
    graphView.setEditable(false);
  }
}

@Override
public void onStop() {
  graphView.setEditable(true);
}
```

If the user-generated graph does not conform to the assumption of having at least one edge the start of the algorithm is canceled and the user is notified by a message. Otherwise we clear a possible selection in the graph view, or else it will be displayed during runtime, and we disable the edit mode. This is undone when the algorithm ends.

Conclusion

In this chapter you learned the basic things about the interface AlgorithmPlugin. The presented code is only a fraction you can or have to do to develop a functioning plugin. It serves as an outlook and may not be runnable without some additional implementations, but it should introduce to the development and show the relevant aspects of a LAVES plugin.

One significant point has been neglected up to here, the runtime environment, which is the issue of chapter The runtime environment.

The runtime environment

An algorithm plugin is no algorithm plugin without implementing the actual algorithm. Leaving chapter Implement AlgorithmPlugin, our plugin has some properties, preference options and customizations, but the implementation of the algorithm is missing. That's what we want to change in this chapter.

The algorithm runtime environment is the heart of our plugin. It cares about executing and visualizing each step that we have defined in our algorithm text (see Load the algorithm text). A concrete runtime environment (rte) inherits from AlgorithmRTE, which is an abstract class and provisions the base functionality. In other words we have to implement a new class that extends AlgorithmRTE and overrides the necessary methods. We can do this in two different ways.

1. We create a new class in our project next to the already existing plugin class (e.g. MyAlgorithmPlugin).

2. We use the concept of inner classes and implement a second class in our plugin class (e.g. MyAlgorithmPlugin). Inner classes are a specific type of a nested class. If you aren't familiar with that, please consider the links below[7][8].

I prefer doing it in the second way. The first approach splits both implementations and leads to less lines of code in the plugin class. But the second one encapsulates the plugin and its rte, and further leads to an easier usage of plugin class members inside the runtime environment. For instance, this implies that we can use our view instances by direct access to the member variable and must not pass them to an external rte class via constructor or setters.

Now, let's implement our class MyAlgorithmRTE as stated above.

```
public class MyAlgorithmPlugin implements AlgorithmPlugin {

  //...

  private class MyAlgorithmRTE extends AlgorithmRTE {

    public MyAlgorithmRTE() throws IllegalArgumentException {
      super(MyAlgorithmPlugin.this, MyAlgorithmPlugin.this.algoText);
    }

    @Override
    protected int executeStep(int stepID, AlgorithmStateAttachment asa)
                                                     throws Exception {
      return 0;
    }

    @Override
    protected void storeState(AlgorithmState state) {
    }

    @Override
    protected void restoreState(AlgorithmState state) {
    }

    @Override
    protected void createInitialState(AlgorithmState state) {
    }

    @Override
    protected void rollBackStep(int stepID, int nextStepID) {
    }

    @Override
    protected void adoptState(int stepID, AlgorithmState state) {
    }

    @Override
    protected View[] getViews() {
      return null;
    }
  }
}
```

[7] https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

[8] https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html

The code above shows all methods we have to override. Afterwards we discuss what the particular methods do.
But before we do, we first look at the constructor. Of course, each rte is based on a concrete plugin and the pseudocode data structure. This is relevant when we create our rte in the initialize method, what we do later. Because we implement the rte as an inner class, we can use the instance of the MyAlgorithmPlugin class and the AlgorithmText member variable directly instead of passing them as arguments in the initialize method.

The first method is executeStep. Here, the implementation of each step takes place. It expects the id of the step that should be executed and returns the id of the step that runs next. The id of a step is specified as one of the parameters that are passed to the AlgorithmStep constructor (see Load the algorithm text). The next method block (storeState, restoreState, and createInitialState) is used to manage the variables of the algorithm. The last of these methods creates the initial variable values and stores them in the AlgorithmState object. With the two other methods the rte controls the values of the variables during the runtime. So after each step the latest variable values are stored using storeState. Because of the user can also step to previous steps of the algorithm, the restoreState method loads the values from the state object and stores them in the variables of the algorithm.

To clarify this, we now implement these methods. Consider the algorithm from section Load the algorithm text. There we have three variables named *a*, *c* and *e*. So we need three equivalents in our MyAlgorithmRTE class.

```java
private class MyAlgorithmRTE extends AlgorithmRTE {

  // algorithm variables
  private int a;
  private int c;
  private int e;
```

The variables *a* and *e* represent edges of the graph. So you may wonder why we use integer variables instead of object variables of type Edge. This is because integer types can be better stored in an AlgorithmState than objects. Furthermore Edge does not implement Serializable, but which is an assumption for objects that should be stored in an AlgorithmState. Now the question is, what does the integer value describe? Since every edge (and vertex) of a graph is identifiable by an unique integer, we use this value to handle the graph objects in our algorithm. In other words, *a* and *e* represent the id of a specific edge. Finally, the variable *c* is our counter.
The state methods just do a mapping between the variables and the equivalents in the state object.

```java
@Override
protected void storeState(AlgorithmState state) {
  state.addInt("a", a);
  state.addInt("c", c);
  state.addInt("e", e);
}

@Override
protected void restoreState(AlgorithmState state) {
  a = state.getInt("a");
  c = state.getInt("c");
  e = state.getInt("e");
}
```

```
@Override
protected void createInitialState(AlgorithmState state) {
  a = state.addInt("a", -1);
  c = state.addInt("c", 0);
  e = state.addInt("e", -1);
}
```

The leftover methods, rollBackStep, adoptState, and getViews, are used to undo a visualization of a certain step, to adopt a solution from an exercise, and to return the views that are used inside the rte to visualize something. We return to rollBackStep after we have discussed the method executeStep. For the other methods take a look at the documentation of class AlgorithmRTE.

Now we take a look at the execution of our algorithm. As a reminder, this is our algorithm:

```
1. Initialization:    Let $a$ be an arbitrary edge and $c:=0$.
2. Iteration:         For each $e \in E$
                        If $w(e) == w(a)$ then
                          $c:=c + 1$
```

As mentioned above, the first argument of executeStep informs us about the step that should be executed in the body. We have separated the pseudocode into four steps with identifiers of 0, 1, 2, and 3. So it is recommended to use a switch-case statement to implement each step.

```
@Override
protected int executeStep(int stepID, AlgorithmStateAttachment asa)
                                                      throws Exception {
  int iNextStepID = -1;

  switch(stepID) {
    case 0: // Let a be an arbitrary edge and c:=0
      break;
    case 1: // For each e in E
      break;
    case 2: // If w(e) == w(a) then
      break;
    case 3: // c:=c+1
      break;
  }

  return iNextStepID;
}
```

That's the framework. We use the iNextStepID variable to store the id of the step that should be executed after the current step. This simple technique lets us control the flow of the algorithm. The first step transitions into the second one, which can easily be managed by setting iNextStep = 1 in the corresponding case scope. The third step (id of 2) is much the same as the first, but differs in execution frequency. That means, he only transitions into step four (id of 3) when the condition of w(e) == w(a) results in true. So we have to distinguish, either we set iNextStep = 3 but only if w(e) == w(a) or we set iNextStep = 1 to continue with the next edge. The last step is easy again. After this step we return to the head of the for-each loop, meaning iNextStep = 1. A bit more complex is the second step (id of 1). Here we have to simulate a for-each loop, which can be done by using a list of edges and each time the second step is passed we pop the first element. But that's not enough. Furthermore we have to store, which edges are already visited. To achieve it we simply store this list in the algorithm state. Now we look at the final implementation.

```
@Override
protected int executeStep(int stepID, AlgorithmStateAttachment asa)
                                                         throws Exception {
  Graph<Vertex, Edge> graph = MyAlgorithmPlugin.this.graphView.getGraph();
  int iNextStepID = -1;

  switch(stepID) {
    case 0: // Let a be an arbitrary edge and c:=0
      a = graph.getEdge(0).getID();
      c = 0;
      forEach = graph.getEdgeByIDSet();
      iNextStepID = 1;
      break;
    case 1: // For each e in E
      if(forEach.size() > 0) {
        e = forEach.get(0);
        forEach.remove(e);
        iNextStepID = 2;
      }
      else
        iNextStepID = -1;
      break;
    case 2: // If w(e) == w(a) then
      final Edge _e = graph.getEdgeByID(e);
      final Edge _a = graph.getEdgeByID(a);
      if(_e.getWeight() == _a.getWeight())
        iNextStepID = 3;
      else
        iNextStepID = 1;
      break;
    case 3: // c:=c+1
      c = c + 1;
      iNextStepID = 1;
      break;
  }

  return iNextStepID;
}
```

Instead of a real arbitrary edge (meaning randomly) we always take the first one. In addition we initialize our for-each list with the set of edges of the user-constructed graph. The variable forEach must be declared at the beginning of the class.

```
private class MyAlgorithmRTE extends AlgorithmRTE {

  // algorithm variables
  private int a;
  private int c;
  private int e;
  private Set<Integer> forEach;
```

We use a mathematical set, but this is not compulsory. Using this data structure it is easier to store the edge data as seen in the implementation of step 1 (case 0).
In case 1 we check whether there are more elements in forEach, and if so we request the next edge and remove it from the list. Otherwise we have reached the end of the algorithm, which means that we return -1. In case 2 we need the concrete objects to

compare the weights, which are attributes of an edge. Therefore, we use the identifiers that are placed in the corresponding algorithm variables e and a. If the weights are equal the algorithm continues with the last step otherwise a new edge must be investigated.

Finally we have to add the new variable forEach to the algorithm state.

```
@Override
protected void storeState(AlgorithmState state) {
  // ...
  state.addSet("forEach", forEach);
}

@Override
protected void restoreState(AlgorithmState state) {
  // ...
  forEach = state.getSet("forEach");
}

@Override
protected void createInitialState(AlgorithmState state) {
  // ...
  forEach = state.addSet("forEach", new Set<Integer>());
}
```

We have so far completed the pure algorithm without any visualization. But this is not the sense of LAVES. To add visualization we could highlight the edges a and e which might lead to a better and easier comprehension. Therefore we implement a specific method in the rte that applies visualization.

```
private void visualizeEdges() {
  GraphView<Vertex, Edge> graphView = MyAlgorithmPlugin.this.graphView;
  GraphView<Vertex, Edge>.VisualEdge ve;

  for(int i = 0; i < graphView.getVisualEdgeCount(); i++) {
    ve = graphView.getVisualEdge(i);

    if(ve.getEdge().getID() == a) {
      ve.setColor(MyAlgorithmPlugin.this.colorA);
      ve.setLineWidth(3);
    }
    else if(ve.getEdge().getID() == e) {
      ve.setColor(MyAlgorithmPlugin.this.colorE);
      ve.setLineWidth(2);
    }
    else {
      ve.setColor(GraphView.DEF_EDGECOLOR);
      ve.setLineWidth(GraphView.DEF_EDGELINEWIDTH);
    }
  }
}
```

In the method we iterate over all edges of the graph view and change the visual appearance. The visual edges that represent a and e should be colorized using the colors from the configuration (Method initialize) and a fix line width. All other edges are rendered using the default color and line width. Now we integrate the method into the algorithm.

```
@Override
protected int executeStep(int stepID, AlgorithmStateAttachment asa)
                                                 throws Exception {
  Graph<Vertex, Edge> graph = MyAlgorithmPlugin.this.graphView.getGraph();
  int iNextStepID = -1;
  switch(stepID) {
    case 0: // Let a be an arbitrary edge and c:=0
      a = graph.getEdge(0).getID();
      c = 0;
      forEach = graph.getEdgeByIDSet();
      sleep(500);
      visualizeEdges();
      sleep(500);
      iNextStepID = 1;
      break;
    case 1: // For each e in E
      if(forEach.size() > 0) {
        e = forEach.get(0);
        forEach.remove(e);
        sleep(500);
        visualizeEdges();
        sleep(500);
        iNextStepID = 2;
      }
      else
        iNextStepID = -1;
      break;
    case 2: // If w(e) == w(a) then
      final Edge _e = graph.getEdgeByID(e);
      final Edge _a = graph.getEdgeByID(a);
      if(_e.getWeight() == _a.getWeight())
        iNextStepID = 3;
      else
        iNextStepID = 1;
      break;
    case 3: // c:=c+1
      c = c + 1;
      iNextStepID = 1;
      break;
  }

  return iNextStepID;
}
```

To visualize the edges (see the sequences highlighted in bold) we use the sleep method of the rte. This method pauses the algorithm for a specified amount of milliseconds. Thereby it is possible to make the visualization visible. Otherwise the algorithm would be executed too fast and a rendering is not possible.

Before we conclude we have to look at the method rollBackStep. As mentioned above, this method should undo visualization. In the example above we visualize the edges in step 1 and 2. So if a user goes from step 2 to step 1 the visualization of step 2 must be undone. Because we always update every edge of the visual graph we can use the method visualizeEdges to achieve this. Our rollBackStep method may look like the following.

```
@Override
protected void rollBackStep(int stepID, int nextStepID) {
  switch(stepID) {
    case 0:
    case 1:
      visualizeEdges();
      break;
  }
}
```

Now we have implemented the runtime environment of our algorithm. The final step is to create an instance of the rte in the initialize method of the plugin and return it in the getRuntimeEnvironment property.

Export your plugin as a JAR

To install a plugin in LAVES we have to export it as a JAR file. The following description refers to the Eclipse IDE.

Select your plugin project in the package explorer and right-click on it. Choose Export … from the popup menu. Next, expand the item "Java" in the tree component of the wizard dialog and select "JAR file". When you click the "Next" button you see a list of your projects where the plugin project should be selected. You only need to specify the export destination, for instance "C:\LAVES Plugins\my-algorithm-plugin.jar". Afterwards you can click on "Finish" and your plugin JAR will be created.

Construct a language file

If you decide to provide a multilingual plugin you have to outsource each text sequence in a language file. The LAVESDK comes with an API to load such files like it was presented in section Method initialize. In this chapter we want to look at the structure of a language file.

A language file has three components. A language label which represents a placeholder in your plugin, one or more language identifiers which contain the translations, and comments.
A line that starts with // is marked as a comment and thus will be skipped. A line that starts with $ indicates a language label. The dollar sign must be followed by the label string, like $ALGO_NAME. Below a label there follow the language identifiers. An identifier must begin with # followed by the identifier name, like #en. The translation is assigned by =, like #en = My algorithm.

As a complete example with multiple translations, the following might be the content of a language file myLangFile.txt.

```
// the name of the algorithm that is implemented in the plugin
$ALGO_NAME
#en = My algorithm
#de = Mein Algorithmus
```

Go to sections Method initialize and Methods to describe the algorithm to see how to load and use the language file programmatically.

Test your plugin

You can export your plugin as a JAR and install it in LAVES to check if it runs successfully. But this needs much effort and prevents you from debugging your application. To reduce the test effort and to give debug support LAVESDK comes with a build-in sandbox that provides a lean version of LAVES with the base functionality. The only thing to do for you is to create a new class (e.g. PluginTest), inherit from Sandbox and add the following functions:

```
import java.awt.EventQueue;

import lavesdk.sandbox.Sandbox;

public class PluginTest extends Sandbox {

  private static final long serialVersionUID = 1L;

  public PluginTest() throws IllegalArgumentException {
    super(new MyAlgorithmPlugin(), "en");
  }

  public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
      public void run() {
        new PluginTest().setVisible(true);
      }
    });
  }


}
```

MyAlgorithmPlugin names your plugin implementation class. After that you can run your PluginTest class and check each method of your plugin. To debug your application you just need to set the breakpoints in a method you want to validate and run PluginTest in debug mode.