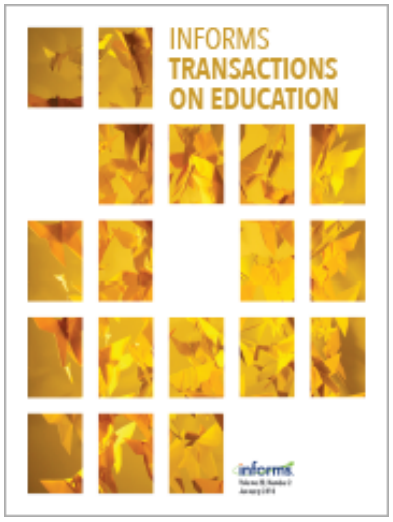


INFORMS Transactions on Education

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

LAVES: An Extensible Visualization Tool to Facilitate the Process of Learning and Teaching Algorithms

Dominik Kress, Jan Dornseifer

To cite this article:
Dominik Kress, Jan Dornseifer (2015) LAVES: An Extensible Visualization Tool to Facilitate the Process of Learning and Teaching Algorithms. INFORMS Transactions on Education 15(3):201-214. https://doi.org/10.1287/ited.2015.0140

Vol. 15, No. 3, May 2015, pp. 201–214
ISSN 1532-0545 (online)
INFORMS Transactions on Education

http://dx.doi.org/10.1287/ited.2015.0140

LAVES: An Extensible Visualization Tool to Facilitate the Process of Learning and Teaching Algorithms

Dominik Kress
Department of Management Information Science, University of Siegen, D-57068 Siegen, Germany,
dominik.kress@uni-siegen.de

Jan Dornseifer
University of Siegen, D-57068 Siegen, Germany, jan.dornseifer@student.uni-siegen.de

The Logistics Algorithms Visualization and Education Software (LAVES) is an open source project at the University of Siegen, aiming at supporting business and business informatics students in understanding the basic concepts of algorithms that are applied to solve problems arising in operations research, especially in logistics, by means of visualization. It allows students to create problem instances click-by-click with direct graphical feedback, offers a set of algorithm-related controls, presents execution-table-views as used by the students when manually processing algorithms, depicts and highlights the related pseudocode (including LaTeX formulas), and includes an exercise mode. LAVES is accompanied by a development kit (LAVESDK) that allows instructors (with Java knowledge) to implement course-specific algorithm visualizations (called plugins) to be used with LAVES. The development kit is generic and provides a broad range of tools. In this paper, we present the basic features of LAVES and LAVESDK and report on first classroom experiences and student feedback.

Keywords: education software; teaching software; algorithm visualization; algorithm animation
History: Received: October 2014; accepted: January 2015.

1. Introduction and Motivation

When teaching Operations Research (OR)/Management Science- (MS) related courses Logistics (required undergraduate-level course), Scheduling (elective graduate-level course), and Decision Support Systems (required graduate-level course) at the University of Siegen in Germany, we have oftentimes found business and business informatics students to be rather unacquainted with some of the very basic concepts of algorithms (loops, if-statements, etc.) and related mathematical notation (especially originating from set theory). As Grossman (2001) points out, this has been observed by many others, and as a consequence, the question of why (and how) even teach algorithms to business students is being debated controversially (see, for example, Grossman 2001, Robinson et al. 2003, Ólafsson 2004, and the references therein). Many argue that specific algorithms should not be part of an introductory OR/MS course in business schools (Horner 2003). However, it is "difficult to come to some general and lasting conclusion" (Ólafsson 2004, p. 33). In our opinion, it is indeed important for business and especially business informatics students to be introduced to some very basic algorithms. This stems from the fact that, when making decisions in their later careers, business students will almost for sure be making use of software tools that directly apply algorithms (enterprise resource planning systems, for example), and will thus need to be able to interpret corresponding output. They will then benefit from, for example, knowing the difference of exact and heuristic approaches that we can best teach by providing concrete examples. Additionally, business informatics graduates will oftentimes be working at "interface" positions between software developers and managers, and are thus expected to have specific knowledge about algorithms.

Grossman (2001, p. 56) warns: "Trying to teach mathematical material to the unprepared [...] will serve to destroy interest in mathematics and the valuable tools of management science." However, he adds that "the innumerate crave numeracy" and that these "students are grateful to a teacher who can enable them to use numbers to think critically." While in this latter context, step-by-step visualization of algorithms on the blackboard in our lectures resulted in positive feedback, our students have continuously been requesting additional explanations and references in our office hours as well as in discussions immediately before and after the lectures. We have furthermore

been faced with the request for additional learning resources in end-of-semester course evaluations. Some exemplary quotes of the students ask for "more tutorials," "more examples," "more extensive examples," "slower speed of explanation when teaching the examples," "more efficient use of the time in the lectures," and "use of additional media" (quotes translated from German). Given the goal to react to these requests without reducing the coverage of our lectures and the fact that there is no possibility to expand office hours or offer additional lectures, an interesting approach is the support of the students (and lecturers) by computer-aided learning activities such as the use of software to visualize algorithms. Prior research has underlined the fact that OR students can benefit from these activities. Seal and Przasnyski (2003, p. 35), for example, analyze several computer-based methods to facilitate the students' learning in OR/MS courses and find that technology can help "students (who come from diverse backgrounds and thus find the mathematical nature of the OR/MS course difficult) to overcome their deficiencies on their own and get up to speed without holding up the class." Similarly, Liebman (1998) states that a considerable reduction of the complexity of learning OR can be achieved by making use of "well-crafted" learning activities. In the visualization context, a well-crafted software tool will have to guide the students "into activities that engage them with the visualization," for example, by letting them provide input to the algorithms being visualized or by interrupting visualizations with questions that have to be answered before the algorithm continues (Naps 2005, p. 50). That is, "the most successful educational uses [...] are those in which the technology is used as a vehicle for actively engaging students in the process of learning algorithms" (Hundhausen et al. 2002, p. 284). From the instructor's perspective, pedagogic benefits can be provided with reasonable investment of effort, resulting in a lot of professional satisfaction and potentially improved course evaluations (Seal and Przasnyski 2003).

The idea of using visualization techniques in helping to comprehend the functionality of algorithms is not new. In fact, algorithm animation (AA) or algorithm visualization (AV) can be considered as a subfield of the scientific discipline of software visualization (SV). In broad terms, it is concerned with "the visualization of the behavior of an algorithm" (Diehl 2007, p. 87). It "depicts the execution of an algorithm as a discrete or continuous sequence of graphical images, the viewing of which is controlled by the user" (Grissom et al. 2003, p. 87). It thus "gives a visual representation [...] on a higher level of abstraction than the actual code" (Karavirta and Korhonen 2006, p. 95). A variety of related tools for educational purposes have been developed. A detailed overview is given on AlgoViz.org (2013) (cf. also Baloukas et al. 2009, Karavirta and Shaffer 2013). The most sophisticated tools are briefly described as follows:

- ALVIE (Crescenzi 2012) is a "postmortem" AV tool, meaning that the user views the visualization after the algorithm has been executed. ALVIE is based on the interesting event paradigm, i.e., the implementation of an algorithm is complemented by visualization information at interesting events of its execution, for instance, when two elements in a sorting algorithm are exchanged.
- ANIMAL (Rößling 2013, Rößling and Freisleben 2002) is an AV system that allows the generation of animations in a graphical user interface, by scripting (AnimalScript), or by using an application programming interface (API). It provides a variety of controls when viewing animations. Users can access a generator framework that allows viewing animations of a number of predefined AVs.
- JHAVÉ (Naps et al. 2011, Naps 2005) is a support environment for multiple AV systems. It provides a drawing context for these systems. Additionally, it provides ways of synchronizing graphical displays with a set of controls, information and pseudocode windows, input generators, stop-and-think questions, and meaningful content generation tools.
- TRAKLA2 (Korhonen 2009, Korhonen et al. 2003) is an AV environment that provides algorithm-related exercises that are graded automatically. The algorithms are randomly instantiated and proceed with the user (step by step) directly manipulating the visual representations of the data structures the algorithm employs. Exercises can be developed by using an API.

All of these tools already include a broad variety of AVs. However, as mentioned by Rößling and Freisleben (2002, p. 341), "the interpretation of what algorithm animation is differs between the tools, and thus each provides a slightly different approach." Indeed, none of the aforementioned tools directly fits all of our goals. There are two main aspects to this. First, our tool needs to be sufficiently flexible to work with self-defined (rather than randomly generated) problem instances, i.e., there is a need for instance editors that enable the user to define any instance of interest. This is because lecturers usually aim at using the software in class as a supporting tool to illustrate the dynamics of algorithms for specific problem instances and because students (partly) use the software to analyze the behavior of algorithms for special cases to gain an in-depth understanding. Second, because it is well understood that an AV exercise mode needs to be sufficiently interactive (Hundhausen et al. 2002, Naps 2005, see above), we aim at including representations of (data and problem) structures (henceforth referred to as "views") that

are directly manipulable by the user. For example, we want vertices to be selectable by clicking rather than by typing their "names" into text fields. This induces a high degree of flexibility when designing exercises. Whereas TRAKLA2 features manipulable views, it is based on random generation of problem instances, and thus lacks the option of working with self-defined problem instances by using instance editors. ALVIE, ANIMAL, and JHAVÉ, in contrast, integrate instance editors but are rather restricted regarding user interaction. Hence we decided to implement a new AV tool from scratch that specifically targets business and business informatics students, and meets the requirements and main concepts described in §2. This eventually led to the development of the open source Logistics Algorithms Visualization and Education Software (LAVES). Because we intended to make this software usable at other universities, we decided to provide an additional software development kit (LAVESDK). LAVES and LAVESDK are written in Java. You can download LAVES and LAVESDK, including comprehensive documentation, at the Department of Management Information Science at the University of Siegen: http://www.wiwi.uni-siegen.de/mis/software/.

The purpose of this paper is to present the basic features of LAVES and LAVESDK and report on first classroom experience and student feedback. By doing so, we hope to motivate others to incorporate visualization tools like LAVES into their didactic concepts. The paper is structured as follows. In §2, we will outline the requirements and concepts upon which the software is built. Subsequently, we will describe the related main features of LAVES in §3. We will then briefly summarize first classroom experience in §4, before describing the basics of LAVESDK in §5. The paper closes with a conclusion and an outlook in §6.

2. Requirements and Main Concepts

Our focus is to provide an interactive learning environment for students and a platform for instructors to (ad hoc) visualize the dynamic processes of algorithms in lectures. To make the software usable at other universities, we need to make sure to

- provide a flexible architecture, allowing easy and fast modification of the software and AVs as well as the development of new AVs,
- allow everyone to download, use, study, share (copy), and modify the software and the source code, and
- provide a platform-independent software.

The latter two requirements have been addressed by implementing in Java and releasing the software under the terms of the GNU General Public License (Free Software Foundation 2014). Regarding the extensibility of the software, we have decided to provide a plugin architecture (as opposed to providing a scripting language or a function library) as a compromise between flexibility and ease of implementation. A plugin architecture enables the user to dynamically install modules (in our case, AVs called plugins) to extend the functionality of the host application (in our case, LAVES) during its runtime.

According to the categorization of Khuri (2001), we are faced with "novice users" (on the student side) who, in our case, are beginners in reading and understanding pseudocode. In this context, Khuri (2001) suggests to implement as many of the following design features as possible:

- Graphical and consistent means of visualization control.
- Similar instructional structure of all visualizations, e.g., when defining user-specific problem instances.
- Comprehensive help files.
- Short "quizzes" or similar opportunities for students to evaluate whether they have understood the material and to exercise the use of the visualization tools (cf. also Hundhausen et al. 2002, Naps 2005).

Based on these desired design features, we have formulated the following requirements and main concepts that LAVES is based on:

- Including intuitive graphical editors and interfaces to easily construct problem instances (especially graphs and networks) using mouse and keyboard.
- Including VCR-like controls to start, pause, resume, and stop the execution of an AV, to set breakpoints, to modify the execution speed, and to manually move backward and forward on the time axis to prior and later states of an algorithm.
- Representing algorithms by pseudocode in natural language (as presented in the lecture notes), including LaTeX notation, symbols, and formulas.
- View-concept: Splitting of AVs into multiple "views" to represent all structures used by students when manually processing algorithms, e.g., graph-views, execution-table-views, matrix-views, etc.
- Including an exercise mode (optional), allowing direct manipulation of the different views on the algorithm, and thus resulting in a high degree of interaction between user and visualization, e.g., by selecting vertices in graph-views, modifying entries in matrix- or execution-table-views, etc.
- Allowing users to customize AVs according to their wishes, e.g., by changing highlighting colors to concentrate on specific effects of an algorithm.
- Giving the software a modern look and feel (in conformance with the expectations of the users).

Table 1 Vogel's Approximation Method

Input: Complete undirected weighted graph $G = (V, E)$ with an even number of vertices	
Output: Perfect matching M	
1. Initialization:	Let $M := \varnothing$. Let $L := V$ be the set of all vertices of the graph.
2. Stop criterion:	If $\lvert L\rvert = 2$, add the edge between the remaining vertices to M and stop. Otherwise, go to step 3.
3. Regret:	For each $v \in L$, determine the regret as follows: Let $v_1(v) \in \arg\min\{c(v, v') \mid v' \in L\}$, $v_2(v) \in \arg\min\{c(v, v') \mid v' \in L \backslash \{v_1(v)\}\}$, $regret(v) := c(v, v_2(v)) - c(v, v_1(v))$.
4. Matching update:	Let $v \in L$ be an arbitrary vertex with the largest regret. Add $(v, v_1(v))$ to the matching M.
5. Update L:	Set $L := L \backslash \{v, v_1(v)\}$ and go to step 2.

• Including multiple languages to support foreign students and to allow lecturers to teach in their native language.

In the subsequent sections of this paper, we will provide details on how we address these requirements.

3. LAVES: The Host Application

Consider a variant of Vogel's approximation method (Reinfeld and Vogel 1958) as an example heuristic that is to be visualized. Given an undirected graph $G = (V, E)$ with vertex set V and edge set E, a matching $M \subseteq E$ in G is defined as a set of pairwise nonadjacent edges. A matching is perfect if every vertex of the graph is an end point of one of the edges of the matching. If we additionally consider edge weights, the weight of a matching is defined as the sum of the weights of its edges. Vogel's approximation method (see Table 1) determines a perfect matching of small weight in a complete undirected weighted graph $G = (V, E)$ (equivalently, denoted by K_n) with $|V| = n$ even, and edge weights $c\colon E \to \mathbb{R}_0^+$.

LAVES, the host application, is divided into five functional areas: menu bar, tool bar, information bar, status bar, and display area. Figure 1 shows these areas after having started the plugin for Vogel's approximation method.

The menu bar contains all of LAVES's functions, whereas the tool bar restricts itself to the most important ones. In particular, the tool bar contains the VCR-functionality, i.e., start/resume, play and pause (execution of the algorithm is paused automatically after having executed the next step), pause, stop, previous step, next step, skip breakpoints, execution speed, etc. Furthermore, the tool bar contains buttons to save and open instance files, start a new plugin, or open the exercise mode. Finally, it includes buttons to construct graphs by defining an adjacency matrix, as well as options to automatically construct complete or complete bipartite graphs or to rearrange vertices and edges of graphs and check graphs for specific features. The elements in the tool bar may vary depending on the plugin. The information bar depicts information on the input needed for the selected algorithm. Furthermore, the user is able to access help files regarding the algorithm. The status bar contains additional information on the specific algorithm and potential runtime errors.

The heart of LAVES is the display area, where the actual AV (as well as the exercise mode) of a specific plugin takes place. Before starting the visualization, the user can open or construct a problem instance, for example, by using a graph editor to add vertices, edges, weights, labels, etc. In Figure 1, the graph editor, including its buttons is located on the right. In addition, the display area depicts the algorithm pseudocode. Here, the user can define breakpoints and open additional information on the steps of the algorithm. Furthermore, the display area includes a legend (specifying the highlighting colors, etc.) and the additional views on the visualization. In the case of Vogel's approximation method, these views include the set L (candidate list), Matching M, adjacency matrix, and an area to determine regret values in an execution-table (Figure 1). The partitioning into multiple views improves the clarity and eases the handling of the visualization. As a result, it improves the usability of the software due to a consistent layout, structure, and look, being in conformance with the expectations of the users with respect to the teaching methods applied in the lectures. The user is free to exclude specific views from the visualization. This can, for example, be helpful if the user wishes to process manipulations of representations of corresponding structures by hand.

Once a problem instance has been defined, the user starts the visualization by setting the execution speed and clicking the play button in the tool bar. LAVES then begins executing the algorithm. One after another, the steps of the algorithm and the related objects in the different views are accentuated. In Figure 2, for example, the regret values in the second iteration of Vogel's approximation method are being determined. As edge $(2, 3)$ has already been added to M, it is highlighted (bold edge) in the graph-view, and the corresponding rows and columns in the adjacency matrix have been crossed out. The vertices of the candidate list L are highlighted in the graph-view (light blue). The regret values for vertices 1 and 4 have already been determined. Green highlights in the adjacency matrix indicate the relevant elements needed to compute $regret(6)$.

During the execution of an algorithm, the user can take different actions. Most important, the user can

Figure 1 Functional Areas of LAVES; the Display Area Includes the Graphical Editor to Construct Graphs

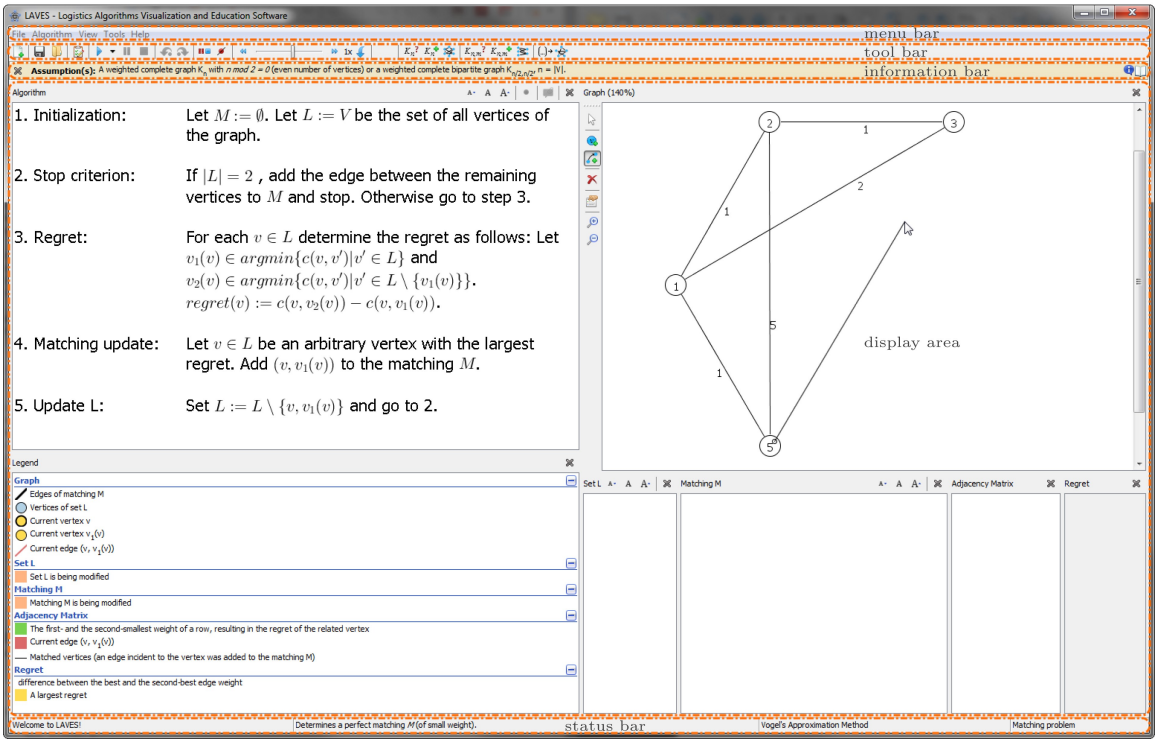


Figure 2 Visualization of Vogel's Approximation Method

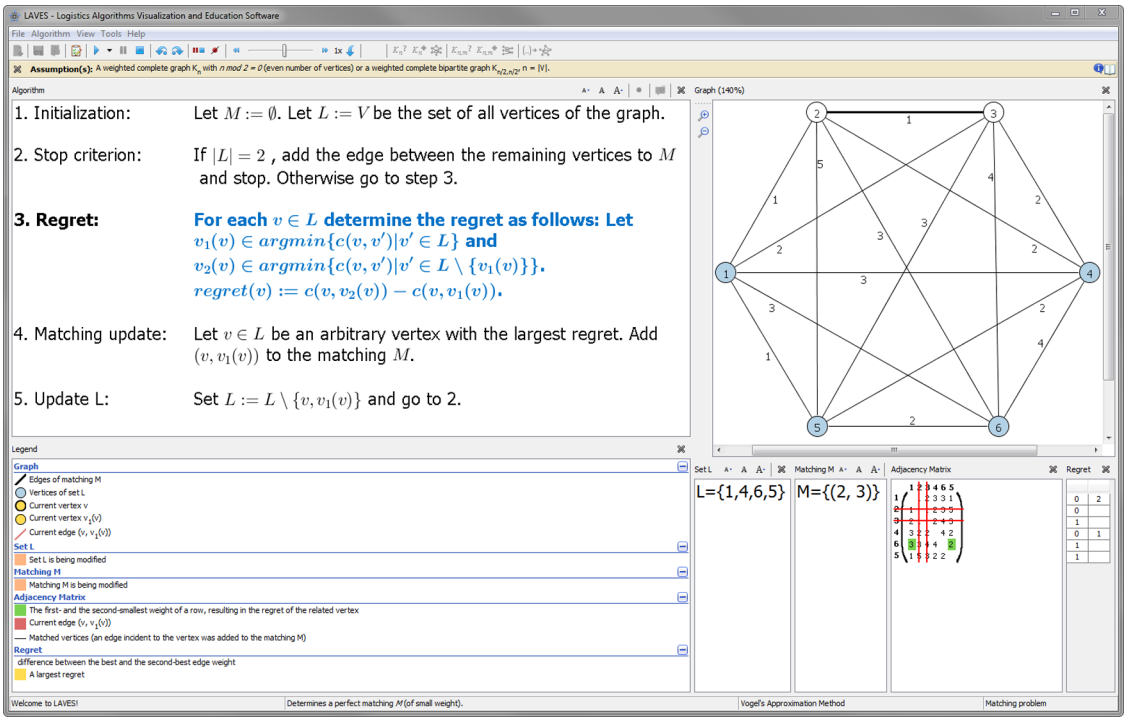
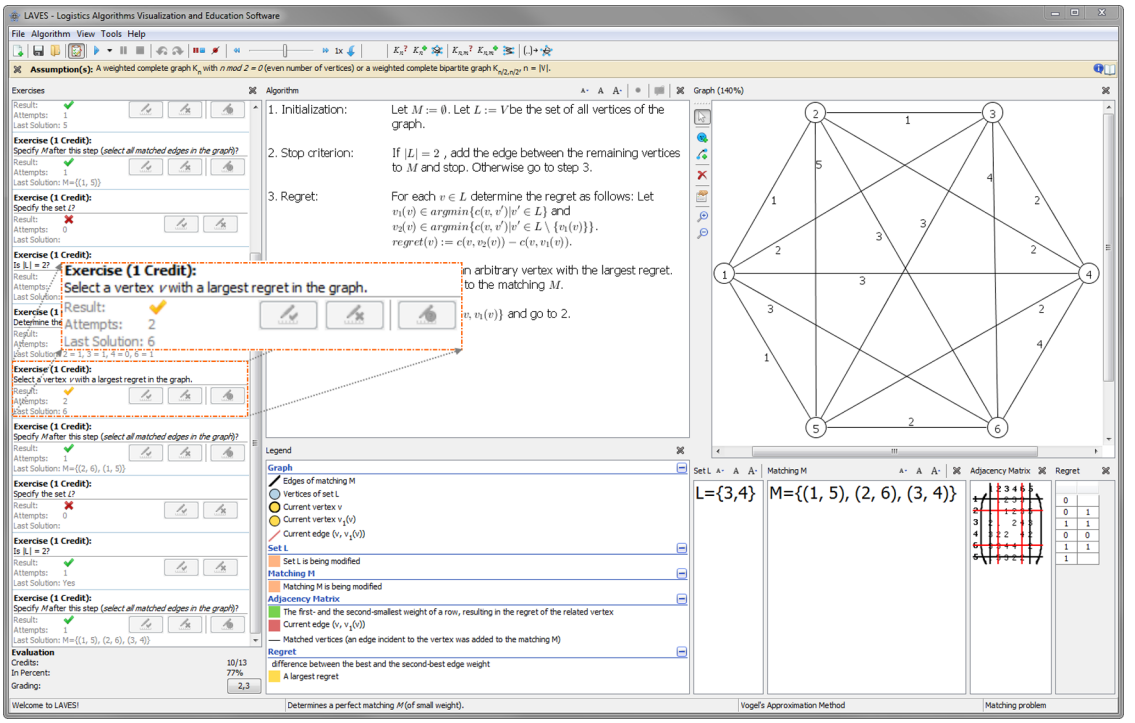


Figure 3 Exercise Mode for Vogel's Approximation Method



modify the execution speed or pause the visualization. By clicking the appropriate buttons in the tool bar, the user can manually move backward and forward on the time axis to prior and later states of the algorithm. When moving backward, visualization effects are automatically undone.

Upon entering the exercise mode, the algorithm automatically pauses at predefined points and asks the user to answer questions (see Figure 3). To answer a specific question, the user is asked to modify views on the algorithm, e.g., by selecting the edges of the current matching M in the graph-view or typing in the regret values in the corresponding execution-table-view. Of course, multiple choice questions or textual answers to specific questions may also be included in the exercise mode. This induces a high degree of flexibility when designing exercises to a specific algorithm. At any point of the exercise mode, questions may be skipped. They are then answered and visualized by the software. The user gets direct visual feedback on the number of attempts needed to solve an exercise (green check mark, yellow check mark, red cross). After finishing the exercise mode, the user is graded.

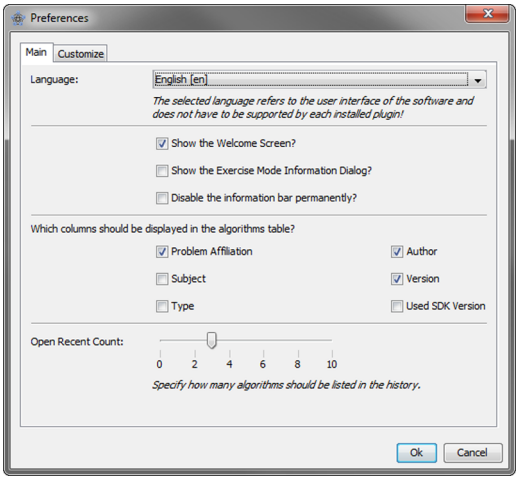
Figure 4 shows some options concerning the user-specific customization of LAVES (Figure 4(a)) and its plugins (Figure 4(b)). The dialogues are accessible through the LAVES menu bar. By selecting colors and other attributes of an AV, the user is, for example, enabled to exclude elements from being shown or to highlight items of the visualization. Hence, specific effects of interest to the user can be accentuated. Languages can easily be edited and added to LAVES by simply editing language files (see Figure 5). So far, LAVES and its plugins are available in English and German.

4. First Classroom Experience

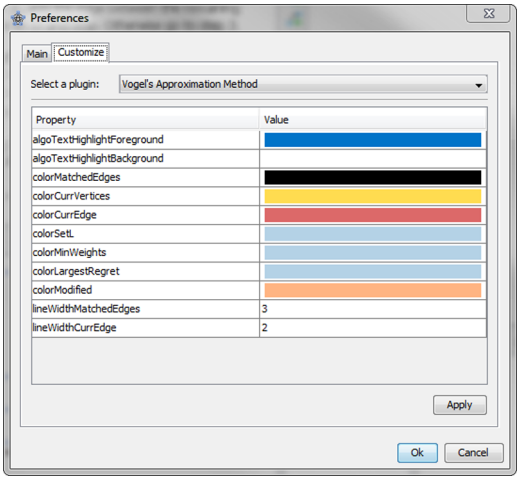
We incorporated LAVES (in its alpha and beta versions) into one of our courses in the summer semester 2014 for the first time. Logistics is a (typically fourth semester) undergraduate-level course at the School of Economic Disciplines (Faculty 3) at the University of Siegen. The (around 40–70) participants are mostly business and business informatics students. In both groups, there is a lack of what we consider classical informatics skills (programming concepts and languages, mathematics, etc.). The course is comprised of basic (graph theoretic-) models and algorithms related to questions arising in logistics. Every week, there is a two-hour lecture and a one-hour tutorial, where algorithms are illustrated based on example instances and voluntary homework assignments are discussed. LAVES provides plugins for most algorithms taught in this course. So far, we provide plugins for heuristic and exact algorithms for the traveling salesman problem (2-opt, nearest neighbor algorithm), vehicle routing problem (savings algorithm), shortest path problem (Dijkstra's algorithm, Floyd-Warshall algorithm), matching problems (Vogel's approximation

Figure 4 User Customization

(a) General settings



(b) Plugin settings



method, Hungarian method, a greedy algorithm), flow problems (Ford-Fulkerson algorithm), and arc routing problems (determining Eulerian cycles).

We used the software in addition to the traditional use of blackboard, presentation slides, and lecture notes in class. While at first, we intended to completely replace the visualization of algorithms on the blackboard by using LAVES, we almost immediately received feedback that students (on average) prefer a mix of blackboard and software usage because this usually gives the students more time to think over the things that have been said. We thus changed the mode very early in the semester to first presenting representations of the structures that the algorithms employ on the blackboard, and afterward running the related plugins in LAVES with the students being aware of the meaning of the plugins' views that are directly related to these structures. We feel that this mode of using LAVES as a supporting tool to illustrate the dynamics of algorithms incorporates smoothly into the lectures, especially when example instance files are generated and loaded ahead of the lectures. The sharing of these example instance files via our course management system has additionally considerably reduced the number of (as well as improved the quality of) algorithm-related questions before and after the lectures and in our office hours, as students were enabled and encouraged to repeat the steps of an algorithm at home at their own speed without having to "rebuild" example instances themselves. The same holds for questions concerning the students' homework assignments as they were enabled to exchange problem instances between each other and with the lecturers. Note that, so far, we have not used the exercise mode of LAVES specifically as part of the grading of the students. However, we have encouraged the students to make use of this mode because it is well suited for self-controlling the understanding of details of an algorithm.

The specific effects of LAVES' use are hard to measure. We have, however, received very positive student feedback in conversations and in the end-of-semester evaluation of our logistics course. Here, students grade features of the course and the lecturers on scales of 1 (best) to 5 (worst). Three statements of the evaluation (in our opinion) directly or indirectly relate to software use (statements translated from German):

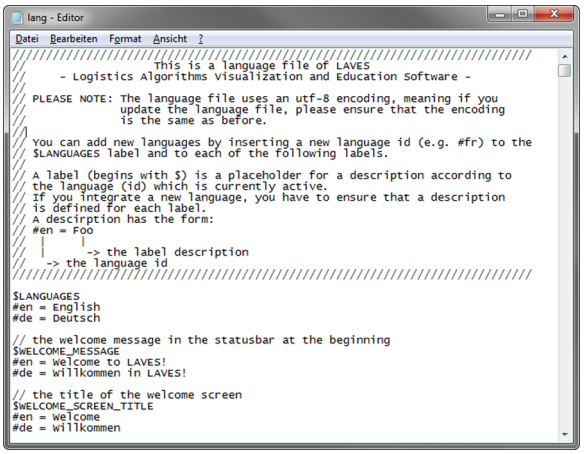
- "The use of media and other practical presentation forms appear to me to be reasonably imbedded in the didactic concept of the course."
- "In my opinion, the learning effect of this course is …"
- "Overall, I evaluate this course as …"

While, of course, not being a scientifically legitimate method to draw final conclusions, first insights into LAVES' use can be gained by comparing the student's grading of these statements before (i.e., summer semester 2013) and after (i.e., summer semester 2014) having incorporated LAVES into the logistics course. Seeing improvements (of the arithmetic mean of the student grades; in percent) of 5.2%, 10.2% and 14.9%, respectively, we find a tendency toward the students benefiting from the software.

Additionally, almost 17% of the students emphasized the positive effects of LAVES in their free-text comments on the evaluation forms, where the students are asked to respond to the following questions (questions and answers are translated from German):

1. What did you particularly enjoy in this course?
2. What do you think could be improved?
3. How would you assess the teaching and learning methods of this course?

Figure 5 LAVES Language File



LAVES-related answers to question 1 include:

- "The material of the course is being enhanced through the use of LAVES. Its use eases the learning process."
- "The fact that the lecturer offers a software tool (LAVES) accompanying the lectures and tutorials."
- Short answers, like "(using) the software (!)," "LAVES."
- "Clear and descriptive presentation."

An answer to question 3 states that "LAVES is a very good tool for self-controlling the learning progress." There are no directly LAVES-related answers to question 2. However, some students criticize that the "speed of the lectures" is generally too high. As mentioned above, this could be because there may not be enough time to think over the things that have been said when traditional teaching methods are pushed aside by the software. After all, writing by hand (even when only copying the things written on the blackboard) promotes and strengthens the thinking and learning process of the students. Future research will have to specifically analyze these and other potential risks resulting from using LAVES.

5. LAVESDK: The Software Development Kit

LAVESDK is a Java- and Swing-based API. It is the foundation of the plugin architecture that is briefly described in §2. It thus allows the development and modification of AVs (plugins) to be used with LAVES. As a result, rather than making a lecture adapt to the software, the software can be designed to address the specific needs of a course by allowing instructors to implement visualizations of algorithms that are being taught in their specific courses. In this section, we will give a brief overview of the components of the development kit. Furthermore, we will present excerpts of example implementations of these components to indicate the level of Java knowledge needed to implement a plugin. We will include comments in these code excerpts to make them sufficiently self-explanatory. Additionally, we will provide some explanatory notes in the text of this section. The excerpts are based on a straight forward algorithm that determines an edge of minimum weight in a weighted graph $G = (V, E)$ with $|E| > 0$ and edge weights c: $E \to \mathbb{R}_0^+$ (see Table 2). In the following, we will refer to this algorithm as min-weight.

Table 2 Find an Edge of Minimum Weight

Input: Weighted graph $G = (V, E)$, $|E| > 0$
Output: Edge e of minimum weight

1. Initialization:	Let $e \in E$ be an arbitrary edge of the graph and set $w := c(e)$. Let $A := E \setminus \{e\}$ be the set of the remaining edges.
2. Stop criterion:	If $A = \varnothing$, then e is an edge of minimum weight; stop. Otherwise, go to step 3.
3. Verification:	Let $a \in A$. If $c(a) < w$, then set $w := c(a)$ and $e := a$. Set $A := A \setminus \{a\}$ and go to step 2.

The interface *AlgorithmPlugin* is the point of entry when developing an AV. Each visualization is defined as an autonomous plugin, being composed of properties characterizing the visualization, customization and event handling methods, as well as methods to open and save instance files. To customize the settings (controlling, for instance, the behavior or representation of the plugin or the visualization) that the user can adjust before opening a plugin, specific methods may be overridden. When developing a new plugin, one will have to create a class that implements the interface AlgorithmPlugin and overrides the available methods.

The class *View* is the heart of the view-concept as described in §§2 and 3. It abstracts a view on an algorithm, in which algorithm-specific data can be visualized. LAVESDK contains multiple preimplemented views such as views for generating, displaying and editing graph and network structures, displaying pseudocode, execution-tables, matrices, legends, and exercises. A specific view for an algorithm can be realized by inheriting from the base class. By creating instances of the class *ViewGroup*, which allows the grouping of multiple views, the views on an algorithm can be arranged vertically and horizontally in a plugin. The relative sizes of the groups are initially defined by the developer of a plugin and are adjustable by the user through draggable vertical or horizontal bars (sashes) during runtime.

Listing 1 depicts an excerpt of a potential implementation of AlgorithmPlugin for min-weight. We restrict ourselves to presenting the most important

Listing 1 Example: AlgorithmPlugin

```
1  public class MinWeightEdgeAlgorithmPlugin implements AlgorithmPlugin {
2    LanguageFile langFile;
3    PluginHost host;
4    AlgorithmText algoText;
5    AlgorithmTextView algoTextView;
6    GraphView<Vertex, Edge> graphView;
7    MinWeightEdgeAlgorithmRTE rte;
8    //...
9    @Override
10   public void initialize(PluginHost host, ResourceLoader resLoader, Configuration config) {
11     try {
12       // Get language file of our plugin and include language file of the host.
13       langFile = new LanguageFile(resLoader.getResourceAsStream(
             "main/resources/langFile.txt"));
14       langFile.include(host.getLanguageFile());
15     } catch (IOException e) {
16       langFile = null;
17     }
18     this.host = host;
19     this.algoText = loadAlgorithmText();
20     // Get the titles of our views from the language file using the language ID that is set in
           the host.
21     final String algoTextViewTitle = LanguageFile.getLabel(langFile, "ALGOTEXTVIEW_TITLE",
           host.getLanguageID(), "Algorithm");
22     final String graphViewTitle = LanguageFile.getLabel(langFile, "GRAPHVIEW_TITLE",
           host.getLanguageID(), "Graph");
23     // Create views.
24     this.algoTextView = new AlgorithmTextView(host, algoTextViewTitle, algoText);
25     this.graphView = new GraphView<Vertex, Edge>(graphViewTitle, new SimpleGraph<Vertex,
           Edge>(false), new DefaultGraphFactory(), null, true, langFile, host.getLanguageID());
26     // Create runtime environment of the algorithm.
27     rte = new MinWeightEdgeAlgorithmRTE();
28     // ...
29   }
30   @Override
31   public AlgorithmRTE getRuntimeEnvironment() {
32     return rte;
33   }
34   @Override
35   public void beforeStart(RTEvent e) {
36     // Check whether the assumptions are fulfilled, otherwise cancel. Furthermore, reset the
           views.
37   }
38   @Override
39   public void onCreate(ViewContainer container, PropertiesListModel creatorPreferences) {
40     // Initialize the graph-view with a new simple graph each time the user creates a new
           plugin entity.
41     graphView.setGraph(new SimpleGraph<Vertex, Edge>(false));
42     // Create a group for the algorithm text view and the graph-view.
43     ViewGroup vg = new ViewGroup(ViewGroup.HORIZONTAL);
44     vg.add(algoTextView);
45     vg.add(graphView);
46     vg.setWeights(new float[] {0.4f, 0.6f});
47     container.setLayout(new BorderLayout());
48     container.add(vg, BorderLayout.CENTER);
49   }
50   // ...
51 }
```

functionalities that need to be implemented to construct an executable plugin. More details are provided in the documentation of LAVESDK.

The initialization routine *initialize*( ) (lines 10–29) is called when the plugin is instantiated by LAVES. Here, all data relevant during runtime of the plugin is initialized, including the language file (lines 11–17) and the generation of the two different views (lines 21–25), one view for depicting the pseudocode (*AlgorithmTextView*) of min-weight (lines 21 and 24) and one view for the generation and visualization of graphs (*GraphView*, lines 22 and 25). Furthermore, the runtime environment of the algorithm is instantiated in line 27 (see below for details). It has to be returned by the public method *getRuntimeEnvironment*( ) (lines 31–33). The initialization routine can include additional initializations (line 28). For example, it can load (or set) visualization colors or other values from the configuration.

The method *beforeStart*( ) (lines 35–37) is called when the algorithm is started by the user. Here, we can, for example, check whether the input data provided by the user is correct and interrupt the execution of the algorithm if this is not the case.

The method *onCreate*( ) (lines 39–49) displays the plugin in LAVES. It provides a *ViewContainer* in which the views of the plugin can be arranged arbitrarily. The method *onCreate*( ) is called on the plugin-instance once the user opens a new plugin in LAVES. Hence we first reset the graph in its corresponding view (line 41) to delete previously depicted graphs. We then create a horizontal ViewGroup (line 43). The views of interest are then added to the group (lines 44–45) and their relative sizes are specified (line 46). Finally, the layout of the ViewContainer is defined (based on the Swing Layout Managers) and the ViewGroup is added to the container (lines 47–48).

The class *AlgorithmText* contains the pseudocode of an algorithm. The pseudocode is divided into paragraphs (*AlgorithmParagraph*). Paragraphs are composed of steps (*AlgorithmStep*). Each step can have a corresponding exercise (*AlgorithmExercise*) and supplementary information.

In our example, the min-weight pseudocode is loaded from the language file in line 19 in Listing 1. An implementation of the corresponding method *loadAlgorithmText*( ) is depicted in Listing 2.

As shown in Table 2, the min-weight pseudocode is composed of three paragraphs (Initialization, Stop criterion, and Verification), which are composed of a number of steps. This structure is represented by constructing corresponding AlgorithmParagraph- and AlgorithmStep-objects in lines 4–17 of Listing 2. The static method *getLabel*( ) of the *LanguageFile* class enables the programmer to load language-specific labels for each paragraph and step of the pseudocode. Consider the example of the first paragraph of min-weight in line 4 in Listing 2: To load the correct label, the language file "langFile" that was declared in line 2 and constructed and loaded in lines 11–17 in Listing 1 is provided with an identifier "AP_INIT" of the paragraph, the language ID (e.g., "en" for English) of the host (adjustable by the user), and a default value "1. Initialization" (refer to Figure 5 for an example language file). The default value is used in case of errors during loading the label from the language file, e.g., when the identifier or the language ID is not found. LaTeX formulas can be integrated by encapsulating them with _latex{ and }. The commands within _latex{ and } have to be supported by the JLaTeXMath API (JLaTeXMath 2014).

The class *AlgorithmRTE* is the runtime environment of an AV. Each *AlgorithmPlugin* has a runtime environment in which the corresponding algorithm is implemented and visualized. It is constructed and modified in a separate or an inner class that inherits from AlgorithmRTE, by overriding protected methods that implement the functionality of the algorithm, including the logic of retaining and restoring specific states of an algorithm (with the states being directly related to the division of the pseudocode into steps), starting and pausing an algorithm, the stepwise execution of an algorithm (backward and forward on the time axis), as well as the undoing of visualization effects when stepping backward. A state of an algorithm is defined by the values of all of its variables. It is retained before executing the corresponding step. If, at one point of the execution of an AV, the user steps back to a prior step, the appropriate state is restored.

Listing 3 shows an excerpt of an example implementation of AlgorithmRTE for steps 1–3 of min-weight. Note that the corresponding step IDs (i.e., 1, 2, 3) are defined in Listing 2. Additional explanations concerning the runtime environment of an AV are provided in the documentation of LAVESDK.

As indicated above, the runtime environment of min-weight is implemented as an inner class in the plugin class MinWeightEdgeAlgorithmPlugin (see also Listing 1). Hence we can directly access the corresponding member variables. The additional variables needed to implement min-weight (defined in Table 2, i.e., e, A, w, and a) are declared in lines 5–8 in Listing 3. Note that the set A contains identifiers of edges rather than the specific edge objects.

The methods *storeState*( ) for retaining variables (lines 13–18) and *restoreState*( ) for restoring variables (lines 20–25) of a state are needed to keep track of the history of the algorithm. This enables the software to move backward and forward on the time axis to prior and later states of min-weight. When restoreState( ) is called, the programmer has to make sure that visualization effects are undone. The corresponding logic is coded in the method *rollBackStep*( ) (lines 73–82).

Listing 2 Example: Loading the Pseudocode from the Language File

```
1  private AlgorithmText loadAlgorithmText() {
2    final AlgorithmText at = new AlgorithmText();
3    // Create paragraphs.
4    final AlgorithmParagraph apInit = new AlgorithmParagraph(at, LanguageFile.getLabel(langFile,
         "AP_INIT", host.getLanguageID(), "1. Initialization:"), 1);
5    final AlgorithmParagraph apStop = new AlgorithmParagraph(at, LanguageFile.getLabel(langFile,
         "AP_STOP", host.getLanguageID(), "2. Stop criterion:"), 2);
6    final AlgorithmParagraph apCheck = new AlgorithmParagraph(at,
         LanguageFile.getLabel(langFile, "AP_CHECK", host.getLanguageID(), "3. Verification:"),
         3);
7    // Create steps and load step description from language file.
8    // 1. Initialization
9    final AlgorithmStep step1 = new AlgorithmStep(apInit, LanguageFile.getLabel(langFile,
         "ASTEP1_INIT", host.getLanguageID(), "Let _latex{$e \\in E$} be an arbitrary edge of the
         graph and set _latex{$w := c(e)$}. "), 1);
10   final AlgorithmStep step2 = new AlgorithmStep(apInit, LanguageFile.getLabel(langFile,
         "ASTEP2_INIT", host.getLanguageID(), "Let _latex{$A := E \\backslash \\{e\\}$} be the
         set of the remaining edges.\n\n"), 2);
11   // 2. Stop criterion
12   final AlgorithmStep step3 = new AlgorithmStep(apStop, LanguageFile.getLabel(langFile,
         "ASTEP3_STOP", host.getLanguageID(), "If _latex{$A = \\varnothing$}, then _latex{$e$} is
         an edge of minimum weight; stop. "), 3);
13   final AlgorithmStep step4 = new AlgorithmStep(apStop, LanguageFile.getLabel(langFile,
         "ASTEP4_STOP", host.getLanguageID(), "Otherwise, go to step 3.\n\n"), 4);
14   // 3. Verification
15   final AlgorithmStep step5 = new AlgorithmStep(apCheck, LanguageFile.getLabel(langFile,
         "ASTEP5_CHECK", host.getLanguageID(), "Let _latex{$a \\in A$}. "), 5);
16   final AlgorithmStep step6 = new AlgorithmStep(apCheck, LanguageFile.getLabel(langFile,
         "ASTEP6_CHECK", host.getLanguageID(), "If _latex{$c(a) < w$}, then set _latex{$w :=
         c(a)$} and _latex{$e := a$}. "), 6);
17   final AlgorithmStep step7 = new AlgorithmStep(apCheck, LanguageFile.getLabel(langFile,
         "ASTEP5_CHECK", host.getLanguageID(), "Set _latex{$A := A \\backslash \\{a\\}$} and go
         to step 2."), 7);
18   return at;
19 }
```

The functionality of processing the steps of min-weight is implemented in the method *executeStep*( ) (lines 34–71). Based on the step ID of the step to be executed, a switch statement chooses the correct logic of the step. The overall procedure is controlled by the return value of executeStep( ) (line 70). The method *sleep*( ) is used to automatically pause the runtime environment of min-weight for a given time interval in milliseconds. Assume that, to visualize min-weight, we want to restrict ourselves to coloring specific edges of the graph (edge a in orange, the edges in A in blue, and the remaining edges in the standard color defined by the user) and increasing the line width of edge e. The method *visualizeEdges*( ) (lines 83–99) runs through the edges of the graph-view, sets the corresponding properties of the edges, and "repaints" the graph-view.

We close this section by noting that LAVESDK provides a broad range of additional (generic) mathematical classes and functions (graphs, networks, matrices, sets, matchings, etc.) and auxiliary algorithms (shortest paths, connected components, augmented paths, breadth-first search, depth-first search, etc.). Furthermore, it comes with a test-application (called Sandbox) to directly test all relevant functions of a plugin in the Java development environment, i.e., without having to install the plugin in LAVES.

6. Conclusion and Outlook

In this paper, we have introduced an open source AV tool, LAVES, and a corresponding software development kit, LAVESDK. The software aims at helping business and business informatics students to comprehend the functionality of basic OR algorithms. It is intended to be used by instructors in class as a tool to represent the dynamics of algorithms, as well as by students to support them in understanding the way specific algorithms work. As opposed to existing visualization tools, LAVES includes intuitive instance editors to define user-specific problem instances and the potential of directly manipulating different views on the visualization in an exercise mode. First classroom

Listing 3 Example: Runtime Environment

```
1  public class MinWeightEdgeAlgorithmPlugin implements AlgorithmPlugin {
2    // ...
3    private class MinWeightEdgeAlgorithmRTE extends AlgorithmRTE {
4      // Algorithm variables
5      private int e;
6      private Set<Integer> A;
7      private float w;
8      private int a;
9      public MinWeightEdgeAlgorithmRTE() {
10       super(MinWeightEdgeAlgorithmPlugin.this, MinWeightEdgeAlgorithmPlugin.this.algoText);
11     }
12     @Override
13     protected void storeState(AlgorithmState state) {
14       state.addInt("e", e);
15       state.addSet("A", A);
16       state.addFloat("w", w);
17       state.addInt("a", a);
18     }
19     @Override
20     protected void restoreState(AlgorithmState state) {
21       e = state.getInt("e");
22       A = state.getSet("A");
23       w = state.getFloat("w");
24       a = state.getInt("a");
25     }
26     @Override
27     protected void createInitialState(AlgorithmState state) {
28       e = state.addInt("e", -1);
29       A = state.addSet("A", new Set<Integer>());
30       w = state.addFloat("w", Float.MAX_VALUE);
31       a = state.addInt("a", -1);
32     }
33     @Override
34     protected int executeStep(int stepID, AlgorithmStateAttachment asa) throws Exception {
35       int nextStep = -1;
36       Graph<Vertex, Edge> graph = MinWeightEdgeAlgorithmPlugin.this.graphView.getGraph();
37       switch(stepID) {
38         case 1:
39           // Let e in E be an arbitrary edge of the graph and set w := c(e).
40           final Edge ae = graph.getEdgeSet().get(0);
41           e = ae.getID();
42           w = ae.getWeight();
43           // Visualize edge e
44           sleep(500);
45           visualizeEdges();
46           sleep(500);
47           nextStep = 2;
48           break;
49         case 2:
50           // Let A := E\{e} be the set of the remaining edges.
51           A = graph.getEdgeByIDSet();
52           A.remove(e);
53           // Visualize the set A
54           sleep(500);
55           visualizeEdges();
56           sleep(500);
```

Listing 3 (Continued)

```
          nextStep = 3;
          break;
        case 3:
          // If A = EmptySet then e is an edge of minimum weight; stop.
          if(A.isEmpty())
            nextStep = -1;
          else
            nextStep = 4;
          // Wait a moment before going to the next step.
          sleep(1000);
          break;
        // ...
      }
      return nextStep;
    }
    @Override
    protected void rollBackStep(int stepID, int nextStepID) {
      switch(stepID) {
        case 1:
        case 2:
          // Repaint the edges using the current values of e, A, and a.
          visualizeEdges();
          break;
        // ...
      }
    }
    private void visualizeEdges() {
      final GraphView<Vertex, Edge> graphView = MinWeightEdgeAlgorithmPlugin.this.graphView;
      GraphView<Vertex, Edge>.VisualEdge vedge;
      // Set the color and line width of all visual edges depending on the values of the
          algorithm's variables e, A, and a.
      for(int i = 0; i < graphView.getVisualEdgeCount(); i++) {
        vedge = graphView.getVisualEdge(i);
        if(vedge.getEdge().getID() == a)
          vedge.setColor(Color.orange);
        else if(A.contains(vedge.getEdge().getID()))
          vedge.setColor(Color.blue);
        else
          vedge.setColor(GraphView.DEF_EDGECOLOR);
        // Edge e is thicker.
        vedge.setLineWidth((vedge.getEdge().getID() == e)? 2: GraphView.DEF_EDGELINEWIDTH);
      }
      graphView.repaint();
    }
  }
}
```

experience and a first course evaluation have indicated that LAVES is well accepted by the students and facilitates the students' understanding of algorithms. However, lecturers have to make sure not to neglect classical teaching methods (blackboard, presentation slides, etc.) because many students feel pure software use is "too fast to follow." Future research will have to specifically analyze the effects of using LAVES.

LAVES and LAVESDK are currently available in version 1.2. So far, we provide plugins related to algorithms for solving problems defined on graphs (see §4 for a list of plugins). However, LAVES is not restricted to be used with problems arising in graph theory. In fact, any algorithm's data structures or problem structures that can reasonably be represented on a piece of paper (for a given instant of time) can be realized. Thus, programmers are mainly restricted by their creativity and ability to abstract. However, the representations will have to be implemented by the plugin developer if they are not supported by LAVESDK, i.e., one may have to implement the related views (see §5 for details), including their manipulability in the

plugin project. Let us consider two examples. First, think of visualizing the simplex algorithm in its tabular form, i.e., by using simplex tableaus (for example, presented by Hillier and Lieberman 2004). LAVESDK in version 1.2 already contains a preimplemented "execution-table-view" that is sufficient to visualize any simplex tableau, including its basic manipulability (needed, for instance, when we think of an exercise where the pivot element is to be selected). Hence, when implementing a plugin for the simplex algorithm, we will not have to implement a completely new view. However, we may need to modify or extend existing views to realize specific kinds of manipulations that are not yet supported. Now, consider a basic machine scheduling algorithm. Usually, related schedules are represented by using Gantt charts (see, for instance, Blazewicz et al. 2007). So far, none of the preimplemented views of LAVESDK directly support those charts. Hence we will have to implement a "Gantt view" before being able to visualize the algorithm of interest. In this case, manipulability may relate to allowing the user to move jobs from one machine to another by dragging and dropping.

We will continue working on including additional AVs and languages (to support foreign students and to allow lecturers to teach in their native language) as well as improving the software itself. This includes the development of plugins for basic scheduling algorithms presented in Jaehn and Pesch (2014) in the near future. Additionally, LAVESDK allows instructors with (advanced) Java knowledge to implement and modify visualizations of algorithms that are being taught in their specific courses themselves (see §5).

References

AlgoViz org (2013) Algorithm visualization catalog. Accessed February 6, 2014, http://algoviz.org/avcatalog.

Baloukas T, Paparrizos K, Sifaleras A (2009) An animated demonstration of the uncapacitated network simplex algorithm. *INFORMS Trans. Ed.* 10(1):34–40.

Blazewicz J, Ecker K, Pesch E, Schmidt G, Weglarz J (2007) *Handbook on Scheduling: From Theory to Applications* (Springer, Berlin).

Crescenzi P (2012) ALViE. Accessed September 22, 2014, http://alvie.algoritmica.org/.

Diehl S (2007) *Software Visualization* (Springer, Berlin).

Free Software Foundation (2014) GNU Licenses. Accessed September 23, 2014, http://www.gnu.org/licenses/licenses.en.html.

Grissom S, McNally MF, Naps T (2003) Algorithm visualization in CS education: Comparing levels of student engagement. *Proc. 2003 ACM Sympos. Software Visualization* (ACM, New York), 87–94.

Grossman TA (2001) Causes of the decline of the business school management science course. *INFORMS Trans. Ed.* 1(2):51–61.

Hillier FS, Lieberman GJ (2004) *Introduction to Operations Research*, 8th ed. (McGraw-Hill, Boston).

Horner PR (2003) Course of action. *OR/MS Today* 30(4):26–29.

Hundhausen CD, Douglas SA, Stasko JT (2002) A meta-study of algorithm visualization effectiveness. *J. Visual Languages Comput.* 13(3):259–290.

Jaehn F, Pesch E (2014) *Ablaufplanung—Einführung in Scheduling* (Springer Gabler, Berlin).

JLaTeXMath (2014) JLaTeXMath—A Java API to render LaTeX. Accessed December 1, 2014, http://forge.scilab.org/index.php/p/jlatexmath.

Karavirta V, Korhonen A (2006) Automatic tutoring question generation during algorithm simulation. *Proc. 6th Baltic Sea Conf. Comput. Ed. Res.* (Kolin Kolistelut-Koli Calling, Kolin, Finland), 95–100.

Karavirta V, Shaffer CA (2013) JSAV: The JavaScript algorithm visualization library. *ITiCSE'13—Proc. ACM Conf. Innovation Tech. Comput. Sci. Ed.* (ACM, New York), 159–164.

Khuri S (2001) A user-centred approach for designing algorithm visualizations. *Informatik/Informatique, Special Issue on Visualization of Software* 8(2):12–16.

Korhonen A (2009) TRAKLA2—Software Visualization Group. Accessed September 22, 2014, http://www.cse.hut.fi/en/research/SVG/TRAKLA2/index.shtml.

Korhonen A, Malmi L, Silvasti P (2003) TRAKLA2: A framework for automatically assessed visual algorithm simulation exercises. *Proc. Third Finnish/Baltic Sea Conf. Comput. Sci. Ed.* (Kolin Kolistelut-Koli Calling, Kolin, Finland), 41–49.

Liebman JS (1998) Teaching operations research: Lessons from cognitive psychology. *Interfaces* 28(2):104–110.

Naps TL (2005) JHAVÉ: Supporting algorithm visualization. *IEEE Comput. Graphics Appl.* 25(5):49–55.

Naps T, Furcy D, Grissom S, McNally M (2011) JHAVÉ. Accessed January 30, 2014, http://jhave.org.

Ólafsson S (2004) OR/MS in a new MBA program. *INFORMS Trans. Ed.* 4(3):28–37.

Reinfeld NV, Vogel WR (1958) *Mathematical Programming* (Prentice-Hall, Englewood Cliffs, NJ).

Robinson S, Meadows M, Mingers J, O'Brien FA, Shale EA, Stray S (2003) Teaching OR/MS to MBAs at Warwick business school: A turnaround story. *Interfaces* 33(2):67–76.

Rößling G (2013) ANIMAL. Accessed January 30, 2014, http://www.algoanim.info/AnimalAV/.

Rößling G, Freisleben B (2002) ANIMAL: A system for supporting multiple roles in algorithm animation. *J. Visual Languages Comput.* 13(3):341–354.

Seal KC, Przasnyski ZH (2003) Using technology to support pedagogy in an OR/MS course. *Interfaces* 33(4):27–40.