**package** de.dakror.wseminar; **import** javafx.animation.Interpolator; **public** **class** Const { **public** **static** **int** cellSize = 24; **public** **static** **int** nodeAmount = 20; **public** **static** **int** edgeAmount = 3; **public** **static** **int** edgesMaxCost = 30; **public** **static** Interpolator overlyEaseIn = Interpolator.SPLINE(.43, .65, .19, 1); **public** **static** **int** visualModeTimeout = 250; **public** **static** **int** defaultCycles = 500; **public** **enum** State { START, GOAL, OPENLIST, CLOSEDLIST, BACKTRACK, REVERT; } } **package** de.dakror.wseminar.controller; **import** java.net.URL; **import** java.util.ResourceBundle; **import** javafx.fxml.FXML; **import** javafx.scene.control.Button; **import** javafx.scene.control.Label; **import** javafx.scene.image.ImageView; **import** javafx.stage.Stage; **public** **class** AlertController { @FXML **private** ResourceBundle resources; @FXML **private** URL location; @FXML **private** ImageView logo; @FXML **private** Label details; @FXML **private** Button okButton; @FXML **private** Label message; @FXML **void** initialize() { okButton.setOnAction(e -> ((Stage) okButton.getScene().getWindow()).close()); } } **package** de.dakror.wseminar.controller; **import** java.net.URL; **import** java.util.Arrays; **import** java.util.ResourceBundle; **import** de.dakror.wseminar.Const; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.algorithm.FruchtermanReingold; **import** de.dakror.wseminar.graph.algorithm.base.Layout; **import** de.dakror.wseminar.graph.generate.GraphGenerator; **import** de.dakror.wseminar.graph.generate.Params; **import** de.dakror.wseminar.graph.generate.Params.SParams; **import** javafx.event.ActionEvent; **import** javafx.event.EventHandler; **import** javafx.fxml.FXML; **import** javafx.scene.control.Button; **import** javafx.scene.control.ChoiceBox; **import** javafx.scene.control.Label; **import** javafx.scene.control.Slider; **import** javafx.scene.control.TextField; **import** javafx.scene.control.TitledPane; **import** javafx.scene.control.TreeView; **import** javafx.scene.image.ImageView; **import** javafx.scene.layout.GridPane; **import** javafx.scene.layout.HBox; **import** javafx.stage.Stage; **import** javafx.util.StringConverter; **public** **class** GenerateGraphDialogController { @FXML **private** ResourceBundle resources; @FXML **private** URL location; @FXML **private** GridPane adv\_abstract; @FXML **private** TitledPane advanced; @FXML **private** Slider graph\_size; @FXML **private** Slider edge\_count; @FXML **private** Button okButton; @FXML **private** HBox okParent; @FXML **private** Label messageLabel; @FXML **private** Button cancelButton; @FXML **private** TextField graph\_seed; @FXML **private** ImageView logo; @FXML **private** Slider node\_count; @FXML **private** ChoiceBox<String> edge\_type; **public** **static** **final** **int** speed = 400; **long** seed = 0; @FXML **void** initialize() { graph\_size.setLabelFormatter(**new** StringConverter<Double>() { String[] sizes = { "Klein", "Mittel", "Groß" }; @Override **public** String toString(Double object) { **return** sizes[object.intValue() - 1]; } @Override **public** Double fromString(String string) { **return** Arrays.asList(sizes).indexOf(string) + 1d; } }); logo.setImage(WSeminar.getImage("new\_graph-50.png")); edge\_type.getItems().addAll("Ungerichtet", "Gerichtet", "Gemischt"); edge\_type.setValue(edge\_type.getItems().get(0)); EventHandler<ActionEvent> close = e -> ((Stage) cancelButton.getScene().getWindow()).close(); cancelButton.setOnAction(close); okButton.setOnAction(e -> { **try** { **if** (graph\_seed.getText().length() == 0) { seed = (**long**) (Math.random() \* Long.MAX\_VALUE); } **else** { seed = Long.decode(graph\_seed.getText()); } } **catch** (Exception e1) { seed = graph\_seed.getText().hashCode(); } Params<String> params = **new** SParams().put("size", (**int**) graph\_size.getValue()).put("seed", seed); params.put("edge\_type", edge\_type.getItems().indexOf(edge\_type.getValue())); **if** (node\_count.getValue() != Const.nodeAmount) params.put("nodes", (**int**) Math.max(8, node\_count.getValue())); **if** (edge\_count.getValue() != Const.edgeAmount) params.put("edges", (**int**) Math.max(2, edge\_count.getValue())); Graph<Integer> graph = **new** GraphGenerator<Integer>().generateGraph(params); WSeminar.instance.paths.clear(); ((TreeView<?>) WSeminar.window.getScene().lookup("#path\_tree")).getRoot().getChildren().clear(); WSeminar.instance.setSourceGraph(graph); WSeminar.instance.setGraphSize((**int**) graph\_size.getValue()); ((Label) WSeminar.window.getScene().lookup("#seed\_label")).setText("Seed: " + WSeminar.seed); Layout<Integer> layout = **new** FruchtermanReingold<Integer>(graph, Const.defaultCycles \* (**int**) graph\_size.getValue(), seed, (**float**) graph\_size.getValue() + 1); WSeminar.instance.setLayout(layout); MainController.doLayoutWithProgress(layout, **null**, **false**, **true**); close.handle(**null**); }); } } **package** de.dakror.wseminar.controller; **import** java.awt.Color; **import** java.lang.reflect.Constructor; **import** java.lang.reflect.Field; **import** java.net.URL; **import** java.util.ResourceBundle; **import** java.util.function.Consumer; **import** com.sun.javafx.charts.Legend; **import** com.sun.javafx.collections.ObservableListWrapper; **import** de.dakror.wseminar.Const.State; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Path; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.algorithm.base.Layout; **import** de.dakror.wseminar.graph.algorithm.base.PathFinder; **import** de.dakror.wseminar.math.Vector2; **import** de.dakror.wseminar.ui.PathLineChart; **import** de.dakror.wseminar.ui.PathTreeItem; **import** de.dakror.wseminar.ui.VisualVertex; **import** de.dakror.wseminar.util.Benchmark.Timestamp; **import** de.dakror.wseminar.util.Benchmark.Type; **import** de.dakror.wseminar.util.Visualizer; **import** javafx.animation.FadeTransition; **import** javafx.animation.Interpolator; **import** javafx.animation.KeyFrame; **import** javafx.animation.Timeline; **import** javafx.application.Platform; **import** javafx.beans.property.ReadOnlyObjectWrapper; **import** javafx.beans.value.ChangeListener; **import** javafx.beans.value.ObservableValue; **import** javafx.event.EventHandler; **import** javafx.fxml.FXML; **import** javafx.geometry.Bounds; **import** javafx.scene.Cursor; **import** javafx.scene.Node; **import** javafx.scene.chart.BarChart; **import** javafx.scene.chart.XYChart; **import** javafx.scene.control.Button; **import** javafx.scene.control.CheckBox; **import** javafx.scene.control.ChoiceBox; **import** javafx.scene.control.Label; **import** javafx.scene.control.Menu; **import** javafx.scene.control.MenuItem; **import** javafx.scene.control.ProgressIndicator; **import** javafx.scene.control.Slider; **import** javafx.scene.control.SplitPane; **import** javafx.scene.control.Tab; **import** javafx.scene.control.TableColumn; **import** javafx.scene.control.TableView; **import** javafx.scene.control.Tooltip; **import** javafx.scene.control.TreeItem; **import** javafx.scene.control.TreeView; **import** javafx.scene.input.Clipboard; **import** javafx.scene.input.ClipboardContent; **import** javafx.scene.input.KeyCode; **import** javafx.scene.input.KeyEvent; **import** javafx.scene.input.MouseEvent; **import** javafx.scene.layout.Pane; **import** javafx.scene.layout.StackPane; **import** javafx.stage.Modality; **import** javafx.stage.Stage; **import** javafx.stage.StageStyle; **import** javafx.util.Duration; **public** **class** MainController { @FunctionalInterface **interface** Macro { **void** run(); } @FXML **private** ResourceBundle resources; @FXML **private** URL location; @FXML **private** TreeView<String> path\_tree; @FXML **private** TreeView<String> path\_tree\_benchmark; @FXML **private** MenuItem relayout\_graph; @FXML **private** Slider zoom; @FXML **private** CheckBox path\_benchmark; @FXML **private** Button path\_find; @FXML **private** PathLineChart<Long, Integer> chart\_timeline; @FXML **private** Pane graph; @FXML **private** Label new\_graph\_label; @FXML **private** MenuItem new\_graph; @FXML **private** Button path\_start; @FXML **private** Button path\_delete; @FXML **private** Menu menu\_graph; @FXML **private** ChoiceBox<String> path\_algorithm; @FXML **private** TreeView<String> graph\_tree; @FXML **private** Button path\_goal; @FXML **private** Button path\_x; @FXML **private** CheckBox path\_animate; @FXML **private** BarChart<String, Long> chart\_alltime; @FXML **private** Tab tab\_benchmark; @FXML **private** SplitPane benchmark; @FXML **private** TableView<Path<Vertex<Integer>>> chart\_table; @FXML **private** Label seed\_label; @FXML **private** Pane presentation; **float** lastX = -1, lastY = -1; **float** dragStartX, dragStartY; **int** currentSlide = -1; **long** last; **boolean** animatingPathFinding; **boolean** batch = **false**; @SuppressWarnings("unchecked") @FXML **public** **void** initialize() { Vector2 scrollMouse = **new** Vector2(); new\_graph\_label.setOnMouseClicked(e -> createGenerateDialog()); graph.visibleProperty().addListener((obs, newVal, oldVal) -> zoom.setDisable(newVal)); zoom.valueProperty().addListener((obs, newVal, oldVal) -> { **if** (WSeminar.instance.getGraph() != **null** && graph.isVisible()) { Pane pane = (Pane) WSeminar.window.getScene().lookup("#graph"); Bounds a = pane.getBoundsInParent(); pane.setScaleX(Math.max(0.1f, Math.min(2, newVal.floatValue() / 200f))); pane.setScaleY(Math.max(0.1f, Math.min(2, newVal.floatValue() / 200f))); Bounds b = pane.getBoundsInParent(); **float** mouseX = (**float**) (scrollMouse.x == 0 ? a.getWidth() / 2 : scrollMouse.x); **float** mouseY = (**float**) (scrollMouse.y == 0 ? a.getHeight() / 2 : scrollMouse.y); pane.setTranslateX(pane.translateXProperty().add(mouseX - (mouseX / a.getWidth() \* b.getWidth())).get()); pane.setTranslateY(pane.translateYProperty().add(mouseY - (mouseY / a.getHeight() \* b.getHeight())).get()); } }); Tooltip tooltip = **new** Tooltip("Klicken, um zu kopieren."); hackTooltipStartTiming(tooltip); seed\_label.setOnMouseClicked(e -> { Clipboard clipboard = Clipboard.getSystemClipboard(); ClipboardContent content = **new** ClipboardContent(); content.putString("" + WSeminar.seed); clipboard.setContent(content); }); Tooltip.install(seed\_label, tooltip); graph.getParent().setOnScroll(e -> { **if** (graph != **null** && graph.isVisible()) { Bounds bounds = graph.getBoundsInParent(); **float** x = (**float**) (e.getX() - graph.getBoundsInParent().getMinX()); **float** y = (**float**) (e.getY() - graph.getBoundsInParent().getMinY()); **if** (x < 0 || x > bounds.getWidth()) x = (**float**) (bounds.getWidth() / 2); **if** (y < 0 || y > bounds.getHeight()) x = (**float**) (bounds.getHeight() / 2); scrollMouse.set(x, y); Slider zoom = ((Slider) WSeminar.window.getScene().lookup("#zoom")); zoom.setValue(zoom.getValue() + e.getDeltaY() \* 0.25f); } }); EventHandler<MouseEvent> eh = e -> { scrollMouse.set((**float**) e.getX(), (**float**) e.getY()); **if** (e.isSecondaryButtonDown() && graph != **null** && graph.isVisible()) { graph.getScene().setCursor(Cursor.MOVE); **if** (lastX != -1) { **float** deltaX = (**float**) (e.getX() - lastX); **float** deltaY = (**float**) (e.getY() - lastY); graph.setTranslateX(graph.getTranslateX() + deltaX); graph.setTranslateY(graph.getTranslateY() + deltaY); } lastX = (**float**) e.getX(); lastY = (**float**) e.getY(); } **else** { **if** (e.isPrimaryButtonDown() && WSeminar.instance.activeVertex != **null**) { **if** (!WSeminar.instance.activeVertex.contains(e.getX(), e.getY())) { WSeminar.instance.activeVertex.setActive(**false**); WSeminar.instance.activeVertex = **null**; } } lastX = -1; lastY = -1; **if** (graph.getScene().getCursor() == Cursor.MOVE) graph.getScene().setCursor(Cursor.DEFAULT); } }; graph.getParent().setOnMouseReleased(eh); graph.getParent().setOnMouseDragged(eh); graph.getParent().setOnMouseExited(e -> scrollMouse.zero()); menu\_graph.getItems().get(0).setOnAction(e -> { **if** (!animatingPathFinding) createGenerateDialog(); }); menu\_graph.getItems().get(1).setOnAction(e -> { **if** (WSeminar.instance.getSourceGraph() != **null** && (System.currentTimeMillis() - last) > 200 && !animatingPathFinding) { MainController.doLayoutWithProgress(WSeminar.instance.getLayout(), **null**, **true**, **true**); last = System.currentTimeMillis(); } }); tab\_benchmark.selectedProperty().addListener((obs, oldVal, newVal) -> { graph.setVisible(!newVal); benchmark.setVisible(newVal); benchmark.autosize(); graph.autosize(); }); path\_start.setOnAction(e -> { **if** (WSeminar.instance.getSourceGraph() == **null**) **return**; WSeminar.instance.selectGoalVertex = **false**; WSeminar.instance.selectStartVertex = **true**; path\_start.getScene().setCursor(Cursor.HAND); }); path\_goal.setOnAction(e -> { **if** (WSeminar.instance.getSourceGraph() == **null**) **return**; WSeminar.instance.selectStartVertex = **false**; WSeminar.instance.selectGoalVertex = **true**; path\_goal.getScene().setCursor(Cursor.HAND); }); path\_x.setOnAction(e -> { **if** (WSeminar.instance.getSourceGraph() == **null**) **return**; **if** (WSeminar.instance.goalVertex != **null**) WSeminar.instance.goalVertex.setState(**null**); WSeminar.instance.selectGoalVertex = **false**; WSeminar.instance.goalVertex = **null**; }); path\_algorithm.getItems().addAll("DFS", "Dijkstra", "AStar"); path\_algorithm.setValue(path\_algorithm.getItems().get(0)); path\_tree.setRoot(**new** PathTreeItem<Integer>("Pfade")); path\_tree\_benchmark.setRoot(**new** PathTreeItem<Integer>("Pfade")); path\_delete.setOnAction(e -> { PathTreeItem<Integer> ti = (PathTreeItem<Integer>) path\_tree.getSelectionModel().getSelectedItem(); **if** (ti.isSpec()) WSeminar.instance.paths.remove(ti.getPathId()); **else** **for** (TreeItem<String> ti2 : ti.getChildren()) WSeminar.instance.paths.remove(((PathTreeItem<Integer>) ti2).getPathId()); ti.getParent().getChildren().remove(ti); }); path\_tree.getSelectionModel().selectedItemProperty().addListener((obs, oldVal, newV) -> { Graph<Vertex<Integer>> g = WSeminar.instance.getGraph(); **if** (newV == **null**) **return**; WSeminar.instance.startVertex = **null**; WSeminar.instance.goalVertex = **null**; Path<Vertex<Integer>> newVal = WSeminar.instance.paths.get(((PathTreeItem<Integer>) newV).getPathId()); path\_delete.setDisable(path\_tree.getRoot().equals(newV)); Visualizer.resetAll(g, **true**, **true**); **if** (newVal == **null**) **return**; **for** (**int** i = 0; i < newVal.size() - 1; i++) { Visualizer.setVertexState(newVal.get(i), State.CLOSEDLIST, **false**); Visualizer.setVertexState(newVal.get(i + 1), State.CLOSEDLIST, **false**); Visualizer.setEdgePath(g.getEdge(newVal.get(i), newVal.get(i + 1)), **true**, **false**); } Visualizer.setVertexState(newVal.get(0), State.START, **false**); Visualizer.setVertexState(newVal.get(newVal.size() - 1), State.GOAL, **false**); WSeminar.instance.startVertex = (VisualVertex<Integer>) graph.lookup("#V" + newVal.get(0).data()); WSeminar.instance.goalVertex = (VisualVertex<Integer>) graph.lookup("#V" + newVal.get(newVal.size() - 1).data()); }); path\_tree.getRoot().addEventHandler(TreeItem.childrenModificationEvent(), e -> path\_tree\_benchmark.setRoot(path\_tree.getRoot())); path\_find.setOnAction(e -> { **if** (WSeminar.instance.startVertex == **null** || WSeminar.instance.startVertex == WSeminar.instance.goalVertex) **return**; **if** (WSeminar.instance.goalVertex == **null**) { **if** (!batch) { Stage stage = WSeminar.createDialog("prompt", "Massen-Wegsuche", WSeminar.window); ((Label) stage.getScene().lookup("#message")).setText("Sicher?"); ((Label) stage.getScene().lookup("#details")).setText("Sie haben nur einen Start- aber keinen Endknoten gewählt. Somit wird eine Wegsuche für jeden anderen Knoten im Graph gestartet. Dies kann sehr zeitaufwendig werden. \nSind Sie sicher, dass sie dies starten möchten?"); ((Button) stage.getScene().lookup("#okButton")).setOnAction(i -> { ((Stage) stage.getScene().getWindow()).close(); batch = **true**; path\_find.fire(); }); **return**; } } **else** batch = **false**; **new** Thread() { Path<Vertex<Integer>> p = **null**; @SuppressWarnings("unchecked") @Override **public** **void** run() { **try** { animatingPathFinding = **true**; menu\_graph.getItems().get(0).setDisable(**true**); menu\_graph.getItems().get(1).setDisable(**true**); Class<?> c = Class.forName("de.dakror.wseminar.graph.algorithm." + path\_algorithm.getValue()); Constructor<?> con = c.getConstructor(Graph.**class**, **boolean**.**class**); Visualizer.setEnabled(path\_animate.isSelected() && !batch); **if** (batch) { **for** (Vertex<Integer> v : WSeminar.instance.getGraph().getVertices()) { **if** (!v.equals(WSeminar.instance.startVertex.getVertex())) { PathFinder<Integer> pf = (PathFinder<Integer>) con.newInstance(WSeminar.instance.getGraph(), **false**); Path<Vertex<Integer>> p = pf.findPath(WSeminar.instance.startVertex.getVertex(), v); **if** (p == **null**) { System.out.println("no path to " + v); **continue**; } Platform.runLater(() -> { PathTreeItem<Integer> pti = **null**; **if** ((pti = ((PathTreeItem<Integer>) path\_tree.getRoot()).insert(p, **true**)) != **null**) { WSeminar.instance.paths.put(p.hashCode(), p); path\_tree.getSelectionModel().select(pti); } }); } } batch = **false**; } **else** { PathFinder<Integer> pf = (PathFinder<Integer>) con.newInstance(WSeminar.instance.getGraph(), path\_animate.isSelected()); **long** sum = 0; **for** (**int** i = 0; i < 100; i++) { pf = (PathFinder<Integer>) con.newInstance(WSeminar.instance.getGraph(), path\_animate.isSelected()); p = pf.findPath(WSeminar.instance.startVertex.getVertex(), WSeminar.instance.goalVertex.getVertex()); sum += p.getBenchmark().getTime(); } System.out.println(sum / 100f); p.getBenchmark().setTime(sum / 100); Visualizer.setEnabled(**true**); animatingPathFinding = **false**; menu\_graph.getItems().get(0).setDisable(**false**); menu\_graph.getItems().get(1).setDisable(**false**); Platform.runLater(() -> { PathTreeItem<Integer> pti = **null**; **if** (p == **null**) { Stage stage = WSeminar.createDialog("alert", "Wegfindung", WSeminar.window); ((Label) stage.getScene().lookup("#message")).setText("Wegfindung fehlgeschlagen"); ((Label) stage.getScene().lookup("#details")).setText("Womöglich konnte der Weg aufgrund eines nicht vollständig zusammenhängenden oder gerichteten Graphen gefunden werden. Bitte wählen Sie andere Endknoten zur Wegfindung."); } **else** **if** ((pti = ((PathTreeItem<Integer>) path\_tree.getRoot()).insert(p, **false**)) != **null**) { WSeminar.instance.paths.put(p.hashCode(), p); path\_tree.getSelectionModel().select(pti); } }); } } **catch** (Exception e) { e.printStackTrace(); } } }.start(); }); chart\_timeline.setAnimated(**false**); chart\_timeline.setCreateSymbols(**true**); chart\_alltime.setAnimated(**false**); path\_tree\_benchmark.getSelectionModel().selectedItemProperty().addListener((obs, oldVal, newV) -> { **if** (newV == **null**) **return**; Path<Vertex<Integer>> newVal = WSeminar.instance.paths.get(((PathTreeItem<Integer>) newV).getPathId()); chart\_timeline.getData().clear(); chart\_alltime.getData().clear(); chart\_table.getItems().clear(); **if** (newV.getParent() == **null**) **return**; TimeLineDataFiller tldf = **new** TimeLineDataFiller(); **if** (newV.getParent().equals(path\_tree\_benchmark.getRoot()) && !newV.isLeaf()) { XYChart.Series<String, Long> sc = **new** XYChart.Series<>(); sc.setName("Gesamtzeit"); chart\_alltime.getData().add(sc); **for** (TreeItem<String> ti : newV.getChildren()) { Path<Vertex<Integer>> path = WSeminar.instance.paths.get(((PathTreeItem<Integer>) ti).getPathId()); XYChart.Data<String, Long> d = **new** XYChart.Data<>(path.getUserData().toString(), path.getBenchmark().getTime() / 1000 / (path.getUserData().toString().contains("anim") ? 1000 : 1)); sc.getData().add(d); Tooltip tt = **new** Tooltip(path.getUserData().toString() + ": " + d.getYValue() + (path.getUserData().toString().contains("anim") ? "m" : "µ") + "s"); hackTooltipStartTiming(tt); **if** (path.getUserData().toString().contains("anim")) d.getNode().setStyle("-fx-background-color: #0D9ED2;"); Tooltip.install(d.getNode(), tt); } tldf.generateColors(newV.getChildren().size() \* 2); **for** (TreeItem<String> ti : newV.getChildren()) { Path<Vertex<Integer>> path = WSeminar.instance.paths.get(((PathTreeItem<Integer>) ti).getPathId()); tldf.fill(path); chart\_table.getItems().add(path); } } **else** **if** (newVal != **null**) { XYChart.Series<String, Long> sc = **new** XYChart.Series<>(); sc.setName("Gesamtzeit"); chart\_alltime.getData().add(sc); XYChart.Data<String, Long> d = **new** XYChart.Data<>(newVal.getUserData().toString(), newVal.getBenchmark().getTime() / 1000 / (newVal.getUserData().toString().contains("anim") ? 1000 : 1)); sc.getData().add(d); Tooltip tt = **new** Tooltip(newVal.getUserData().toString() + ": " + d.getYValue() + (newVal.getUserData().toString().contains("anim") ? "m" : "µ") + "s"); hackTooltipStartTiming(tt); ((StackPane) d.getNode()).setPrefSize(8, 8); Tooltip.install(d.getNode(), tt); tldf.generateColors(2); tldf.fill(newVal); chart\_table.getItems().add(newVal); } **for** (**int** i = 0; i < chart\_timeline.getData().size(); i++) { XYChart.Series<Long, Integer> s = chart\_timeline.getData().get(i); Path<Vertex<Integer>> path = newV.getParent().equals(path\_tree\_benchmark.getRoot()) && !newV.isLeaf() ? WSeminar.instance.paths.get(((PathTreeItem<Integer>) newV.getChildren().get(i / Type.values().length)).getPathId()) : newVal; Color c = tldf.palette[(i % Type.values().length) \* (tldf.palette.length / Type.values().length) + i / Type.values().length]; s.getNode().setStyle(String.format("-fx-stroke: #%02x%02x%02x", c.getRed(), c.getGreen(), c.getBlue())); **for** (XYChart.Data<Long, Integer> d : s.getData()) { Tooltip tt = **new** Tooltip(path.getUserData().toString() + "(" + d.getXValue() + (path.getUserData().toString().contains("anim") ? "m" : "µ") + "s): " + d.getYValue() + " " + s.getName()); hackTooltipStartTiming(tt); ((StackPane) d.getNode()).setPrefSize(8, 8); d.getNode().setStyle(String.format("-fx-background-color: #%02x%02x%02x, white", c.getRed(), c.getGreen(), c.getBlue())); Tooltip.install(d.getNode(), tt); } } Legend l = (Legend) chart\_timeline.getChartLegend(); l.setItems(**new** ObservableListWrapper<>(l.getItems().subList(0, Type.values().length))); **for** (Node n : l.lookupAll(".chart-legend-item")) { n.setOnMouseClicked(e -> { **if** (!n.getStyleClass().contains("disabled")) n.getStyleClass().add("disabled"); **else** n.getStyleClass().remove("disabled"); **boolean** ds = n.getStyleClass().contains("disabled"); **for** (**int** i = Type.getByDesc(((Label) n).getText()).ordinal(); i < chart\_timeline.getData().size(); i += Type.values().length) { XYChart.Series<Long, Integer> s = chart\_timeline.getData().get(i); s.getNode().setVisible(!ds); s.getData().forEach(d -> d.getNode().setVisible(!ds)); } }); } **for** (**int** i = 0; i < l.getItems().size(); i++) { Color c = tldf.palette[(i % Type.values().length) \* (tldf.palette.length / Type.values().length)]; l.getItems().get(i).getSymbol().setStyle(String.format("-fx-background-color: #%02x%02x%02x, white;", c.getRed(), c.getGreen(), c.getBlue())); } }); TableColumn<Path<Vertex<Integer>>, String> tc = **new** TableColumn<>("Pfad"); tc.setCellValueFactory(p -> **new** ReadOnlyObjectWrapper<String>(p.getValue().getUserData().toString())); chart\_table.getColumns().add(tc); **for** (Type t : Type.values()) { **if** (t.name().endsWith("SIZE")) { TableColumn<Path<Vertex<Integer>>, Integer> tc2 = **new** TableColumn<>("min. " + t.desc); tc2.setCellValueFactory(p -> **new** ReadOnlyObjectWrapper<Integer>((**int**) p.getValue().getBenchmark().getMin(t))); chart\_table.getColumns().add(tc2); tc2 = **new** TableColumn<>("max. " + t.desc); tc2.setCellValueFactory(p -> **new** ReadOnlyObjectWrapper<Integer>((**int**) p.getValue().getBenchmark().getSum(t))); chart\_table.getColumns().add(tc2); } **else** { TableColumn<Path<Vertex<Integer>>, Integer> tc2 = **new** TableColumn<>(t.desc); tc2.setCellValueFactory(p -> **new** ReadOnlyObjectWrapper<Integer>((**int**) p.getValue().getBenchmark().getSum(t))); chart\_table.getColumns().add(tc2); } } fade(presentation, **false**); Consumer<Boolean> change = next -> { fade(graph, **false**); fade(presentation, **true**); currentSlide += next ? 1 : -1; currentSlide = Math.max(0, currentSlide); presentation.setStyle("-fx-background-image: url(assets/img/pres/pres-" + currentSlide + ".png); -fx-background-color: #f6f6f6; -fx-background-position: center center; -fx-background-repeat: no-repeat; -fx-background-size: contain"); }; presentation.sceneProperty().addListener(i -> { presentation.getScene().addEventFilter(KeyEvent.KEY\_PRESSED, e -> { **switch** (e.getCode()) { **case** SPACE: fade(presentation, **false**); fade(graph, **true**); **break**; **case** LEFT: **case** RIGHT: **case** ENTER: **if** (presentation.getOpacity() == 0 && e.getCode() != KeyCode.ENTER) **break**; change.accept(e.getCode() != KeyCode.LEFT); presentation.requestFocus(); e.consume(); **break**; **default**: **break**; } }); }); } **public** **static** **void** fade(Node node, **boolean** in) { FadeTransition ft = **new** FadeTransition(Duration.millis(200), node); ft.setFromValue(node.getOpacity()); ft.setToValue(in ? 1 : 0); ft.setInterpolator(Interpolator.EASE\_OUT); ft.play(); } **public** **static** **void** createGenerateDialog() { WSeminar.createDialog("generate\_graph\_dialog", "Neues Netz generieren", WSeminar.window); } **public** **static** **void** hackTooltipStartTiming(Tooltip tooltip) { **try** { Field fieldBehavior = tooltip.getClass().getDeclaredField("BEHAVIOR"); fieldBehavior.setAccessible(**true**); Object objBehavior = fieldBehavior.get(tooltip); Field fieldTimer = objBehavior.getClass().getDeclaredField("activationTimer"); fieldTimer.setAccessible(**true**); Timeline objTimer = (Timeline) fieldTimer.get(objBehavior); objTimer.getKeyFrames().clear(); objTimer.getKeyFrames().add(**new** KeyFrame(**new** Duration(50))); } **catch** (Exception e) { e.printStackTrace(); } } **public** **static** **void** doLayoutWithProgress(Layout<Integer> layout, String message, **boolean** transition, **boolean** setGraphAnimate) { **if** (layout.getSourceGraph().getVertices().size() < 100) { Graph<Vertex<Integer>> render = layout.render(); **if** (transition) WSeminar.instance.transitionTo(render); **else** WSeminar.instance.setGraph(render, setGraphAnimate); **return**; } Stage progress = showProgressDialog(layout.progress, message); **new** Thread() { @Override **public** **void** run() { Graph<Vertex<Integer>> render = layout.render(); Platform.runLater(**new** Runnable() { @Override **public** **void** run() { progress.close(); **if** (transition) WSeminar.instance.transitionTo(render); **else** WSeminar.instance.setGraph(render, setGraphAnimate); } }); } }.start(); } **public** **static** Stage showProgressDialog(ObservableValue<? **extends** Number> ov, String message) { Stage progress = WSeminar.createDialog("progress", "Fortschritt", WSeminar.window, StageStyle.TRANSPARENT, Modality.NONE); **if** (message == **null**) { progress.getScene().setFill(**null**); progress.getScene().getRoot().setStyle("-fx-background-color: transparent"); } Label msgLabel = (Label) progress.getScene().lookup("#message"); msgLabel.setVisible(message == **null**); msgLabel.setText(message); progress.setAlwaysOnTop(**true**); ProgressIndicator pi = ((ProgressIndicator) progress.getScene().lookup("#progress")); pi.setProgress(0); ChangeListener<Number> cl = (obs, newVal, oldVal) -> Platform.runLater(() -> { pi.setProgress(Math.round(newVal.doubleValue() \* 100) / 100.0); }); ov.addListener(cl); progress.setOnHiding(e -> ov.removeListener(cl)); **return** progress; } **class** TimeLineDataFiller { Color palette[]; **int** highestX = 0; **int** highestY = 0; **void** generateColors(**int** n) { n \*= Type.values().length; Color[] cols = **new** Color[n]; **for** (**int** i = 0; i < n; i++) { cols[i] = Color.getHSBColor(i / (**float**) (n - 1), 0.9f, i % 2 == 0 ? 0.85f : 1.0f); } palette = cols; } **void** fill(Path<Vertex<Integer>> path) { **for** (Type t : Type.values()) { XYChart.Series<Long, Integer> series = **new** XYChart.Series<>(); series.setName(t.desc); **for** (Timestamp ts : path.getBenchmark().get(t)) { **if** (ts.time > highestX) highestX = (**int**) ts.time; **if** (ts.stamp > highestY) highestY = (**int**) ts.stamp; XYChart.Data<Long, Integer> d = **new** XYChart.Data<>((**long**) (ts.time / (path.getUserData().toString().contains("anim") ? 1000f : 1)), (**int**) ts.stamp); series.getData().add(d); } chart\_timeline.getData().add(series); } } } } **package** de.dakror.wseminar.controller; **import** java.net.URL; **import** java.util.ResourceBundle; **import** javafx.fxml.FXML; **import** javafx.scene.control.Label; **import** javafx.scene.control.ProgressIndicator; **public** **class** ProgressController { @FXML **private** ResourceBundle resources; @FXML **private** URL location; @FXML **private** ProgressIndicator progress; @FXML **private** Label message; @FXML **void** initialize() {} } **package** de.dakror.wseminar.controller; **import** java.net.URL; **import** java.util.ResourceBundle; **import** javafx.fxml.FXML; **import** javafx.scene.control.Button; **import** javafx.scene.control.Label; **import** javafx.scene.image.ImageView; **import** javafx.stage.Stage; **public** **class** PromptController { @FXML **private** ResourceBundle resources; @FXML **private** URL location; @FXML **private** ImageView logo; @FXML **private** Label details; @FXML **private** Button okButton; @FXML **private** Button cancelButton; @FXML **private** Label message; @FXML **void** initialize() { okButton.setOnAction(e -> ((Stage) okButton.getScene().getWindow()).close()); cancelButton.setOnAction(e -> ((Stage) okButton.getScene().getWindow()).close()); } } **package** de.dakror.wseminar.graph.algorithm; **import** **static** de.dakror.wseminar.util.Benchmark.Type.\*; **import** java.util.ArrayList; **import** java.util.TreeSet; **import** de.dakror.wseminar.Const.State; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Path; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.Heuristics; **import** de.dakror.wseminar.graph.VertexData.Position; **import** de.dakror.wseminar.graph.WeightedEdge; **import** de.dakror.wseminar.graph.algorithm.base.PathFinder; **import** de.dakror.wseminar.util.Visualizer; **public** **class** AStar<V> **extends** PathFinder<V> { TreeSet<Vertex<V>> openList; ArrayList<Vertex<V>> closedList; **public** AStar(Graph<Vertex<V>> graph, **boolean** animate) { **super**(graph, animate); openList = **new** TreeSet<>((a, b) -> Float.compare(a.get(Heuristics.**class**).F(), b.get(Heuristics.**class**).F())); closedList = **new** ArrayList<>(); metaClasses = **new** Class<?>[] { Heuristics.**class** }; } @SuppressWarnings("unchecked") @Override **public** Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to) { Visualizer.resetAll(graph, **true**, **false**); BM.time(); Heuristics<V> h = **new** Heuristics<>(); h.H = distance(from, to); from.add(h); openList.add(from); BM.add(OPEN\_LIST\_SIZE); Vertex<V> last = **null**; **while** (**true**) { **if** (openList.size() == 0) **return** **null**; Vertex<V> v = openList.pollFirst(); BM.add(SORTS); BM.sub(OPEN\_LIST\_SIZE); closedList.add(v); BM.add(CLOSED\_LIST\_SIZE); **if** (v.get(Heuristics.**class**).parent != **null**) Visualizer.setEdgePath(graph.getEdge(v, v.get(Heuristics.**class**).parent), **true**, **true**); Visualizer.setVertexState(v, State.CLOSEDLIST, **true**); **if** (takeStep(last, v, to)) **break**; last = v; } Path<Vertex<V>> p = **new** Path<Vertex<V>>(); p.setUserData("AStar" + (animate ? " anim" : "") + " " + from.data() + "->" + to.data()); Vertex<V> v = to; **for** (Edge<Vertex<V>> e : graph.getEdges()) { Visualizer.setEdgePath(e, **false**, **true**, **false**); } **while** (v != **null**) { p.add(0, v); **if** (v.get(Heuristics.**class**).parent != **null**) { Visualizer.setEdgePath(graph.getEdge(v, v.get(Heuristics.**class**).parent), **true**, **true**); } Visualizer.setVertexState(v, State.BACKTRACK, **false**); v = v.get(Heuristics.**class**).parent; BM.add(PATH\_CREATION); } p.calculateCost(graph); p.setBenchmark(BM); BM.time(); cleanup(); Visualizer.resetAll(graph, **true**, **false**); **return** p; } @SuppressWarnings("unchecked") @Override **protected** **boolean** takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to) { **if** (node.equals(to)) { to.add(node.get(Heuristics.**class**)); **return** **true**; } **float** nG = node.get(Heuristics.**class**).G; **for** (Edge<Vertex<V>> e : graph.getEdgesFrom(node)) { Vertex<V> v = e.getOtherEnd(node); BM.add(v); **if** (v.get(Heuristics.**class**) == **null**) { Heuristics<V> h = **new** Heuristics<>(); h.G = nG + weight(e); h.H = distance(v, to); h.parent = node; v.add(h); openList.add(v); Visualizer.setEdgeActive(e, **true**, **true**); Visualizer.setVertexState(v, State.OPENLIST, **false**); BM.add(OPEN\_LIST\_SIZE); } **else** **if** (nG + weight(e) < v.get(Heuristics.**class**).G) { BM.add(OVERRIDES); v.get(Heuristics.**class**).G = nG + weight(e); Visualizer.setEdgeActive(graph.getEdge(v.get(Heuristics.**class**).parent, v), **false**, **true**); v.get(Heuristics.**class**).parent = node; Visualizer.setEdgeActive(e, **true**, **false**); } } **return** **false**; } **float** weight(Edge<Vertex<V>> e) { **return** e **instanceof** WeightedEdge ? ((WeightedEdge<Vertex<V>>) e).getWeight() : 1; } **float** distance(Vertex<V> v, Vertex<V> to) { **return** v.get(Position.**class**).pos.dst(to.get(Position.**class**).pos); } } **package** de.dakror.wseminar.graph.algorithm.base; **import** java.util.Random; **import** javafx.beans.property.DoubleProperty; **import** javafx.beans.property.SimpleDoubleProperty; **import** javafx.geometry.BoundingBox; **import** javafx.geometry.Bounds; **import** de.dakror.wseminar.Const; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.Position; **import** de.dakror.wseminar.math.Vector2; **public** **abstract** **class** Layout<V> { **protected** **final** **float** EPSILON = 0.00001f; **protected** Graph<V> sourceGraph; **protected** Graph<Vertex<V>> graph; **protected** **int** maxCycles; **protected** **long** seed; **protected** **boolean** isInit = **false**; **protected** Random r; **protected** **float** minX = Integer.MAX\_VALUE; **protected** **float** minY = Integer.MAX\_VALUE; **protected** **float** maxX = 0; **protected** **float** maxY = 0; **protected** Bounds bounds; **public** DoubleProperty progress = **new** SimpleDoubleProperty(0); **public** Layout(Graph<V> sourceGraph) { **this**(sourceGraph, Const.defaultCycles); } **public** Layout(Graph<V> sourceGraph, **int** maxCycles) { **this**(sourceGraph, maxCycles, (**long**) (Math.random() \* Long.MAX\_VALUE)); } **public** Layout(Graph<V> sourceGraph, **int** maxCycles, **long** seed) { **this**.sourceGraph = sourceGraph; **this**.maxCycles = maxCycles; **if** (seed != WSeminar.seed) { **this**.seed = seed; **this**.r = **new** Random(seed); } **else** r = WSeminar.r; } **public** Graph<Vertex<V>> render() { **return** render(**true**); } **public** Graph<Vertex<V>> render(**boolean** init) { **return** render(maxCycles, **true**); } **public** Graph<Vertex<V>> render(**int** steps, **boolean** init) { **if** (init || !isInit) init(); isInit = **true**; progress.set(0); **for** (**int** i = 0; i < steps; i++) { step(); progress.set(progress.get() + 1.0 / steps); } finish(); progress.set(1); **return** graph; } **public** **abstract** **void** init(); **public** **abstract** **void** step(); **public** **void** finish() { reposition(); } **public** Graph<Vertex<V>> getGraph() { **return** graph; } **public** **long** getSeed() { **return** seed; } **public** Graph<V> getSourceGraph() { **return** sourceGraph; } **public** **int** getMaxCycles() { **return** maxCycles; } **protected** **void** reposition() { graph.getVertices().forEach(v -> { Vector2 p = v.get(Position.**class**).pos; **if** (p.x < minX) minX = p.x; **if** (p.y < minY) minY = p.y; **if** (p.x > maxX) maxX = p.x; **if** (p.y > maxY) maxY = p.y; }); bounds = **new** BoundingBox(minX, minY, maxX - minX, maxY - minY); graph.getVertices().forEach(v -> v.get(Position.**class**).pos.sub(minX, minY)); } **public** Bounds getBounds() { **return** bounds; } } **package** de.dakror.wseminar.graph.algorithm.base; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Path; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.PathCommons; **import** de.dakror.wseminar.util.Benchmark; **public** **abstract** **class** PathFinder<V> { **protected** Benchmark<Vertex<V>> BM; **protected** Graph<Vertex<V>> graph; **protected** Class<?>[] metaClasses = { PathCommons.**class** }; **protected** **boolean** animate; **public** PathFinder(Graph<Vertex<V>> graph, **boolean** animate) { **this**.graph = graph; BM = **new** Benchmark<>(); **this**.animate = animate; } **public** Graph<Vertex<V>> getGraph() { **return** graph; } **public** **abstract** Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to); **protected** **void** cleanup() { **for** (Vertex<V> v : graph.getVertices()) { **for** (Class<?> c : metaClasses) v.remove(c); } } **public** Benchmark<Vertex<V>> getBenchmark() { **return** BM; } **protected** **abstract** **boolean** takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to); } **package** de.dakror.wseminar.graph.algorithm; **import** **static** de.dakror.wseminar.util.Benchmark.Type.\*; **import** java.util.HashMap; **import** java.util.List; **import** java.util.stream.Collectors; **import** de.dakror.wseminar.Const.State; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Path; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.PathCommons; **import** de.dakror.wseminar.graph.WeightedEdge; **import** de.dakror.wseminar.graph.algorithm.base.PathFinder; **import** de.dakror.wseminar.util.Visualizer; **public** **class** DFS<V> **extends** PathFinder<V> { HashMap<Vertex<V>, PathCommons<V>> meta; **public** DFS(Graph<Vertex<V>> graph, **boolean** animate) { **super**(graph, animate); meta = **new** HashMap<>(); } @Override **public** Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to) { Visualizer.resetAll(graph, **true**, **false**); BM.time(); **if** (!takeStep(**null**, from, to)) **return** **null**; Path<Vertex<V>> p = **new** Path<Vertex<V>>(); p.setUserData("DFS" + (animate ? " anim" : "") + " " + from.data() + "->" + to.data()); Vertex<V> v = to; **while** (meta.get(v).parent != **null**) { p.add(0, v); v = meta.get(v).parent; BM.add(PATH\_CREATION); } p.add(0, from); BM.add(PATH\_CREATION); p.calculateCost(graph); p.setBenchmark(BM); BM.time(); cleanup(); Visualizer.resetAll(graph, **false**, **false**); **return** p; } @Override **protected** **boolean** takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to) { PathCommons<V> pc = **new** PathCommons<>(); pc.parent = parent; meta.put(node, pc); Visualizer.setVertexState(node, State.OPENLIST, **false**); BM.add(OPEN\_LIST\_SIZE); **if** (node.equals(to)) **return** **true**; List<Edge<Vertex<V>>> edges = graph.getEdgesFrom(node).stream().filter(e -> { Vertex<V> v = e.getOtherEnd(node); BM.add(v); **boolean** free = meta.get(v) == **null**; Visualizer.setEdgeActive(e, free, **false**); **return** free; }).sorted((a, b) -> Float.compare(a **instanceof** WeightedEdge ? ((WeightedEdge<Vertex<V>>) a).getWeight() : 0, b **instanceof** WeightedEdge ? ((WeightedEdge<Vertex<V>>) b).getWeight() : 0)).collect(Collectors.toList()); BM.add(SORTS); **for** (Edge<Vertex<V>> e : edges) { Vertex<V> oe = e.getOtherEnd(node); BM.add(oe); **if** (oe.equals(to)) { BM.sub(OPEN\_LIST\_SIZE); BM.add(CLOSED\_LIST\_SIZE); Visualizer.setVertexState(node, State.CLOSEDLIST); Visualizer.setEdgePath(e, **true**); **return** takeStep(node, oe, to); } } **for** (Edge<Vertex<V>> e : edges) { Vertex<V> oe = e.getOtherEnd(node); BM.add(oe); Visualizer.tick(); BM.sub(OPEN\_LIST\_SIZE); BM.add(CLOSED\_LIST\_SIZE); Visualizer.setVertexState(node, State.CLOSEDLIST); Visualizer.setEdgePath(e, **true**); **for** (Edge<Vertex<V>> e1 : edges) { **if** (e1 == e) **continue**; Visualizer.setEdgeActive(e1, **false**, **false**); } **if** (takeStep(node, oe, to)) **return** **true**; **for** (Edge<Vertex<V>> e1 : edges) { **if** (e1 == e) **continue**; Visualizer.setEdgeActive(e1, **true**, **false**); } Visualizer.setEdgePath(e, **false**); } BM.add(BACK\_TRACKS); Visualizer.setVertexState(node, Visualizer.isEnabled() ? State.BACKTRACK : **null**, **false**); **return** **false**; } } **package** de.dakror.wseminar.graph.algorithm; **import** **static** de.dakror.wseminar.util.Benchmark.Type.\*; **import** java.util.ArrayList; **import** java.util.Collections; **import** de.dakror.wseminar.Const.State; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Path; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.InfPath; **import** de.dakror.wseminar.graph.WeightedEdge; **import** de.dakror.wseminar.graph.algorithm.base.PathFinder; **import** de.dakror.wseminar.util.Benchmark.Type; **import** de.dakror.wseminar.util.Visualizer; **public** **class** Dijkstra<V> **extends** PathFinder<V> { ArrayList<Vertex<V>> list; **public** Dijkstra(Graph<Vertex<V>> graph, **boolean** animate) { **super**(graph, animate); list = **new** ArrayList<>(); metaClasses = **new** Class<?>[] { InfPath.**class** }; } @SuppressWarnings("unchecked") @Override **public** Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to) { Visualizer.resetAll(graph, **true**, **false**); BM.time(); **for** (Vertex<V> v : graph.getVertices()) { v.add(**new** InfPath<>()); **if** (v.equals(from)) v.get(InfPath.**class**).d = 0; list.add(v); BM.add(v); BM.add(OPEN\_LIST\_SIZE); Visualizer.setVertexState(v, State.OPENLIST, **true**); }; **boolean** found = **false**; Vertex<V> v = **null**; **while** (!list.isEmpty()) { Collections.sort(list, (a, b) -> { **int** c = Float.compare(a.get(InfPath.**class**).d, b.get(InfPath.**class**).d); **if** (c == 0) **return** a.data().toString().compareTo(b.data().toString()); **return** c; }); BM.add(SORTS); v = list.remove(0); BM.sub(OPEN\_LIST\_SIZE); BM.add(CLOSED\_LIST\_SIZE); Visualizer.setVertexState(v, State.CLOSEDLIST, **true**); **if** (v.equals(to)) { found = **true**; **break**; } takeStep(**null**, v, to); } **if** (!found) **return** **null**; Path<Vertex<V>> p = **new** Path<Vertex<V>>(); p.setUserData("Dijkstra" + (animate ? " anim" : "") + " " + from.data() + "->" + to.data()); **for** (Edge<Vertex<V>> e : graph.getEdges()) { Visualizer.setEdgePath(e, **false**, **false**, **false**); } **while** (v != **null**) { p.add(0, v); **if** (v.get(InfPath.**class**).parent != **null**) { Visualizer.setEdgePath(graph.getEdge(v, v.get(InfPath.**class**).parent), **true**, **true**); } Visualizer.setVertexState(v, State.BACKTRACK, **false**); v = v.get(InfPath.**class**).parent; BM.add(PATH\_CREATION); } p.calculateCost(graph); p.setBenchmark(BM); BM.time(); cleanup(); Visualizer.resetAll(graph, **true**, **false**); **return** p; } @SuppressWarnings("unchecked") @Override **protected** **boolean** takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to) { **for** (Edge<Vertex<V>> e : graph.getEdges(node)) { Vertex<V> oe = e.getOtherEnd(node); BM.add(oe); **if** (list.contains(oe)) { **float** alt = node.get(InfPath.**class**).d + (e **instanceof** WeightedEdge ? ((WeightedEdge<Vertex<V>>) e).getWeight() : 1); **if** (alt < oe.get(InfPath.**class**).d) { oe.get(InfPath.**class**).d = alt; oe.get(InfPath.**class**).parent = node; Visualizer.setEdgePath(e, **true**, **true**); BM.add(Type.OVERRIDES); } } } **return** **false**; } } **package** de.dakror.wseminar.graph.algorithm; **import** de.dakror.wseminar.Const; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.Force; **import** de.dakror.wseminar.graph.VertexData.Position; **import** de.dakror.wseminar.graph.algorithm.base.Layout; **import** de.dakror.wseminar.math.Vector2; **public** **class** Eades<V> **extends** Layout<V> { **float** stretch = 0.7f; **int** repulsionRange = 10; **float** forceMultiplier = 1.0f / 3.0f; **float** desiredLength = 3; **public** Eades(Graph<V> sourceGraph) { **super**(sourceGraph, Const.defaultCycles); } **public** Eades(Graph<V> sourceGraph, **int** maxCycles) { **super**(sourceGraph, maxCycles, (**long**) (Math.random() \* Long.MAX\_VALUE)); } **public** Eades(Graph<V> sourceGraph, **int** maxCycles, **long** seed) { **super**(sourceGraph, maxCycles, seed); } @Override **public** **void** init() { graph = sourceGraph.getVertexGraph(Position.**class**, Force.**class**); **for** (Vertex<V> v : graph.getVertices()) v.get(Position.**class**).pos.set(r.nextFloat(), r.nextFloat()); } @Override **public** **void** step() { **for** (Vertex<V> v : graph.getVertices()) { Force f = v.get(Force.**class**); f.F.scl(1 / 4.0f); f.Fe.zero(); f.Fr.zero(); } relaxEdges(); calculateRepulsion(); moveNodes(); } **void** relaxEdges() { **for** (Edge<Vertex<V>> e : graph.getEdges()) { Vector2 dif = e.getFrom().get(Position.**class**).pos.clone().sub(e.getTo().get(Position.**class**).pos); **float** length = Math.max(dif.length(), EPSILON); **float** f = (forceMultiplier \* (desiredLength - length) / length) \* (**float**) Math.pow(stretch, graph.getEdges(e.getFrom()).size() + graph.getEdges(e.getTo()).size() - 2); dif.scl(f); e.getFrom().get(Force.**class**).Fe.add(dif); e.getTo().get(Force.**class**).Fe.sub(dif); } } **void** calculateRepulsion() { **for** (Vertex<V> v : graph.getVertices()) { Vector2 d = **new** Vector2(); **for** (Vertex<V> u : graph.getVertices()) { **if** (v == u) **continue**; Vector2 dif = v.get(Position.**class**).pos.clone().sub(u.get(Position.**class**).pos); **float** distance = dif.length(); **if** (distance == 0) { d.add(r.nextFloat(), r.nextFloat()); } **else** **if** (distance < repulsionRange) { d.add(dif.scl(1 / (distance \* distance))); } } **float** dlen = d.length() / 2; **if** (dlen > 0) { v.get(Force.**class**).Fr.add(d.scl(1 / dlen)); } } } **void** moveNodes() { **for** (Vertex<V> v : graph.getVertices()) { Force f = v.get(Force.**class**); f.F.add(f.Fe).add(f.Fr); **float** maxMovement = 5; v.get(Position.**class**).pos.add(Math.max(-maxMovement, Math.min(maxMovement, f.F.x)), Math.max(-maxMovement, Math.min(maxMovement, f.F.y))); } } } **package** de.dakror.wseminar.graph.algorithm; **import** de.dakror.wseminar.Const; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.Disposition; **import** de.dakror.wseminar.graph.VertexData.Position; **import** de.dakror.wseminar.graph.algorithm.base.Layout; **import** de.dakror.wseminar.math.Vector2; **public** **class** FruchtermanReingold<V> **extends** Layout<V> { **float** k; **float** width, height; **float** size = 2; **public** FruchtermanReingold(Graph<V> sourceGraph) { **this**(sourceGraph, Const.defaultCycles); } **public** FruchtermanReingold(Graph<V> sourceGraph, **int** maxCycles) { **this**(sourceGraph, maxCycles, WSeminar.seed); } **public** FruchtermanReingold(Graph<V> sourceGraph, **int** maxCycles, **long** seed) { **this**(sourceGraph, maxCycles, seed, 1); } **public** FruchtermanReingold(Graph<V> sourceGraph, **int** maxCycles, **long** seed, **float** width, **float** height) { **super**(sourceGraph, maxCycles, seed); **this**.width = width; **this**.height = height; } **public** FruchtermanReingold(Graph<V> sourceGraph, **int** maxCycles, **long** seed, **float** size) { **this**(sourceGraph, maxCycles, seed, sourceGraph.getVertices().size() \* size, sourceGraph.getVertices().size() \* size); **this**.size = size; } @Override **public** **void** init() { graph = sourceGraph.getVertexGraph(Position.**class**, Disposition.**class**); **for** (Vertex<V> v : graph.getVertices()) v.get(Position.**class**).pos.set(r.nextFloat() \* width, r.nextFloat() \* height); calculateK(graph.getVertices().size()); } @Override **public** **void** step() { **float** temperature = 1f / 10 \* width; **float** step = temperature / maxCycles; **for** (Vertex<V> v : graph.getVertices()) { Disposition d = v.get(Disposition.**class**); d.disp.zero(); **for** (Vertex<V> u : graph.getVertices()) { **if** (u != v) { Vector2 delta = v.get(Position.**class**).pos.clone().sub(u.get(Position.**class**).pos); **float** length = Math.max(EPSILON, delta.length()); **float** fr = getFr(length); d.disp.add(delta.x / length \* fr, delta.y / length \* fr); } } } **for** (Edge<Vertex<V>> e : graph.getEdges()) { Vector2 delta = e.getFrom().get(Position.**class**).pos.clone().sub(e.getTo().get(Position.**class**).pos); **float** length = Math.max(EPSILON, delta.length()); **float** fa = getFa(length); **float** dx = delta.x / length \* fa; **float** dy = delta.y / length \* fa; e.getFrom().get(Disposition.**class**).disp.sub(dx, dy); e.getTo().get(Disposition.**class**).disp.add(dx, dy); } **for** (Vertex<V> v : graph.getVertices()) { Position p = v.get(Position.**class**); Disposition d = v.get(Disposition.**class**); **float** length = Math.max(EPSILON, d.disp.length()); **float** min = Math.min(length, temperature); p.pos.add(d.disp.x / length \* min, d.disp.y / length \* min); p.pos.x = Math.min(width - r.nextFloat(), Math.max(r.nextFloat(), p.pos.x)); p.pos.y = Math.min(height - r.nextFloat(), Math.max(r.nextFloat(), p.pos.y)); } temperature -= step; } **void** calculateK(**int** vertexCount) { k = (**float**) (Math.sqrt((width \* height / (size / 2)) / vertexCount)); } **float** getFa(**float** d) { **return** (d \* d) / k; } **float** getFr(**float** d) { **return** (k \* k) / d; } } **package** de.dakror.wseminar.graph; **import** java.util.ArrayList; **import** java.util.List; **import** java.util.stream.Collectors; **public** **class** DefaultGraph<V> **implements** Graph<V> { List<Edge<V>> edges = **new** ArrayList<>(); List<V> vertices = **new** ArrayList<>(); @Override **public** List<Edge<V>> getEdges() { **return** edges; } @Override **public** List<Edge<V>> getEdgesFrom(V from) { } @Override **public** List<Edge<V>> getEdgesTo(V to) { } @Override **public** List<Edge<V>> getEdges(V v) { } @Override **public** List<V> getVertices() { **return** vertices; } @Override **public** **void** addEdge(Edge<V> edge) { **if** (!containsVertex(edge.getFrom())) addVertex(edge.getFrom()); **if** (!containsVertex(edge.getTo())) addVertex(edge.getTo()); edges.add(edge); } @Override **public** **boolean** removeEdge(V from, V to) { } **public** **boolean** containsEdge(Edge<V> edge) { **return** edges.contains(edge); } @Override **public** **void** addVertex(V v) { vertices.add(v); } @Override **public** **boolean** removeVertex(V v) { **return** edges.removeIf(e -> e.getFrom().equals(v) || e.getTo().equals(v)) && vertices.add(v); } **public** **boolean** containsVertex(V v) { **return** vertices.contains(v); } @Override **public** **boolean** areConnected(V from, V to) { **return** getEdge(from, to) != **null**; } @Override **public** Edge<V> getEdge(V from, V to) { **return** edges.stream().filter(e -> e.isDirected() ? (e.getFrom().equals(from) && e.getTo().equals(to) || e.getTo().equals(from) && e.getFrom().equals(to)) : (e.getFrom().equals(from) || e.getFrom().equals(to)) && (e.getTo().equals(from) || e.getTo().equals(to))).findFirst().orElse(**null**); } @Override **public** **float** getWeight(V from, V to) { Edge<V> edge = getEdge(from, to); **return** edge **instanceof** WeightedEdge ? ((WeightedEdge<V>) edge).getWeight() : 0; } @Override **public** Graph<Vertex<V>> getVertexGraph(Class<?>... vertexData) { Graph<Vertex<V>> graph = **new** DefaultGraph<>(); **for** (V v : vertices) { Vertex<V> vertex = **new** Vertex<V>(v); vertex.decorate(vertexData); graph.addVertex(vertex); } **for** (Edge<V> edge : edges) { Vertex<V> from = graph.getVertices().get(vertices.indexOf(edge.getFrom())); Vertex<V> to = graph.getVertices().get(vertices.indexOf(edge.getTo())); Edge<Vertex<V>> e = edge **instanceof** WeightedEdge ? **new** WeightedEdge<Vertex<V>>(from, to, ((WeightedEdge<V>) edge).getWeight()) : **new** Edge<Vertex<V>>(from, to); e.setDirected(edge.isDirected()); graph.addEdge(e); } **return** graph; } @SuppressWarnings("hiding") @Override **public** <V> **boolean** isConnected(V v, Edge<V> edge) { **return** edge.isDirected() ? edge.getFrom().equals(v) : edge.getFrom().equals(v) || edge.getTo().equals(v); } } **package** de.dakror.wseminar.graph; **public** **class** Edge<V> **implements** Comparable<Edge<V>> { V from, to; **boolean** directed; **public** Edge(V from, V to) { **this**.from = from; **this**.to = to; } **public** Edge(V from, V to, **boolean** directed) { **this**.from = from; **this**.to = to; **this**.directed = directed; } **public** V getFrom() { **return** from; } **public** Edge<V> setFrom(V from) { **this**.from = from; **return** **this**; } **public** V getTo() { **return** to; } **public** Edge<V> setTo(V to) { **this**.to = to; **return** **this**; } **public** Edge<V> setDirected(**boolean** directed) { **this**.directed = directed; **return** **this**; } **public** V getOtherEnd(V end) { **if** (end.equals(from)) **return** to; **else** **return** from; } **public** **boolean** isDirected() { **return** directed; } @Override **public** String toString() { **return** "E[" + from + " " + (directed ? "" : "<") + "-> " + to + "]"; } @Override **public** **int** hashCode() { **return** toString().hashCode(); } @Override **public** **int** compareTo(Edge<V> o) { **return** 0; } } **package** de.dakror.wseminar.graph.generate; **import** de.dakror.wseminar.Const; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.DefaultGraph; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.WeightedEdge; **public** **class** GraphGenerator<V> { @SuppressWarnings("unchecked") **public** Graph<V> generateGraph(Params<String> params) { **long** seed = params.get("seed"); WSeminar.setSeed(seed); Graph<V> graph = **new** DefaultGraph<>(); **int** nodeAmount = params.orElse("nodes", Const.nodeAmount); **int** nodes = (WSeminar.r.nextInt(nodeAmount / 2) + nodeAmount / 2) \* (**int**) params.get("size"); **for** (**int** i = 0; i < nodes; i++) { **try** { graph.addVertex((V) (Integer) i); } **catch** (Exception e) { **throw** **new** IllegalStateException("Generics not matching graph type!", e); } } System.out.println("Added " + graph.getVertices().size() + " nodes to the graph."); **int** edgesPlaced = 0; **int** edge\_type = params.get("edge\_type"); **for** (**int** i = 0; i < nodes; i++) { **int** edges = Math.max(WSeminar.r.nextInt(Math.min(graph.getVertices().size() / 2 - 1, params.orElse("edges", Const.edgeAmount))), 1); **for** (**int** j = 0; j < edges; j++) { **int** index = i; **do** { index = WSeminar.r.nextInt(nodes); } **while** (index == i || graph.areConnected(graph.getVertices().get(i), graph.getVertices().get(index))); Edge<V> edge = **new** WeightedEdge<V>(graph.getVertices().get(i), graph.getVertices().get(index), WSeminar.r.nextInt(Const.edgesMaxCost)); **if** (edge\_type == 1 || (edge\_type == 2 && WSeminar.r.nextFloat() > WSeminar.r.nextFloat())) edge.setDirected(**true**); graph.addEdge(edge); } edgesPlaced += edges; } System.out.println("Made " + edgesPlaced + " connections."); **return** graph; } } **package** de.dakror.wseminar.graph.generate; **import** java.util.HashMap; **import** java.util.Map; **public** **abstract** **class** Params<K> { **public** **static** **class** SParams **extends** Params<String> {} **protected** Map<K, Object> map; **public** Params() { map = **new** HashMap<>(); } **public** Params<K> put(K k, Object v) { map.put(k, v); **return** **this**; } **public** <T> T get(K k) { **return** orElse(k, **null**); } @SuppressWarnings("unchecked") **public** <T> T orElse(K k, T o) { **if** (map.containsKey(k)) **return** (T) map.get(k); **else** **return** o; } } **package** de.dakror.wseminar.graph; **import** java.util.List; **public** **interface** Graph<V> { **public** List<Edge<V>> getEdges(); **public** List<Edge<V>> getEdgesFrom(V from); **public** List<Edge<V>> getEdgesTo(V to); **public** List<Edge<V>> getEdges(V v); **public** List<V> getVertices(); **public** **void** addEdge(Edge<V> edge); **public** **boolean** removeEdge(V from, V to); **public** **void** addVertex(V v); **public** **boolean** removeVertex(V v); **public** **boolean** areConnected(V from, V to); @SuppressWarnings("hiding") **public** <V> **boolean** isConnected(V v, Edge<V> edge); **public** Edge<V> getEdge(V from, V to); **public** **float** getWeight(V from, V to); **public** Graph<Vertex<V>> getVertexGraph(Class<?>... vertexData); } **package** de.dakror.wseminar.graph; **import** java.util.ArrayList; **import** java.util.Collection; **import** java.util.Iterator; **import** de.dakror.wseminar.util.Benchmark; **public** **class** Path<V> **implements** Iterable<V> { ArrayList<V> nodes; Object userData; Benchmark<V> benchmark; **float** cost; **public** Path() { nodes = **new** ArrayList<>(); } **public** Benchmark<V> getBenchmark() { **return** benchmark; } **public** **void** setBenchmark(Benchmark<V> benchmark) { **this**.benchmark = benchmark; } @Override **public** Iterator<V> iterator() { **return** nodes.iterator(); } **public** **int** size() { **return** nodes.size(); } **public** **boolean** isEmpty() { **return** nodes.isEmpty(); } **public** **boolean** contains(Object o) { **return** nodes.contains(o); } **public** V get(**int** index) { **return** nodes.get(index); } **public** **boolean** add(V e) { **return** nodes.add(e); } **public** **void** add(**int** index, V element) { nodes.add(index, element); } **public** V remove(**int** index) { **return** nodes.remove(index); } **public** **boolean** remove(Object o) { **return** nodes.remove(o); } **public** **void** clear() { nodes.clear(); } **public** **boolean** addAll(Collection<? **extends** V> c) { **return** nodes.addAll(c); } **public** **void** calculateCost(Graph<V> graph) { **float** cost = 0; **for** (**int** i = 0; i < size() - 1; i++) { cost += graph.getWeight(nodes.get(i), nodes.get(i + 1)); } **this**.cost = cost; } **public** **float** getCost() { **return** cost; } @Override **public** **int** hashCode() { **return** toString().hashCode(); } **public** **void** setUserData(Object userData) { **this**.userData = userData; } **public** Object getUserData() { **return** userData; } @Override **public** **boolean** equals(Object obj) { **if** (obj == **null**) **return** **false**; **return** obj.hashCode() == hashCode(); } **public** String toGrouperString() { **return** nodes.get(0) + " -> " + nodes.get(nodes.size() - 1); } **public** String toSpecString() { **return** userData + " { " + (nodes.size() - 1) + " } = " + cost; } **public** String toString(**boolean** spec) { **return** spec ? toSpecString() : toGrouperString(); } @Override **public** String toString() { **return** (userData != **null** ? userData + " " : "") + nodes.get(0) + " -> { " + (nodes.size() - 2) + " } -> " + nodes.get(nodes.size() - 1) + " = " + cost; } } **package** de.dakror.wseminar.graph; **import** java.util.HashMap; **import** java.util.Map; **public** **class** Vertex<T> { T data; Map<Class<?>, Object> vertexData = **new** HashMap<>(); **public** Vertex(T data) { **this**.data = data; } **public** T data() { **return** data; } **public** **void** set(T data) { **this**.data = data; } **public** **void** add(Object... vd) { **for** (Object v : vd) vertexData.put(v.getClass(), v); } **public** **void** decorate(Class<?>... vd) { **for** (Class<?> c : vd) { **try** { add(c.newInstance()); } **catch** (InstantiationException | IllegalAccessException e) { e.printStackTrace(); } } } @SuppressWarnings("unchecked") **public** <V> V get(Class<V> cl) { **return** (V) vertexData.get(cl); } **public** Object[] getAll() { **return** vertexData.values().toArray(**new** Object[] {}); } **public** **boolean** remove(Class<?> cl) { **return** vertexData.remove(cl) != **null**; } @SuppressWarnings("unchecked") @Override **public** **boolean** equals(Object obj) { **if** (obj **instanceof** Vertex) **return** data.equals(((Vertex<T>) obj).data); **return** **false**; } @Override **public** **int** hashCode() { **return** data.hashCode(); } @Override **public** String toString() { **return** "( " + data + " )"; } } **package** de.dakror.wseminar.graph; **import** de.dakror.wseminar.math.Vector2; **public** **class** VertexData { **public** **static** **class** Delay { **public** **float** delay; } **public** **static** **class** Disposition { **public** Vector2 disp = **new** Vector2(); } **public** **static** **class** Force { **public** Vector2 Fe = **new** Vector2(), Fr = **new** Vector2(), F = **new** Vector2(); } **public** **static** **class** Position { **public** Vector2 pos = **new** Vector2(); } **public** **static** **class** PathCommons<V> { **public** Vertex<V> parent; **public** **boolean** visited = **true**; } **public** **static** **class** Heuristics<V> **extends** PathCommons<V> { **public** **float** G, H; **public** **float** F() { **return** G + H; } } **public** **static** **class** InfPath<V> **extends** PathCommons<V> { **public** **float** d = Float.MAX\_VALUE; } } **package** de.dakror.wseminar.graph; **public** **class** WeightedEdge<V> **extends** Edge<V> { **float** weight; **public** WeightedEdge(V from, V to, **float** weight) { **super**(from, to); **this**.weight = weight; } **public** WeightedEdge(V from, V to, **boolean** directed, **float** weight) { **super**(from, to, directed); **this**.weight = weight; } **public** **float** getWeight() { **return** weight; } **public** Edge<V> setWeight(**float** weight) { **this**.weight = weight; **return** **this**; } @Override **public** String toString() { **return** "WE[" + from + " " + (directed ? "" : "<") + " " + weight + " > " + to + "]"; } @Override **public** **int** compareTo(Edge<V> o2) { **return** Float.compare(weight, o2 **instanceof** WeightedEdge ? ((WeightedEdge<V>) o2).getWeight() : 0); } } **package** de.dakror.wseminar.math; **public** **class** Vector2 **implements** Comparable<Vector2> { **public** **float** x, y; **public** Vector2() {} **public** Vector2(**float** x, **float** y) { **this**.x = x; **this**.y = y; } **public** Vector2 add(Vector2 o) { add(o.x, o.y); **return** **this**; } **public** Vector2 add(**float** x, **float** y) { **this**.x += x; **this**.y += y; **return** **this**; } **public** Vector2 sub(Vector2 o) { add(-o.x, -o.y); **return** **this**; } **public** Vector2 sub(**float** x, **float** y) { add(-x, -y); **return** **this**; } **public** Vector2 set(**float** x, **float** y) { **this**.x = x; **this**.y = y; **return** **this**; } **public** Vector2 set(Vector2 v) { set(v.x, v.y); **return** **this**; } **public** **float** length() { **return** (**float**) Math.sqrt(x \* x + y \* y); } **public** Vector2 nor() { **float** length = length(); x /= length; y /= length; **return** **this**; } **public** Vector2 normalize() { **return** clone().nor(); } **public** Vector2 scl(**float** skalar) { mul(skalar, skalar); **return** **this**; } **public** Vector2 zero() { scl(0); **return** **this**; } **public** Vector2 mul(**float** scX, **float** scY) { x \*= scX; y \*= scY; **return** **this**; } @Override **public** Vector2 clone() { **return** **new** Vector2(x, y); } **public** **float** dst(Vector2 o) { **float** dx = x - o.x; **float** dy = y - o.y; **return** (**float**) Math.sqrt(dx \* dx + dy \* dy); } @Override **public** String toString() { **return** "[" + x + ", " + y + "]"; } @Override **public** **boolean** equals(Object obj) { **return** obj **instanceof** Vector2 ? x == ((Vector2) obj).x && y == ((Vector2) obj).y : **false**; } @Override **public** **int** compareTo(Vector2 o) { **if** (x == o.x) **return** Float.compare(y, o.y); **else** **return** Float.compare(x, o.x); } } **package** de.dakror.wseminar.ui; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.WeightedEdge; **import** javafx.scene.Node; **import** javafx.scene.control.Label; **import** javafx.scene.control.TextField; **import** javafx.scene.control.TreeCell; **import** javafx.scene.control.TreeItem; **import** javafx.scene.input.KeyCode; **import** javafx.scene.layout.HBox; **import** javafx.scene.layout.Priority; **public** **class** GraphTreeCell **extends** TreeCell<String> { **final** HBox graphic = **new** HBox(); **final** Label name = **new** Label(); **final** TextField value = **new** TextField(); @SuppressWarnings("unchecked") **public** GraphTreeCell() { getStyleClass().add("hierarchy-tree-cell"); name.setMinWidth((-1.0D / 0.0D)); value.setDisable(**true**); value.getStyleClass().addAll("hierarchy-readwrite-label", "label"); value.setOnKeyPressed(e -> { **if** (e.getCode() == KeyCode.ENTER) graphic.requestFocus(); }); value.focusedProperty().addListener((obs, oldVal, newVal) -> { **if** (!newVal) { String val = value.getText(); Node node = ((GraphTreeItem) getTreeItem()).getNode(); **if** (node **instanceof** VisualEdge) { **float** num = 0; **try** { num = (**float**) Double.parseDouble(val); } **catch** (NumberFormatException e) { value.setText(((VisualEdge<Integer>) node).text.getText()); **return**; } ((VisualEdge<Integer>) node).text.setText((**int**) num + ""); ((WeightedEdge<Vertex<Integer>>) ((VisualEdge<Integer>) node).edge).setWeight(num); **for** (TreeItem<String> vti : getTreeView().getRoot().getChildren()) { **for** (TreeItem<String> eti : vti.getChildren()) { **if** (((GraphTreeItem) eti).getParam().equals(((GraphTreeItem) getTreeItem()).getParam())) { String value = eti.getValue(); eti.setValue(**null**); eti.setValue(value); } } } } } }); HBox.setHgrow(value, Priority.ALWAYS); graphic.getStyleClass().add("tree-cell-graphic"); graphic.getChildren().addAll(**new** Node[] { name, value }); selectedProperty().addListener((obs, oldVal, newVal) -> { **if** (getTreeItem() != **null** && isSelected()) { Node node = ((GraphTreeItem) getTreeItem()).getNode(); **if** (node **instanceof** VisualVertex && node != WSeminar.instance.activeVertex) { **if** (WSeminar.instance.activeVertex != **null**) WSeminar.instance.activeVertex.setActive(**false**); WSeminar.instance.activeVertex = (VisualVertex<Integer>) node; ((VisualVertex<?>) node).setActive(**true**); } **if** (node **instanceof** VisualEdge && node != WSeminar.instance.activeEdge) { **if** (WSeminar.instance.activeEdge != **null**) WSeminar.instance.activeEdge.setActive(**false**); WSeminar.instance.activeEdge = (VisualEdge<Integer>) node; ((VisualEdge<?>) node).setActive(**true**); } } }); } @Override **protected** **void** updateItem(String item, **boolean** empty) { **super**.updateItem(item, empty); value.setDisable(**true**); **if** (empty) { setText(**null**); setGraphic(**null**); } **else** { **if** (getIndex() == 0 && !getStyleClass().contains("hierarchy-first-cell")) getStyleClass().add("hierarchy-first-cell"); name.setText(item); Node node = ((GraphTreeItem) getTreeItem()).getNode(); **if** (node **instanceof** VisualVertex) { value.setText(((VisualVertex<?>) ((GraphTreeItem) getTreeItem()).getNode()).getVertex().data() + ""); } **else** **if** (node **instanceof** VisualEdge) { value.setText(((VisualEdge<?>) ((GraphTreeItem) getTreeItem()).getNode()).text.getText()); value.setDisable(!(((VisualEdge<?>) node).edge **instanceof** WeightedEdge)); } value.setMaxWidth(50); value.setVisible(value.getText().length() > 0); setGraphic(graphic); } } } **package** de.dakror.wseminar.ui; **import** javafx.scene.Node; **import** javafx.scene.control.TreeItem; **public** **class** GraphTreeItem **extends** TreeItem<String> { Node node; Object param; **public** GraphTreeItem() {} **public** GraphTreeItem(Node node, String name) { **super**(name); **this**.node = node; } **public** <T> GraphTreeItem(Node node, String name, T param) { **super**(name); **this**.node = node; **this**.param = param; } **public** <T> **void** setParam(T param) { **this**.param = param; } @SuppressWarnings("unchecked") **public** <T> T getParam() { **return** (T) param; } **public** Node getNode() { **return** node; } } **package** de.dakror.wseminar.ui; **import** javafx.beans.NamedArg; **import** javafx.collections.ObservableList; **import** javafx.scene.Node; **import** javafx.scene.chart.Axis; **import** javafx.scene.chart.LineChart; **public** **class** PathLineChart<X, Y> **extends** LineChart<X, Y> { **public** PathLineChart(@NamedArg("xAxis") Axis<X> xAxis, @NamedArg("yAxis") Axis<Y> yAxis) { **super**(xAxis, yAxis); } **public** PathLineChart(@NamedArg("xAxis") Axis<X> xAxis, @NamedArg("yAxis") Axis<Y> yAxis, @NamedArg("data") ObservableList<Series<X, Y>> data) { **super**(xAxis, yAxis, data); } **public** Node getChartLegend() { **return** **super**.getLegend(); } } **package** de.dakror.wseminar.ui; **import** de.dakror.wseminar.graph.Path; **import** de.dakror.wseminar.graph.Vertex; **import** javafx.scene.control.TreeItem; **public** **class** PathTreeItem<V> **extends** TreeItem<String> { **int** pathId = -1; **public** PathTreeItem(String text) { **super**(text); } **public** PathTreeItem(Path<Vertex<V>> p, **boolean** spec) { **super**(p.toString(spec)); **if** (spec) pathId = p.hashCode(); } **public** **int** getPathId() { **return** pathId; } **public** **boolean** isSpec() { **return** pathId != -1; } **public** PathTreeItem<V> insert(Path<Vertex<V>> p, **boolean** batch) { **if** (p == **null**) **return** **null**; **for** (TreeItem<String> ti : getChildren()) { **if** (ti.getValue().equals(p.toGrouperString()) || (batch && isSameBatch(ti.getValue(), p.toGrouperString()))) { **for** (TreeItem<String> ti2 : ti.getChildren()) **if** (ti2.getValue().equals(p.toSpecString())) **return** **null**; PathTreeItem<V> pti = **new** PathTreeItem<V>(p, **true**); **if** (batch && !ti.getValue().endsWith("( \* )")) { ti.setValue(ti.getValue().substring(0, ti.getValue().indexOf("->") + 2) + " ( \* )"); } ti.getChildren().add(pti); **return** pti; } } String gr = p.toGrouperString(); PathTreeItem<V> pti = **new** PathTreeItem<V>(batch ? (gr.substring(0, gr.lastIndexOf("(")) + "( \* )") : gr); PathTreeItem<V> pti2 = **new** PathTreeItem<V>(p, **true**); pti.getChildren().add(pti2); getChildren().add(pti); **return** pti2; } **boolean** isSameBatch(String s1, String s2) { **return** s1.substring(0, s1.indexOf("->")).equals(s2.substring(0, s2.indexOf("->"))); } } **package** de.dakror.wseminar.ui; **import** java.text.NumberFormat; **import** de.dakror.wseminar.Const; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.Position; **import** de.dakror.wseminar.graph.WeightedEdge; **import** de.dakror.wseminar.math.Vector2; **import** javafx.beans.value.ChangeListener; **import** javafx.event.EventHandler; **import** javafx.scene.control.Slider; **import** javafx.scene.input.MouseEvent; **import** javafx.scene.input.ScrollEvent; **import** javafx.scene.layout.Pane; **import** javafx.scene.paint.Color; **import** javafx.scene.shape.Line; **import** javafx.scene.shape.Polygon; **import** javafx.scene.shape.StrokeLineCap; **import** javafx.scene.text.Text; **public** **class** VisualEdge<V> **extends** Line { Color color; **public** Text text; Polygon p; Edge<Vertex<V>> edge; **boolean** active; **boolean** path; **boolean** left; **public** VisualEdge(Edge<Vertex<V>> edge, Pane pane) { **super**(edge.getFrom().get(Position.**class**).pos.x \* Const.cellSize + Const.cellSize / 2, edge.getFrom().get(Position.**class**).pos.y \* Const.cellSize + Const.cellSize / 2, edge.getTo().get(Position.**class**).pos.x \* Const.cellSize + Const.cellSize / 2, edge.getTo().get(Position.**class**).pos.y \* Const.cellSize + Const.cellSize / 2); **this**.edge = edge; setId("E" + edge.hashCode()); getStyleClass().add("visual-edge"); left = WSeminar.r.nextBoolean(); text = **new** Text(); **if** (edge **instanceof** WeightedEdge) { NumberFormat nf = NumberFormat.getNumberInstance(); nf.setMaximumFractionDigits(2); text.setText(nf.format(((WeightedEdge<Vertex<V>>) edge).getWeight())); text.setOpacity(0); pane.getChildren().add(text); } p = **new** Polygon(-5, 4, 5, 4, 0, -6); **if** (edge.isDirected()) { p.setOpacity(0); pane.getChildren().add(p); } ChangeListener<Number> cl = (obs, newVal, oldVal) -> { setColor(color); text.setTranslateX(0.5f \* (getStartX() + getEndX()) - (left ? text.getBoundsInLocal().getWidth() : 0)); text.setTranslateY(0.5f \* (getStartY() + getEndY())); Vector2 dir = **new** Vector2((**float**) getEndX() - (**float**) getStartX(), (**float**) getEndY() - (**float**) getStartY()).nor(); Vector2 back = dir.clone().scl(18); p.setTranslateX(getEndX() - back.x); p.setTranslateY(getEndY() - back.y); p.setRotate(Math.toDegrees(Math.atan2(dir.y, dir.x)) + 90); }; opacityProperty().addListener((obs, newVal, oldVal) -> { **if** (newVal.doubleValue() == 1 && oldVal.doubleValue() == 0) **return**; text.setOpacity(newVal.doubleValue()); p.setOpacity(newVal.doubleValue()); }); setColor(Color.DARKGRAY); EventHandler<MouseEvent> en = e -> { **if** (!path) setColor(Color.valueOf("#5f5f5f")); }; EventHandler<MouseEvent> ex = e -> { **if** (!path) setColor(active ? Color.valueOf("#2279e5") : Color.DARKGRAY); }; pane.getParent().addEventHandler(ScrollEvent.ANY, e -> { **if** (getScene() == **null**) **return**; Slider zoom = ((Slider) getScene().lookup("#zoom")); setStrokeWidth(2 + Math.max(0, 15 \* (100 - zoom.getValue()) / 100)); }); setOnMouseEntered(en); setOnMouseExited(ex); p.setOnMouseEntered(en); p.setOnMouseExited(ex); startXProperty().addListener(cl); startYProperty().addListener(cl); endXProperty().addListener(cl); endYProperty().addListener(cl); setStrokeWidth(2); setStrokeLineCap(StrokeLineCap.ROUND); p.setStrokeWidth(1); } **void** setColor(Color color) { **this**.color = color; setStroke(color); p.setStroke(color.darker()); p.setFill(color); } **public** **void** setActive(**boolean** active) { **if** (path && !active) **return**; **if** (active) setColor(Color.valueOf("#2279e5")); **else** setColor(Color.DARKGRAY); **if** (active) text.setFill(Color.valueOf("#2279e5").darker()); **else** text.setFill(Color.BLACK); **this**.active = active; } **public** **void** setPath(**boolean** path) { setPath(path, **false**); } **public** **void** setPath(**boolean** path, **boolean** soft) { **if** (path) setColor(Color.valueOf("#3c009f")); **else** { **if** (soft) setActive(active); **else** setColor(Color.DARKGRAY); } **if** (path) text.setFill(Color.valueOf("#3c009f").darker()); **else** { **if** (soft) setActive(active); **else** text.setFill(Color.BLACK); } **this**.path = path; } **public** **boolean** isPath() { **return** path; } **public** **boolean** isActive() { **return** active; } } **package** de.dakror.wseminar.ui; **import** de.dakror.wseminar.Const; **import** de.dakror.wseminar.Const.State; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.Position; **import** javafx.scene.Cursor; **import** javafx.scene.control.TreeItem; **import** javafx.scene.control.TreeView; **import** javafx.scene.input.MouseButton; **import** javafx.scene.shape.Circle; **public** **class** VisualVertex<V> **extends** Circle { Vertex<V> vertex; State prevState, state; **boolean** active; @SuppressWarnings("unchecked") **public** VisualVertex(Vertex<V> v) { vertex = v; setCenterX(Const.cellSize / 2); setCenterY(Const.cellSize / 2); setRadius(12); setTranslateX(v.get(Position.**class**).pos.x \* Const.cellSize); setTranslateY(v.get(Position.**class**).pos.y \* Const.cellSize); getStyleClass().add("visual-vertex"); setOnMouseClicked(e -> { **if** (e.getButton() != MouseButton.PRIMARY) **return**; **if** (WSeminar.instance.selectStartVertex || WSeminar.instance.selectGoalVertex) { **if** (WSeminar.instance.selectStartVertex) { **if** (WSeminar.instance.startVertex != **null**) WSeminar.instance.startVertex.setState(**null**); setState(State.START); WSeminar.instance.startVertex = (VisualVertex<Integer>) VisualVertex.**this**; } **if** (WSeminar.instance.selectGoalVertex) { **if** (WSeminar.instance.goalVertex != **null**) WSeminar.instance.goalVertex.setState(**null**); setState(State.GOAL); WSeminar.instance.goalVertex = (VisualVertex<Integer>) VisualVertex.**this**; } WSeminar.instance.selectStartVertex = **false**; WSeminar.instance.selectGoalVertex = **false**; getScene().setCursor(Cursor.DEFAULT); **return**; } **if** (WSeminar.instance.activeVertex == VisualVertex.**this**) **return**; **if** (WSeminar.instance.activeVertex != **null**) WSeminar.instance.activeVertex.setActive(**false**); setActive(**true**); WSeminar.instance.activeVertex = (VisualVertex<Integer>) VisualVertex.**this**; TreeView<String> tv = ((TreeView<String>) getScene().lookup("#graph\_tree")); **for** (TreeItem<String> item : tv.getRoot().getChildren()) { **if** (item **instanceof** GraphTreeItem) { GraphTreeItem gti = (GraphTreeItem) item; **if** (gti.node == VisualVertex.**this**) { tv.getSelectionModel().select(gti); **break**; } } } }); } @SuppressWarnings("unchecked") **public** **void** setActive(**boolean** active) { **if** (getStyleClass().contains("active")) getStyleClass().remove("active"); **else** **if** (active) getStyleClass().add("active"); getParent().getChildrenUnmodifiable().stream().filter(n -> (n **instanceof** VisualEdge) && WSeminar.instance.getGraph().isConnected(vertex, ((VisualEdge<V>) n).edge)).forEach(n -> ((VisualEdge<V>) n).setActive(active)); **this**.active = active; } **public** **void** setState(State state) { **if** (state == State.REVERT) { revertState(); **return**; } **if** (**this**.state == state) **return**; **if** ((**this**.state == State.GOAL || **this**.state == State.START) && state != **null**) { **return**; } **if** (**this**.state != **null**) getStyleClass().remove(**this**.state.name().toLowerCase()); **if** (state != **null**) getStyleClass().add(state.name().toLowerCase()); prevState = **this**.state; **this**.state = state; } **public** **void** resetState(**boolean** start) { **if** ((state != State.GOAL && state != State.START) || start) setState(**null**); } **public** **void** revertState() { state = prevState; prevState = **null**; } **public** Vertex<V> getVertex() { **return** vertex; } } **package** de.dakror.wseminar.util; **import** java.util.ArrayList; **import** java.util.List; **import** com.sun.javafx.collections.ObservableListWrapper; **import** de.dakror.wseminar.graph.Edge; **import** javafx.collections.ObservableList; **public** **class** Benchmark<V> { **public** **static** **class** Timestamp **implements** Comparable<Timestamp> { **public** **final** **long** time; **public** **final** **float** stamp; **public** Timestamp(**long** time, **float** stamp) { **this**.time = time; **this**.stamp = stamp; } @Override **public** **int** compareTo(Timestamp o) { **return** Long.compare(time, o.time); } } **public** **static** **enum** Type { VERTICES("Knoten"), VERTICES\_UNIQUE("Verschiedene Knoten"), SORTS("Sortierungen"), OPEN\_LIST\_SIZE("OpenList-Größe"), CLOSED\_LIST\_SIZE("ClosedList-Größe"), BACK\_TRACKS("Rückschritte"), PATH\_CREATION("Wegerstellung"), OVERRIDES("Überschreibungen"); **public** **final** String desc; **private** Type(String desc) { **this**.desc = desc; } **public** **static** Type getByDesc(String desc) { **for** (Type t : values()) { **if** (t.desc.equals(desc)) **return** t; } **return** **null**; } } **long** time, time1; **long** firstTime; ObservableList<Timestamp>[] values; List<V> vertices; List<Edge<V>> edges; @SuppressWarnings("unchecked") **public** Benchmark() { values = **new** ObservableList[Type.values().length]; **for** (**int** i = 0; i < values.length; i++) values[i] = **new** ObservableListWrapper<>(**new** ArrayList<>()); vertices = **new** ArrayList<>(); edges = **new** ArrayList<>(); } **public** **void** add(Type t, **float** f) { **long** time = System.nanoTime() / 1000; **if** (firstTime == 0) firstTime = time; **float** last = values[t.ordinal()].size() > 0 ? values[t.ordinal()].get(values[t.ordinal()].size() - 1).stamp : 0; values[t.ordinal()].add(**new** Timestamp(time - firstTime, last + f)); } **public** **void** add(Type t) { add(t, 1); } **public** **void** sub(Type t) { add(t, -1); } **public** **void** add(V v) { add(Type.VERTICES); **if** (!vertices.contains(v)) { vertices.add(v); add(Type.VERTICES\_UNIQUE); } } **public** ObservableList<Timestamp> get(Type t) { **return** values[t.ordinal()]; } **public** ObservableList<Timestamp>[] getAll() { **return** values; } **public** **void** time() { **if** (time1 == 0) time1 = System.nanoTime(); **else** { time = System.nanoTime() - time1; time1 = 0; } } **public** **long** getTime() { **return** time; } **public** **void** setTime(**long** time) { **this**.time = time; } **public** **float** getMin(Type t) { **float** m = Float.MAX\_VALUE; **for** (Timestamp ts : get(t)) **if** (ts.stamp < m && ts.stamp > 0) m = ts.stamp; **if** (m == Float.MAX\_VALUE) **return** 0; **return** m; } **public** **float** getSum(Type t) { **float** m = 0; **for** (Timestamp ts : get(t)) **if** (ts.stamp > m) m = ts.stamp; **return** m; } } **package** de.dakror.wseminar.util; **import** java.io.File; **import** java.io.IOException; **import** java.util.Arrays; **import** javafx.application.Application; **import** javafx.css.CssMetaData; **import** javafx.fxml.FXMLLoader; **import** javafx.scene.Node; **import** javafx.scene.Parent; **import** javafx.scene.Scene; **import** javafx.stage.Stage; **public** **class** FXML2CSS **extends** Application { **public** **static** **void** main(String[] args) { **if** (args.length == 0) **throw** **new** IllegalArgumentException("No file specified"); launch(args); } @Override **public** **void** start(Stage primaryStage) **throws** Exception { File fxml = **new** File(getParameters().getRaw().get(0)); **try** { FXMLLoader l = **new** FXMLLoader(fxml.toURI().toURL()); Scene scene = **new** Scene(l.load()); Stage stage = **new** Stage(); stage.setOpacity(0); stage.setScene(scene); stage.show(); printRec(stage.getScene().getRoot()); } **catch** (IOException e) { e.printStackTrace(); } System.exit(0); } @SuppressWarnings({ "unchecked", "rawtypes" }) **public** **static** **void** printRec(Parent parent) { **for** (Node c : parent.getChildrenUnmodifiable()) { **if** (c **instanceof** Parent) { printRec((Parent) c); } **else** **if** (c.getId() != **null**) { System.out.println(); System.out.println("#" + c.getId() + "{"); **for** (CssMetaData css : c.getCssMetaData()) { Object value = css.getStyleableProperty(c).getValue(); **if** (value != **null** && !value.equals(css.getInitialValue(c))) { String str = value.toString(); **if** (value **instanceof** Double[]) { **if** (((Double[]) value).length == 0) **continue**; str = Arrays.toString((Double[]) value); } System.out.println(" " + css.getProperty() + ": " + str + ";"); } } System.out.println("}"); } } } } **package** de.dakror.wseminar.util; **import** de.dakror.wseminar.Const.State; **import** de.dakror.wseminar.WSeminar; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.ui.VisualEdge; **import** de.dakror.wseminar.ui.VisualVertex; **import** javafx.application.Platform; **import** javafx.scene.Node; **public** **class** Visualizer { **static** **boolean** enabled = **true**; **public** **static** **void** setEnabled(**boolean** enable) { enabled = enable; } **public** **static** **boolean** isEnabled() { **return** enabled; } **public** **static** <V> **void** setVertexState(Vertex<V> v, State state) { setVertexState(v, state, **true**); } @SuppressWarnings("unchecked") **public** **static** <V> **void** setVertexState(Vertex<V> v, State state, **boolean** tick) { **if** (enabled) Platform.runLater(() -> { ((VisualVertex<V>) lookup("#V" + v.data())).setState(state); }); **if** (tick) tick(); } **public** **static** <V> **void** setEdgePath(Edge<Vertex<V>> e, **boolean** path) { setEdgePath(e, path, **true**); } @SuppressWarnings("unchecked") **public** **static** <V> **void** setEdgePath(Edge<Vertex<V>> e, **boolean** path, **boolean** tick) { **if** (enabled) Platform.runLater(() -> { ((VisualEdge<V>) lookup("#E" + e.hashCode())).setPath(path); }); **if** (tick) tick(); } @SuppressWarnings("unchecked") **public** **static** <V> **void** setEdgePath(Edge<Vertex<V>> e, **boolean** path, **boolean** soft, **boolean** tick) { **if** (enabled) Platform.runLater(() -> { ((VisualEdge<V>) lookup("#E" + e.hashCode())).setPath(path, soft); }); **if** (tick) tick(); } **public** **static** <V> **void** setEdgeActive(Edge<Vertex<V>> e, **boolean** active) { setEdgeActive(e, active, **true**); } @SuppressWarnings("unchecked") **public** **static** <V> **void** setEdgeActive(Edge<Vertex<V>> e, **boolean** active, **boolean** tick) { **if** (enabled) Platform.runLater(() -> { ((VisualEdge<V>) lookup("#E" + e.hashCode())).setActive(active); }); **if** (tick) tick(); } @SuppressWarnings("unchecked") **public** **static** <V> **void** resetAll(Graph<Vertex<V>> graph, **boolean** full, **boolean** start) { **if** (enabled) Platform.runLater(() -> { **for** (Edge<Vertex<V>> e : graph.getEdges()) { ((VisualEdge<V>) lookup("#E" + e.hashCode())).setActive(**false**); **if** (full) ((VisualEdge<V>) lookup("#E" + e.hashCode())).setPath(**false**); VisualVertex<V> f = ((VisualVertex<V>) lookup("#V" + e.getFrom().data())); VisualVertex<V> t = ((VisualVertex<V>) lookup("#V" + e.getTo().data())); **if** (full) { f.resetState(start); t.resetState(start); } f.setActive(**false**); t.setActive(**false**); } }); } **public** **static** **void** tick() { **try** { **if** (enabled) Thread.sleep(300); } **catch** (InterruptedException e) { e.printStackTrace(); } } **static** Node lookup(String selector) { Node n = WSeminar.window.getScene().lookup(selector); **if** (n == **null**) System.err.println(selector); **return** n; } } **package** de.dakror.wseminar; **import** java.io.IOException; **import** java.util.ArrayList; **import** java.util.HashMap; **import** java.util.Random; **import** de.dakror.wseminar.controller.GenerateGraphDialogController; **import** de.dakror.wseminar.graph.Edge; **import** de.dakror.wseminar.graph.Graph; **import** de.dakror.wseminar.graph.Path; **import** de.dakror.wseminar.graph.Vertex; **import** de.dakror.wseminar.graph.VertexData.Delay; **import** de.dakror.wseminar.graph.algorithm.base.Layout; **import** de.dakror.wseminar.ui.GraphTreeCell; **import** de.dakror.wseminar.ui.GraphTreeItem; **import** de.dakror.wseminar.ui.VisualEdge; **import** de.dakror.wseminar.ui.VisualVertex; **import** javafx.animation.FadeTransition; **import** javafx.animation.Interpolator; **import** javafx.animation.KeyFrame; **import** javafx.animation.KeyValue; **import** javafx.animation.ParallelTransition; **import** javafx.animation.ScaleTransition; **import** javafx.animation.Timeline; **import** javafx.animation.TranslateTransition; **import** javafx.application.Application; **import** javafx.fxml.FXMLLoader; **import** javafx.geometry.Bounds; **import** javafx.geometry.Pos; **import** javafx.scene.Node; **import** javafx.scene.Parent; **import** javafx.scene.Scene; **import** javafx.scene.control.Label; **import** javafx.scene.control.MenuButton; **import** javafx.scene.control.MenuItem; **import** javafx.scene.control.Slider; **import** javafx.scene.control.TreeItem; **import** javafx.scene.control.TreeView; **import** javafx.scene.image.Image; **import** javafx.scene.layout.Pane; **import** javafx.scene.paint.Color; **import** javafx.scene.shape.Line; **import** javafx.scene.text.Font; **import** javafx.scene.text.FontWeight; **import** javafx.stage.Modality; **import** javafx.stage.Stage; **import** javafx.stage.StageStyle; **import** javafx.stage.Window; **import** javafx.util.Duration; **public** **class** WSeminar **extends** Application { **public** **static** WSeminar instance; **public** **static** Window window; **public** **static** Random r; **public** **static** **long** seed; **static** HashMap<String, Image> imgCache = **new** HashMap<>(); Graph<Integer> sourceGraph; Layout<Integer> layout; Graph<Vertex<Integer>> graph; **int** graphSize; **public** VisualVertex<Integer> activeVertex, startVertex, goalVertex; **public** VisualEdge<Integer> activeEdge; **public** HashMap<Integer, Path<Vertex<Integer>>> paths = **new** HashMap<>(); **final** **int** duration = 200; **public** **boolean** selectStartVertex, selectGoalVertex; @Override **public** **void** start(Stage primaryStage) **throws** Exception { instance = **this**; window = primaryStage; primaryStage.setOnCloseRequest(e -> System.exit(0)); primaryStage.minWidthProperty().set(1280); primaryStage.minHeightProperty().set(720); primaryStage.setScene(createScene("main")); primaryStage.setTitle("PathFinder - WSeminar Extrema"); primaryStage.getIcons().addAll(getImage("mind\_map-24.png"), getImage("mind\_map-32.png")); Rectangle2D r = Screen.getScreens().get(2).getBounds(); primaryStage.setX(r.getMinX()); primaryStage.setY(r.getMinY()); primaryStage.setWidth(r.getWidth()); primaryStage.show(); } **public** **void** setSourceGraph(Graph<Integer> sourceGraph) { getMenuItem("#menu\_graph", "relayout\_graph").setDisable(sourceGraph == **null**); **this**.sourceGraph = sourceGraph; } **public** Graph<Integer> getSourceGraph() { **return** sourceGraph; } **public** **void** setLayout(Layout<Integer> layout) { **this**.layout = layout; } **public** Layout<Integer> getLayout() { **return** layout; } **public** **void** setGraphSize(**int** graphSize) { **this**.graphSize = graphSize; } **public** **int** getGraphSize() { **return** graphSize; } **public** Graph<Vertex<Integer>> getGraph() { **return** graph; } **public** **void** transitionTo(Graph<Vertex<Integer>> graph) { **if** (**this**.graph == **null**) { setGraph(graph, **true**); **return**; } Pane pane = (Pane) WSeminar.window.getScene().lookup("#graph"); ParallelTransition pt = **new** ParallelTransition(); **for** (**int** i = 0; i < graph.getVertices().size(); i++) { Vertex<Integer> v = graph.getVertices().get(i); Node node = pane.lookup("#V" + v.data()); Node text = pane.lookup("#VT" + v.data()); TranslateTransition tt = **new** TranslateTransition(Duration.millis(GenerateGraphDialogController.speed), node); VisualVertex<Integer> newCircle = **new** VisualVertex<Integer>(v); tt.setToX(newCircle.getTranslateX()); tt.setToY(newCircle.getTranslateY()); TranslateTransition tt2 = **new** TranslateTransition(Duration.millis(GenerateGraphDialogController.speed), text); tt2.setToX(newCircle.getTranslateX()); tt2.setToY(newCircle.getTranslateY()); pt.getChildren().addAll(tt, tt2); } **for** (**int** i = 0; i < graph.getEdges().size(); i++) { Edge<Vertex<Integer>> e = graph.getEdges().get(i); Line node = (Line) pane.lookup("#E" + e.hashCode()); VisualVertex<Integer> newFrom = **new** VisualVertex<Integer>(e.getFrom()); VisualVertex<Integer> newTo = **new** VisualVertex<Integer>(e.getTo()); Timeline tl = **new** Timeline(**new** KeyFrame(**new** Duration(GenerateGraphDialogController.speed), **new** KeyValue(node.startXProperty(), newFrom.getTranslateX() + Const.cellSize / 2), **new** KeyValue(node.startYProperty(), newFrom.getTranslateY() + Const.cellSize / 2), **new** KeyValue(node.endXProperty(), newTo.getTranslateX() + Const.cellSize / 2), **new** KeyValue(node.endYProperty(), newTo.getTranslateY() + Const.cellSize / 2))); pt.getChildren().add(tl); } pt.play(); **this**.graph = graph; } @SuppressWarnings("unchecked") **public** **void** setGraph(Graph<Vertex<Integer>> graph, **boolean** animate) { activeVertex = **null**; **this**.graph = graph; Node n = window.getScene().lookup("#newGraph"); **if** (n != **null**) n.setVisible(graph == **null**); Pane pane = (Pane) window.getScene().lookup("#graph"); pane.getChildren().clear(); ((Slider) window.getScene().lookup("#zoom")).setValue(200); pane.setScaleX(1); pane.setScaleY(1); TreeView<String> tv = ((TreeView<String>) window.getScene().lookup("#graph\_tree")); tv.setCellFactory(tree -> **new** GraphTreeCell()); GraphTreeItem root = **new** GraphTreeItem(**null**, "Graph"); tv.setRoot(root); ArrayList<Vertex<Integer>> addedVertices = **new** ArrayList<>(); **for** (**int** i = 0; i < graph.getEdges().size(); i++) { Edge<Vertex<Integer>> e = graph.getEdges().get(i); VisualEdge<Integer> edge = **new** VisualEdge<>(e, pane); **for** (Vertex<Integer> v : **new** Vertex[] { e.getFrom(), e.getTo() }) { **int** index = addedVertices.indexOf(v); VisualVertex<Integer> circle = **new** VisualVertex<Integer>(v); circle.setId("V" + v.data()); GraphTreeItem gti = **new** GraphTreeItem(circle, "Vertex", v.data()); **if** (v.equals(e.getFrom()) || !e.isDirected()) { TreeItem<String> item = index == -1 ? gti : root.getChildren().get(index); item.getChildren().add(**new** GraphTreeItem(edge, "Kante " + (e.isDirected() ? "" : "<") + "-> " + e.getOtherEnd(v).data(), i)); } **if** (index > -1) **continue**; root.getChildren().add(gti); Label l = **new** Label(v.data() + ""); l.setId("VT" + v.data()); l.setTextFill(Color.BLACK); l.setFont(Font.font(**null**, FontWeight.NORMAL, 15)); l.setMinSize(Const.cellSize, Const.cellSize); l.setAlignment(Pos.CENTER); l.setTranslateX(circle.getTranslateX()); l.setTranslateY(circle.getTranslateY()); FadeTransition ft = **new** FadeTransition(Duration.millis(duration), l); ft.setFromValue(0); ft.setToValue(1); ft.setInterpolator(Interpolator.EASE\_OUT); FadeTransition ft2 = **new** FadeTransition(Duration.millis(duration), circle); ft2.setFromValue(0); ft2.setToValue(1); ft2.setInterpolator(Interpolator.EASE\_OUT); ScaleTransition st = **new** ScaleTransition(Duration.millis(duration), circle); st.setFromX(0); st.setFromY(0); st.setToX(1); st.setToY(1); st.setInterpolator(Const.overlyEaseIn); l.setDisable(**true**); ParallelTransition pt = **new** ParallelTransition(circle, ft, ft2, st); pane.getChildren().add(circle); pane.getChildren().add(l); **float** delay = (**float**) (Math.random() \* 400); Delay d = **new** Delay(); d.delay = delay; v.add(d); pt.setDelay(Duration.millis(delay)); **if** (animate) pt.play(); addedVertices.add(v); } pane.getChildren().add(0, edge); FadeTransition ft2 = **new** FadeTransition(Duration.millis(duration), edge); ft2.setFromValue(0); ft2.setToValue(1); ft2.setInterpolator(Interpolator.EASE\_OUT); ScaleTransition st = **new** ScaleTransition(Duration.millis(duration), edge); st.setFromX(0); st.setFromY(0); st.setToX(1); st.setToY(1); ParallelTransition pt = **new** ParallelTransition(edge, ft2, st); pt.setDelay(Duration.millis(Math.max(e.getFrom().get(Delay.**class**).delay, e.getTo().get(Delay.**class**).delay))); **if** (animate) pt.play(); } root.getChildren().sort((a, b) -> Integer.compare(((GraphTreeItem) a).getParam(), ((GraphTreeItem) b).getParam())); Bounds pB = pane.getParent().getParent().getBoundsInParent(); pane.setTranslateX((pB.getWidth() - layout.getBounds().getWidth() \* Const.cellSize) / 2); pane.setTranslateY((pB.getHeight() - layout.getBounds().getHeight() \* Const.cellSize) / 2); } **public** **static** **void** setSeed(**long** seed2) { r = **new** Random(seed2); seed = seed2; } **public** **static** Scene createScene(String resource) { **try** { FXMLLoader l = **new** FXMLLoader(WSeminar.**class**.getResource("/assets/fxml/" + resource + ".fxml")); Parent pane = (Parent) l.load(); Scene scene = **new** Scene(pane); scene.getStylesheets().addAll("assets/css/Theme.css", "assets/css/style.css"); **return** scene; } **catch** (IOException e) { e.printStackTrace(); **return** **null**; } } **public** **static** Stage createDialog(String resource, String title, Window owner) { **return** createDialog(resource, title, owner, StageStyle.DECORATED, Modality.APPLICATION\_MODAL); } **public** **static** Stage createDialog(String resource, String title, Window owner, StageStyle style, Modality modality) { Stage stage = **new** Stage(); stage.setResizable(**false**); stage.initStyle(style); stage.getIcons().addAll(WSeminar.getImage("mind\_map-24.png"), WSeminar.getImage("mind\_map-32.png")); stage.setScene(WSeminar.createScene(resource)); stage.setTitle(title); stage.initModality(modality); stage.initOwner(owner); stage.show(); **return** stage; } **public** **static** MenuItem getMenuItem(String menuSelector, String id) { MenuButton menu = (MenuButton) window.getScene().lookup(menuSelector); **for** (MenuItem mi : menu.getItems()) **if** (mi.getId().equals(id)) **return** mi; System.out.println("NOP"); **return** **null**; } **public** **static** Image getImage(String resource) { **if** (imgCache.containsKey(resource)) **return** imgCache.get(resource); **else** { Image img = **new** Image(WSeminar.**class**.getResourceAsStream("/assets/img/" + resource)); imgCache.put(resource, img); **return** img; } } **public** **static** **void** main(String[] args) { Thread.setDefaultUncaughtExceptionHandler((t, e) -> { e.printStackTrace(); Stage stage = createDialog("alert", "Fehler!", window); ((Label) stage.getScene().lookup("#message")).setText(e.getClass().getSimpleName()); ((Label) stage.getScene().lookup("#details")).setText(e.getLocalizedMessage()); }); launch(args); } }