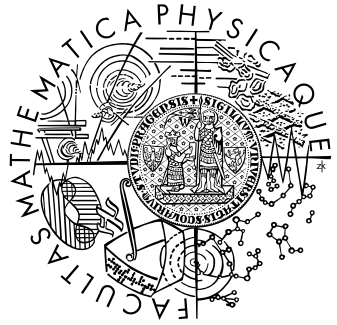
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Petr Koupý

Visualization of problems of motion on a graph

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Pavel Surynek, Ph.D.

Study program: Computer Science, Programming

2010

I would like to thank my supervisor, Pavel Surynek, not only for giving me an idea on this thesis but also for his guidance, swift support, suggestions and numerous pieces of advice.

I hereby declare that I wrote the thesis myself using only the referenced sources. I also agree with lending and publishing of this thesis.

In Prague, May 17, 2010 Petr Koupý

Contents

Title: Visualization of problems of motion on a graph

Author: Petr Koupý
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: RNDr. Pavel Surynek, Ph.D.
Supervisor's e-mail address: pavel.surynek@mff.cuni.cz

Abstract: A software tool visualizing the movement of entities on a graph is presented in this thesis. Such model is often used to abstract environment where the given set of entities must be reordered from an initial to a certain goal configuration in space. Software solvers of these problems usually produce sub-optimal solutions in the textual form, which is generally hard to explore by a human. Thus, the visualization tool can be utilized by a researcher when analyzing the quality of such solutions. In order to visualize solutions, the presented tool handles a set of problems – embedding the graph into a plane, controlling the animation, capturing the output to images or video files, managing colors and validating movements in the solution. The thesis provides detailed information about the implementation of the tool including the choice of suitable algorithms, architecture and technologies.

Keywords: visualization, graph embedding, movement on graph, video capturing

Název práce: Visualization of problems of motion on a graph

Autor: Petr Koupý
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: RNDr. Pavel Surynek, Ph.D.
E-mail vedoucího: pavel.surynek@mff.cuni.cz

Abstrakt: Práce popisuje softwarový nástroj pro vizualizaci pohybu entit po grafu. Toto modelové prostředí je často použito jako abstrakce problémů, kde daná množina entit musí být přeuspořádána z nějaké počáteční do určitě cílové konfigurace v prostoru. Programy řešící tyto problémy obvykle používají sub-optimální algoritmy pro vygenerování řešení textového charakteru, která jsou obecně nevhodná pro prohlížení člověkem. Vizualizační nástroj tedy může být využit výzkumníkem při analýze kvality takových řešení. Nástroj pro vizualizaci řeší několik problémů – kreslení grafu do roviny, ovládání animace, zaznamenávání výstupu do obrazových nebo video souborů, správu barev a validaci pohybů. Práce poskytuje detailní informace o implementaci nástroje včetně výběru vhodných algoritmů, architektury a technologií.

Klíčová slova: vizualizace, kreslení grafu, pohyb po grafu, zaznamenávání videa

1 Introduction

Movement of entities on a graph is a basic abstraction for many real-life tasks, as stated in [17]:

> *These tasks include rearranging containers in storage yards, coordination of movements of a large group of automated agents, or optimization of dense traffic. However, this is not the only motivation. ... An example may be data transfer with limited buffers at communication nodes, a coordination of a group of agents in strategic computer games, or planning movements in mass scenes in computer-generated imagery.*

The environment for these tasks is abstracted as a graph, where vertices represent discrete positions and edges possible paths between them. Cars, containers or packets are represented by entities that move among such graph. The problem is then defined by an initial and goal positions of entities on the graph together with restrictions that must be considered when planning movements – physical environments for example do not allow two or more entities to be located in one place at the same time. Solution to this problem consists of a sequence of entity movements that transforms the initial configuration to the final one while not breaking any of defined restrictions. This leads to variants of problem, two of which are widely known as *pebble motion on a graph* [10] and *multi-robot path planning* [16]. Notice the problem is solved in a centralized manner, where entities do not move autonomously. Several existing algorithms produce solutions to these problems. The currently best algorithm is presented in [17]. Generally, the shorter the solution is in terms of the number of movements the better. However, as mentioned in [17] and proved in [15] and [18]:

> *A natural additional requirement is to produce shortest possible solutions (...). Unfortunately, this requirement makes the problem intractable (namely NP-hard).*

Therefore, the solutions produced by any reasonably fast algorithm are expected to contain various types of redundancies, some of which might be possibly identified and removed by a certain set of additional algorithms. Because available solvers produce solutions only in a textual form, which is not suitable for analysis by a human, the process of designing such algorithms would quickly lead to a problem of not knowing the exact visual form of a solution. Thus, there is a motivation for a visualization tool (Figure 1). This thesis describes the development and implementation of such tool.

Figure 1. From a practical motivation over the textual solution to the abstract *visualization*.

1.1 Tool Overview

GraphRec [11] is a visualization tool oriented on the animation of moving entities on the generic middle-sized graph containing tens to hundreds of nodes. The tool provides an **animation engine** for the entity movement together with features designed to support the observation of the solution time line. In particular, the **graph must be embedded** on the screen be-

fore the visualization can even occur. Additionally, the tool can **capture the animation** into various image or video formats. Any similar tool has not been available up until now. With the existing graph visualization software (e.g. *Graphviz* [1]) it is neither possible to represent entities nor move them among graph nodes. Here is a brief description of how program works:

1) As an input, user provides the problem solution containing graph definition, initial positions of entities and the complete list of entity movements.
2) The tool loads the input and calculates positions of nodes to produce a graph layout.
3) User adjusts the colors of nodes and entities.
4) In a similar way as with video recorder, user navigates through the animation with the possibility to record it.

1.2 Tool Purpose

The purpose of the tool is to act as a visual frontend for the existing text-based applications, which solve planning problems that can be abstracted as the movement of entities on a graph. In this way, the tool can be used to analyze solutions of these problems visually – either to **identify various redundancies** or simply to **compare differently optimized solutions** of the same problem. Since the automatic detection of redundancies with unknown characteristics is not possible, the initial analysis by a human is essential. Because humans are mainly visual-oriented, the visualization of the problem seems to be a suitable approach.

The tool can also find inconsistencies in a given solution by verifying its movements against constraints specified in the definition of the variant of motion problem. Solution validation is necessary to prevent the corruption of the animation. However, the validation can also be utilized when **debugging solving algorithms**.

Moreover, the tool is designed for **teaching and presentation** purposes, where the visualization helps people to understand how the presented solving algorithm works. Captured **media files** might accompany articles either on a paper or on the web.

1.3 Tool Scope

The tool should not be considered as a graph embedding software. Although the implementation contains two **embedding** algorithms to provide decent looking graphs, it is not a main goal. Program also should not be mistaken with applications that actually solve planning problems. Core functionality is focused on **animation**, interactive **user interface** and production of **image and video output**. In a model situation where there are three types of software – one for creating a problem definition, one for generating a solution and one for visualization of the solution – the presented tool is supposed to be the third one. Program is **strictly GUI application** not providing any kind of command line interface.

1.4 Thesis Structure

The visualization tool and its motivation are introduced in **section 1**. Closer look at the problem of visualization of entity movement on a graph together with the detailed analysis of design decisions, picked algorithms and chosen technologies is provided in **section 2**. User's guide containing user stories and description of the graphical interface is located in **section 3**. Programmer's documentation consisting of compilation, architecture analysis and class description is the content of **section 4**. Summarization of the thesis and achieved results are presented in **section 5**.

2 Design Analysis

Visualization tool deals with a **diverse set of problems**. Not only does it provide support for the animation, but also resolves various affiliated tasks – from graph embedding to capturing a video. Before going into detail, there is a list of targeted features:

Input

- User can load multiple files at once.
- User can choose what to load when file contains more than one solution.
- It is possible to validate a solution and review its errors.

GUI

- Tabbed interface where each tab contains one opened solution.
- Support for zooming, scrolling and rotating.
- User can edit properties (location, color) of multiple graph elements at once.

Layout

- Support for the automatic embedding of a graph into 2-dimensional plane.
- It is possible to adjust graph layout manually by selecting and dragging nodes.

Animation

- Controls allowing user to start, step, stop and seek the animation.
- Animation speed is adjustable.
- Moving entities can be visually highlighted.
- It is possible to run parallel animation of more than one solution at once.

Capturing

- Image snapshots can be taken manually or scheduled with adjustable time interval.
- Support for both raster and vector image formats.
- Animation can be captured into popular video formats.

Output

- Solution can be saved together with graph layout, coloring and current view state.
- All opened tabs can be saved into a single file at once.

2.1 Graph Embedding

Before the visualization can even occur, the graphs on which the movement will be animated have to be embedded on the screen. In order to create universal tool, there is no assumption about the graph characteristics. Since general graphs are not necessarily *planar*, the embedding algorithm should at least **reduce** the amount of **crossing edges** while maintaining *Euclidean* distances between nodes proportional to some reasonably chosen metric (e.g. *shortest paths*).

GraphRec implements two widely used *force-directed* planar embedding algorithms described in [6] and [9]. Both methods are based on the simulation of a certain physical model. Whereas the model introduced in [6] considers nodes as repulsive particles and edges as contracting springs, another interpretation where chosen free node is connected by springs to the rest of anchored nodes is proposed in [9]. Owing to their physical background, force-directed algorithms often produce **expected and intuitive layouts**.

2.1.1 Fruchterman-Reingold Method

Embedding method discovered in [6] is based on the idea that nodes are **repulsive particles** and edges are **contracting springs** (Figure 2). Considering the repulsive power of nodes, they

are all equal. Same stands for edges – all of them have the same stiffness. The equilibrium of the springs is set to zero length, so that edges tend to contract all the time. Algorithm gradually provides better and better layout for **all nodes at each cycle**. Whereas single iteration calculates repulsive powers among all nodes, attractive powers are calculated only between adjacent nodes (those connected by =an edge). Utilized force model is **elastic**, which allows user to interact with running algorithm. GraphRec uses slightly modified version of the algorithm (Algorithm 1).

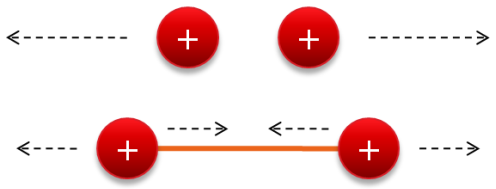


Figure 2. *Repulsive* and *attractive* forces.

Algorithm 1. Modified version of *Fruchterman-Reingold* [6] embedding algorithm.

FRUCHTERMANREINGOLD$(G(V,E), disp)$
{initial random positions of nodes, target displacement of nodes }
1 **forever**
{calculate new positions}
2 **for each** $v = (x, y, acc_x, acc_y) \in V$ **do**
3 $v.acc_x \leftarrow 0$ {accumulator for horizontal position change}
4 $v.acc_y \leftarrow 0$ {accumulator for vertical position change}
{accumulate repulsive forces}
5 **for each** $u = (x, y, acc_x, acc_y) \in V \setminus \{v\}$ **do**
6 **let** $d_x \leftarrow |v.x - u.x|$
7 **let** $d_y \leftarrow |v.y - u.y|$
8 **let** $d \leftarrow d_x^2 + d_y^2$
9 **let** $f_r \leftarrow disp^2/\sqrt{d}$ {repulsive force multiplier}
10 $v.acc_x \leftarrow v.acc_x + (d_x \cdot f_r)/d$
11 $v.acc_y \leftarrow v.acc_y + (d_y \cdot f_r)/d$
{accumulate attractive forces}
12 **for each** $e = (m,n) \in E_v = \{e \in E \mid \exists w \in V : e = (v,w) \vee e = (w,v)\}$ **do**
13 **let** $d_x \leftarrow |e.m.x - e.n.x|$
14 **let** $d_y \leftarrow |e.m.y - e.n.y|$
15 **let** $d \leftarrow d_x^2 + d_y^2$
16 **let** $f_a \leftarrow d/disp$ {attractive force multiplier}
17 $v.acc_x \leftarrow v.acc_x - (|E_v| \cdot d_x \cdot f_a)/d$
18 $v.acc_y \leftarrow v.acc_y - (|E_v| \cdot d_y \cdot f_a)/d$
{update positions}
19 **let** $static \leftarrow 0$ {counter for nodes with insignificant change}
20 **for each** $v = (x, y, acc_x, acc_y) \in V$ **do**
21 **if** $v.acc_x < \varepsilon \wedge v.acc_y < \varepsilon$ **then**
22 $static \leftarrow static + 1$
23 **else**
24 $v.x \leftarrow v.x + v.acc_x$
25 $v.y \leftarrow v.y + v.acc_y$
26 **if** $static = |V|$ **then**
27 **break**
28 **return** $G(V,E)$

Time complexity of the single iteration is $O(|V|^2 + |E|)$. Original algorithm is terminated by *simulated annealing* method, which gradually weakens the forces allowing the algorithm to focus more and more on minor rather than radical changes. Unfortunately, simulated annealing is not suitable for interactive embedding, because the process can be altered by a user any

time. Therefore, modified algorithm is terminated simply by checking when all vertices are moved only insignificantly.

Moreover, modified algorithm emphasizes the dependency between an attractive force and the number of edges that are connected to the vertex. Since the attractive force is additionally multiplied by the number of connected edges, poorly connected vertices on the outer boundary of the graph are not so much attracted by adjacent nodes. This approach gives better layout especially in the case of the regular grid, where boundary nodes tend to be dragged to the center of the graph otherwise. Another difference is that modified algorithm does not calculate some of the squaring and root extractions – algorithm then behaves more dynamically while a user is dragging some node (graph follows dragged node better). It should be noted that these modifications were made in the trial and error manner.

2.1.2 Kamada-Kawai Method

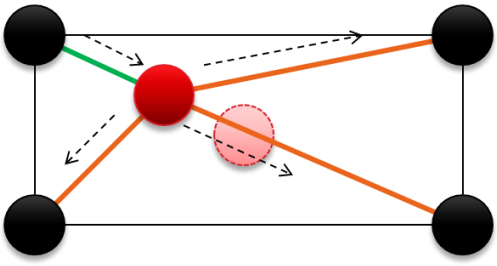


Figure 3. Node moving to the *equilibrium*.

Method described in [9] is also based on the simulation of a certain physical system. Target Euclidean distances between nodes are proportional to their graph-theoretical distances (shortest paths). This time, a single iteration improves position of **only one node**, which renders the algorithm **non-elastic** (it does not react to the alteration by a user). When calculating position of a chosen node, all other nodes are considered as solid anchors for springs that are hooked together in a location of the currently processed node. **Springs are either contractive or repulsive** depending on their current stretch. Stiffness and equilibrium of each spring is different – it is proportional to the shortest path between the two nodes. Algorithm utilizes several methods and principles from differential calculus, linear algebra, graph theory and material mechanics. Following explanation is a compact and customized retelling of the method discovered in [9], which is necessary to understand GraphRec source code.

All-pairs shortest-paths problem can be solved in $O(|V|^3)$ by *Floyd-Warshall* algorithm (Algorithm 2), which is explained in [4].

Algorithm 2. *Floyd-Warshall* [4] algorithm for finding all-pairs shorthes-paths.

FLOYDWARSHALL$(G(V,E))$

1 **let** $D = \{d_{ij}\} \leftarrow \begin{cases} 0, & i = j \\ 1, & v_i, v_j \in V, (v_i, v_j) \in E \\ \infty, & v_i, v_j \in V, (v_i, v_j) \notin E \end{cases}$
2 **for** $k \leftarrow 1, \dots, |V|$ **do**
3 **for** $i \leftarrow 1, \dots, |V|$ **do**
4 **for** $j \leftarrow 1, \dots, |V|$ **do**
5 $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$
6 **return** D

Spring counteracts its elongation/contraction by the force $F(x) = K(x - x_0)$, which is linearly proportional to the deflection x from its equilibrium x_0. Accumulated potential energy corresponds to the integral of force:

$$E(x) = \int F(x)\,dx = \frac{1}{2}K(x - x_0)^2$$

Factor K is defined for nodes $v_i, v_j \in V$ as $k_{ij} = K_0/d_{ij}^2$, where K_0 is an arbitrary constant. Thus, spring is more solid between closer nodes (because of a shorter path). Similarly, equili-

brium length of the spring is defined as $l_{ij} = disp \cdot d_{ij}$, where $disp$ is an external parameter specifying the target displacement of nodes. Finally, considering a set of nodes $|V| = n$, where $v_i \in V$ has Euclidean coordinates x_i, y_i, the potential energy of the whole system can be defined as a sum of potential energies for all springs:

$$E(x_1, \dots, x_n, y_1, \dots, y_n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} \left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - l_{ij} \right)^2$$

At each iteration, algorithm chooses such node v_m that has maximum size of the gradient vector ∇E_m, where $E_m(x_m, y_m)$ is the function E whose variables (apart from x_m, y_m) are considered as fixed constants:

$$\nabla E_m = \left(\frac{\partial E}{\partial x_m}, \frac{\partial E}{\partial y_m} \right), |\nabla E_m| = \sqrt{\left(\frac{\partial E}{\partial x_m} \right)^2 + \left(\frac{\partial E}{\partial y_m} \right)^2}$$

$$\frac{\partial E}{\partial x_m} = \sum_{i=1}^{n} k_{mi} \left((x_m - x_i) - \frac{l_{mi}(x_m - x_i)}{\sqrt{(x_m - x_i)^2 + (y_m - y_i)^2}} \right)$$

$$\frac{\partial E}{\partial y_m} = \sum_{i=1}^{n} k_{mi} \left((y_m - y_i) - \frac{l_{mi}(y_m - y_i)}{\sqrt{(x_m - x_i)^2 + (y_m - y_i)^2}} \right)$$

Gradient vector is then gradually decreased by moving the chosen node to the local minimum of E_m. *Newton-Raphson* numerical method (Figure 4) approximates the zero of the function $f(x)$ by iterating through the equation $x_{n+1} = x_n - f(x_n)/f'(x_n)$, which can be also expressed as $f'(x_n) \cdot dx = -f(x_n)$, where $dx = (x_{n+1} - x_n)$. Let us apply this method to approximate the zero of the gradient vector ∇E_m, which effectively finds the local minimum of E_m. This includes calculating the *Jacobian* of ∇E_m:

$$J_{\nabla E_m} \cdot \begin{pmatrix} dx \\ dy \end{pmatrix} = -\nabla E_m$$

$$\begin{bmatrix} \frac{\partial^2 E}{\partial x_m^2} & \frac{\partial^2 E}{\partial x_m \partial y_m} \\ \frac{\partial^2 E}{\partial y_m \partial x_m} & \frac{\partial^2 E}{\partial y_m^2} \end{bmatrix} \cdot \begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} -\frac{\partial E}{\partial x_m} \\ -\frac{\partial E}{\partial y_m} \end{bmatrix}$$

$$\frac{\partial^2 E}{\partial x_m^2} = \sum_{i=1}^{n} k_{mi} \left(1 - \frac{l_{mi}(y_m - y_i)^2}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}} \right)$$

$$\frac{\partial^2 E}{\partial x_m \partial y_m} = \frac{\partial^2 E}{\partial y_m \partial x_m} = \sum_{i=1}^{n} k_{mi} \left(\frac{l_{mi}(x_m - x_i)(y_m - y_i)}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}} \right)$$

$$\frac{\partial^2 E}{\partial y_m^2} = \sum_{i=1}^{n} k_{mi} \left(1 - \frac{l_{mi}(x_m - x_i)^2}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}} \right)$$

In order to solve the above system of two linear equations, let us use *Cramer's rule*:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix}$$

$$x = \frac{ed - bf}{ad - bc}, y = \frac{af - ec}{ad - bc}$$

Finally, the pseudo-code for *Kamada-Kawai* method can be written as in Algorithm 3.

Algorithm 3. *Kamada-Kawai* [9] embedding algorithm.

KAMADAKAWAI($G(V,E), disp, K_0$)
{initial random positions of nodes, target displacement of nodes , stiffness factor}
1 **let** $D = \{d_{ij}\} \leftarrow$FLOYDWARSHALL($G(V,E)$) {shortest paths}
2 **let** $L = \{l_{ij} \leftarrow disp \cdot d_{ij}\}$ {equilibrium lengths}
3 **let** $K = \{k_{ij} \leftarrow K_0 / d_{ij}^2\}$ {stiffness factors of springs}
4 **while** $max_{i=1}^{n}(|\nabla E_i|) > \varepsilon$ **do**
5 **let** $\nabla E_m \leftarrow \nabla E_j : |\nabla E_j| = \max_{i=1}^{n}(|\nabla E_i|), j = 1, \dots, n$
6 **let** $v \leftarrow v_i \in V : i = m, i = 1, \dots, n$
7 **while** $|\nabla E_m| > \varepsilon$ **do**
{solve the linear system $J_{\nabla E_m} \cdot \begin{pmatrix} dx \\ dy \end{pmatrix} = -\nabla E_m$ }
8 **let** $J_{\nabla E_m} \leftarrow$ JACOBIAN(∇E_m)
9 **let** $\begin{pmatrix} dx \\ dy \end{pmatrix} \leftarrow$ CRAMER ($J_{\nabla E_m}, -\nabla E_m$)
10 $v.x \leftarrow v.x + dx$
11 $v.y \leftarrow v.y + dy$
12 **return** $G(V,E)$

Function E is in its local minimum when all of its first partial derivatives are equal to zero. Algorithm 3 approximates this target by gradually decreasing greatest ∇E_i gradient vectors, one at a time, until all of them have its elements (first partial derivatives) sufficiently close to the zero. Apparently, partial derivative can be computed in $O(|V|)$. Outer loop calculates $|V|$ gradients each consisting of 2 partial derivatives, which leads to $O(|V|^2)$. Inner loop calculates 4 derivatives to produce Jacobian and 2 derivatives in order to update ∇E_m. Since T representing the number of inner loop iterations (Newton-Raphson method) depends un-trivially on the node count, node positions and graph structure, resulting cost of the inner loop would be $O(T|V|)$. It should be noted that, when several conditions hold, Newton-Raphson method is proven in [20] to converge quadratically to the zero of the given function (the number of significant digits doubles after each iteration). However, convergence rate can be more than quadratic or might even fail when those condition are not met (e.g. the initial picked value is too far from the actual zero). Thus, GraphRec puts the upper limit on the number of inner loop executions in order to prevent lock-ups. This effectively means that the overall cost of the inner loop is only $O(|V|)$. Hence the complexity of a single iteration of the outer loop is $O(|V|^2)$.

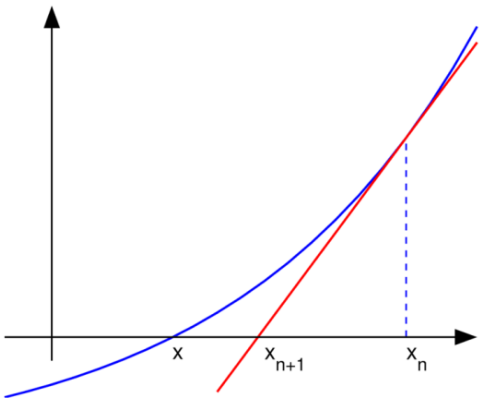


Figure 4. *Newton-Raphson* method.

2.1.3 Methods Comparison

Methods can be compared regarding both their speed and quality of produced layout. As Figure 5 indicates, Kamada-Kawai method produces visually more appealing layout. In fact, such result is expected. Whereas Fruchterman-Reingold method models ideal distances only between adjacent nodes, Kamada-Kawai takes into account all pairs of nodes. As for execution speed, Kamada-Kawai needs larger number of iterations to find equilibrium and is therefore slower despite the lower complexity of a single iteration.

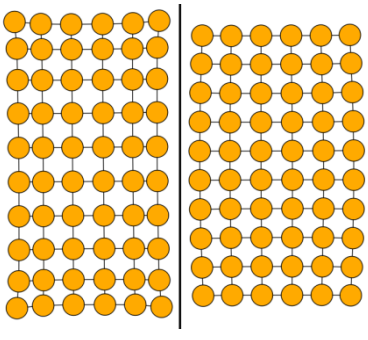


Figure 5. *FR* (left), *KK* (right).

2.2 Problem Variants

Generally, a problem of motion on a graph is a task to find a sequence of entity movements that transforms given initial placement of entities into the requested target placement. As described in [17], there are **two interesting variants** of the problem varying in the number of constraints applied to a single movement. Let us take problem definitions from [17] and [18] to formalize how the supposed solutions of these problems look like.

Definition 1 (*solution of pebble motion on a graph*). Let $G = (V, E)$ be an undirected graph. Let $R = \{\overline{r_1}, \overline{r_2}, \dots, \overline{r_n}\}$ be a set of entities (denoted by constant symbols) such that $|R| < |V|$. Any given *solution of pebble motion on a graph* for a set of entities R is a sequence $(F_R^t)_{t=0}^m$, where $F_R^t: R \to V$ is a uniquely invertible function for every time step $t \in \{0,1,\dots,m\}$. Either $F_R^t(r) = F_R^{t+1}(r)$ or $\{F_R^t(r), F_R^{t+1}(r)\} \in E$ must hold for every $r \in R$ and every $t \in \{0,1,\dots,m-1\}$. A movement of the given entity $r \in R$ at the given time step $t \in \{0,1,\dots,m-1\}$ is *allowed* if and only if the statement $F_R^t(r) \neq F_R^{t+1}(r) \Longrightarrow \forall s \in R: F_R^t(s) \neq F_R^{t+1}(r)$ holds. All the movements of the solution must be *allowed*. □

Definition 2 (*solution of multi-robot path planning*). Let $G = (V, E)$ be an undirected graph. Let $R = \{\overline{r_1}, \overline{r_2}, \dots, \overline{r_n}\}$ be a set of entities (denoted by constant symbols) such that $|R| < |V|$. Any given *solution of multi-robot path planning* for a set of entities R is a sequence $(F_R^t)_{t=0}^m$, where $F_R^t: R \to V$ is a uniquely invertible function for every time step $t \in \{0,1,\dots,m\}$. Either $F_R^t(r) = F_R^{t+1}(r)$ or $\{F_R^t(r), F_R^{t+1}(r)\} \in E$ must hold for every $r \in R$ and every $t \in \{0,1,\dots,m-1\}$. A movement of the given entity $r \in R$ at the given time step $t \in \{0,1,\dots,m-1\}$ is *allowed* if and only if one of the following statements holds:

(i) $F_R^t(r) \neq F_R^{t+1}(r) \Longrightarrow \forall s \in R: F_R^t(s) \neq F_R^{t+1}(r)$

(ii) $F_R^t(r) \neq F_R^{t+1}(r) \Longrightarrow \exists s \in R: s \neq r \wedge F_R^t(s) = F_R^{t+1}(r) \wedge F_R^t(s) \neq F_R^{t+1}(s)$, where the movement of the entity s at the time step t must be *allowed*.

All the movements of the solution must be *allowed*. □

Because every time step in a solution is represented by a uniquely invertible function, it is ensured that any given node is occupied by a single entity at any given time. The main difference between the two variants is that *multi-robot path planning* allows the situation in which the node is both the source and the target for two simultaneous movements.

2.2.1 Solution Validation

In practice, a solution is a sequence of an arbitrary number of movements. Every move is specified only by its **source node**, **target node** and a **time step** at which occurs. Notice there is no information about the entity. Because it is not guaranteed that input data are completely correct, there is a need for validation. Depending on a character of input data, movements are validated against either Definition 1 or Definition 2. Process is described in the following list:

1) All moves are sorted by the time step.
2) Moves are split into groups that are characterized by the same time step. Each group is then filtered. Moves that are **discarded** meet one of these criteria:
    a) The source node is equal to the target node, which effectively results in a loop.
    b) There is no edge between the source and the target node.
    c) The source node does not contain an entity in the respective time step.
    d) There is already different approved move that begins in the same source node as the currently examined move.
    e) There is already different approved move that ends in the same target node as the currently examined move.

f) There is already different approved node that begins in the target node and ends in the source node of the currently examined move (inverse move).

3) Second pass through every filtered group determines the final valid moves. Moves that are ultimately **approved** meet one of these criteria:
   a) The source node contains an entity and the target node is empty.
   b) (**only** *multi-robot path planning*) Both the source node and the target node contain an entity. Recursive search proves that the target node is freed by some other move occurring in the same time step and that the whole chain of such moves is terminated by an empty node.

2.3 Movement Animation

The **animation of moving entities** is the core feature of the application. Since the solution is built over discrete time steps, these should be possible to play through or even step through in order to increase controllability of the observation. When examining certain part of the solution it is also necessary to provide adjustable speed of the animation and the possibility to jump quickly between various time steps.

Animation of the solution can be controlled in a similar way as playing a movie on a video recorder. Firstly, user adjusts the animation speed and specifies the starting time step. Then, it is possible to play or step through the animation time line. GraphRec supports the synchronized animation of more than one solution at once, which is for example useful when comparing differently optimized solutions for the same problem.

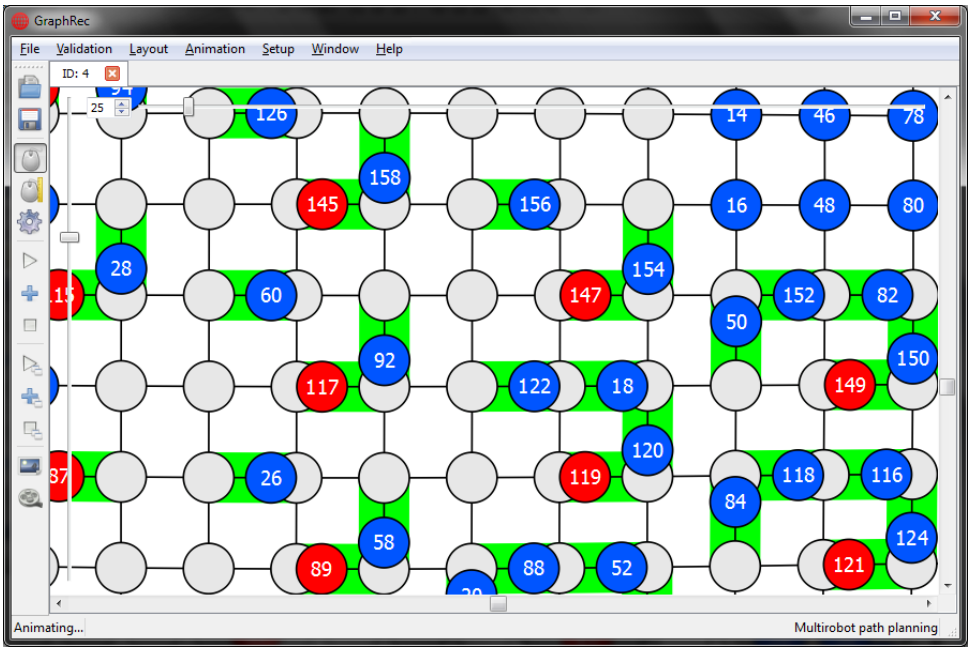


Figure 6. *Moving entities* emphasized by highlighted edges.

2.4 Color Management

The clearness of the animation must be taken into attention as well. It appears that *highlighting* of moving entities greatly improves the overall perception of where the motion actually occurs (Figure 6). The demand for user vigilance might be further reduced by *distinguishing* between entities that are already in their final positions and that are not.

Consequently, the tool enables all graph elements to be assigned with various colors. This is especially important in scenarios such as observation of the movement of one particular entity or even group of entities, where **color differentiation** considerably simplifies their traceability.

2.5 Chosen Technologies

From the beginning of the development, GraphRec was intended to be **multiplatform open-source software**. In order to achieve these properties, implementation is based on **Qt cross-platform application and GUI framework** [12] developed by Nokia Corporation. Since Qt itself is implemented in C++, the natural choice of a programming language was C++ along with its standard library. Because Qt framework offers very good data structures and some advanced abstractions, the usage of C++ standard library was reduced to minimum (e.g. numeric operations).

Not only does Qt serve as a bridge among various platforms, but also heavily simplifies the design of a GUI application. In particular, GraphRec takes advantage of the *signal and slot* mechanism, window layouts, graphical subsystem, support for raster and vector image formats, concurrency, XML[1] API and regular expressions. The only thing that is not covered by Qt is video rendering. In order to support capturing of video, GraphRec uses **FFmpeg video library** [1], which is both multiplatform and open-source while providing wide range of popular video encoders.

Program was developed under Qt Creator, which is a multiplatform IDE specially designed for the development of Qt applications. Both Qt framework and FFmpeg library are licensed under either GNU GPL[2] or GNU LGPL[3]. Since these licenses are *viral*, GraphRec must be licensed under these or less restrictive licenses. Currently, GraphRec is **English-only** application. However, since Qt provides very good support for localization, every string for GUI is wrapped in a translation function. This is a good starting point for any possible localization in the future.

2.6 Operating Environment

Because of chosen technologies, GraphRec can be compiled for Windows, Linux and Mac OS from the single source code. As will be explained later, it is possible to create **compressed and statically linked binary**, which is almost completely self-contained not expecting any non-standard preinstalled libraries on the host system. Since GraphRec consists of a single binary file, it can be simply installed and run from anywhere. Depending on the platform, configuration is stored, independently on the binary location, in registry (Windows) or in conf files (Linux).

In practice, GraphRec was tested and is distributed under Windows XP SP3 and above or Ubuntu 8.10 and above. In addition to the requirements of mentioned operating systems, the minimal environment for GraphRec execution should include a processor with MMX[4] and SSE[5] support, 150MB of free random access memory (when encoding video), colored display, classic keyboard and a mouse with the roller.

[1] eXtensible Markup Language (http://www.w3.org/XML/)
[2] GNU General Public License (http://www.gnu.org/licenses/gpl.html)
[3] GNU Lesser General Public License (http://www.gnu.org/licenses/lgpl.html)
[4] MultiMedia eXtensions
[5] Streaming SIMD Extensions

2.7 Extensibility

Because of the intended open-source nature of the application, one of the design principles was to produce easily extensible and maintainable code. As a prerequisite for such aim, code was written to be self-explaining, well formatted and easily readable without the need for distracting comments. A certain subset of *Hungarian notation* was used for naming conventions. However, the extensibility must be primarily reflected in the architecture. Opportunities for extensions are obvious – more input or output file formats, various graph embedding algorithms, different solution validation schemes. All those modules can be divided into groups that are characterized by the same functionality and data but by the different algorithms. Such situation can be abstracted by the well-known *Strategy pattern* described in [7] and shown in Diagram 1. Strategy pattern is the main design principle of the GraphRec architecture.

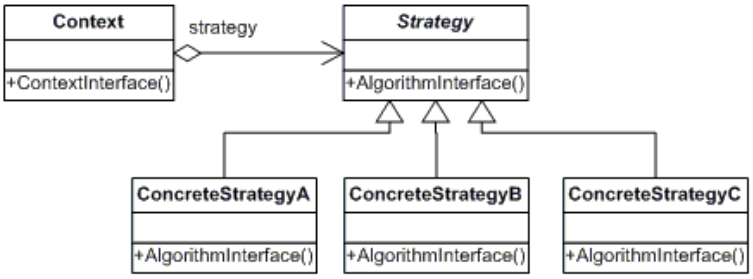


Diagram 1. *Strategy pattern* UML scheme.

2.8 User Interface

The design of the graphical user interface was made to be highly flexible, simple and intuitive. Since there is a natural requirement to compare solutions among themselves, the possibility to work with arbitrary number of them is essential. GraphRec GUI is built primarily around this requirement. As a result, GUI is equipped with **tabbed interface**, where each independent tab contains a context of the single solution (Figure 7). These tabs can be manipulated in such a way that more of them are visible at the same time. Tabs are completely insulated and orthogonal with the only exception of the video encoding feature, which is exclusively assigned to a single tab at a time.

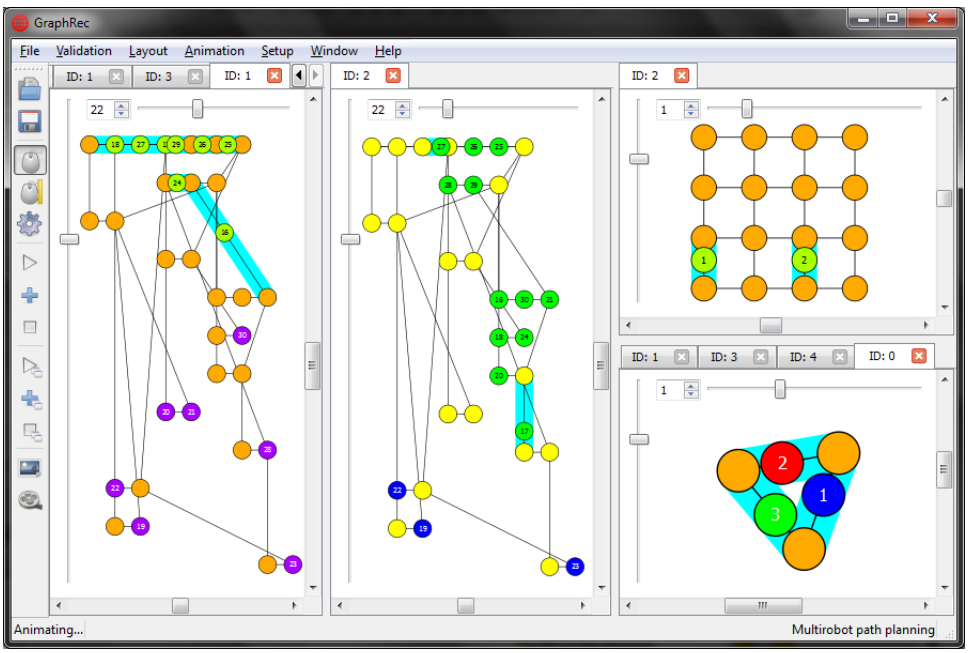


Figure 7. Extensive usage of *tabbed user interface*.

Following design principles are also reflected in the GUI of Graphrec:

- Main window and dialogs are resizable.
- Window controls adapt to various sizes of their parents.
- Frequently used controls are pinned to the toolbar.
- Dialogs are non-modal if possible.
- It is possible to terminate long operations.
- User interface stays responsive even during extensive calculations.
- User is informed by status messages or progress bars about what is happening.
- When there is more similar objects (e.g. files, nodes, list entries), it must be possible to work with more of them at once.

2.9 Video Capturing

GraphRec uses concurrency while encoding video. Rendering and encoding is done on different threads. Thus, rendering function acts as a producer, who puts video frames into the buffer of a limited size, and encoding function acts as a consumer, who reads frames from the buffer and encodes them into an output file. Both threads are synchronized by the two semaphores. The function for acquiring semaphore normally blocks until resources are available. Since GraphRec must stay responsive during video encoding, the semaphore acquiring is periodically requested only for a small amount of time after which the application message queue is inspected. General scheme for *producer-consumer synchronization* is described in [1] and in Algorithm 4.

Algorithm 4. *Producer-Consumer* [1] synchronization scheme.

```
SYNCHRONIZE(size, timeout)
1  let buffer ← ALLOCATEBUFFER(size)
2  let free ← CREATESEMAPHORE(size)
3  let used ← CREATESEMAPHORE(0)

4  let PRODUCE ← λ(  while TRYACQUIRE(free, timeout) = FALSE do
5                        PROCESSMESSAGES
6                    WRITEDATA(buffer)
7                    RELEASE(used)
8                 )

9  let CONSUME ← λ(  while TRYACQUIRE(used, timeout) = FALSE do
10                       PROCESSMESSAGES
11                   READDATA(buffer)
12                   RELEASE(free)
13                )

14 let producer ← CREATETHREAD(PRODUCE)
15 let consumer ← CREATETHREAD(CONSUME)
16 STARTTHREAD(producer)
17 STARTTHREAD(consumer)
18 WAITTHREAD(producer)
19 WAITTHREAD (consumer)
```

3 User's Guide

Guide is organized in the following way. Firstly, the opening of the input file is explained. Immediately after that, the environment is introduced in the descriptive linear manner. Then, the description of mouse and keyboard controls follows. Ultimately, there are several user stories on how to achieve certain goals.

3.1 Opening Input File

Input file contains graph definition, positions of nodes, colors of graph elements and some additional information. File can be opened either by toolbar button or by **File – Open...** in the main menu. This action opens dialog containing two lists (Figure 8). List on the left side must be first filled with input files before proceeding further. To do that, just click on **Add files...** button and choose one or more input files in the standard open dialog. Note that you must choose appropriate file format. After the files are searched, left list contains one line for each graph found in the input files. Each line provides some additional information for a given graph:

- **Name** – Each graph in the input file should be marked by its ID number. If ID is not specified, dash is displayed instead.
- **Line** – Line at which the graph definition starts in the respective file. This can be useful information for the manual revision of large input files.
- **Nodes** – Number of nodes in the graph.
- **Edges** – Number of edges in the graph.
- **Entities** – Number of entities that occupy nodes.
- **Timesteps** – Number of time steps the solution scenario consists of.
- **Movements** – Number of entity movements in the whole solution scenario.
- **Validator** –Shows either the default validator or the one preferred by the input file. Selected validator will be used for the initial validation. Validator can be changed by right clicking on the respective line. Note that validator can be also changed later in the main menu.

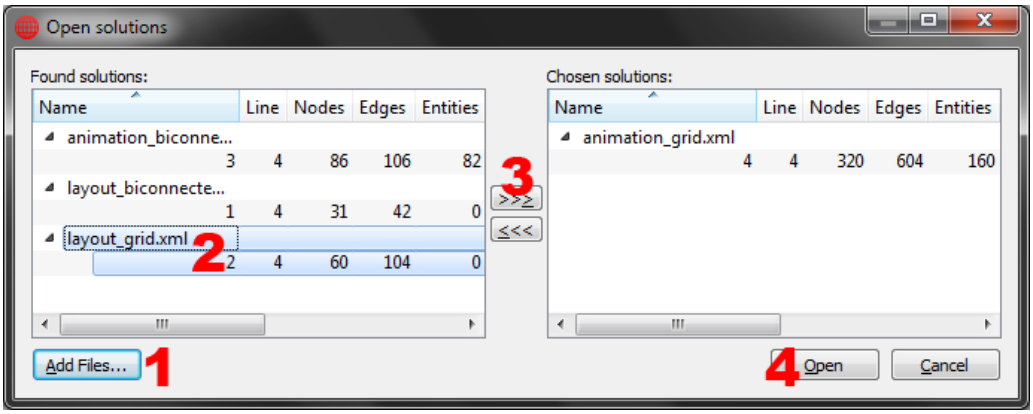


Figure 8. *Open dialog.*

List can be sorted by each of these columns. Irrelevant files can be folded in order to save space. Lines can be multi selected (by mouse dragging or by holding *Ctrl/Shift* and left mouse clicking) and then moved to the list on the right side (by >>> button or by double-clicking). Note that if you select a header containing the file name, all graphs specified by this file will

be moved to the right list. List on the right side serves as a basket for graphs you are willing to open. Graphs can be moved between the two lists until you are satisfied with the selection. To confirm dialog just click **Open** button and wait until selected graphs are loaded.

3.2 Environment Description

Following subsections describes the main application window and its parts.

3.2.1 Main Menu

Menu consists of several sections:

- **File** menu contains actions related to the file handling.
  - **Open...** shows advanced dialog for choosing input files.
  - **Save...** shows standard save dialog for saving single file.
  - **Save All...** shows save dialog for saving all opened tabs into a single file.
  - **Exit** terminates the application.
- **Validation** menu contains actions related to the solution validation.
  - **Validator** submenu shows all available validators. Currently selected validator is ticked.
  - **Log...** shows dialog for error logging and reviewing.
- **Layout** menu contains actions related to the graph embedding.
  - **Mode** submenu shows a list of embedding modes.
    - **Manual (continuous)** grants the full manual control over the positions of nodes.
    - **Manual (discrete)** shows the background grid that defines separation granularity.
    - **Automatic** enables selected layouter and its embedding algorithm.
  - **Layouter** submenu shows all available layouters. Currently selected layouter is ticked.
  - **Recalculate** move all nodes to the random positions and enables currently selected layouter. Available only in the *Automatic* mode.
  - **Snap to Grid** aligns positions of all nodes to the closest valid place in the grid. Available only in the *Manual (discrete)* mode.
- **Animation** menu contains actions related to the animation control and capturing. Almost all actions in this menu appear in two versions – one for the currently selected tab and one for all foreground (currently visible) tabs. Following list will describe only single tab actions.
  - **Play** starts the animation from the currently selected time step.
  - **Step** animates only the currently selected time step and then stops.
  - **Stop** finishes the currently animated time step and then stops animation.
  - **Seek...** shows dialog for the time step selection.
  - **Reset** sets the scenario to its beginning.
  - **Synchronize All** enables or disables time step synchronization between the tabs when playing animation on more of them at once.
  - **Snapshot...** shows the dialog for capturing images.
  - **Sequence...** shows the dialog for capturing image sequences or videos.
- **Setup** menu contains actions related to the configuration.
  - **Colors...** shows the dialog for coloring graph elements.
  - **Options...** shows the main configuration dialog.
- **Window** menu contains actions related to the graphical user interface.

    - **Split Horizontally** splits the current tab group horizontally.
    - **Split Vertically** splits current tab vertically.
    - **Unsplit** recursively joins two tab groups on the selected level of the hierarchy.
    - **Unsplit All** removes all splits.
    - **Toolbar** button enables or disables a visibility of the application tool bar.
    - **Controls** button enables or disables a visibility of the additional controls placed inside the tabs.
- **Help** menu contains actions related to the documentation and application information.
    - **Documentation...** shows a simple browser containing this guide.
    - **About...** shows the about box containing version, contact and license information for GraphRec.
    - **About Qt...** shows about box containing version, contact and license information for Qt framework (used by GraphRec).
    - **About FFmpeg...** shows about box containing version, contact and license information for FFmpeg video library (used by GraphRec).

3.2.2 Tool Bar

Application tool bar contains frequently used actions from the main menu (Figure 9). These actions are (in order of appearance) **Open...**, **Save...**, **Manual (continuous)**, **Manual (discrete)**, **Automatic**, **Play**, **Step**, **Stop**, **Play All**, **Step All**, **Stop All**, **Snapshot...**, **Sequence...**. Note that currently there is no possibility to add or remove actions from the tool bar. Tool bar can be enabled or disabled by **Window – Toolbar** in the main menu. It can be also moved over the screen or docked to arbitrary side of the application window.

Figure 9. Application *tool bar*.

3.2.3 Status Bar

Left side of the status bar contains additional information about current activity of the application. Messages can indicate idle state, file opening, file saving, validation, error detection, graph embedding, running animation, video capturing and image saving. Note that some of these messages can be displayed only for a limited amount of time. Right side of the status bar shows a label of the currently selected validator.

3.2.4 Tabs

Each opened solution is contained in its exclusive tab. Tab header contains the close button and ID number of the solution. Context help of each tab contains the name of its source file. Order of the tabs in one tab group can be changed by mouse dragging. Note that it is not currently possible to drag and drop the tab from one tab group to another tab group. Tab groups are described in the next section. Each tab contains a horizontal slider and a spin box, whose values specify the current time step of the solution scenario. In fact, horizontal slider should be considered as a time line, similarly as in audio or video players. There are also two vertical sliders on the left. The upper one is intended for adjusting the speed of the animation. The lower one specifies the overall displacement of nodes – the value that is used as a hint by the active layouter.

3.2.5 Window Splitting

In order to provide a way to work with more tabs at a time, it is possible to split the initial set of tabs into more groups. The tab group must contain at least two tabs; otherwise, it is not possible to split it. Splitting can be stacked horizontally (**Window – Split Horizontally**) or vertically (**Window – Split Vertically**). Before splitting, it is necessary to select the tab that will be the first one in a new group. Consequently, splitting is not possible if you select the first tab – new tab group would be the same as the current one. After the group is split, you are free to do further sub splitting, which effectively results in a binary tree hierarchy. Window space for each tab group can be adjusted by dragging dividers between them. When there is more than one tab visible, user interface still controls only single tab - the one that has focus. To switch focus between tabs, you just click anywhere into their window area. Active tab is always emphasized by the surrounding frame. There is also an inverse operation to splitting. Tab groups can be rejoined by **Window – Unsplit**. Before unsplitting, it is necessary to give focus to any tab in one of the two tab groups you want to join. Since any of these two tab groups can be already sub split, it is recursively joined to the bottom of the tree hierarchy. In order to rejoin all tab groups into a single flat tab group just click on **Window – Unsplit All**.

3.3 Controls

This section provides the description of mouse and keyboard usage, which is not obvious at the first sight.

Mouse

- **Move** nodes in a graph by clicking on them and dragging them while holding the left mouse button. Not working while the animation is running.
- **Scroll** the graph view by clicking anywhere into the free space and then by dragging the view while holding the left mouse button.
- **Zoom** the graph view by rolling the mouse wheel.
- **Select** nodes by holding *Ctrl* key while clicking on them or while dragging selection frame over them (left mouse button). In order to move selected group of nodes, *Ctrl* key must be held during the operation otherwise nodes will be unselected.

Keyboard

- **Scroll** the graph view by arrow keys.
- **Move** the graph by pressing *IKJL* keys (not allowed in **Automatic** embedding mode).
- **Rotate** the graph by pressing *AD* keys (not allowed in **Automatic** embedding mode).
- **Zoom** the graph view by pressing *WS* keys.
- **Randomize** positions of nodes by pressing *R* key.

3.4 Validating Solution

Each solution scenario is validated by the selected validator during a file loading. Validator checks whether all entity movements are valid and non-conflicting. All incorrect movements are logged so the animation engine would ignore them. If there are any errors, the notification will appear in the status bar. GraphRec currently provides two validators:

- **Pebble** validator for the *pebble motion on a graph* (Definition 1). The movement is valid if and only if there is an edge between the source and target node, if there is an entity in the source node and if the target node is empty in the time step just before the movement.

- **Multirobot** validator for the *multi-robot path planning* (Definition 2). The movement is valid if and only if there is an edge between the source and target node, if there is an entity in the source node and if the target node is either already empty or freed in the same time step as the movement occurs. This permits chain movements of multiple entities.

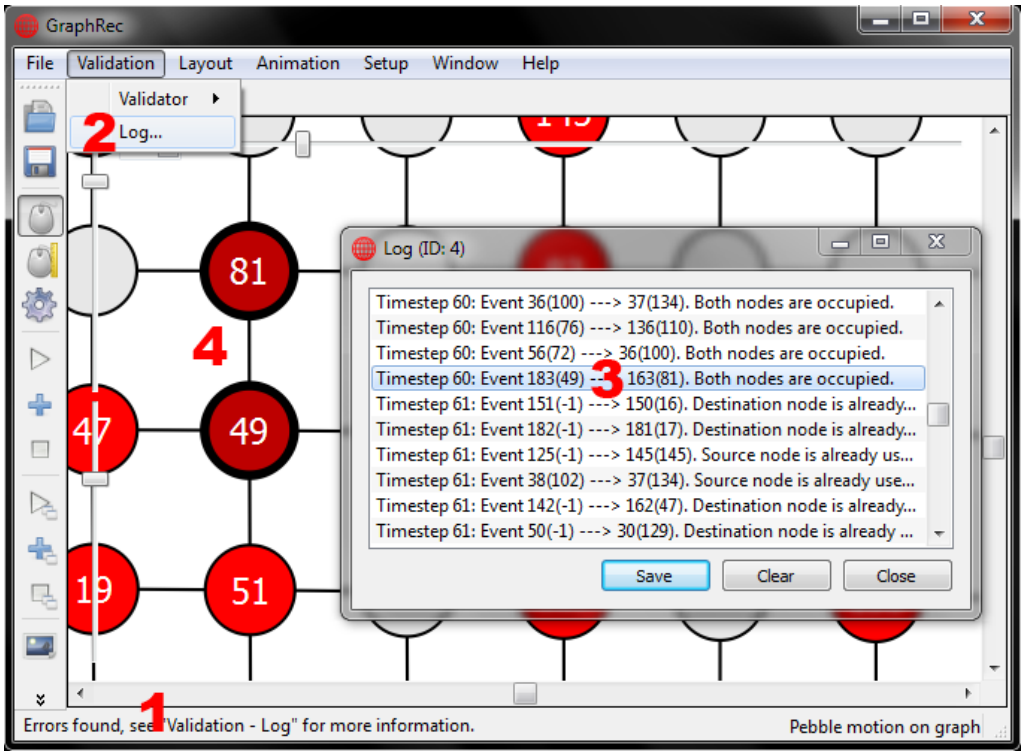


Figure 10. Reviewing *error log*.

Validator can be changed by the **Validation – Validator** menu or by **Setup – Options...**. Errors can be reviewed in a special dialog accessible through the **Validation – Log...** menu action. Whole log can be saved into a text file. Each line in the error log specifies a single problem. It consists of the time step specification, movement details and error description. Movement details are logged in the format `SN(E1) ---> DN(E2)`, where `SN` stands for *source node ID*, `DN` for *destination node ID* and both `E1` and `E2` are ID numbers of the entities that occupy respective nodes before the invalid movement. Note that entity IDs that are less or equal zero are reserved for empty nodes. Errors can be visually tracked down by clicking on log entries – corresponding time step is set and affected nodes are highlighted (Figure 10).

3.5 Embedding Graph

Generally, initial positions of the graph nodes are random. However, if the graph is biconnected and the input file specifies its circles, nodes are aligned into lines that correspond to those circles. In most cases, the initial layout should be further modified. Usually, the first step to improve layout is to enable automatic layouter, which will iteratively modify the layout by its embedding algorithm. Layouter can be chosen from the **Layout – Layouter** menu or by **Setup – Options...**. To enable selected layouter, toggle **Layout – Mode – Automatic**. GraphRec currently provides two layouters:

- **Fruchterman-Reingold** layouter belongs to the group of force-directed layouters. Nodes behave as repulsive particles whereas edges behave like springs. Since main

principle is based on physical laws, layouter is very well predictable and intuitive. Layout is elastic and can be influenced by dragging nodes by the mouse during embedding.

- **Kamada-Kawai** layouter is also force-directed. One at a time, nodes are hooked by springs to the anchored rest and gradually moved to the equilibrium. Since nodes are not updated all at once, layouter is less predictable and intuitive. Layout cannot be influenced during embedding. Generally, Kamada-Kawai is slower but provides slightly better layout.

Automatic layouters have usually more than one local minimum at which layout is considered done. If the current local minimum is not good enough, it helps to drag nodes to its desired positions while the layouter is still enabled. This action will usually result in finding another, probably better, local minimum. It is also possible to start again with the random layout either by pressing *R* key or by clicking **Layout – Recalculate**. Another way to interact with the layouter during its work is to adjust the node displacement dynamically by the vertical slider on the lower left side of the graph view (Figure 11). Sliding the displacement up and down for a few times may remove little imperfections in the layout, especially in the case of grid graphs.

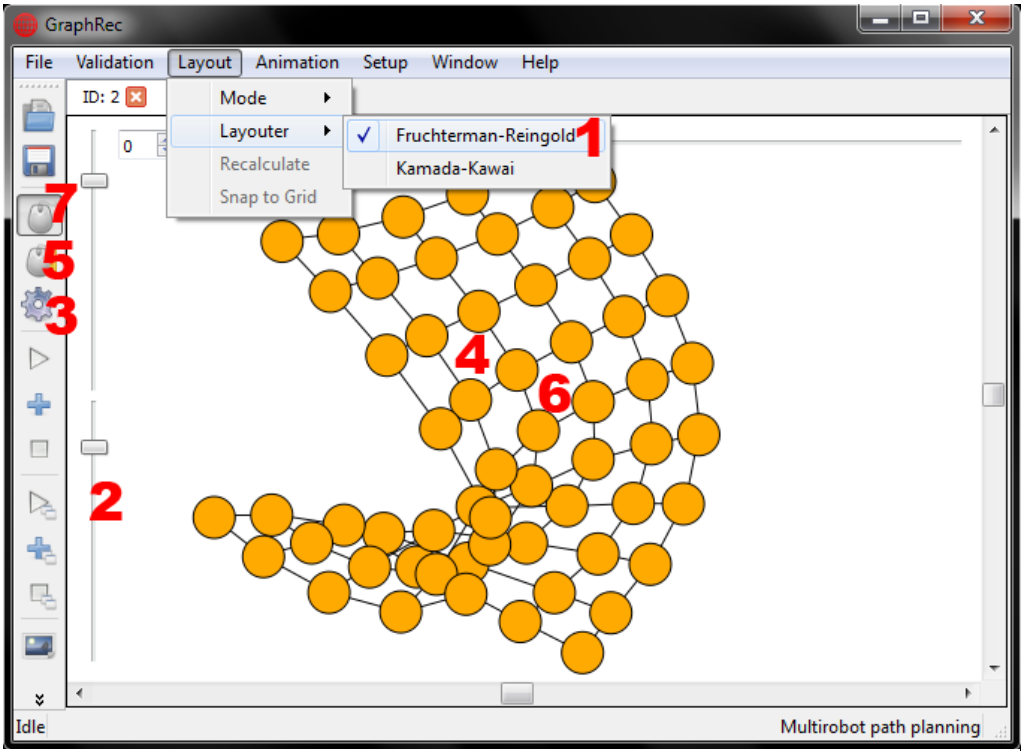


Figure 11. Managing *graph layout*.

After the automatic layout is finished, it is recommended to switch to either **Layout – Mode – Manual (continuous)** or **Layout – Mode – Manual (discrete)** in order to refine the layout further. At this point, graph will probably need some zooming, scrolling, rotating and node moving. Whereas a continuous mode gives no restrictions on node positioning, discrete mode allows only positions defined by its background grid. Granularity of the background grid can be changed in **Setup – Options...**. Discrete mode is intended for layouts that need exact orthogonality for the esthetic purposes. Before aligning nodes manually to the grid, you may find useful **Layout – Snap to Grid**, which aligns all nodes to the closest valid position in the grid.

3.6 Settings

Configuration can be changed in the dialog accessible through **Setup – Options...** in the main menu. Note that settings are propagated only to the tab that has focus. If you want to change settings globally, hit the **Save Default** button, close/save all tabs and reload them. Explanation of the configuration items follows:

Animation

- **Length** of a single time step in milliseconds.
- **Style** of the animation. Whereas *Linear* animation has the same speed all the time, *EasyInOut* animation starts slowly, then goes steady and finish slowly again.

Layout

- **Layouting mode** determines whether the layout will be done automatically or not.
- **Layouter** shows the list of available layouters.
- **Node displacement** serves as a hint for layouter. This entry should be used only for exact value specification. For approximate alteration of the node displacement, it is better to use vertical slider on the lower left side of the graph view.
- **Grid offset** defines the granularity of the background grid, which is visible while the manual discrete mode is enabled.

Validation

- **Validator** shows the list of available validators.

Visual

- **Node description** determines what should be displayed on the node surface. Node labels can be either empty or can contain various combinations of entity and node identification numbers.
- **Controls visibility** enables or disables sliders and spin box in the graph view.
- **Interactive capturing** determines whether the user is able to interact (e.g. zooming, scrolling) with the graph view during video or image sequence capturing. If disabled, capturing is indicated only by a progress bar and graph view is not rendered in order to speed up video encoding.

3.7 Coloring Graph Elements

Graph colors can be adjusted in a dialog displayed after clicking on the **Setup – Colors...** menu entry (Figure 12). First, choose a type of the graph element you would like to color from the tabbed panel on the left. Then select one or multiple items from the list. List items can be multi selected by dragging the mouse or by holding *Ctrl/Shift* and left mouse clicking. To select all items hit the **Select All** button. Since the dialog is not modal, it is also possible to switch back to the main window and select graph nodes by holding *Ctrl* together with the left mouse button while dragging mouse over them – the selection will be propagated into the color dialog. List can be sorted by any column, which for example makes it easier to select all items of the same color. After selecting the desired set of items, click on the **Set Color** button and choose what color you would like to change. Note that this can be also achieved by the right mouse click anywhere into selected set of items. After clicking on the appropriate color type, you can choose exact color in a standard dialog for color picking. Both nodes and entities have their background and foreground color. Because foreground color is used for labels displayed on nodes, it should be in the contrast with the background color. Moreover, entities have another pair of colors used for their final state. Final state of the entity is reached after its last movement in the solution scenario.

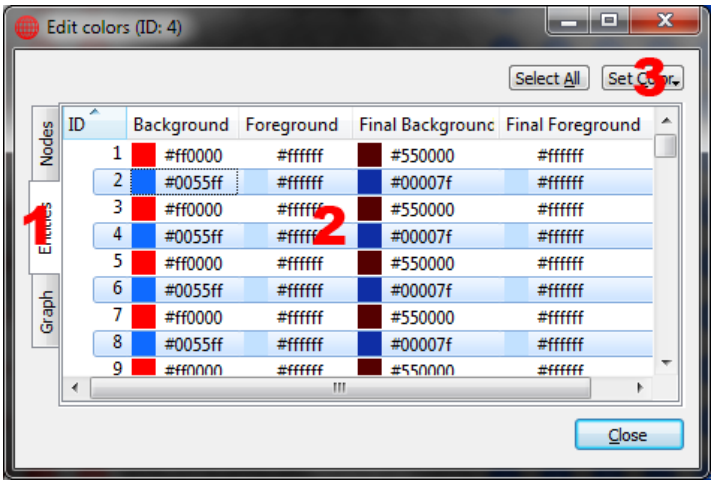


Figure 12. *Adjusting colors* of graph elements.

Special explanation should be given to the third tab with the *Graph* label. List in the third tab is intended for colors that are more general:

- **Edge highlight** is a color used for the wide line rendered under all moving entities. This color should be bright with a very strong contrast against the graph background.
- **Graph background** is a color used for the background of the whole graph view.
- **Outlines and edges** is a color used for all lines that represent edges and node boundaries. This color should be in very strong contrast with the graph background.

3.8 Saving Output File

Current tab can be saved into the file by clicking on the **File – Save...** menu entry and choosing appropriate file format. Information that will be saved includes graph definition, solution scenario, node positions, coloring, current view and chosen validator. If you want to save all opened tabs into a single file at once just click on the **File – Save All...** menu entry.

3.9 Controlling Animation

Actions for controlling the animation are all accessible through **Animation** menu. Frequently used actions are also pinned to the tool bar. All actions are divided into the two groups – the first group is intended for controlling the tab that has currently focus (e.g. **Play**) whereas the second group controls all foreground tabs (e.g. **Play All**).

Whole solution scenario consists of several time steps that can be played through (**Play**/**Play All**) or stepped through (**Step**/**Step All**). If you want to play the animation from one particular time step, click on the **Seek...**/**Seek All...** menu entry and specify it. Other possibility is either to adjust the horizontal slider on the top side of the graph view or to enter the exact time step into the spin box located in the top left corner of the graph view. Animation speed can be dynamically adjusted by the vertical slider on the upper left side of the graph view. Animation can be stopped any time by clicking on the **Stop**/**Stop All** button. Just note that the currently animated time step have to be finished so that if the animation length was set too long, this might take a while. In order to rewind the whole scenario back to the beginning, set time step to zero or click **Reset**/**Reset All** button.

Running more parallel animations by **Play All** can be done in two modes by toggling **Synchronize All**:

- **Synchronized** animation is suitable for the tabs that have equal animation durations. Synchronization ensures that the tabs are timed collectively. Note that when synchro-

nizing tabs with different animation durations, all tabs must wait for the slowest one every time step.

- **Non-synchronized** animation is suitable for the tabs that have different animation durations. In this mode, tabs are timed independently. This implies that two parallel animations with the same duration might get slightly desynchronized after a few time steps.

3.10 Capturing Media Files

GraphRec provides two ways to capture media files. There is a possibility to take raster and vector screenshots of the graph view through the **Animation – Snapshot...** dialog or to capture image sequences and video through the **Animation – Sequence...** dialog. Dialog is almost the same for both actions. The only difference is that in the sequence mode it is possible to select video encoder and some additional settings. Dialog items are explained in the following list:

- **Encoder** shows a list of available media encoders. When the encoder is selected, its settings are shown on the right side of the dialog. Specifics of these encoders are described in following sections.
- **Width** of the resulting image or video (in pixels).
- **Height** of the resulting image or video (in pixels).
- **Start** specifies the time step when the capturing begins. (available only in the sequence mode)
- **Stop** specifies the time step when the capturing stops (resulting media will not include this time step). (available only in sequence mode)
- **Interval** determines the number of time steps that will be skipped between two captured images in the sequence mode. For example if the interval is set to 3, only every fourth time step will be captured. (available only in the sequence mode for non-video encoders)
- **Name** of the resulting media file. If the location already contains file with such name, the specified name will be automatically extended with the numeral in order not to overwrite any data.
- **Path** to the destination directory for media files. Be sure there is enough free space when capturing the video.

If all settings are set, capturing can be started by hitting the **Capture** button. What happens next depends on the settings. In the snapshot mode, image is simply saved and notification is displayed in the status bar. The dialog is not closed and thus ready to take as many images as needed. Note that in the snapshot mode, the dialog is modal so you are free to interact with the rest of the application. This is convenient for taking images during the animation. It is suggested to increase animation length so it would be easier to take the image at the intended moment.

Sequence mode behaves differently. The dialog is not modal so it is not possible to interact with the rest of the application. After the **Capture** button is clicked, the dialog is closed and the encoding starts running from the specified starting time step until manually stopped or the stopping time step is reached. If the capturing was set to be interactive in the settings, it is possible to interact with the graph view during encoding (e.g. zooming, scrolling). Note however that while capturing interactively, the animation speed depends on the speed of processor, which means that it does not match real time. If the interactive capturing is disabled, the

graph view is not rendered in order to increase performance. In this case, only simple progress bar informs about the status of the encoding job.

Note: When capturing the sequence with image encoders, images are taken only between time steps. This means that if the capturing consists of ten time steps, only ten images will be saved to the destination directory. File names of these images will contain time step number.

Note: Proportions of resulting image or video frame do not necessarily match the visible area of the graph view. This happens due to the selected resolution that, in most cases, has different aspect ratio than the visible area of the graph view. At first, capturing frame is aligned to the top left corner of the graph view and then either its width or height is increased to match the output ratio. This ensures that the resulting image will certainly contain intended part of the graph view while having correct aspect ratio.

3.10.1 Images

GraphRec currently supports encoding to three raster formats (PNG[6], JPG, BMP) and one vector format (SVG[7]). Raster formats are included in *Raster Image e*ncoder. The only possible setting for the raster image is to adjust the quality of the output. SVG format is provided by the *SVG Generator* encoder. Pay attention to the settings of the resolution and DPI[8], because inappropriate values may result in very thin or very wide strokes in the resulting vector image. To avoid this you should increase the resolution when increasing DPI and vice versa.

3.10.2 Video Standards

This section provides a list of standard settings, which should be accepted by FFmpeg encoders. Encoders, such as H.263 or RV10, are especially strict on the offered settings. On the contrary, MPEG encoders provide much more freedom in the custom settings selection. Note also that the visual output provided by GraphRec is less complex and more static in comparison with the real life movies. Due to this fact, it is convenient to use lower bitrates than suggested ones in order to save additional space.

- **MPEG4**
  - **High Definition**, 1280x720x30fps, 1280x720x25fps, 8000 kbps
  - **Home Theater**, 720x480x30fps, 720x576x25fps, 4000 kbps
  - **Portable**, 640x480x30fps, 640x480x25fps, 352x240x30fps, 352x288x25fps, 768kbps
  - **Handheld**, 176x144x15fps, 200 kbps
- **MPEG2**
  - **HD 1080p**, 1920x1080x30fps, 1920x1080x25fps, 15000 kbps
  - **HD 720p**, 1280x720x30fps, 1280x720x25fps, 9000 kbps
  - **DVD**, 720x480x30fps, 720x576x25fps, 4000 kbps
  - **SVCD**, 480x480x30fps, 480x576x25fps, 2376 kbps
- **MPEG1**
  - **VCD**, 352x240x30fps, 352x288x25fps, 1150 kbps
- **RV10**
  - **Download**, 640x480x30fps, 1000 kbps
  - **Broadband**, 560x420x30fps, 650 kbps
  - **Midband**, 320x240x15fps, 300 kbps

[6] Portable Network Graphics (http://www.w3.org/TR/PNG/)
[7] Scalable Vector Graphics (http://www.w3.org/TR/SVG/)
[8] dots per inch

    - **Narrowband**, 192x144x8fps, 100 kbps
- **FLV1**
    - **Best**, 320x240x25fps, 1200 kbps
    - **Normal**, 320x240x25fps, 960 kbps
    - **Optimal**, 352x288x15fps, 780 kbps
    - **Low**, 320x240x15fps, 640 kbps
    - **Least**, 176x144x15fps, 480 kbps
- **H.263**
    - **Extended**, 176x144x15fps, 240 kbps
    - **Highest**, 176x144x15fps, 104 kbps
    - **Standard**, 176x144x12fps, 64 kbps
    - **Balanced**, 128x96x10fps, 140 kbps
    - **Smallest**, 128x96x10fps, 44 kbps

3.10.3 Video Settings

Currently only two additional parameters are provided by the user interface:

- **GOP**[9] is the distance between two *I-frames* (the images containing full information). Note that MPEG1/2 standard is constrained by relatively small GOP sizes (18 for 30fps, 15 for 25fps). Other formats should be used with the GOP set between 100 and 300 frames.
- **Buffer** specifies the size of the internal buffer, which is used by a chosen codec to encode frames. Default size is 8MB, which should be enough for all reasonable settings. However, if you plan to encode a video in extremely high resolutions or bitrates, size should be increased in order to avoid video corruption.

Note: Advanced settings are currently hidden from the user in order to keep things simple. These settings are set internally to provide the best quality at reasonable encoding time and decoding performance. For MPEG4 in particular, this means, that GMC[10], AIC[11] and MV4[12] are enabled. General settings for all encoders are also set to higher values (this includes *macro block decision algorithm*, *motion estimation comparison function* and *Trellis quantization*). Number of *B-frames* (bi-directional frames) is set to two frames per GOP for MPEG4 and MPEG2 (other encoders use their default value).

[9] Group of Pictures
[10] Global Motion Compensation
[11] Advanced Intra Coding
[12] four Motion Vectors by macro block

4 Programmer's Documentation

Documentation consists of various topics concerning the application architecture. The first section provides a detailed description of the compilation. Second section contains architecture overview that briefly describes the whole application in a top-down fashion. The overview should be read before any other of the following sections, because these are organized inversely (bottom-up) and thus not providing any hindsight. In each section, it is assumed that the reader knows information from the previous sections. Documentation is intended to be read together with the corresponding parts of the source code and the *Doxygen* [8] generated call graphs on the included CD-ROM.

Since it is not in the scope of the documentation, it is assumed that programmer understands specific aspects of Qt programming – mainly the *signal/slot mechanism* and *GUI designing*. It should be only noted here that signal/slot mechanism allows a construction of more flexible architecture at the cost of some performance (sending a signal is approximately 10 times slower than the normal function call). Qt GUI designing is built around *qmake*, which takes XML document describing the GUI and compiles it into a C++ code. XML description of the GUI can be done either manually or with the help of QtDesigner. Generated C++ class for the GUI is then accessible by a special pointer. For those and other relevant topics, please refer to the Qt documentation [13].

4.1 Compilation

GraphRec is dependent on Qt framework and FFmpeg video library, both of which are large enough that it is not reasonable to include their source code into the redistributable package. Once those prerequisites are prepared and configured alongside, GraphRec can be built solely from the files that are included in the redistributable package. Since both Qt and FFmpeg are multiplatform projects, it should be possible to perform the compilation on all major platforms. However, testing was done only on Windows and Linux (specifically Windows XP, Windows Vista, Windows 7, Ubuntu 8.10, Ubuntu 9.04, Ubuntu 9.10, Ubuntu 10.04). Steps in the following subsections produce the statically linked build of GraphRec compiled by GNU compilers and compressed by UPX packer [14]. Compilation steps are configured to be compatible with the processors containing MMX and SSE instruction set extensions, which are present on the majority of modern processors. More uncommon extensions (e.g. 3DNow!, SSE2) are disabled. Note that all proposed paths can be changed, however they must not contain any spaces. Because the compilation process is quite complicated to be done manually, the included CD-ROM contains sub-directory with all required packages together with automated build script.

4.1.1 Preparing Environment

Windows

Download following packages from http://sourceforge.net/projects/mingw/files/:

```
binutils-2.19.1-mingw32-bin.tar.gz
mingwrt-3.15.2-mingw32-dll.tar.gz
mingwrt-3.15.2-mingw32-dev.tar.gz
mingw32-make-3.81-20080326-3.tar.gz
gcc-core-4.4.0-mingw32-bin.tar.gz
gcc-core-4.4.0-mingw32-dll.tar.gz
gcc-c++-4.4.0-mingw32-bin.tar.gz
gcc-c++-4.4.0-mingw32-dll.tar.gz
w32api-3.13-mingw32-dev.tar.gz
```

```
pthreads-w32-2.8.0-mingw32-dll.tar.gz
gmp-4.2.4-mingw32-dll.tar.gz
libiconv-1.13-mingw32-dll-2.tar.gz
mpfr-2.4.1-mingw32-dll.tar.gz
```

Unpack all downloaded packages into `c:\mingw\` and confirm all overwrite warnings.

Linux

Make sure you have installed following packages and their dependencies from repositories:

```
binutils
binutils-static
make
gcc
libgcc
libc6
libc6-dev
libglib
libglib-dev
cpp
g++
libstdc++6
libstdc++6-dev
libfontconfig
libfontconfig-dev
libxrender
libxrender-dev
```

Although the goal is to create statically linked executable, on Linux it is dangerous to statically link against system libraries. Thus, the resulting executable will be linked statically only against Qt and FFmpeg. Since libraries like `libc6` or `libstdc++6`, both of which will be linked dynamically, are not backwards compatible, it is advised to carry out the build process with older versions of listed packages. On the other hand, mentioned libraries are forwards compatible up until now. Therefore, the older the packages will be, the more systems will be able to run the executable. However, there must be done a trade-off between the amount of compatible systems and the efficiency of produced code, assuming the newer versions of libraries remove bugs and improve performance.

4.1.2 Building Qt

Windows

Download the following package from http://get.qt.nokia.com/qt/source/:

```
qt-everywhere-opensource-src-4.6.2.zip
```

Unpack the archive into `c:\qt\`.

Open the file `c:\qt\mkspecs\win32-g++\qmake.conf` and edit the following line:

```
QMAKE_LFLAGS = -static -static-libgcc -enable-stdcall-fixup -Wl,-
               enable-auto-import -Wl,-enable-runtime-pseudo-reloc
```

Execute the following batch script from within `c:\qt\`:

```
PATH = %PATH%;c:\mingw\bin\;c:\qt\bin\
configure.exe -nomake tools -nomake examples -nomake demos -nomake
              docs -nomake translations -release -opensource -
              confirm-license -static -ltcg -fast -no-exceptions -no-
              accessibility -stl -no-sql-mysql -no-sql-psql -no-sql-
              oci -no-sql-odbc -no-sql-tds -no-sql-db2 -no-sql-sqlite
              -no-sql-sqlite2 -no-sql-ibase -no-qt3support -no-opengl
              -no-openvg -qt-zlib -no-gif -qt-libpng -no-libmng -no-
              libtiff -qt-libjpeg -no-dsp -no-vcproj -no-
              incredibuild-xge -plugin-manifests -qmake -process -
              rtti -mmx -no-3dnow -sse -no-sse2 -no-openssl -no-dbus
              -no-phonon -no-multimedia -no-audio-backend -no-webkit
              -no-script -no-scripttools -no-declarative -no-style-
              plastique -no-style-cleanlooks -no-style-motif -no-
              style-cde -no-native-gestures -no-iwmmxt -no-crt -no-
              cetest -no-freetype -no-s60
```

```
qmake.exe projects.pro -o Makefile -spec win32-g++
mingw32-make.exe
```

Linux

Download the following package from http://get.qt.nokia.com/qt/source/:

```
qt-everywhere-opensource-src-4.6.2.tar.gz
```

Unpack the archive into `/tmp/qt/`.

Execute the following shell script from within `/tmp/qt/` with elevated privileges:

```
#!/bin/sh
./configure -nomake tools -nomake examples -nomake demos -nomake docs
            -nomake translations -release -opensource -confirm-
            license -static -fast -no-exceptions -no-accessibility -
            stl -no-sql-mysql -no-sql-psql -no-sql-oci -no-sql-odbc -
            no-sql-tds -no-sql-db2 -no-sql-sqlite -no-sql-sqlite2 -
            no-sql-sqlite_symbian -no-sql-ibase -no-qt3support -no-
            xmlpatterns -no-multimedia -no-audio-backend -no-phonon -
            no-phonon-backend -no-webkit -no-javascript-jit -no-
            script -no-scripttools -no-declarative -no-3dnow -no-sse2
            -qt-zlib -no-gif -no-libtiff -qt-libpng -no-libmng -qt-
            libjpeg -no-openssl -no-nis -no-cups -no-iconv -no-dbus -
            no-gtkstyle -no-nas-sound -no-opengl -no-openvg -no-sm -
            xshape -xsync -no-xinerama -no-xcursor -no-xfixes -no-
            xrandr -xrender -mitshm -fontconfig -no-xinput -no-xkb -
            glib
make
make install
```

4.1.3 Building FFmpeg

Windows

Create the following directories:

```
c:\msys\
c:\msys\bin\
c:\msys\mingw\
```

Download the following packages from http://sourceforge.net/projects/mingw/files/:

```
binutils-2.19.1-mingw32-bin.tar.gz
mingwrt-3.15.2-mingw32-dll.tar.gz
mingwrt-3.15.2-mingw32-dev.tar.gz
gcc-core-3.4.5-20060117-1.tar.gz
gcc-g++-3.4.5-20060117-1.tar.gz
w32api-3.13-mingw32-dev.tar.gz
```

Unpack downloaded packages into `c:\msys\mingw\` and confirm all overwrite warnings.

Download the following packages from http://sourceforge.net/projects/mingw/files/:

```
coreutils-5.97-2-msys-1.0.11-bin.tar.lzma
coreutils-5.97-2-msys-1.0.11-ext.tar.lzma
```

Unpack the archives into `c:\msys\bin\`.

Download the following package from http://sourceforge.net/projects/mingw/files/:

```
MSYS-1.0.11.exe
```

Run the installer and set the installation path to `c:\msys\`. Installation will be automatically finished by the post-installation batch script, where, upon request, the MinGW path must be set to `c:/msys/mingw` (note forward slashes).

Download the following package from http://ffmpeg.org/releases/:

```
ffmpeg-0.5.tar.bz2
```

Unpack the archive into `c:\ffmpeg\`.

Run the MSYS environment from the Start menu, change the directory to `c:/ffmpeg/` (note forward slashes) and execute the following bash script:

```
#!/bin/sh
./configure --enable-gpl --disable-ffmpeg --disable-ffplay --disable-
            ffserver --enable-swscale --disable-vhook --disable-
            network --disable-ipv6 --disable-mpegaudio-hp --enable-
            memalign-hack --disable-encoders --enable-encoder=flv --
            enable-encoder=h263 --enable-encoder=h263p --enable-
            encoder=mpeg1video --enable-encoder=mpeg2video --enable-
            encoder=mpeg4 --enable-encoder=rv10 --disable-decoders --
            disable-muxers --enable-muxer=avi --enable-muxer=flv --
            enable-muxer=h263 --enable-muxer=matroska --enable-
            muxer=mov --enable-muxer=mp4 --enable-muxer=mpeg1system -
            -enable-muxer=mpeg1vcd --enable-muxer=mpeg1video --
            enable-muxer=mpeg2dvd --enable-muxer=mpeg2svcd --enable-
            muxer=mpeg2video --enable-muxer=rm --enable-muxer=swf --
            enable-muxer=tgp --disable-demuxers --disable-parsers --
            disable-bsfs --disable-protocol=pipe --disable-devices --
            disable-filters --disable-altivec --disable-amd3dnow --
            disable-amd3dnowext --disable-mmx2 --disable-ssse3 --
            disable-armv5te --disable-armv6 --disable-armv6t2 --
            disable-armvfp --disable-iwmmxt --disable-mmi --disable-
            neon --disable-vis --disable-debug
make
make install
```

Copy `c:\msys\local\include` to `c:\qt\include`.

Copy `c:\msys\local\lib` to `c:\qt\lib`.

Linux

Download the following package from http://ffmpeg.org/releases/:

```
ffmpeg-0.5.tar.bz2
```

Unpack the archive into `/tmp/ffmpeg/`.

Execute the following shell script from within `/tmp/ffmpeg/` with elevated privileges:

```
#!/bin/sh
./configure --enable-gpl --disable-ffmpeg --disable-ffplay --disable-
            ffserver --enable-swscale --disable-vhook --disable-
            network --disable-ipv6 --disable-mpegaudio-hp --enable-
            memalign-hack --disable-encoders --enable-encoder=flv --
            enable-encoder=h263 --enable-encoder=h263p --enable-
            encoder=mpeg1video --enable-encoder=mpeg2video --enable-
            encoder=mpeg4 --enable-encoder=rv10 --disable-decoders --
            disable-muxers --enable-muxer=avi --enable-muxer=flv --
            enable-muxer=h263 --enable-muxer=matroska --enable-
            muxer=mov --enable-muxer=mp4 --enable-muxer=mpeg1system -
            -enable-muxer=mpeg1vcd --enable-muxer=mpeg1video --
            enable-muxer=mpeg2dvd --enable-muxer=mpeg2svcd --enable-
            muxer=mpeg2video --enable-muxer=rm --enable-muxer=swf --
            enable-muxer=tgp --disable-demuxers --disable-parsers --
            disable-bsfs --disable-protocol=pipe --disable-devices --
            disable-filters --disable-altivec --disable-amd3dnow --
            disable-amd3dnowext --disable-mmx2 --disable-ssse3 --
```

```
            disable-armv5te --disable-armv6 --disable-armv6t2 --
            disable-armvfp --disable-iwmmxt --disable-mmi --disable-
            neon --disable-vis --disable-debug
make
make install
```

4.1.4 Building GraphRec

Windows

Download the following package from http://koupy.net/download/:

```
GraphRec-1.0.0-Win32.zip
```

Unpack the archive into `c:\graphrec\`.

Execute the following batch script from within `c:\graphrec\src\`:

```
PATH = %PATH%;c:\mingw\bin\;c:\qt\bin\
qmake.exe GraphRec.pro -spec win32-g++ -r CONFIG+=release
          CONFIG+=static QTPLUGIN+=qjpeg DEFINES+=G_GRSTATIC
mingw32-make.exe
```

Resulting executable is `c:\graphrec\src\release\GraphRec.exe`.

Linux

Download the following package from http://koupy.net/download/:

```
GraphRec-1.0.0-X11.tgz
```

Unpack the archive into `/temp/graphrec/`.

Execute the following shell script from within `/temp/graphrec/src/`:

```
#!/bin/sh
PATH=/usr/local/Trolltech/Qt-4.6.2/bin:$PATH
export PATH
LD_LIBRARY_PATH=/usr/local/lib
export LD_LIBRARY_PATH
qmake GraphRec.pro -spec linux-g++ -r CONFIG+=release CONFIG+=static
      QTPLUGIN+=qjpeg DEFINES+=G_GRSTATIC
make
```

Resulting executable is `/tmp/graphrec/src/GraphRec`.

4.1.5 Compressing Executable

Windows

Download the following package from http://upx.sourceforge.net/download/:

```
upx304w.zip
```

Unpack the archive into `c:\upx\`.

Copy `c:\graphrec\src\release\GraphRec.exe` to `c:\upx\`.

Execute the following batch script from within `c:\upx\`:

```
upx.exe --best --lzma GraphRec.exe
```

Resulting executable is `c:\upx\GraphRec.exe`.

Linux

Download the following package from http://upx.sourceforge.net/download/:

```
upx-3.04-i386_linux.tar.bz2
```

Unpack the archive into `/tmp/upx/`.

Copy `/tmp/graphrec/src/GraphRec` to `/tmp/upx/`.

Execute the following shell script from within `/tmp/upx/`:

```
upx --best --lzma GraphRec
```

Resulting executable is `/tmp/upx/GraphRec`.

4.1.6 Redistributable Package

Package structure:

- `bin` folder contains the main executable.
- `src` folder contains sources and resources (images, icons)
- `doc` folder contains documentation files. Entry point is named `index.html`.
- `samples` folder contains input data for testing purposes.

Both packages, for Windows and Linux, contain the application launcher in their top-level directory. The launcher is a simple batch/shell script that runs the main executable located in `bin` directory.

The installer is made by the third-party software called *InstallJammer* [5]. It is a free, cross-platform install builder with the high level of configurability. It supports both self-unpacking installers and archives.

4.1.7 Licensing

Both Qt and FFmpeg are licensed under the dual license – either GNU GPL or GNU LGPL. GraphRec uses FFmpeg *swscale* support for the highly optimized conversion between RGB[13] and YUV[14] images. Since *swscale* support is GPL-only, GraphRec must be also licensed under the GPL. This implies that redistributable package of GraphRec must contain the complete source code and full text of GNU GPL license. GPL license also demands each source code file beginning with the license stub. By the viral nature of the GPL license, all changes and additions to GraphRec must be released under the same or less restrictive license. Concerning the packed executable, UPX is also licensed under GPL. However, as stated on the project website [14], UPX decompression stub inserted into the compressed executable is not a subject to the GPL, which means that it is compatible with the arbitrary license.

4.2 Architecture Overview

On the top of the hierarchy, there is a `GraphRec` class, which in fact represents the functionality and behavior of the main window and its menu, tool bar and status bar. `GraphRec` also handles the file opening/saving and manages a collection of `GraphView` instances. Each `GraphView` stands for one tab that contains a graph. Because `GraphRec` provides user controls that are shared by all `GraphView` instances, there must be a mechanism for sending user input to the correct `GraphView` (the one that is foreground and focused). This is flexibly achieved by the signal/slot mechanism, which allows literally connecting/disconnecting respective `GraphView` to/from `GraphRec`. `GraphView` class covers the rest of the application functionality – validation, embedding, animation and capturing. `GraphView` also stores the graph representation and is the owner of almost all dialogs (apart from file handling dialogs that are owned by

[13] red (R), green (G), blue (B) color space
[14] luminance (Y), chrominance (U and V) color space

`GraphRec`). This implies that each `GraphView` has its own set of dialogs and thus its own settings. However, configuration of a single `GraphView` can be saved to the persistent storage (registry/file) and then it serves as a template for all new instances.

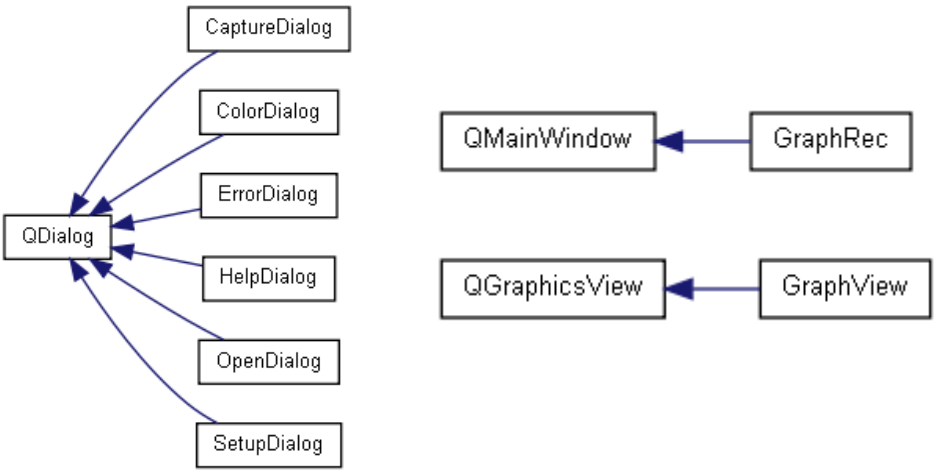


Diagram 2. *User interface* classes.

Since `GraphView` responsibility is very wide, it is distributed onto several other classes and dialogs. In order to pass data among these classes, there is a common data structure `Context`. It serves as a container for several collections containing graph representation, movement calendar and some settings. Idea is that all those helper classes alter one common `Context`, which is then displayed by `GraphView`. Data collections in the `Context` are assembled from classes that represent primitives – `Node`, `Edge` and `Entity`. Graph itself is represented either by list of `Node` instances, each of which contains list of connected `Edge` instances, or by list of `Edge` instances, each of which contains its `Node` pair.

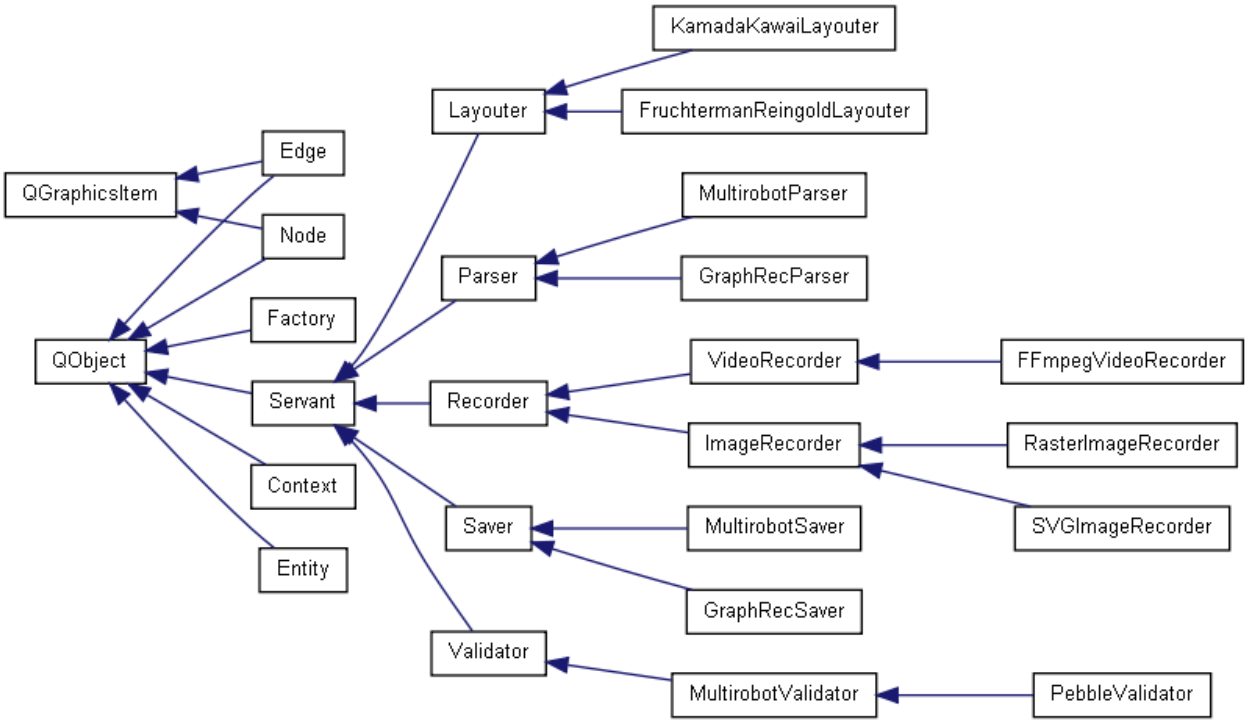


Diagram 3. Structure of the *underlying model* based on the *Strategy pattern*.

As was said earlier, `GraphView` utilizes other classes to achieve some tasks. These classes can be split into two groups – dialogs and servants. Dialog usually exposes part of the `Context` to

the user (e.g. color editing) or provides the interface for an additional functionality (e.g. video capturing). All dialogs are derived from `QDialog` class, thereby forming a simple two-level hierarchy (Diagram 2). Servants are more complicated. Each servant provides specialized functions that can be called by its owner. Servants form a three-level hierarchy (Diagram 3) – first level is constituted of `Servant` interface, which is further inherited by more specialized second-level interfaces (e.g. `Layouter` interface for graph embedding). Third level is composed from actual implementations of second-level interfaces (e.g. `FruchtermanReingoldLayouter` using specific algorithm for graph embedding). `GraphRec`, `GraphView` and dialogs are programmed against servant interfaces located in the first and second level. Implementations on the third level are identified by their name and evidenced in the `Factory` class. When the owner wants to construct a servant of a certain type (or even a name), its request is passed to the `Factory`, which returns the `Servant` pointer to the specified servant. Described mechanism, also known as *Strategy pattern*, brings modularity and extensibility into GraphRec source code. Programmer who wants to implement additional servant (e.g. for layouting) needs to know only graph primitives, `Context` and respective servant interface. Rest of the application is isolated from him/her.

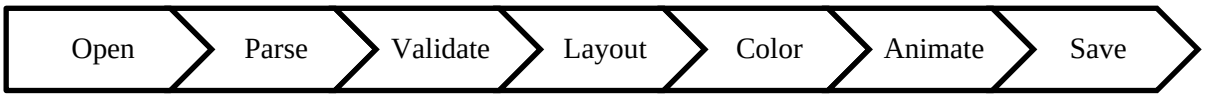


Diagram 4. *Data flow* through the application.

To provide better understanding of the data flow through the application, example of basic user story is described in the following paragraph (Diagram 4). Let us say that the user wants to open, edit and save single solution of *multi-robot path planning*. He/she invokes `OpenDialog` (owned by the `GraphRec`) and loads file into it. `OpenDialog` asks its `MultirobotParser` servant to find solutions in the file. After the intended solution is selected and dialog is confirmed, solution location is handed to the `GraphView`, which in turn creates its own `MultirobotParser` servant and asks it to initialize the `Context` by collecting information from the provided file location. Immediately afterwards, `Context` is validated by the `MultirobotValidator` servant and finally displayed by the `GraphView`. Initialized `GraphView` is then connected to the `GraphRec` user interface and user can start to do some work. As an example, user might decide to activate embedding algorithm. In that case, `GraphView` asks the `FruchtermanReingoldLayouter` servant to calculate node positions in the `Context`. User also might not be satisfied with the graph coloring. Therefore, he/she invokes `ColorDialog`, in which a color of any primitive in the `Context` can be changed. On the other hand, animation and rendering is handled by `GraphView` itself. If user decides to save the solution, `GraphRec` creates `MultirobotSaver` servant and provides it to the respective `GraphView`, which in turn asks it to save the `Context`.

4.3 Graph Primitives

Graph is represented by three primitives – `Node`, `Edge` and `Entity` – all of which are described in following subsections. Whereas `Entity` is essentially only a data container, both `Node` and `Edge` inherits `QGraphicsItem` and implements its paint event handler. Because paint handlers are called quite frequently and `QGraphicsItem` might need some unrelated data to paint itself (e.g. `Node` needs the current time step to infer the color), it is reasonable to maintain local copies of those data rather than asking for them at each paint event. Propagation of such information is done via the signal/slot mechanism – thus, to continue the example, when the time step is changed in the `GraphView`, it is also changed in all `Node` instances.

4.3.1 Entity Class

`Entity` is a simple class containing its identification (`m_id`), final time step (`m_timestepFinal`) and color information. Final time step is the one after which `Entity` does not move anymore (stops changing owners). Each *Entity* stores its normal (`m_clBackground`, `m_clForeground`) and final (`m_clBackgroundFinal`, `m_clForegroundFinal`) color set. `Node` owning the `Entity` is colored by the final color set when and after the final time step is reached. Color set is a pair of colors that are used by the hosting `Node` for both background and label. Note that `Entity` itself is actually not visible in the `GraphView` – it only provides information to the hosting `Node`.

4.3.2 Node Class

`Node` is a class inherited from the `QGraphicsItem`. It contains its identification (`m_id`), reference to the currently contained `Entity` (`m_entity`) and the list of references to connected `Edge` instances (`m_edges`). For purposes of the animation, `Node` provides shallow copy constructor `CloneShallow()`. Shallow copy is not connected to the graph and is intended only as a temporary object used for depicting the movement of `Entity` between two `Node` instances. Since `Node` is one of the visual elements rendered by `GraphView`, it must be able to paint itself accordingly to its inner state and contained `Entity`:

- In case of null reference to `Entity`, `Node` is colored by its own color set (`m_clBackground`, `m_clForeground`, `m_clBoundary`). Otherwise, it is colored by colors provided by the hosted `Entity`. `Node` keeps track of the current time step (`m_timestep`), which is compared at every paint event with the final time step discovered from the `m_entity`. This information is needed to determine whether to use final color set or not.
- Optionally, `Node` is able to display its identification or even identification of the contained `Entity`. Possible combinations are listed in `NodeDescription` enumeration and saved in `m_description` variable.
- `Node` keeps track of its current position. In the case of discrete positioning (`m_discreteDisplacementEnabled`), `Node` snaps itself to the closest allowed location (`m_discreteDisplacementOffset`). This functionality is implemented in the event handler `itemChange()` for positional changes and in the `AlignPoint()` function. Change of the position is reported to the `GraphView` via `NodePositionChanged()` signal.
- At every position change, all `Edge` instances connected to the `Node` (`m_edges`) are requested to update their positions as well (`Edge::Adapt()`).

When selected or moved by the user, Z coordinate of `Node` is elevated over the rest of graph elements displayed by the `GraphView`. Z coordinate is returned to its original value after the action is finished. Third level of Z-axis is reserved for normal `Node` instances, fourth level for the selected instances and finally fifth level is reserved for the grabbed instance (the one dragged by a mouse). Note that all three levels are defined above the levels reserved for `Edge`. Also, note that there is an automatic sub hierarchy between graphic elements that share the same Z coordinate. Refer to `mousePressEvent()` and `mouseReleaseEvent()`.

4.3.3 Edge Class

Likewise the `Node`, `Edge` is another class inherited from the `QGraphicsItem` and rendered by the `GraphView`. It contains references (`m_nodeSource`, `m_nodeDestination`) and positions (`m_ptSource`, `m_ptDestination`) of its two connected `Node` instances. Positions are stored for efficiency, since it is assumed that `Edge` paint event is called more frequently than node positions are changed. `Edge` must be able to render itself in a normal and highlighted mode (`m_highlight`), which is represented by a different color and increased thickness. Each time when the `Entity`, represented by the shallow copy of the hosting `Node` (`Node::CloneShallow()`), is moving along the trajectory depicted by the respective `Edge`, the shallow copy of this `Edge` (`Edge::CloneShallow()`) is switched to the highlighted mode (`m_highlight`) and placed exactly under its prototype on the Z-axis. This approach ensures that highlighting thick line will be always painted on the background while not overlapping lines of non-highlighted `Edge` instances. As was stated in the `Node` description, shallow copy is intended only for the mentioned purpose. First level of Z-axis is reserved for the highlighted `Edge` instances and the second level is reserved for normal `Edge` instances. Note that both levels are defined below the levels reserved for `Node`.

4.4 Passing Common Data

`Context` is a class containing common data structures used by `GraphView`, dialogs and servants. Each `Context` is owned by a single `GraphView` and passed by a reference to other objects. Note that `Context` can be extended by additional data entries. However, current data entries must remain the same, because many classes depend on them. Description of data entries follows:

- Graph is identified by its `filePosition` in the file of name `fileName` located in `filePath`. For convenience, `fileName` and `filePath` are joined in the `fileCompleteName`. Graph can also have a title `graphName`, which is currently used as a placeholder for graph ID optionally provided by input file. Both `graphName` and `fileName` are exposed to the user – `graphName` as a tab label and `fileName` as a tab context help.
- Hashed map `nodes` maps node identifications to respective `Node` instances.
- Hashed map `entities` maps entity identifications to respective `Entity` instances
- Hashed map `edges` maps pairs of node identifications to respective `Edge` instances. It is intended for the fast lookup of `Edge` instance when only identifications of its two nodes are known.
- `CalendarEvent` is a structure consisting of `timestep` and `move`, which is expressed as a pair of source node identification and destination node identification. `CalendarEvent` also contains two flags – `valid` determines whether the event was approved by a validator and `reverse` determines the movement direction. Thus, if the direction suggested by the input file is evaluated as invalid, validator has an option to approve the inverse direction instead (if it is valid). All `CalendarEvent` structures are stored in an ordered list `calendar`. List is ordered by `timestep` values. Since more than one `CalendarEvent` can have the same `timestep` value and `Context` uses unstable sorting algorithm, order between events with the same time step is not defined.
- Hashed map `timesteps` maps each time step to the index of its first occurrence in ordered `calendar`. It is intended for the fast lookup of events belonging to the same time step.

- `Frame` is a hashed map that maps `Node` instances to `Entity` instances. It should be interpreted as definition of locations for all `Entity` instances at one particular time step. All `Frame` maps are stored in the list `frames`, which is indexed by time steps.
- Color of the `GraphView` scene is saved in the `sceneBackground`.
- Information about the currently viewed part of the `GraphView` scene is saved in `sceneMatrix` and `sceneViewCenter`. Whereas `sceneMatrix` stores a transformation matrix for zooming, `sceneViewCenter` stores a point in the scene that is aligned to the center of the visible area (*viewport*). To provide a backwards compatibility with the alpha version of GraphRec, there is also a `sceneAngle`, which determines the rotation of the scene. However, `sceneAngle` is now *obsolete*, because rotation is applied on the graph itself instead of the scene.
- String `validatorName` stores the name of the last validator that validated `Context`.
- Flag `enabledColoring` specifies whether the input file has provided explicit coloring information or not. If not, `enabledColoring` is *true* and serves as a hint stating that implicit colors should be applied.
- Flag `enabledLayouting` specifies whether input file has provided explicit positional information or not. If not, `enabledLayouting` is *true* and serves as a hint stating that embedding should be applied automatically.
- Intended displacement of nodes is stored in a `layoutDisplacement` variable. It serves as a hint for layouters. Note that the value is relative and each layouter can interpret it differently.

4.5 Producing Servants

`Servant` is an abstract class, which is intended as a basic interface provided by its implementers. Basic interface includes the function `Name()` that returns a servant name and the function `Description()` that returns its description. Whereas the name serves as a unique identification of the servant, description is intended for a usage by the GUI (menu entries, status bar labels etc.). Each servant must also provide static version of the name function, called `GetName()`, so that its name can be discovered without the instantiation. Both `Name()` and `GetName()` must return the same string.

Classes that implement `Servant` are catalogued in the `Factory` class. Servants are grouped according to their types, which are listed in the `ServantType` enumeration. Each type corresponds to an abstract class that inherits `Servant` and specifies some additional functions. These specialized interfaces are described in the following subsections. When a certain object needs a servant for some purpose and knows its type and name, it calls `CreateServant()`, which is a `Factory` static function returning the instance of the requested servant. `Factory` also provides static function `GetServantNames()` for discovering all available servants of one particular type.

4.5.1 Parser Interface

`Parser` is an abstract class, which inherits `Servant` and imposes implementation of two functions:

- `ParseFile()` should search the `file` for graphs, **count statistics** for each found one and insert those data into the table according to provided `header`. Table is composed from root and its children, all of them `TreeWidgetItem` instances. Whereas root only holds file name and file path (as a tool tip), each child stands for one line in the table. Idea is that, when displaying more than one analyzed file to the user, each table can be

folded into its root. Collected information includes the number of graph elements, solution length, preferred servants and file location. Order of these items in the table row is specified by parameter `header`, which is a list of entries from `HeaderItem` enumeration. Each `HeaderItem` entry in the list serves as a hint on what information to put in what table column. Function returns reference to the table root. `TreeWidgetItem` inherits `QTreeWidgetItem`, which serves as a basic element for many of Qt data visualization widgets (lists, trees, grids). Only difference between the two is that `TreeWidgetItem` implements differently its compare function for sorting – it has more universal behavior when comparing various combinations of text and number values.

- `ParseGraph()` should search the `file` on the location specified in the `context`, **completely analyze** a graph on this location and insert all data into the `context`. It is expected that the function builds `nodes`, `entities`, `edges`, `calendar` and fills `graphName`, `enabledColoring`, `enabledLayouting` in the `context`. Optionally it can also fill `validatorName`, `sceneBackground`, `sceneMatrix` and `sceneViewCenter`. Note that `calendar` should get sorted before leaving the function. While parsing the file, it is possible to emit some error messages through `Error()` signal.

Note that the `Name()` function of the `Parser` must return a string containing information about the file suffix in the following format: `*.suffix` (e.g. `MyParser (*.txt)`).

4.5.2 Saver Interface

`Saver` is an abstract class, which inherits `Servant` and imposes implementation of three functions:

- `Open()` should open the provided file and accomplish an initialization of the saver. It is also intended for writing a file header etc.
- `Save()` should save the provided `context` into the output `file`. The structure and amount of data that are going to be saved is entirely up to the `Save()` function.
- `Close()` should safely close the output file (if needed). It is also intended for writing a file footer etc.

Note that the `Name()` function of the `Saver` must return a string containing information about the file suffix in the following format: `*.suffix` (e.g. `MySaver (*.txt)`).

4.5.3 Validator Interface

`Validator` is an abstract class, which inherits `Servant` and imposes implementation of two functions:

- `Validate()` is responsible for exploring the `calendar` and building the `frames` in the provided `context`. It should set `valid` and `reverse` flags in every `CalendarEvent`. While going through the `calendar`, function should progressively build `Frame` maps from valid movements and insert these maps into the `frames` in the `context`. Since `frames` is the main structure needed for an animation, `Validate()` is the most responsible function in a matter of what exactly will be animated. While validating the `calendar`, it is possible to emit some error messages through `Error()` signal. Function should also set a `validatorName` in the `context` as a signature.
- `GetColor()` returns certain color for the every value specified in `ColorScheme` enumeration. An idea behind this function is that all graphs validated by one particular `Validator` should be visually distinguishable from the others. Note that this function serves only as a hint and should be used only if the input file did not specified any ex-

plicit coloring of the graph elements. Since `Validator` interface implements this function itself, its reimplementation is optional in the implementer.

4.5.4 Layouter Interface

`Layouter` is an abstract class, which inherits `Servant` and imposes implementation of a `Layout()` function. It alters positions of `Node` instances contained in the `nodes` map in the provided `context`. Positions are set accordingly to the implemented embedding algorithm. Since some embedding algorithms are iterative, it is assumed that function will be repeatedly called by the owner, probably on the timer timeout or on the separate thread. Consequently, the function must return boolean value specifying whether the layout is finished (*true*) or not (*false*). Non-iterative algorithm, which calculates the layout in a single call, should simply return *true*.

4.5.5 Recorder Interface

`Recorder` is an abstract class, which inherits `Servant` and imposes implementation of a `GetSettingsWidget()` function. It should prepare and return a `QWidget`, which contains controls connected to the custom slots in the implementer. Returned widget is presented to the user as a part of `CaptureDialog`. When the user edits any of these controls in the dialog, all changes are directly sent to the `Recorder` implementer. This mechanism allows the implementer to have almost any specific additional settings that are not already covered by the interface functions.

ImageRecorder Interface

`ImageRecorder` is an abstract class, which inherits `Recorder` and imposes implementation of two functions:

- `GetPaintDevice()` should prepare and return `QPaintDevice` into which the owner will render the visual data. Since some Qt classes derived from the `QPaintDevice` need various information for their construction, function takes four arguments – `path` and `name` of the target file (for devices that directly saves the data), `height` and `width` of the image (for raster devices).
- `SaveImage()` should encode the provided `device` to the target file specified by a `path` and `name`. After the owner renders data to the `QPaintDevice` obtained by `GetPaintDevice()`, it can call `SaveImage()` passing the `device` as an argument. Function is intended for devices that do not save the data directly to the persistent storage.

Note: Both functions take the target file name as an argument. This name is **incomplete** and serves only as a template. Function should search the destination for any name conflicts and appropriately edit the provided file name to be unique (e.g. by adding number). Format suffix must be also appended to the `name`.

VideoRecorder Interface

`VideoRecorder` is an abstract class, which inherits `Recorder` and imposes implementation of three functions:

- `GetFPS()` returns the intended frame rate for the video stream.
- `Start()` function should initialize the encoder and then run a consumer thread, which will encode `QImage` instances from the global buffer `G_GRVideoBuffer` of the size `G_GRVideoBufferSize`. Function takes four arguments – `path` and `name` of

the target file, `height` and `width` of the video frame. The `name` is incomplete and serves only as a template. Function should search the destination for the name conflicts and appropriately edit the provided file `name` to be unique (e.g. by adding number). Format suffix must be also appended to the `name`. Access to the global buffer is guarded by two semaphores – `G_GRVideoFree` and `G_GRVideoUsed`. All these global variables with `G_GR` prefix are declared in the `main.cpp` file and can be accessed by only one producer and one consumer at a time. Since GraphRec architecture allows the user to request more than one video encoding job at a time, there is a danger of having more than one concurrent consumer. It would certainly lead to the corrupted video. Thus, before doing anything with those global variables, `Start()` function have to check whether the `G_GRVideoOwner` pointer is null and does not point to some other `VideoEncoder` implementer. In the case that the pointer is null, it should be initialized by a self-reference of the implementer. This approach guarantees the exclusive access to the global variables.

- `Stop()` function should wait until the `G_GRVideoBuffer` is emptied and then should safely terminate the consumer thread. After that, the output file should be ended (probably with some footer or trailer) and closed. Before the function returns, the `G_GRVideoOwner` pointer has to be set to null.

4.6 GraphView Class

`GraphView` class inherits `QGraphicsView`, which is a class for advanced two-dimensional graphics. `GraphView` extends its ancestor by a several custom-made functions and by a usage of additional dialogs, controls and servants. Since `GraphView` is quite complex class, its description is divided into several subsections. This introductory section will describe only a `GraphView` construction and its connections with other classes.

Usually, it is not very useful to describe class constructor in the documentation. However, in the case of `GraphView` constructor, a systematic description supports very well the overall idea over the low-level application structure and data flow:

1) Arguments passed to the constructor consists of the graph location (`file`, `filePosition`) and servant names (`parserName`, `validatorName`, `layouterName`).
2) `QGraphicsScene` is created and initialized by calling the inherited `setScene()` function. Scene is the core part of every `QGraphicsView`. For more information about `QGraphicsView`, `QGraphicsScene` and its coordinate system, please refer to the Qt documentation [13].
3) `Context` (further accessible through `m_context`) is created and information about the graph location (`file`, `filePosition`) is stored into it.
4) `ErrorDialog` (further accessible through `m_dialogError`) is created.
5) `Parser` corresponding to the `parserName` is obtained from th4 `Factory` and its `Error()` signal is connected to the `Log()` slot in the `ErrorDialog`.
6) Input `file` is parsed by the `Parser::ParseGraph()` on the specified `filePosition`, effectively filling almost all information in the `m_context`.
7) Scene background, scene matrix and viewport central point are set accordingly to the `m_context`.
8) `Validator` that corresponds to the `validatorName` is obtained from the `Factory` and its `Error()` signal is connected to the `Log()` slot in the `ErrorDialog`.
9) Structures in the `m_context` are validated, which results in the construction of the `m_context->frames`. If the input file did not specified any coloring, default colors of

the chosen `Validator` are injected into graph elements in the `m_context`. Otherwise, coloring has been already done by a parser.

10) All instances of the `QGraphicsItem` from the `m_context` (that is, `Node` and `Edge` instances) are added into the scene.
11) `Layouter` corresponding to the `layouterName` is obtained from the `Factory` and saved into the `m_layouter`.
12) `QTimer` for the embedding (further accessible through `m_timerLayout`) is created and connected to the `on_timerLayout_timeout()` handler.
13) If the input file did not specified positioning, layout is firstly randomized by the `LayoutRandomize()` and then properly calculated by the `m_layouter`.
14) All nodes are connected to `GraphView` signals and slots, which are further utilized to distribute several parameters into and from the nodes on the user actions or during the animation. Especially note the `NodeMoved()` slot, which notifies the `GraphView` about layout changes.
15) `QTimeLine` for the animation (further accessible through `m_animationTimeLine`) is created and connected to the `AnimateTimeStep()` slot.
16) Signal `AnimationStepDone()` is connected to the `AnimateTimeStep()` slot.
17) Additional visual controls are created - two `QSlider` instances (`m_sliderTimeStep`, `m_sliderDisplacement`) and one `QSpinBox` instance (`m_spinBoxTimeStep`). Both controls referring to the time step are connected with each other to reflect the same information all the time. All three controls are connected to the `GraphView` slots (`SetTimeStep()` or `SetDisplacement()`). Note also that their focus is forwarded to the `GraphView` (this is very important, because a user expects the `GraphView` would connect to the `GraphRec` by clicking anywhere into its area).
18) Widgets created in the previous step are placed onto the `GraphView` surface.
19) `ColorDialog` (further accessible through `m_dialogColor`) is created.
20) Settings are fetched from the persistent storage.

Following list classifies output connections (signals) from the `GraphView` to other classes.
Connections to the `GraphRec` (can be fired at once by calling `UpdateConnections()`):

`HasFocus()`, `ValidatorNameChanged()`, `LayouterNameChanged()`, `ValidatorDescriptionChanged()`, `LayouterDescriptionChanged()`, `Message()`, `DiscreteDisplacementEnabled()`, `LayoutingEnabled()`, `NodeLabelsChanged()`, `LayoutingInProgress()`, `AnimationInProgress()`

Connections to all `Node` instances:

`TimeStepChanged()`, `DiscreteDisplacementEnabled()`, `DiscreteDisplacementOffsetChanged()`

Connections to additional controls:

`TimeStepChanged()`, `DisplacementChanged()`, `DurationChanged()`

Connections to the `GraphView` itself:

`AnimationStepDone()`

4.6.1 Error Dialog

`ErrorDialog` is a class, which inherits `QDialog` and has its GUI part accessible through the `m_ui` pointer. Class provides a public `Log()` slot, which can be used by other classes to post error messages into the `QListWidget` (`m_ui->listWidgetErrorLog`) in the `ErrorDialog`. Whenever the user selects some error message (`QListWidgetItem`), it is compared to the regular expression `m_errorRegExp`. If the expression matches, `ErrorDialog` emits the `ErrorSelected()` signal, which is connected to the `GraphView::SelectEvent()` slot. Selected event

is then tracked down in the GraphView::m_context. If the event exists, timeline is set to its time step and corresponding nodes are highlighted. Whole log can be also saved into a simple text file (on_buttonSave_clicked()). ErrorDialog is non-modal and is constructed in the GraphView constructor and destroyed in its destructor. During the lifetime of the owner, ErrorDialog can be only shown or hidden through GraphView::m_dialogError.

4.6.2 Color Dialog

ColorDialog is a class, which inherits QDialog and has its GUI part accessible through the m_ui pointer. GUI is represented by a QTabWidget with three tabs, each containing one QTreeWidget. Order of tabs is inferred from the TabOrder enumeration. Header of the first tree widget is constructed according to the Node::NodeColorType enumeration. Similarly, header of the second tree widget corresponds to the Entity::EntityColorType enumeration. For the fast determination of both color types, column numbers are mapped to the types in the nodeColorTypes and entityColorTypes maps. Whereas the first and the second tree widget is intended for listing nodes and entities (both obtained from m_context argument passed to the constructor), third tree widget contains more general items – namely m_itemBackground, m_itemBoundary and m_itemHighlight. Column order and their captions can be easily changed in the constructor.

ColorDialog is designed to be non-modal, effectively allowing the selection of nodes or entities directly from the GraphView. Whereas GraphView::keyPressEvent() on *Ctrl* key enables the nodes selection by a mouse dragging, GraphView::keyReleaseEvent() on *Ctrl* key appends all selected nodes to the list and passes it to the ColorDialog::SelectItems() function, which locates and selects corresponding items in the tree widget (either nodes or entities, depending on what tree widget is currently visible).

Color changes are done in a flexible but rather complicated way. Every time a dialog tab is changed, on_tabWidget_currentChanged() calls CreateMenu() function, which constructs a customized menu for the current tree widget. Menu actions correspond to the tree widget columns and are connected through a QSignalMapper to the SetColor() slot. Menu is then embedded into the m_ui->buttonSetColor and serves also as a tree widget context menu displayed by the ShowContextMenu() function. Whenever user selects one or more items in the tree widget and hits a certain menu action, SetColor() is called with the column number obtained from the signal mapper. Color palette is then showed (static function QColorDialog::getColor()) and after the user chooses the appropriate color, it is saved for all selected items into the m_context, each time inferring color type from the nodeColorTypes or entityColorTypes. Note that m_itemBoundary and m_itemHighlight must be treated in a special way, since it affects more than one graph element in the m_context. Another exception is m_itemBackground, which is connected to the GraphView through the BackgroundColorChanged() signal.

ColorDialog is constructed in the GraphView constructor and destroyed in its destructor. During the lifetime of the owner, ColorDialog can be only shown or hidden through GraphView::m_dialogColor. Since it is not possible to change colors elsewhere, ColorDialog contains actual color information for all graph elements at any time.

4.6.3 Embedding

Mechanism of the automatic continuous embedding is based on the ticking of m_timerLayout and its handler on_timerLayout_timeout(), which calls repeatedly m_layouter->Layout() until the *false* value is returned informing that the layout is finished. Automatic

continuous embedding is enabled by calling the `SetLayoutingEnabled()`, which sets the `m_context->enabledLayouting` variable. By calling `SetDisplacement()`, it is possible to change `m_context->layoutDisplacement`, which is fetched by the `Layouter::Layout()` from the `m_context`. Timer `m_timerLayout` can be controlled by the `LayoutStart()` and `LayoutStop()` functions. Note that `m_timerLayout` can be also launched indirectly by moving any node while the automatic embedding is enabled – it signals the `NodeMoved()` slot, which calls `LayoutStart()`.

Discrete embedding is done differently. It can be enabled by calling the `SetDiscreteDisplacementEnabled()`, which sets the `m_discreteDisplacementEnabled` variable. Function `drawBackground()` then repeatedly paints the grid consisting of horizontal and vertical lines on the background of the `GraphView` scene. The color of the lines is inversed RGB value of the background color. The offset between lines in the grid can be set by calling the `SetDiscreteDisplacementOffset()`, which sets the `m_discreteDisplacementOffset`. Both `m_discreteDisplacementEnabled` and `m_discreteDisplacementOffset` are sent to all nodes via `DiscreteDisplacementEnabled()` and `DiscreteDisplacementOffsetChanged()` signals whenever changed. Every node then calls `Node::AlignPoint()` each time its position is altered (`Node::itemChange()`), effectively snapping itself to the closest intersection of the horizontal and vertical line. All nodes can be snapped to the grid at once by calling `LayoutDiscrete()`.

It should be noted that node positions are locked during the animation by the `LayoutLock()` function, because it would otherwise lead to entities visually missing their destinations. When both `m_context->enabledLayouting` and `m_discreteDisplacementEnabled` are disabled, nodes can be freely moved by the user. When needed, layout can be randomized by the `LayoutRandomize()` function.

4.6.4 Scene Actions

All functions described in this section are usually called from the `keyPressEvent()` handler. Graph can be moved by calling the `ScrollGraph()`, which in fact move every node from the *m_context* by calling the `Node::moveBy()`. Graph movement is only possible when the layouter is not working, because otherwise it might destabilize its algorithm. Note that viewport scrolling, which is handled by the `QGraphicsView` itself, is still possible by a mouse dragging even when the embedding is in progress. Zooming is done by the `ScaleView()`. Provided scaling factor is first tried on the scene matrix in order to prevent very large/small zoom, and then applied by calling the `scale()` function inherited from the `QGraphicsView`. `ScaleView()` is usually called by the `whellEvent()` handler, which calculates the scaling factor from the mouse wheel rotation.

The most complicated action is rotation. Its implementation can be found in the `RotateGraph()` function. It should be noted that alpha version of GraphRec rotated the whole scene. Since this approach lead to certain inconsistencies with other features, graph is currently rotated by changing positions of its nodes instead. This also implies that `Context::sceneAngle` is no longer needed and only kept for backwards compatibility with files created by the alpha version (such files are now transformed in the `GraphView` constructor to be compatible with a new approach). Another limitation, similarly to the scrolling, is that the graph can be rotated only when the layouter is not working due to the possible destabilization of its algorithm. Discrete positioning is also ignored during the rotation, because it is in conflict with the implemented rotation mechanism. Rotation itself is done in the following way. All nodes are grouped into the `QGraphicsItemGroup` and the current cursor position is determined by the

`QCursor::pos()`. Then, `QTransform` is applied onto the group by virtually moving the whole group to the cursor position, rotating it here by the given angle and moving it back to its original position. This ensures that graph is rotated over the cursor position, which is more flexible than rotation over the fixed viewport center or graph center (which is expensive to calculate).

4.6.5 Animation

Usually, before the animation is started, user sets the initial time step for the animation by calling the `SetTimeStep()`. It simply takes all nodes and updates their entities according to the `m_context->frames`. In order to do that, `GraphView` stores information about the current time position in the `m_calendarPosition` and `m_timestep` – both of which are kept synchronized with the help of `m_context->timesteps`. Whenever the `m_timestep` variable is altered by some function, the function also emits the `TimeStepChanged()` in order to update the time step in all nodes, so that nodes know whether they are in the final position or not. From the hindsight, animation is a sequence of `AnimateTimeStep()` calls. Every `AnimateTimeStep()` in the sequence either starts the `m_animationTimeLine` for a visible animation or emits `AnimationStepDone()` for the fast skip. Both signals, `m_animationTimeLine->finished()` and `AnimationStepDone()`, are connected back to the `AnimateTimeStep()`, effectively acting as an endless loop. The loop can be controlled by the `AnimationStart()` and `AnimationStop()` functions. `AnimationStart()` sets the `m_animationIsRunning` flag, so that other functions can discover whether the animation is in progress, and calls the `AnimateTimeStep()` to start the loop. `AnimationStop()` only sets the `m_animationStopRequest` flag, which is periodically checked by the `AnimateTimeStep()`. There is also the function `AnimationStep()`, which is only the simple sequence of two calls – `AnimationStart()` and `AnimationStop()`. Considering the animation, there are two embedding problems that deserve a special attention. First problem is that the animation assumes static positions of graph elements – animated movements are precalculated one time step ahead. Thus, embedding must be paused during the animation, because animated entities would otherwise miss their possibly moving destinations. Second problem is that the user can change embedding settings during the animation. In order to react correctly to these situations, `AnimationStart()` locks node positions (`LayoutLock()`) and sets the pair of flags - `m_layoutingWasEnabled` (automatic continuous embedding mode was enabled at the moment of the animation start) and `m_layoutingWasRunning` (layouter was still working at the moment of the animation start).

`AnimateTimeStep()` itself is quite a large function, which is composed from the several logical parts (recommended reading order is 3, 4, 1, 2):

1) **Processing data structures from the last call**. All shallow copies from the `m_nodeBuffer` are removed from the scene and destroyed; their entities are inserted into corresponding destination nodes (also discovered from the `m_nodeBuffer`). Shallow copies from the `m_edgeBuffer` are also removed from the scene and destroyed. Finally, animations from the `m_animationBuffer` are destroyed, leaving all three data structures empty for the next round.
2) While **deciding whether to stop the animation loop**, `m_animationStopRequest` is checked. If it is *true*, `m_animationIsRunning` flag is reset and node positions are unlocked by the `LayoutLock()`. Note that there are two situations, in which the layouter should be immediately launched. First situation occurs when the layouter was running just right before the animation and automatic embedding is still enabled now (means that the layouter was interrupted). Second situation occurs when the automatic embedding was not enabled before the animation but is enabled now. Both cases can be in-

ferred from the `m_context->enabledLayouting`, `m_layoutingWasEnabled` and `m_layoutingWasRunning` flags.

3) **Preparing data structures for the animation and the next call**. Every valid `CalendarEvent` structure with the same time step as the `m_timestep` is fetched from the `m_context->calendar`. Shallow copy of the event source node is inserted into the scene and its `QGraphicsItemAnimation` is created, connected to the `m_animationTimeLine` and appended to the `m_animationBuffer` list. Each `QGraphicsItemAnimation` infers its starting and ending point from the position of associated source and destination node. Idea is that the `m_animationTimeLine` acts as a director, who sets the time in all connected `QGraphicsItemAnimation` instances, which, in turn, move their embedded `QGraphicsItem` along the predefined pathway. Entity is removed from the source node immediately after the shallow copy is created, effectively leaving the copy as the sole entity carrier. Both shallow copy and destination node are always appended as a pair into the `m_nodeBuffer`. Note that shallow copy is not connected to any of `GraphView` signals. Thus, it is safe to increment the `m_timestep` even while the entities are not yet in their destinations. Edge between the event source node and destination node is also shallow copied and the copy is switched to the highlighted mode. All such edges are appended to the `m_edgeBuffer`.
4) **Emitting signals to continue the animation loop.** Depending whether just animating or capturing (`m_recordingEnabled`), `m_animationTimeLine->start()` is called (internally emitting signals) or `AnimationStepDone()` is emitted. Signal is emitted also in the case of empty `m_animationBuffer` or in the case of the zero duration of the `m_animationLine` (inferred from the `m_timeoutZero` flag), because the animation would be invisible and only slowing down the process.

4.6.6 Setup Dialog

`SetupDialog` is a class, which inherits `QDialog` and has its GUI part accessible through the `m_ui` pointer. `SetupDialog` is implemented in a straightforward way providing the get and set functions for almost all of its control widgets. Dialog is utilized by the `GraphView::ShowSetupDialog()`, which, at first, inserts `GraphView` variables into the `SetupDialog` controls, then executes it as a modal dialog and finally fetches its values back to the `GraphView` variables. The only interesting thing about the dialog implementation is that change signals of its control widgets are connected to the `ChangeTrigger()` slot, which emits `Changed()` signal that is connected to the `m_ui->buttonDefault` enabler. Thus, if the settings are changed and the user hits the just enabled `m_ui->buttonDefault`, they are saved as a default global settings for any subsequent `GraphView` constructor.

4.6.7 Capture Dialog

`CaptureDialog` can be invoked from the main menu through two different actions that are connected to the corresponding slots in the `GraphView` – `Snapshot()` or `Sequence()`. Both slots call `GraphView::ShowCaptureDialog()` with appropriate `CaptureDialog::Mode` as a parameter. `ShowCaptureDialog()` closes and destroys the existing `GraphView::m_dialogCapture` and constructs the new one. After the dialog is constructed, settings are injected into it, its signals for accepting and rejecting are connected to the `GraphView::CaptureDialogHandler()` and finally, depending on the passed mode, it is invoked as a modal (`mSequence` mode) or non-modal (`mSnapshot` mode) dialog.

CaptureDialog is a class, which inherits QDialog and has its GUI part accessible through the m_ui pointer. Most of the capture settings provided by the dialog are implemented in a straightforward get/set way. The only interesting is m_ui->comboBoxRecorder containing the list of available recorders (discovered from the Factory). Every time a recorder is selected (on_comboBoxRecorder_currentIndexChanged()), its instance is retrieved from the Factory and saved into the m_recorder pointer. Recorder interface defines the Recorder::GetSettingsWidget() function that returns a QWidget, which, upon call, is saved into the m_widgetRecorder and embedded in the CaptureDialog window. Behavior of the CaptureDialog depends on the Mode passed to the constructor. First of all, each mode has a specific set of enabled controls. More importantly, whereas mSnapshot mode only allows the listing of recorders that implement ImageRecorder, mSequence allows all available recorders. The behavior of the dialog confirmation handler (on_buttonCapture_clicked()) is also different – in the mSnapshot mode, it only emits the signal but leaves the dialog opened.

Back in the GraphView, CaptureDialogHandler() is called as a reaction on the CaptureDialog confirmation. Depending on whether the dialog was accepted or not, settings are then fetched from it into the GraphView private variables and the selected recorder is saved to the GraphView::m_recorder pointer. Further solving of the task is leaved for either GraphView::SnapshotHandler() or GraphView::SequenceHandler(), both of which are described in the next section.

4.6.8 Rendering

Rendering is covered by the Render() function. For now, let us only say that the function behaves differently on what GraphView::RenderMode is stored in the m_renderMode variable – it is either directly saving the image to the persistent storage (rmDirectSave mode) or inserting the image into the G_GRVideoBuffer (rmBufferSave mode). Render() function might be invoked from four places in the source code. These entry points represent four different approaches how the rendering can be handled:

1) SnapshotHandler() sets m_renderMode to rmDirectSave and simply calls the Render(), which renders and saves a **single image**.
2) SequenceHandler() infers that m_recorder is ImageRecorder. Thus, m_renderMode is set to rmDirectSave, because expected result is a sequence of images. Scene is rendered at the every m_captureInterval right between the two corresponding time steps. Relevant time steps are bounded by the m_captureTimeStepBegin and m_captureTimeStepEnd. SequenceHandler() further decides depending on the m_recordingInteractive variable:
    a) **Non-interactive image sequence capturing** is handled directly by the SequenceHandler(). Time steps are cycled through by the SetTimeStep() in the loop, whose each iteration calls the Render() function. In order to improve performance and responsiveness of the application, scene is not rendered, only a simple QProgressDialog is shown (m_dialogProgress) and the application message loop is emptied during each iteration.
    b) **Interactive image sequence capturing** is handled by the AnimateTimeStep() function. SequenceHandler() only sets the initial time step, toggles the m_recordingEnabled flag and calls the AnimationStart(). AnimateTimeStep() will be normally preparing and animating time steps, each time checking for the correct combination of m_recordingEnabled and m_renderMode, which allows calling the Render().

3) `SequenceHandler()` infers that `m_recorder` is `VideoRecorder`. Thus, `m_renderMode` is set to `rmBufferSave`, because the expected result is a **video file**. Firstly, the frame rate is retrieved from the `m_recorder` by calling the `VideoRecorder::GetFPS()` and saved into the `m_fps` variable. Then, `m_recorder` is started by calling the `VideoRecorder::Start()`, which takes the destination file path (`m_captureFilePath`, `m_captureFileName`) and the resolution (`m_captureWidth`, `m_captureHeight`) as arguments. From now on, `m_recorder` acts as a consumer, who progressively fetches visual data from the `G_GRVideoBuffer` and saves it into the file. Producer will be the `Render()` function together with the `AnimateTimeStep()`, which is immediately invoked after calling the `AnimationStart()`. Note that the difference between interactive and non-interactive capturing (`m_recordingInteractive`) is not as significant as with the image sequences – it is now handled by the same code in the `AnimateTimeStep()` and the only difference is the visibilty of the scene and the utilization of the `m_dialogProgress`. Rendering in `AnimateTimeStep()` is done efficiently – after checking the correct combination of `m_recordingEnabled` and `m_renderMode`, nodes are animated only at the positions required by a video frame rate. Thus, instead of calling `m_animationTimeLine->start()`, position of every `QGraphicsItemAnimation` from the `m_animationBuffer` is explicitly set in the loop that iterates as many times as there are frames that fit, according to the `m_fps`, into the `m_animationTimeLine->duration()`. Note that the position is still inferred from the `m_animationTimeLine`, because it might not be linearly dependent on the time. Every loop iteration calls the `Render()` and empties the application message loop due to the responsiveness of GUI. After the loop is finished, `AnimationStepDone()` signal is emitted. When all time steps are captured, `AnimationStop()` function calls the `VideoRecorder::Stop()` from the `m_recorder` in order to safely finish the recording. This approach ensures that the video is rendered as fast as possible and the quality of output is not dependent on the processor speed (e.g. dropped frames). However, this also implies that, due to the processor speed, interactive video capturing is not real time – from the user's point of view, it might be either extremely slow or extremely fast, both of which are not ideal for the interactivity.

`Render()` function can be described in three steps:

- **Preparing** `QPainter`:
  - In `rmDirectSave` mode, painter is constructed from the `QPaintDevice`, which is retrieved from the `m_recorder` by calling the `ImageRecorder::GetPaintDevice()`. Note that, apart from the `m_captureWidth` and `m_captureHeight`, which are quite expectable, the function takes also the `m_captureFilePath` and `m_captureFileName` as arguments, because some paint devices save data directly to the persistent storage while rendering (e.g. XML file in the case of the SVG file format).
  - In `rmBufferSave` mode, painter is constructed from the `QImage`, which itself is constructed according to the `m_captureWidth` and `m_captureHeight`.
- **Rendering** image. Since viewport of the `GraphView` might have different aspect ratio than the one calculated from the `m_captureWidth` and `m_captureHeight`, a rectangle representing the exposed area of the scene must be appropriately extended. At first, the rectangle is aligned to the top left corner of the viewport and then either its width or height is increased to match the output ratio. This ensures that the resulting image will certainly contain intended part of the scene and will have the correct aspect ratio. Finally, exposed area is rendered by the painter into its embedded device.

- **Saving** `QPaintDevice`:
  - In `rmDirectSave` mode, device is saved by the `ImageRecorder::SaveImage()`. Note that, this function might actually do nothing since data might have been already saved by the device itself.
  - In `rmBufferSave` mode, the device, which in fact is a `QImage`, is saved into the `G_GRVideoBuffer` that is guarded by two semaphores – `G_GRVideoFree` and `G_GRVideoUsed` – in a manner of producer/consumer synchronization. Current buffer position is stored in the `m_bufferPosition` variable. Note that acquiring the semaphore is regularly interrupted by emptying the application message loop due to the responsiveness of GUI.

4.6.9 Video Encoding

Video encoding is implemented in the `FFmpegVideoRecoder` class, which inherits `VideoRecorder` interface. Class embeds the `EncoderThread`, which is derived from the `QThread` and intended as a consumer for the `G_GRVideoBuffer`. Moreover, the class is heavily dependent on API functions of the FFmpeg video library. In order to manage and distribute data in a uniform way among these API functions and the embedded thread, class defines the structure `Data`, which contains all required parameters and FFmpeg data structures. `Data` structure, created in the `FFmpegVideoRecoder` constructor, is accessible through the `m_data` or `EncoderThread::m_recdata` pointer (since both classes refer to the same data, `FFmpegVideoRecoder` is the owner responsible for the deletion). Following description focuses on how the encoding is handled in detail. In order to provide a clear explanation, API calls are, in most cases, not mentioned explicitly. It should be noted that due to the lack of the proper FFmpeg documentation, API calls are deduced from the `output-example.c` provided in FFmpeg redistributable package [2].

Let us assume that the `Data` structure already contains some initialized entries, which have been set by the user via slots connected to the widget provided to the `CaptureDialog`:

- **Initialization** of the `Data` structure is further done by the `Start()` function:
  a) File suffix and video format are inferred from the `m_data->formatString`. Both `m_data->format` and `m_data->context` structures are initialized by FFmpeg API calls. Since FFmpeg guesses the format from a given string, it is easy to add more video formats if needed.
  b) Structure `m_data->stream` is initialized by calling the `CreateStream()` function, which further calls API. Note that the `CreateStream()` also contains all codec settings, some of which are currently hard coded.
  c) By a series of API calls, `OpenVideo()` function opens a codec, whose settings were just set in the `CreateStream()`. After that, the encoding buffer is allocated (`m_data->buffer`, `m_data->bufferSize`). Finally, a pair of `AVFrame` structures is initialized by the `AllocateFrame()` function – whereas the format of `m_data->frameTemp` must be compatible with `QImage` (RGB), `m_data->frameFinal` is intended as a codec input (where the most suitable format is YUV).
  d) Output file is created and opened (`Start()` takes the path and name as arguments). Format header is written into the file by the API call.
  e) Encoding thread is invoked by calling the `m_thread->start()`, which effectively starts the `EncoderThread::run()` on the different thread.
- **Encoding** of images from the `G_GRVideoBuffer` is done by the `EncoderThread::run()` function running on the worker thread:

a) First of all, relevant data are copied from the `EncoderThread::m_recdata` into the local variables of the `EncoderThread::run()` function. Thus, data (mainly pointers to FFmpeg structures) are now located on the local stack and one level of indirection is avoided. Note that this is **not** intended as a protection against race conditions – FFmpeg structures are still accessible from both threads becuase only pointers are copied. Moreover, copying is not guarded by any mutex. None of this is problem, because the worker thread has an exclusive access assuming the current architecture (main thread only prepares those structures before starting the worker thread – there is no concurrency between them concerning the shared data).
b) Function enters the infinite loop that is encoding the images from the `G_GRVideoBuffer` until the buffer is empty and the `EncoderThread::m_terminate` flag is toggled. Since anything that is done in the loop has direct impact on the encoding speed, the loop must be implemented efficiently. At first, every `QImage` (RGB) is copied from the `G_GRVideoBuffer` into the local `frameTemp` (RGB). Note that the copying is the only thing guarded by semaphores. Thus, the main thread, as a producer, is not slowed down by waiting on the actual frame encoding.
c) Visual data in the `frameTemp` are converted from RGB to YUV format and saved into the `frameFinal`. Highly optimized algorithm for this conversion is implemented in the `swscale()` function provided by the GPL version of FFmpeg.
d) Image stored in the `frameFinal` is encoded by a chosen codec and saved into the output file (both actions done by API calls). Note that the encoding function utilizes the `buffer`, whose `bufferSize` is set to 8MB by default (might be changed by the user through GUI). If the `buffer` is too small, recording either immediately fails or the resulting video will be corrupted.

- **Deinitialization** and memory deallocation is done by the `Stop()` function:
  a) `EncoderThread::SafelyTerminate()` is called, which effectively terminates the worker thread by setting its `m_terminate` flag.
  b) Format trailer is written into the output file by the API call.
  c) `CloseVideo()` and `DestroyStream()` deletes all allocated FFmpeg structures.
  d) Output file is closed.

4.7 Main Window

`GraphRec` is a class, which inherits `QMainWindow` and has its GUI part accessible through the `m_ui` pointer. Class represents the main application window consisting of the menu bar (`m_ui->menuBar`), tool bar (`m_ui->toolBar`), status bar (`m_ui->statusBar`) and the central widget (`m_ui->centralWidget` or `m_ui->gridLayout`). Menu bar is composed of several submenus and actions (`QAction`). Tool bar provides a subset of frequently used menu actions. Note that the tool bar is designed by a programmer and cannot be edited by the user at runtime. During the runtime, it is only possible to turn the tool bar on/off or dock it to various sides of the window. Status bar contains two labels – one for displaying the application status (`m_labelStatus`) and the other for displaying the validator name (`m_labelValidator`). Status bar also allows posting some temporary messages over the labels.

Initially, the central widget is empty and almost all parts of the window and menu are disabled. When user clicks on the **File – Open** button, function `on_actionOpen_triggered()` is called. At first, it invokes `OpenDialog`, which is later described in its own subsection. How-

ever, when the dialog is confirmed by the user, the function searches its `acceptedSolutions` (list of `OpenDialog::SolutionInfo` structures) and process them one after the other. Each `SolutionInfo` provides file name, file location and names of the preferred parser, validator and layouter. The function opens the file and creates a new `GraphView` by passing it the opened file and the information from the `SolutionInfo`. Created `GraphView` is then added as a tab into either existing or newly constructed `QTabWidget` (depending on whether there is already one). Newly constructed `QTabWidget` is appended into the `m_tabWidgets` list and embedded into the central widget of the window. Function `ResetControls()` is then called in order to enable/disable menu actions. When user closes the tab (`TabCloseRequested()`), its `GraphView` is destroyed and the tab is removed from its tab widget. If the tab widget has no more tabs it is also destroyed and removed from both `m_tabWidgets` and central widget. Function `ResetControls()` is called again.

Previous paragraph hints that each tab widget can contain more than one `GraphView` instance. However, it is even more complicated, because the tab widget can be split into more tab widgets. Splitting tab widgets is described in the next subsection. Anyway, since the majority of menu actions are only handles that further calls functions in the `GraphView`, it is clear that there must be some mechanism for delivering these calls to the right `GraphView` instance. Note that this mechanism must deliver calls also in the opposite direction (from the `GraphView` to the `GraphRec`) in order to update labels and certain menu selections. Whole problem is resolved by the usage of signal/slot mechanism. Many of `GraphRec` actions or signals are connected to the `GraphView` slots and vice versa. Every time the `GraphView` is changed (it can be done by either the tab change or the focus change), `FocusedGraphViewChanged()` is called. The function disconnects the `GraphRec` from the `m_currentGraphView` and then reconnects it to the `GraphView` provided in an argument `who`. After the reconnection is done, connected `GraphView` is requested to emit its status by calling its function `UpdateConnections()`. In order to make all this working, every `GraphView` has its `HasFocus()` signal connected to the `GraphRec` immediately after the construction.

`GraphRec` class is also partially responsible for the file saving. When the user clicks on either **File – Save** or **File – Save All** button, the function `on_actionSave_triggered()` or `on_actionSaveAll_triggered()` is called. Both functions invoke the `QFileDialog` while filling its file format combo box with names of the `Saver` servants obtained from the `Factory`. After the user chooses the path, name and suffix, the file is opened and the corresponding saver is built. At first, saver is initialized by the `Saver::Open()`. Then, it is passed to the `SaveContext()` function of the currently connected `GraphView` (or to every `GraphView` in the case of the **Save All** action). In the end, file is terminated by the `Saver::Close()` and closed.

4.7.1 Splitting

Central area of the main window is represented by the `m_ui->centralWidget` with the `m_ui->gridLayout` applied on it. In a basic case, there is only one instance of `QTabWidget`, whose parent is already mentioned `m_ui->gridLayout`. Let us assume that the tab widget contains more than one tab (each tab containing instance of a `GraphView`) and the currently selected tab is not the first one (left most). Then, it is possible to split this tab widget by the `Split()` function into two tab widgets – first containing tabs up to (not including) the selected tab, second containing the rest. Original tab widget is then replaced by the instance of `QSplitter` (supports a movable boundary between its children) into which those two new instances of `QTabWidget` are added. Described process can be repeated on the arbitrary level of the hierarchy represented by a binary tree (Diagram 5), whose inner vertices are instances of the `QSplitter` and leafs are instances of the `QTabWidget`. The tab widget that is going to be split

is inferred from the parent pointer of the currently focused tab (`m_currentGraphView`). Every split can be done either horizontally or vertically.

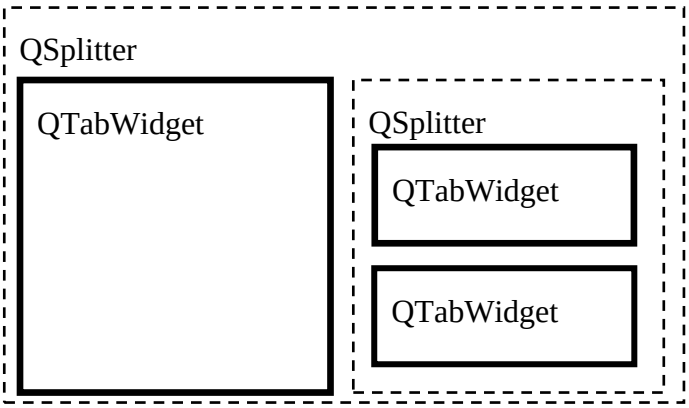


Diagram 5. Binary tree hierarchy of the *tab splitting*.

There is also inverse operation to the splitting. `Unsplit()` function takes a `QSplitter` as an argument and, by recursively calling itself on the splitter's children, it returns the `QTabWidget` containing all tabs from accessible leafs. If the user unsplits a tab widget by calling the `on_actionUnsplit_triggered()`, its parental `QSplitter` is passed to the `Unsplit()` function, effectively merging the focused tab widget with its unambiguous neighbour branch (vertical or horizontal). Resulting tab widget is put on the place where the destroyed parental splitter originally was. As for the tab closing, if the tab is closed and there are no more tabs in the tab widget, the widget is destroyed and its parental splitter is replaced by the sibling (either `QSplitter` or `QTabWidget`). Note that both `Split()` and `Unsplit()` are maintaining the current set of tab widgets in the `m_tabWidgets` list.

In the situation, when there is more than one tab visible to the user, it is possible to run animation on all of them at once. As was explained in the animation description, every animation is independently timed by its own `QTimeLine`, which behaves similarly to a `QTimer`. Since timers are not precise on the non-realtime operating system, it is difficult to synchronize more of them running in parallel. This implies that two parallel animations with the same duration might get desynchronized after a few time steps. To prevent this from happening, the user can toggle **Animation – Synchronize All** action, which modifies behavior of the certain functions. When the synchronization is toggled, `on_actionPlayAll_triggered()` function adds all foreground `GraphView` instances into the `m_synchronizedGraphViews` and, instead of calling `GraphView::AnimationStart()`, it calls `GraphView::AnimationStep` on all of them. Since the construction in `on_actionOpen_triggered()`, every `GraphView` has its `Stepped()` signal connected to the `GraphRec::SynchronizeAll()` slot. Thus, when any `GraphView` finishes an animation of the time step, `SynchronizeAll()` is called. It evaluates whether all members of the `m_synchronizedGraphViews` have already finished their step and if so, it calls the `GraphView::AnimationStep` on all of them to continue the animation. Note that in the case of different animation duration among tabs, synchronization always waits for the slowest `GraphView` leaving others stopped for a while. In order to assure consistent behavior with other features, `m_synchronizedGraphViews` is also managed by other menu actions related to the animation (mainly removing of some `GraphView` instances from the list).

4.7.2 Open Dialog

`OpenDialog` is a class, which inherits `QDialog` and has its GUI part accessible through the `m_ui` pointer. GUI is represented by the two `QTreeWidget` instances (`m_ui->listFound`, `m_ui->listChosen`) and some buttons. Both tables have its header derived from a `headerTemplate`, a list of `HeaderItem` values from the `Parser` interface, specified in the `OpenDialog` constructor. Note that table columns containing important but not interesting information for the user are hidden by the `HideColumn()` function.

The handler of **Add files** button (`on_buttonAddFiles_clicked()`) invokes `QFileDialog` while filling its file format combo box with the names of `Parser` servants obtained from the `Factory`. After the user chooses appropriate format and files to open, corresponding `Parser` is built and its `ParseFile()` function is called for the every selected file. The function call returns the `QTreeWidgetItem` acting as a root for multiple other `QTreeWidgetItem` instances each of which represents one table row showing the statistics for one graph. Since the `headerTemplate` is also passed to the `ParseFile()`, it is ensured that the row item order will always correspond to the header item order. Before appending the root item to the `m_ui->listFound`, root is passed to the `FillMissingInfo()`, which fills into its children some default values that might not be found in the input file (mainly layouter and validator name). When all items are added into the `m_ui->listFound` table, user can select arbitrary number of them and move them to the `m_ui->listChosen` table or vice versa. Handlers of the mouse double clicks and handlers of the corresponding buttons (>>>, <<<) utilize the `Move()` function. Since the user can select an arbitrary combination of multiple root items (when intending to open all graphs in the file) or only some of their children, the `Move()` must very carefully create, destroy and copy items in both tables. After the user has finished the selection and hits the **Open** button, `on_buttonOpen_clicked()` builds a `SolutionInfo` structure for every item in the `m_ui->listFound` and appends it to the `acceptedSolutions` list. `SolutionInfo` is intended as a hint for the `GraphView` constructor – where to find a graph (file name, file position), how to parse it (parser name), how to interpret it (validator name) and how to embed it (layouter name). Note that the file name is not located in every table item, but only in the tool tips of root items. Publicly accessible `acceptedSolutions` list acts as output storage of the `OpenDialog`.

There is one tiny feature with quite complicated implementation that should be described. Since the input file might not specify validators for its graphs, attempt to open larger file with some default validator might appear to be very slow because of the intensive error logging. In that case, user would be also forced to change the validator manually in every opened `GraphView`. Assuming the user knows what validator should be used for a given file; there is a possibility to change it directly in the `OpenDialog` before the actual opening. User selects multiple (root or normal) items in the table and by the right mouse button click invokes `customContextMenuRequested()`, which is a signal of a `QTreeWidget` connected to the `ShowContextMenu()` slot of the `OpenDialog`. `ShowContextMenu()` opens the `m_contextMenu` (`QMenu` instance), whose actions correspond to the validator names obtained from the `Factory`, and sets the `m_senderTreeWidget` to remember what `QTreeWidget` the context menu was requested from. Every menu action is connected through the `m_signalMapperValidators` (`QSignalMapper` instance) to the `SetValidator()` slot. `SetValidator()` discovers all selected items in the `m_senderTreeWidget` table and alters their validator column with a given validator name.

4.7.3 Help Dialog

`HelpDialog` is a class, which inherits `QDialog` and has its GUI part accessible through the `m_ui` pointer. Class acts as a simple help viewer of the file `index.html` in the folder `/../doc/` (relatively to the executable). Since it is expected that the help file is encoded in HTML[15], the class uses a `QTextBrowser` (`m_ui->textBrowser`) for the rendering. `HelpDialog` currently does not support the text searching and indexing. Thus, the help file should contain the table of contents and should be well structured with a usage of hypertext links. Note that the `HelpDialog` is non-modal and has no parent (when created by the `GraphRec`) in order to be accessible even when other modal dialog is shown.

4.8 Persistent Settings

GraphRec uses Qt multiplatform approach to save settings persistently between user sessions. `QSettings` class provides abstraction for the uniform access to the settings, which are saved in various locations – either the registry on Windows (`HKEY_CURRENT_USER\Software\` or `HKEY_LOCAL_MACHINE\Software\` or `HKEY_LOCAL_MACHINE\Software\WOW6432node`) or the conf files on Unix (`$HOME/.config/` or `/etc/xdg/`). Because the application name (GraphRec) and domain (koupy.net) is set explicitly in the `GraphRec` class constructor, it is possible to work with the settings anywhere in the code without specifying it again. It is sufficient to make the `QSettings` instance on the stack and then call its `value()` function for fetching entries or `setValue()` function for saving entries. GraphRec settings are currently located either in the constructors of GUI classes or in the functions that prepares/deletes modal dialogs.

[15] HyperText Markup Language (http://www.w3.org/TR/REC-html40/)

5 Conclusion

Presented thesis described the implementation of the visualization tool for the entity movement on a graph. Initially, the design phase was inspected including the choice of suitable algorithms and technologies. Following section introduced a comprehensive guide for users. The last section explored the application architecture extensively.

Resulting open-source application *GraphRec* [11] is able to run on multiple platforms, contains flexible GUI and provides various features ranging from the graph embedding to the video capturing. Because a similar tool is not currently available, the code base was written in an extensible modular manner to be possibly further reused.

The presented visualization proved itself an effective way for discovering the nature of expected redundancies in solutions. As stated in [19], acquired knowledge was used to formalize redundancies and to design methods for their removal.

Although all planned features were successfully implemented, there are still possibilities for the future development. Currently, the interface between the visualization tool and problem solvers is represented only by the input and output file formats. It would be possible to wrap a GUI around the problem solver and to design a common interface to enable the communication between both components. Such cooperation could lead to the creation of the module for simulation management providing some advanced statistics. As an example, after the removal of a certain vertex from a graph, the application could immediately run the simulation and present an impact of such action.

Another opportunity to extend the application is to implement the parser of the file format used by some advanced graph embedding application (for example *Graphviz* [3]), so the graph layout could be created by a specialized tool. Finally, it would be challenging to rewrite the animation engine to be scalable to much larger graphs – possibly thousands to millions of vertices.

Bibliography

[1] Andrews, G. R. *Producer/Consumer Synchronization*. Foundations of Multithreaded, Parallel, and Distributed Programming, pp. 56-57, Addison Wesley, 1999.

[2] Bellard, F. *FFmpeg – a complete, cross-platform solution to record, convert and stream audio and video.* Project web page, http://ffmpeg.org, 2010.

[3] Bilgin, A., Ellson, J., Gansner, E., Hu, Y., Koren, Y., North, S. *Graphviz - Graph Visualization Software.* Project web page, http://www.graphviz.org, 2010.

[4] Cormen, T. H., Leiserson, Ch. E., Rivest, R. L. *The Floyd–Warshall algorithm.* Introduction to Algorithms (1st ed.), pp. 558–565, MIT Press and McGraw-Hill, 1990.

[5] Courtney, D. *InstallJammer – a multiplatform GUI installer*. Project web page, http://www.installjammer.com, 2010.

[6] Fruchterman, T. M. J., Reingold, E. M. *Graph Drawing by Force-Directed Placement.* Software: Practice and Experience, Volume 21, pp. 1129-1164, John Wiley & Sons, 1991.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J. M. *Strategy pattern*. Design Patterns: Elements of Reusable Object-Oriented Software, pp. 349–359, Addison-Wesley, 1994.

[8] Heesch, D. *Doxygen – source code documentation generator tool*. Project web page, http://www.doxygen.org, 2010.

[9] Kamada, T., Kawai, S. *An algorithm for drawing general undirected graphs.* Information Processing Letters, Volume 31, pp. 7-15, Elsevier, 1989.

[10] Kornhauser, D., Miller, G. L., Spirakis, P. G. *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.

[11] Koupý, P. *GraphRec – a visualization tool for entity movement on graph.* Project web page, http://www.koupy.net/graphrec.php, 2010.

[12] Nokia Corp. *Qt – Cross-platform application and UI framework.* Project web page, http://qt.nokia.com, 2010.

[13] Nokia Corp. *Qt Online Reference Documentation.* Web page, http://qt.nokia.com/doc/, 2010.

[14] Oberhumer, M. F. X. J., Molnar, L., Reiser, J. F. *UPX – the Ultimate Packer for eXecutables*. Project web page, http://upx.sourceforge.net, 2010.

[15] Ratner, D., Warmuth, M. K. *Finding a Shortest Solution for the N×N Extension of the 15-PUZZLE Is Intractable*, Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann Publishers, 1986.

[16] Ryan, M. R. K. *Exploiting subgraph structure in multi-robot path planning.* Journal of Artificial Intelligence Research (JAIR), Volume 31, pp. 497-542, AAAI Press, 2008.

[17] Surynek, P. *An Application of Pebble Motion on Graphs to Abstract Multi-robot Path Planning.* Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), pp. 151-158, IEEE Press, 2009.

[18] Surynek, P. *An Optimization Variant of Multi-Robot Path Planning is Intractable.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), article submitted for publication, 2010.

[19] Surynek, P., Koupý, P. *Vizualizace jako prostředek k získání znalostí o kvalitě řešení problémů pohybu po grafu*. Proceedings of the Conference Znalosti 2010, pp. 129-141, Nakladatelství Oeconomica, 2010.

[20] Süli. E., Mayers, D. *Relaxation and Newton's Method*. An Introduction to Numerical Analysis, pp. 19-24, Cambridge University Press, 2003.

Appendix A

Included CD-ROM structure

`/build/`

Automated build process for the Windows platform. All necessary packages are included on the CD-ROM. Before running the build script, please read the enclosed instruction file.

`/doc/`

PDF[16] documents describing GraphRec. Among included is the original specification, separate user's guide, separate programmer's documentation, conference article [19] and this thesis. The thesis is also placed in the root directory of the CD-ROM.

`/doxy/`

Documentation generated automatically by the *Doxygen* tool. Includes several UML[17] diagrams and call graphs.

`/redist/`

Redistributable packages of GraphRec for Windows and Linux platform. Both versions are provided in uncompressed, archived and installer-based package.

[16] Portable Document Format (http://www.adobe.com/devnet/pdf/pdf_reference.html)
[17] Unified Modeling Language (http://www.omg.org/spec/UML/2.0/)

Appendix B

Multirobot File Format

Multirobot was the initial file format supported by GraphRec. It is derived from the output of the planning software developed by my supervisor. Consequently, some data fields are not relevant for GraphRec. On the other hand, GraphRec puts some optional extensions to the original format (e.g. positioning and coloring). GraphRec opens either the original or the extended format but saves only the extended one. Format specification consists of the grammar written in EBNF[18] (with case insensitive terminals) followed by the semantic description and a short example. Lines that are irrelevant for GraphRec are substituted by the undefined auxiliary non-terminal (grammar is thus incomplete).

```
file = { graph } , '<EOF>' ;
graph = { aux } , [ id ] ,
        { aux } , vertex block ,
        { aux } , edge block ,
        { aux } , [ circle block ] ,
        { aux } , [ validator block ] ,
        { aux } , [ color block ] ,
        { aux } , [ position block ] ,
        { aux } , solution block ,
        { aux } , length ,
        { aux } ;
id = 'id:' , uint , nl ;
vertex block = 'V =' , nl , { vertex }, nl ;
vertex = '(' , uint , ':' , uint , ')' ,
         '[' , sint , ':' , sint , ':' , sint '] ' ,
         { uintwh } , nl ;
edge block = 'E =' , nl , { edge }, nl ;
edge= '{' , uint , ',' , uint , '} (' , uint , ')' , nl ;
circle block = 'C =' , nl , { circle }, nl ;
circle = uintwh , '(' , uint , ',' , uint , '): ' , { uintwh } ,
         ' [' , { uintwh } , '] ' , '{' , { uintwh } , '}' , nl ;
validator block = 'MOD =' , nl , ( 'M:IMMEDIATE' | 'M:TRANSITIVE' ) , nl ;
color block = 'COL =' , nl , [ scene ] , [ borders ] ,
              [ highlight ] , { color } , nl ;
scene = 'B_SCN:A:' , color value , nl ;
borders = 'P_BRD:A:' , color value , nl ;
highlight = 'P_HLT:A:' , color value , nl ;
color = ( 'B' | 'P' ) , '_' , ( 'EMP' | 'INH' | 'FIN' ) ,
        ':' , uint , ':' , color value , nl ;
position block = 'POS =' , nl , [ matrix ] , [ angle ] ,
                 [ center ] , { position } , nl ;
matrix = ' MATRIX:' , float , ':' , float , ':' , float , ':' ,
         float , ':' , float , ':' , float , nl ;
angle = 'ANGLE:' , float , nl ;
center = 'CENTER:X' , float , ':Y' , float , nl ;
position = uint , ':X' , float , ':Y' , float , nl ;
solution block = 'Solution' , nl , { move } , nl ;
move = uintwh , '---> ' , uintwh , [ move extension ] ,
       '(' , uint , ')' , nl ;
```

[18] Extended Backus–Naur Form
(http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip)

```
move extension = '[' , sint , ' ---> ' , sint , '] ' ;
length = 'Length:' , uint , nl ;
aux = ? unknown additional information ? , nl ;
nl = new line , { new line } ;
new line = '<LF>' | '<CR>' | '<LF>' , '<CR>' | '<CR>' , '<LF>' ;
numeral = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
uint = numeral , { numeral } ;
uintwh = uint , ' ' ;
sint = [ '+' ] , uint | '-' , uint ;
float = sint , '.' , uint | sint ;
hnumeral = numeral | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' ;
color value = '#' , hnumeral , hnumeral , hnumeral , hnumeral ,
              hnumeral , hnumeral ;
```

- `(<node_id>:<IGNORED>)[<initial_entity_id>:<IGNORED>:<IGNORED>]`
  stands for the vertex definition. Entity identifications less or equal zero are reserved for the empty nodes.
- `{<source_node_id>,<destination_node_id>} (<IGNORED>)`
  stands for the edge definition.
- `<circle_id> (<source_node_id>,<destination_node_id>): <whole_circle> [<new_arc>] {<existing_arc>}`
  stands for the circle definition. Last three tokens are lists of node identifications. Circle is created by joining the new arc to the source and destination node, both of which are joints of an existing arc.
- `M:IMMEDIATE` selects the validator for *pebble motion on a graph*.
- `M:TRANSITIVE` selects the validator for *multi-robot path planning*.
- `B_SCN:A:<color>` is a background color for the scene.
- `P_BRD:A:<color>` is a color for edge strokes and node borders.
- `P_HLT:A:<color>` is a color for edge highlighting.
- `B_EMP:<node_id>:<color>` is a background color for an empty node.
- `P_EMP:<node_id>:<color>` is a foreground color for an empty node.
- `B_INH:<entity_id>:<color>` is a non-final background color for the entity.
- `P_INH:<entity_id>:<color>` is a non-final foreground color for the entity.
- `B_FIN:<entity_id>:<color>` is a final background color for the entity.
- `P_FIN:<entity_id>:<color>` is a final foreground color for the entity.
- `MATRIX:<m11>:<m12>:<m21>:<m22>:<dx>:<dy>`
  is a transformation matrix of the scene.
- `ANGLE:<angle>` is a rotation angle of the scene (in degrees). Currently **obsolete**, because the scene is no longer rotated – rotation is saved in the node positions instead.
- `CENTER:X<x>:Y<y>` is a point in the scene aligned to the center of the viewport.
- `<node_id>:X<x>:Y<y>` is a position of the specified node.
- `<source_node_id> ---> <destination_node_id> [<IGNORED> ---> <IGNORED>] (<time_step>)`
  defines one particular move of an unspecified entity between the two specified nodes at the specified time step.

```
id:1
V =
(1:0)[1:0:0]
(2:0)[2:0:0]
(3:0)[3:0:0]
E =
{1,2} (0)
{2,3} (0)
{3,1} (0)
MOD =
M:TRANSITIVE
COL =
B_SCN:A:#ffffff
P_BRD:A:#000000
P_HLT:A:#00ffff
B_EMP:1:#ffaa00
B_EMP:2:#ffaa00
B_EMP:3:#ffaa00
P_EMP:1:#000000
P_EMP:2:#000000
P_EMP:3:#000000
B_INH:1:#0000ff
B_INH:2:#ff0000
B_INH:3:#00ff00
P_INH:1:#ffffff
P_INH:2:#ffffff
P_INH:3:#ffffff
B_FIN:1:#00007f
B_FIN:2:#aa0000
B_FIN:3:#005500
P_FIN:1:#ffffff
P_FIN:2:#ffffff
P_FIN:3:#ffffff
POS =
MATRIX:1.68179:0:0:1.68179:0:0
ANGLE:0
CENTER:X-7.13524:Y-35.6762
1:X-18.7568:Y-8.08399
2:X4.0907:Y-71.4452
3:X-62.2215:Y-59.842
Solution
2 ---> 3 [0 ---> 0] (0)
3 ---> 1 [0 ---> 0] (0)
1 ---> 2 [0 ---> 0] (0)
3 ---> 1 [0 ---> 0] (1)
2 ---> 3 [0 ---> 0] (1)
1 ---> 2 [0 ---> 0] (1)
3 ---> 1 [0 ---> 0] (2)
2 ---> 3 [0 ---> 0] (2)
1 ---> 2 [0 ---> 0] (2)
Length:9
```

Appendix C

GraphRec File Format

GraphRec file format is a refined alternative to the Multirobot file format. Format is specified in XML and is designed against the requirement to provide better locality and encapsulation of the information than the Multirobot format. Format is specified by the semantic description and a short example. Optional tags or attributes are enclosed in square brackets.

File is composed from the XML header, document type `<!DOCTYPE graphrec>` and a single root element `<graphrec version="<version_number>"></graphrec>`. Root element acts as a container for one or more `<solution [id="<solution_id>"]></solution>` definitions, which are further composed from the following elements:

- `[<scene [bg="<background_color>"]></scene>]` contains the scene definition:
  - `[<viewport x="<x_position>" y="<y_position>"/>]`
    specifies a point in the scene aligned to the center of the viewport.
  - `[<matrix m11="<m11>" m12="<m12>" m21="<m21>" m22="<m22>" dx="<dx>" dy="<dy>"/>]` defines the transformation matrix of the scene.
- `<graph></graph>` contains the graph definition:
  - `<entity id="<entity_id>" [bg="<background_color>"] [bgf="<final_background_color>"] [fg="<foreground_color>"] [fgf="<final_foreground_color>"]/>`
    stands for entity definition. Entity identification must be greater than zero.
  - `<node id="<node_id>" [ent="<initial_entity_id>"] [x="<x_position>" y="<y_position>"] [bg="<background_color>"] [fg="<foreground_color>"] [bnd="<boundary_color>"]/>`
    stands for vertex definition. Entity identification for the empty node is zero.
  - `<edge n1="<first_node_id>" n2="<second_node_id>" [ln="<line_color>"] [hgl="<highlight_color>"]/>`
    stands for the edge definition.
- `<scenario [validator="<validator_name>"]></scenario>`
  contains all movements of the solution:
  - `<move tms="<time_step>" src="<source_node_id>" dst="<destination_node_id>"/>`
    defines one particular move of an unspecified entity between the two specified nodes at the specified time step.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE graphrec>
<graphrec version="1.0">
    <solution id="1">
        <scene bg="#ffffff">
            <viewport x="-1.18921" y="-29.7302"/>
            <matrix m11="1.68179" m12="0" m21="0"
                    m22="1.68179" dx="0" dy="0"/>
        </scene>
        <graph>
            <entity id="1" bg="#0000ff" bgf="#00007f"
                           fg="#ffffff" fgf="#ffffff"/>
            <entity id="2" bg="#ff0000" bgf="#aa0000"
                           fg="#ffffff" fgf="#ffffff"/>
            <entity id="3" bg="#00ff00" bgf="#005500"
                           fg="#ffffff" fgf="#ffffff"/>
            <node id="1" ent="1" x="-18.7568" y="-8.08399"
                         bg="#ffaa00" fg="#000000" bnd="#000000"/>
            <node id="2" ent="2" x="4.0907" y="-71.4452"
                         bg="#ffaa00" fg="#000000" bnd="#000000"/>
            <node id="3" ent="3" x="-62.2215" y="-59.842"
                         bg="#ffaa00" fg="#000000" bnd="#000000"/>
            <edge n1="1" n2="2" ln="#000000" hgl="#00ffff"/>
            <edge n1="2" n2="3" ln="#000000" hgl="#00ffff"/>
            <edge n1="3" n2="1" ln="#000000" hgl="#00ffff"/>
        </graph>
        <scenario validator="Multirobot">
            <move tms="0" src="3" dst="1"/>
            <move tms="0" src="1" dst="2"/>
            <move tms="0" src="2" dst="3"/>
            <move tms="1" src="1" dst="2"/>
            <move tms="1" src="2" dst="3"/>
            <move tms="1" src="3" dst="1"/>
            <move tms="2" src="1" dst="2"/>
            <move tms="2" src="2" dst="3"/>
            <move tms="2" src="3" dst="1"/>
        </scenario>
    </solution>
</graphrec>
```