Journal homepage: https://devma.com.ua/en

DEVELOPMENT MANAGEMENT

UDC 519.1
DOI: 10.57111/devt.20(4).2022.36-44

Vol. 21, No. 4. 2022

Visualization of algorithms on graphs with a large number of vertices: The features of applications design

Liudmyla Gryzun[1*], Oleksandr Shcherbakov[1], Yurii Parfonov[1], Liliia Bodnar[2]

[1]Simon Kuznets Kharkiv National University of Economics
61166, 9A Nauka Ave., Kharkiv, Ukraine

[2]South Ukrainian National Pedagogical University named after K.D. Ushynsky
65020, 26 Staroportofrankivska Str., Odessa, Ukraine

Abstract. The task of visualization of large graphs as a special data structure and algorithms on them is considered by scientists and practitioners as a complex and non-trivial problem. The analysis of scientific works and existing software applications that implement similar functions of the subject domain testifies the relevance of expanding exploration in the lines of identifying the features of the development of applications for the visualization of large graphs and algorithms on them. The formulation of features and recommendations for the development of such software and presentation of the software module designed by the authors is the aim of the article. In the course of the work, the main features of the development of a program for the visualization of graphs with a large number of vertices were identified and formulated using methods of analysis and graph theory. Special recommendations on the essence of each of the stages of development of such applications were provided and those steps that are most important for developers in terms of the complexity of processing and visualization of large graphs, metrics of their layout in the application screen, etc. were identified. A software module developed by the authors, that provides a unified application programming interface for visualizing any algorithm on graphs, which allows to save time working on utility software and focus more on solving algorithmic problems is also presented. The presented module was developed by the authors taking into account the identified recommendations. A comparative analysis of the developed software module and analogues was carried out, which proved the extended functionality of the module for the visualization of graphs with a large number of vertices. The module is a practically valuable tool for data structures researchers and other experts working on graph algorithms, since it enables data visualization at debugging software and simplifies the analysis of large data structures

Keywords: large graphs; problems of algorithms visualization; a module for visualization of algorithms on graphs; graph layout; unified Application Programming Interface

Article's History: Received: 29.08.2022; Revised: 24.11.2022; Accepted: 20.12.2022

INTRODUCTION

Graph theory makes a powerful theoretical basis for modelling relationships between objects and solving variety of practical problems in different subject domains. In computer science, graphs are one of the most common and widely used data structures. Current complex scientific and technical problems expect to store and process huge data amount for their solutions. By far, a lot of graph algorithms have been developed which enable to solve great range of problems: from search of the shortest ways to the optimisation and numerical problems.

There are some cases, when it is quite complicated to use or modify the algorithm without its visualization which gets even more urgent when the problem expects using graphs with big number of vertexes (and edges, correspondingly). For instance, the graph representation of the car network of a developed European state or a big university social network may demand tens of thousands graph vertexes and edges.

Thus, it makes obvious the urgency of the extension of the investigations in the lines of revealing core development features of the applications which enable visualization of big graphs and design of a module providing unified Application Programming Interface (API) for visualization of any graph algorithm.

Suggested Citation:
Gryzun, L., Shcherbakov, O., Parfonov, Yu., & Bodnar, L. (2022). Visualization of algorithms on graphs with a large number of vertices: The features of applications design. *Development Management,* 21(4), 36-44. doi: 10.57111/devt.20(4).2022.36-44.

*Corresponding author

According to studies [1; 2], a graph with large number of vertices (or so called "large graph") is a graph with approximately 10K vertices and/or edges. The way the graph is presented on the screen is called graph layout. In fact, this is the result of a graph visualization algorithm that positioned all the vertices on the screen.

The researchers D. Lande and I. Subach [1], V. Babkov, M. Serik [2], O. Demianchuk [3] emphasize core problems of large graphs visualization, which are readability, speed and algorithmic complexity. In this context, the researchers S. Iguana [4], F. Beck [5], A. Noack [6] emphasize the difficulties of aesthetic metrics determination for the criteria of "good" layout.

There are distinguished three main ways to visualize a graph: dimension reduction approach [4; 5], force-directed and energy-based approach [6], and features-based layout which are analysed in terms of speed concerns in the papers [7-9], where it is admitted that graph visualization algorithms mostly have bad algorithmic complexity (quadratic or cube one).

Thus, the analysis of the research papers [1; 3; 4] and existing applications which implement similar functions of the subject domain [4; 9] enables to testify, that despite the variety of scientific papers related to graph visualization problems, and, consequently, the number of software applications based on those researches, none of them has an option to visualize advanced or custom algorithms using API. The best analogues that have been found, could only display predefined algorithms and usually only basic ones (Depth First Search (DFS) [10], Dijkstra [11], etc.). They do not have sufficient functionality that can be used mostly for educational purposes.

Therefore, it can be concluded that the proceeding and visualization of large graphs is an urgent issue. In addition, in some cases graph data can be dependent on external factors and the basic algorithm should be modified to tackle the problem properly, which often causes new mistakes and bugs. The most efficient way to find them is to represent the data visually. Development of such applications is really time-consuming and focuses rather on implementing utility applications than working on real tasks solving.

Thus, the aim of this work was to reveal the core development features of an application for visualization of graphs with large number of vertices, and represent the authors' software module that provides a unified API to visualize any graph algorithm to save time working on utility software and focus more on solving problems.

MATERIALS AND METHODS

The set of theoretical and practical methods were used during the work. The theoretical literature analysis allowed to reveal the challenges of visualization of graphs with large number of vertices and the development of an application for this purpose.

Graph theory methods and exactly graph visualization algorithms were used at the initial stages of the application design and were taken into account at the formulation of the core features of its development.

There were used three main ways to visualize a graph which are characterised below: force-directed and energy-based approach, dimension reduction approach, and features-based layout. Force-directed and energy-based approach includes the family of methods based on physical systems simulation. Vertices are treated as charged particles that repulse each other, and edges model elastic strings. These methods simulate the dynamics of this system or find out a minimum of energy. Important methods of this family are ForceAtlas [9], Fruchterman-Reingold [10], Kamada-Kawai [11] and OpenOrd [12]. The last one uses optimisation techniques to speed up computation. As a useful side effect, graph gets more clustered. Such methods typically provide good result, and final plots reflect the graph layout very well. However, they are also computationally hard and have a lot of parameters to adjust, which was taken into consideration at their usage.

According to dimension reduction approach [4], a graph can be defined as an adjacency matrix $N \times N$, where N is the number of nodes. This matrix can also be considered as a table of N objects in N-dimensional space. This representation allows to use general-purpose dimension-reduction methods (UMAP, tSNE, PCA, and others). Another way to do it is to calculate theoretical distances between nodes and then to conserve proportion moving to lower-dimensional space. The ideas of this approach were also relevantly used at the appropriate stages of the application design discussed below. Approach of features-based layout is based on the idea that graph data reflect some objects of the real world. Thus, vertices and edges can have their own features according to object properties, real-life conditions, etc. Therefore, these features were used to represent them on the plane. It was possible to deal with node features as with usual tabular data using dimension reduction approaches or drawing a scatter plot for pairs of features.

It is important to emphasise the problems of these approaches implementation which arise in terms of speed concerns. For instance, one of the most common algorithms from force-directed set of algorithms, the Fruchterman-Reingold [10], in its regular variant has a total runtime of $O(|V|^2+|E|)$, where $|V|$ is the number of vertices in a graph and $|E|$ is the number of edges connecting the vertices. The grid-variant of this algorithm allows to reduce its runtime to $O(|V|+|E|)$: it divides the graph plot area into a grid of squares and applies repulsion forces between the nodes inside of adjacent squares, excluding the iteration of nodes further away. However, it is admitted that this runtime is only achieved in a best-case scenario, remaining quadratic for the worst-case scenario. Among the force-directed family, it was also developed the GEM algorithm [6] with the expectation to outperform in terms of runtime both the Kamada-Kawai algorithm and the Fruchterman-Reingold algorithm. According to [12-14], the total runtime of the GEM algorithm is $O(|V|(|V|^2+|E|))$.

Similar level of complexity is inherited also to the algorithms of dimension reduction approach. For instance, Principle Component Analysis (PCA) algorithm has two computationally crucial steps: computing the covariance matrix and computing the eigenvalue decomposition of the covariance matrix. The computational complexity of the covariance matrix computations is $O(NM \times \min(N, M))$, which is a result of multiplying two matrices of size $M \times N$ and $N \times M$, respectively. The worst-case complexity of the algorithms of eigenvalue decomposition is $O(M^3)$ for a matrix of size $M \times M$. Therefore, the overall complexity can be estimated as $O(NM \times \min(N, M)+M^3$ [15].

At the stage of design and development of the said module contemporary specialised software and systems such as Figma, Simple and Fast Multimedia Library (SFML), Texus' Graphical User Interface (TGUI) library, and C++ were used. The peculiarities of their usage are described in details in the relevant subsections of the work.

RESULTS AND DISCUSSION

The core features development of an application for visualization of graphs with large number of vertices can be formulated and characterised as following steps.

One of the basic steps for such an application is to provide users an ability to build a graph which later will be used for algorithm visualization. The three most common ways to represent a graph using data structures are adjacency list, adjacency matrix, and incidence matrix. Graph data structures comparison for each of three ways in terms of complexity of basic operations (graph storing, addition (removing) of a vertex, addition (removing) of an edge, etc.) are given in the Table 1 using big O notation ($|V|$ is the number of vertices in a graph and $|E|$ is the number of edges connecting the vertices).

Table 1. Graph data structures comparison for each of three ways of representation

	Adjacency list	Adjacency matrix	Incidence matrix
Store graph	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add vertex	$O(1)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add edge	$O(1)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Remove vertex	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Remove edge	$O(\|V\|)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Adjacency check	$O(\|V\|)$	$O(1)$	$O(\|E\|)$
Conclusions	Slow to remove vertices and edges, as it is necessary to find all vertices or edges	Slow to add or remove vertices, as matrix must be resized/ copied	Slow to add or remove vertices and edges, as matrix must be resized/copied

Source: developed by the authors based on [14-16]

The core peculiarity of this step which should be taken into account is the following. Since graphs used by the potential application are expected to have a large size (>10 000 vertices) it is crucial not to have a storage overhead. For this reason, an adjacency list is a preferred way to store graph data.

Graph visualization itself, being a complicated problem, determines its own peculiarities which should be minded by the developers. The application on this purpose should at least implement feature-based layouts. At the same time, support of other methods would be recommended for advanced versions of the application.

As it is expected to visualize a custom algorithm on a graph, it is necessary to provide users with such an ability. For this reason, there should be realised an API that helps to create an algorithm steps file to reproduce these steps later in the application.

One of the important features of such applications development is prediction of its possible using for educational purposes and for quick verification of basic graph properties such as connectivity, looking for bridges, etc. In this context, it is recommended to implement some basic graph algorithms like Breadth First Search (BFS) [17], DFS, etc.

In this context, it is also relevant to follow the criteria of "good" graph layout. It is important to apply some aesthetic metrics offered in research [4]:

(1) There should be minimum of edges intersection, as too many intersections make the layout look messy.

(2) Adjacent vertices should be closer to each other than not adjacent ones, as connected nodes in such a way will look also closer, which is true in graph by definition.

(3) The set of vertices (communities) should be grouped into clusters and they should look like a dense cloud.

(4) There should be minimum of edges and nodes overlapping.

Other researchers [5; 6] also formulated additional aesthetic criteria for graph visualization, such as reduction of visual clutter, reduction of spatial aliases and maximisation of compactness. It is also concluded that in real practical use, some of these criteria are in conflict with each other.

In terms of proving the choice of the technological tools for the application development, it is relevant to keep in mind some ideas. One of the important features of the application functionality is to make it easier to debug different algorithm implementations. Since different developers can use different Operational Systems, it will be a right decision to make the software cross-platform. On the other hand, the application of this purpose is going to deal with huge amount of data and to have the options for potential extension in the future. Therefore, requirements for the programming language include high performance and preferably it should be object-oriented to simplify the introduction of new features. Thus, all of these requirements are met by C++ [18]. Since C++ standard library does not provide tools for work with graphics, there should be a set of libraries chosen for this purpose. In particular, SFML [19] is helpful to create a display window and most of the graphics. TGUI [20] library allows to provide a proper user interface.

Both of these libraries support multiple platforms which is essential for a tool like this. The list of supported platforms includes Windows, Linux, macOS, and even Android is partially supported.

Development is preferably to be performed using a Linux-based operating system, as it provides a lot of tools [21] to facilitate the development process.

The application of such a purpose has to be open-source, due to the important benefits of this approach: users from all over the world can contribute to the tool development, there is feedback mechanism on both implementation details and code quality, possibility for end-users to modify the application according to their needs.

The revealed and formulated features of development of the applications for visualization of graphs with large

number of vertices made a necessary theoretical and technological base for design of a module that provides a unified API to visualize any graph algorithm to save time working on utility software and focus more on solving problems.

Developed module does not contain any databases, as they are not applicable in this scenario. Instead of it, data interchange formats and solutions are applied, as it was presented above. A graph is represented in a text format and visualization steps include two core stages.

At the first stage, the graph is stored in a METIS-based graph representation format, which is going to be extended since the module is expected to cover all possible graph types in future. Possible options for graph types that will be supported are: directed/undirected, edge-weighted, node-weighted.

As it was pointed out, since proceeding graphs are expected to have over >10 000 vertices and are not going to be stored overhead, an adjacency list was chosen as a way to store graph data.

A graph $G=(V,E)$ with N vertices is stored in plain text format in a file with N+1 lines. First line contains two integers N and F that specify vertex number and a graph format.

Since there are several possible combinations of graph types (which can be extended later), it was decided to use a common programming technique to store this information in a single integer where each bit specifies whether some feature is used or not. For example, if one attribute has a value of 0x08 and another one 0x02 to specify that the graph will use both of them, a bit-wise OR is applied to these values and the resulting number (10 in this case) will be stored as a graph type. All these calculations will be definitely done behind the scenes and the user will be provided with a user-friendly API. If the graph edges are weighted value should contain 0x01, in case of node weights 0x02, if the nodes' position can be specified via coordinates (can be used in case of feature-based layouts and also to restore vertex positions after the graph building stage) 0x04, directed graphs contain 0x08.

The remaining N lines of the file store information about the current graph structure. In particular, the i-th line contains information about the i-th vertex. Depending on the value of F, the information stored in each line is somewhat different. In the most general form (when F=15) each line has the following structure:

$$c\,x\,y\,v_1\,w_1\,v_2\,w_2\ldots, \quad (1)$$

where c – node weight, xy – corresponding coordinates on the plane, v_i – vertex adjacent to the current one and w_i – weight of the edge to the adjacent vertex.

Next, visualization step, which collects information about every pass of the visualization process (for example, highlighting vertices and nodes). Some algorithms may require more sophisticated situations to visualize (such as partitioning levels [22]), but the majority of algorithms do not need it. Thus, provided functionality of a module is sufficient.

To enhance user experience, the API provides an opportunity to combine several visualization steps into groups. It is caused by the fact that, if for the sake of simplicity, there was used BFS [22] as an algorithm to analyse, there is no need to examine every topology check, and it will be enough to observe the state of the graph only after every algorithm iteration. JavaScrip Object Notation (JSON) as a data storing approach was preferred to native one, as it allows serialization of custom objects and also supports arrays, which is crucial in this case.

Following the formulated features of technological tools, the module was developed by the authors in C++ language with attraction of SFML and TGUI libraries, and using a Linux-based operating system.

In this part of the work the developed module is characterised.

Since the module architecture is complex and it is hard to see all the details in one diagram, every software component is discussed separately.

The starting point of the program is the Application class which structure is presented in Figure 1 in the form of diagram and built by the authors in the course of the module design. It contains the main loop, handles the framerate, and delegates user events to other classes. Window class gathers user events, and handles camera movement and everything related to updating and displaying the main window of the application. To keep the user's screen clean and not messy the program is working in a single-window instance and creates additional windows only to notify user about wrong usage of the software. StateManager allows to transit application from one state to another.

There are three main states which correspond to three main screens of the application:

1) MainMenuState is a simple screen that allows choosing whether the user wants to run a graph visualizer or a graph builder;

2) GraphBuilderState (Fig. 2) is a screen that allows users to build a graph manually. It has several modes such as 'Hand', 'Add vertex', 'Add edge'. It is much easier to build a graph using a UI tool rather than writing down raw values into a plain text file.

3) VisualizationState, which is the most complicated part of the module. It includes graph layout algorithms, and sample graph algorithms to demonstrate the work of the application and allows to visualize an algorithm provided by the user.

IGraphAlgorithm interface is used for predefined algorithms within the program. For example, as shown in Figure 3, there are two basic algorithms that were implemented for software testing purposes. Both of them implement run() method which executes the algorithm using the provided graph and returns a record for the visualization process.

Graph class contains an adjacency list which is essentially a vector of vertices. Each vertex contains a list of outgoing topologies (in other words, all adjacent vertices), and a colour specified to this vertex. Graph class also includes ResourceProvider which is a useful utility that provides shapes for vertices and edges.

AlgorithmRecorder class is designed to store and to replay the visualization process on a graph. This class is also provided in the application's API in order to allow users to create algorithm steps data.

The module developed according to the core features formulated above was tested on purpose of verification of its functional requirements. As a result, it was possible to conclude that the designed software module meets all of them.

The developed module is implemented on the basis of a desktop application (exactly the Ubuntu 20.04 distribution), which is one of the most popular Linux-based distributions.

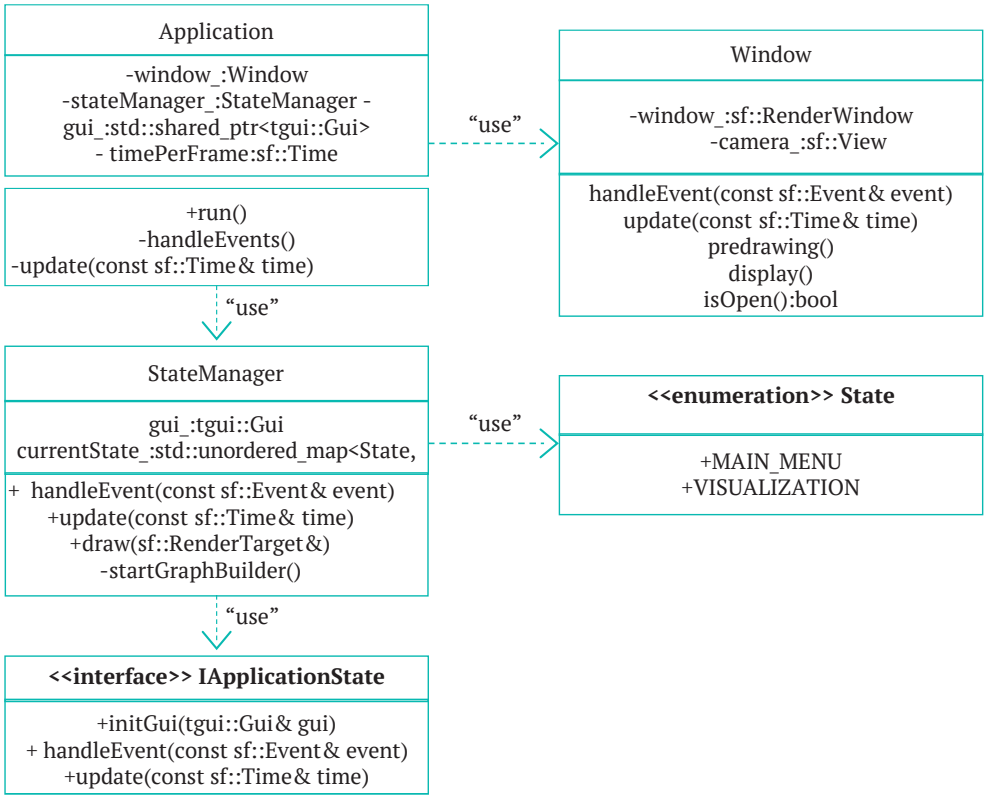

Figure 1. Diagram of fundamental classes

Source: developed by the authors in the course of the module design

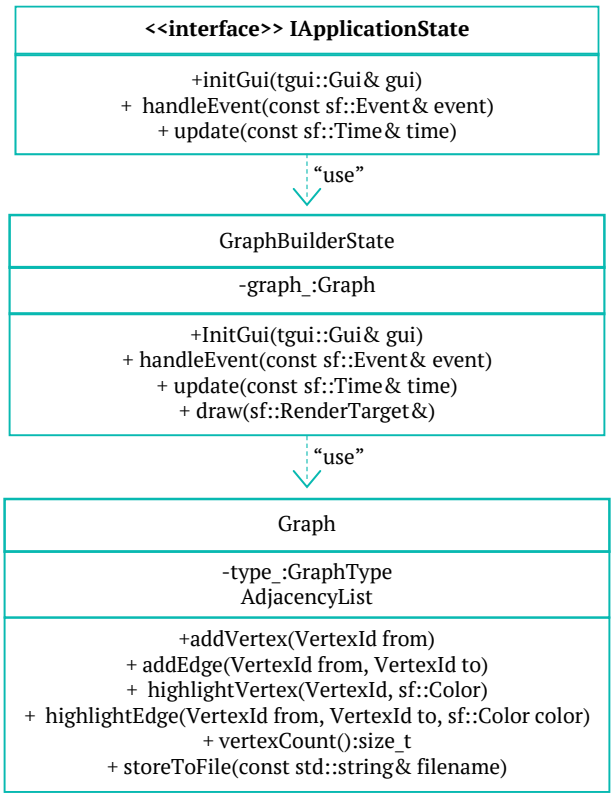

Figure 2. Diagram of GraphBuilderState

Source: developed by the authors in the course of the module design

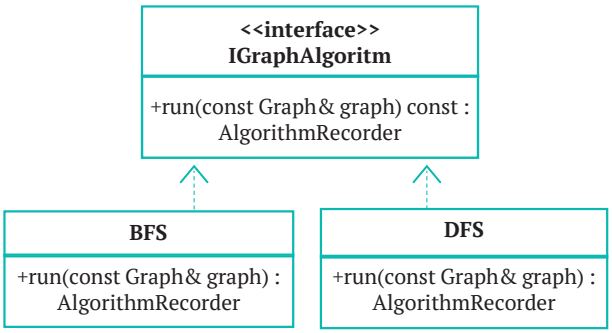

Figure 3. Diagram of IGraphAlgorithm interface
Source: developed by the authors in the course of the module design

According to the set goals, the core development features of an application for visualization of graphs with large number of vertices are revealed and formulated, which is a unique result that has not been highlighted in the research sources before.

The formulated features of development of the applications were used as a theoretical and technological base for design of the software module (presented above) that provides a unified API to visualize any graph algorithm to save time working on utility software and focus more on solving problems. In terms of practical significance of the work, the module designed following the said features and peculiarities makes a valuable tool for data scientists and other experts who are specialised in working on graph algorithms, as it enables data visualization for debugging and analysing large data structures.

In comparison with other applications that realize similar functions of the subject domain which characteristics are covered in [4; 9; 12; 23], the developed module has advanced functionality in terms of visualization of graphs with large number of vertices, that can be characterised as follows.

The module provides reading graph from a file: a properly written graph data is read and displayed successfully. In case of incorrectly written/corrupted file, the user is notified that file doesn't meet file format requirements.

API to write graphs and algorithm steps to corresponding files programmatically is realized, and it works properly producing valid files.

Algorithm provided by a user is visualized correctly. In case of invalid file, the proper notification is displayed.

A graph is built with the help of graph builder, and manual building of files is present. Manually built graph can be stored in a hard drive correctly. Sample algorithms (BFS and DFS) are provided for visualization tool. Both of them validate input and warn user in case of invalid input.

Random graph generation is also provided by the module and works properly: graph can be generated with predefined number of vertices and edges; advanced graph layout visualization algorithms is implemented; Fruchterman-Reingold layout is calculated correctly.

Selected episodes of the module work according to its functionality are presented in Figures 4-7.

Comparative characteristic of the developed software module with other applications which realise similar functions of the subject domain is generalised in the Table 2. The characteristic is done based on the papers [4; 9; 12; 23] and the module functionality presented above.

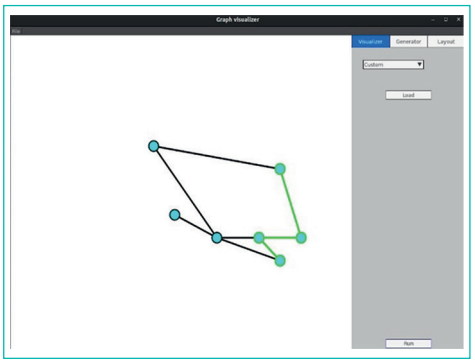

Figure 4. Algorithm visualized using external API
Source: developed by the authors

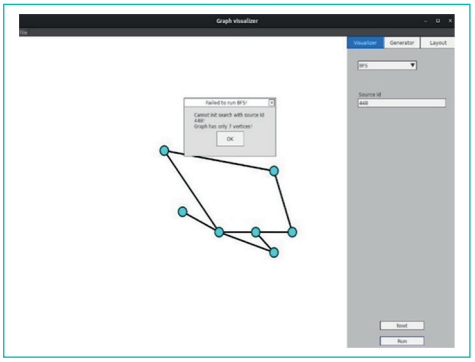

Figure 5. BFS algorithm input validation
Source: developed by the authors

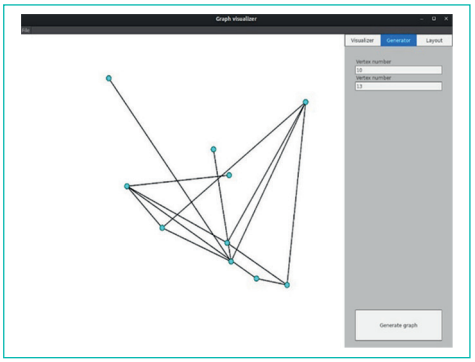

Figure 6. Randomly generated graph
Source: developed by the authors

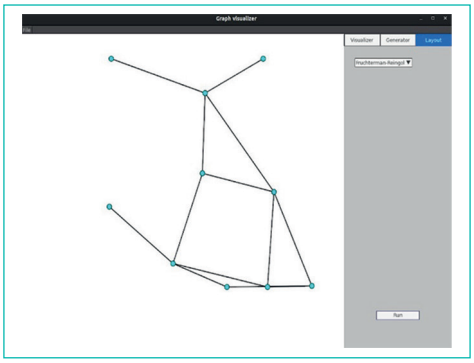

Figure 7. Result of applying Fruchterman-Reingold layout

Source: developed by the authors

Table 2. Comparative characteristic of the developed software module with analogues

Application name	Visualgo	USFCA visualizer	Gephi	Graphia	Developed software module
Platform	Web application	Web application	Desktop (Windows/ MacOS/Linux)	Desktop (Windows/ Linux)	Desktop (the Ubuntu 20.04 distribution)
Inputs a custom graph	Only manual	No	Yes	Only converting graph formats of other applications	Yes
Visualizes predefined algorithms	Yes	Yes	No	Yes	Yes
Supports different types of graph visualization	No	No	Yes	It has only one force-directed layout and very limited ways to tune it	Yes
Reads graph from a file and visualizes it	No	No	No	No	Yes
API to write graphs and algorithm steps to corresponding files	No	No	No	No	Yes
A graph is built with the help of graph builder, and manual building of files	No	No	No	No	Yes
User's algorithm is visualized	No	No	No	No	Yes
Notes	Not applicable for big graphs due to support of only manual input. It is seen as a good educational tool.	Poor functionality. Can be used only to get familiar with some basic algorithms, but the previous analogue deals with it much better.	Advanced visualization tool. Extremely good for data scientists. However, does not provide algorithm visualization.	Rendering options are not good for large graphs.	Performed using Linux-based operating system and all advantages of C++. Implements feature-based layouts, at the same time, supporting other methods. Has advanced functionality in terms of visualization of user's large graphs and their algorithms.

Source: developed by the authors

The table analysis testifies that the developed module has advanced functionality in terms of visualization of graphs with large number of vertices in comparing with the analogues presented in the papers.

CONCLUSION

According to the aim of the work, the main features of the development of an application for visualization of graphs with large number of vertices were revealed and formulated. The core steps of the applications development along with their characteristics recommended to mind in the progress of design of such software were distinguished and detailed.

The software module designed by the authors following the mentioned above features is presented with the details of its development and characteristics of its core functions. It provides a unified API to visualize any graph algorithm to save time working on utility software and focus more on solving problems. Therefore, in practical aspects, the developed and presented software module makes a valuable tool for data scientists and other experts who are specialised in working on graph algorithms, as it enables data visualization for debugging and analysing large data structures.

In addition, the comparative analysis of the developed module and the analogues has been realised in the paper.

The analysis testifies that the authors' software module has advanced functionality in terms of visualization of graphs with large number of vertices.

The developed software has lots of points for an extension due to its well-designed interface and topic depth. There can be applied new layout algorithms, and extended support for other graph features such as multilevel partitioning, multigraphs, etc. Nevertheless, performance improvements will be crucial to support graphs with even bigger amounts of vertices and edges. In addition, the tool can be optimised (for example OpenGL can be used instead of SFML) in order to work with graphics on a lower level. The mentioned extensions can prove the prospect of further research.

REFERENCES

[1] Lande, D., & Subach, I. (2021). *Visualization and analysis of network structures*. Kyiv: Igor Sikorsky Kyiv Polytechnic Institute, Politekhnika.

[2] Babkov, V., & Serik, M. (2011). Methods of visualization of data of complex structure based on tree-like maps. *Scientific papers of Donetsk National Technical University. Series: Informatics, Cybernetics and Computer Science*, 14(188), 163-170.

[3] Demianchuk, O. (2022). Principles of data visualization. Retrieved from https://dou.ua/forums/topic/39157/.

[4] Iguana, S. (2019). Large graph visualization tools and approaches. Retrieved from https://towardsdatascience.com/large-graph-visualization-tools-and-approaches-2b8758a1cd59.

[5] Beck, F., Burch, M., & Diehl, S. (2009). Towards an aesthetic dimensions framework for dynamic graph visualisations. In *2009 13th International Conference Information Visualisation* (pp. 592-597). Barcelona: IEEE. doi: 10.1109/IV.2009.42.

[6] Noack, A. (2019). Energy models for graph clustering. *Journal of Graph Algorithms and Applications*, 11(2), 453-480. doi: 10.7155/jgaa.00154.

[7] Schwab, M., Strobelt, H., Tompkin, J., Fredericks, C., Huff, C., Higgins, D., Strezhnev, A., Komisarchik, M., King, G., & Pfister, H. (2017). An education system with hierarchical concept maps and dynamic nonlinear learning plans. *IEEE Transactions on Visualization and Computer Graphics*, 23(1), 571-580. doi: 10.1109/TVCG.2016.2598518.

[8] Wilke, C. (2019). *Fundamentals of data visualization: A primer on making informative and compelling figures*. Sebastopol: O'Reilly Media Inc.

[9] Jacomy, M., Venturini, T., Heymann, S., & Bastian, M. (2014). ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLoS ONE*, 9(6): article number e98679. doi: 10.1371/journal.pone.0098679.

[10] Fruchterman, T.M.J., & Reingold, E.M. (2014). Graph drawing by force-directed placement. *Journal of Software: Practise and Experience*, 21(11), 1129-1164. doi: 10.1002/spe.4380211102.

[11] Cheong, S., & Si, Y. (2015). Accelerating Kamada-Kawai for boundary detection in mobile ad hoc network. *ACM Transactions on Sensor Networks*, 13(1), article number 3. doi: 10.1145/3005718.

[12] Shawn, M., Brown, W.M., Klavans, R., & Boyack, K.W. (2011). OpenOrd: An open-source toolbox for large graph layout. *Visualization and Data Analysis 2011. Proceedings of the SPIE*, 7868, article number 786806. doi: 10.1117/12.871402.

[13] Frick, A., Ludwig, A., & Mehldau, H. (1995). A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In Tamassia, R., Tollis, I.G. (Eds.), *Graph Drawing* (pp. 388-403). Berlin: Springer. doi: 10.1007/3-540-58950-3_393.

[14] Sund, D. (2016). *Comparison of visualization algorithms for graphs and implementation of visualization algorithm for multi-touch table using JavaFX*. Linköping: Linköpings Universitet.

[15] Banerjee, A. (2020). Computational complexity of PCA. Retrieved from https://alekhyo.medium.com/computational-complexity-of-pca-4cb61143b7e5.

[16] Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., & Wagner, D. (2010). Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 15, article number 2.3. doi: 10.1145/1671970.1671976.

[17] Holdsworth, J. (1999). *The nature of breadth-first search*. Townsville: James Cook University.

[18] Meyers, S. (2014). *Effective modern C++*. Newton: O'Reilly Media.

[19] SFML documentation. (n.d.). Retrieved from https://www.sfml-dev.org/documentation/2.5.1/.

[20] TGUI documentation. (n.d.). Retrieved from https://tgui.eu/documentation/.

[21] Shotts, W. (2019). *The Linux command line: A complete introduction*. (2nd ed.). San Francisco: No Starch Press.

[22] Kurant, M, Markopoulou, A., & Thiran, P. (2010). On the bias of BFS (Breadth First Search). In *2010 22nd International Teletraffic Congress (ITC 22)* (pp. 1-8). Amsterdam: IEEE. doi: 10.1109/ITC.2010.5608727.

[23] Shynhalov, D. (2018). Investigation of the software for analysis and visualization of social graph structures. *Control, Navigation and Communication Systems. Academic Journal*, 5(51), 128-131. doi: 10.26906/SUNZ.2018.5.128.

Візуалізація алгоритмів на графах з великою кількістю вершин: особливості проєктування застосунків

Людмила Едуардівна Гризун[1], Олександр Всеволодович Щербаков[1], Юрій Едуардович Парфьонов[1], Лілія Василівна Боднар[2]

[1]Харківський національний економічний університет імені Семена Кузнеця
61166, просп. Науки, 9А, м. Харків, Україна

[2]Південноукраїнський національний педагогічний університет імені К.Д. Ушинського
65020, вул. Старопортофранківська, 26, м. Одеса, Україна

Анотація. Завдання візуалізації великих графів як спеціальної структури даних та алгоритмів на них розглядається вченими і практиками як складна і нетривіальна проблема. Аналіз наукових робіт та існуючих програмних додатків, що реалізують подібні функції предметної області, засвідчує актуальність розширення розвідок у напрямках виявлення особливостей розробки додатків для візуалізації великих графів та алгоритмів на них. Формулювання особливостей і рекомендацій щодо розробки такого програмного забезпечення та представлення спроєктованого авторами програмного модуля є метою статті. У ході роботи за допомогою методів аналізу та теорії графів виявлено та сформульовано основні особливості розробки програми для візуалізації графів з великою кількістю вершин. Надано окремі рекомендації щодо сутності кожного з етапів розробки таких додатків та виявлено ті кроки, які є найбільш важливими для розробників у термінах складності обробки та візуалізації великих графів, метрик їх розташування на екрані додатку тощо. Також представлено розроблений авторами модуль, який забезпечує уніфікований інтерфейс програмування додатків для візуалізації будь-якого алгоритму на графах, що дозволяє заощадити час на роботі над службовим програмним забезпеченням і більше зосередитися на розв'язанні алгоритмічних задач. Представлений модуль розроблено авторами з урахуванням виявлених особливостей та рекомендацій. Проведено порівняльний аналіз розробленого програмного модуля та аналогів, який засвідчив розширену функціональність модуля щодо візуалізації графів з великою кількістю вершин. Модуль є практично значущим інструментом для дослідників у галузі структур даних та інших експертів, які працюють над алгоритмами на графах, оскільки дає змогу візуалізувати дані при налагодженні програмного забезпечення та спрощує аналіз великих структур даних

Ключові слова: великі графи; проблеми унаочнення алгоритмів; модуль візуалізації алгоритмів на графах; компонування графа; уніфікований програмний інтерфейс