Scientific Visualization, 2025, volume 17, number 4, pages 39 - 56, DOI: 10.26583/sv.17.4.05

This paper is an extended version of a contribution presented
at the Graphicon 2025 conference.

Method of Identification of Implicit Relations in Solving Analytical Problems

I.D. Sokolov[1], K. V. Ionkina[2], M. S. Ulizko[3], E.V. Antonov[4], E. N. Bazhanova[5], A. A. Artamonov[6]

National Research Nuclear University "MEPhI", Moscow, Russia

[1] ORCID: 0009-0008-4467-1347, idsokolov@kaf65.ru
[2] ORCID: 0000-0001-6850-2443, kionkina@ka65.ru
[3] ORCID: 0000-0003-2608-8330, msulizko@kaf65.ru
[4] ORCID: 0000-0003-1498-9131, eantonov@kaf65.ru
[5] ORCID: 0000-0001-7714-0579, ebazhanova@kaf65.ru
[6] ORCID: 0000-0002-9140-5526, AAArtamonov@mephi.ru

Abstract

Visualization tools enable the transformation of large datasets into user-friendly graphical representations. This paper presents the development of a graph visualization tool designed to uncover implicit relationships among information entities. We describe a method for identifying such implicit relationships and introduce an interactive graph visualization system that allows users to explore the graph through filtering. The implemented functionality includes a specialized query language for dynamically modifying the appearance of nodes and edges.

The proposed method and the developed tool were evaluated on two real-world datasets: (1) detecting potential violations of nuclear non-proliferation commitments and (2) identifying promising areas for scientific collaboration among organizations. The results confirm the practical relevance of the proposed approach.

Keywords: graph visualization, graph construction, heterogeneous data, heterogeneous data processing, ReactJS.

1. Introduction

Currently, numerous data visualization tools are available. However, each comes with certain limitations. Commercial software solutions may be inaccessible due to high costs, while others may lack sufficient functionality or be overly complex to learn. For instance, Gephi [1], a widely used application for graph visualization, demands a high level of user expertise due to its complex interface and multi-step graph construction process. Similarly, VOSviewer [2] operates only with predefined data structures and addresses a narrow, domain-specific task—namely, bibliometric analysis.

This highlights the need for specialized software that integrates interactive filtering mechanisms and methods for highlighting key nodes and edges, enabling the analysis of large, heterogeneous datasets and the discovery of hidden patterns—without being constrained to a fixed data schema.

The demand for a flexible and functional tool becomes especially pressing when analyzing heterogeneous objects, where uncovering implicit relationships and dependencies is critical.

To determine a relationship between two objects, A and B, one must first extract their characteristics and identify shared attributes that indicate a connection. Suppose there exists

an object C that shares characteristics with both A and B. In this case, C serves as a common feature linking A and B, thereby establishing an indirect relationship between them.

The paper "Discovering Hidden Dependencies Between Objects Based on Large-Scale Bibliographic Data Analysis" [3] addresses the problem of assessing thematic proximity between journals and conferences by employing several approaches to measure an author's affinity to a research field. The author proposes constructing co-authorship graphs where edge weights represent connections between journals. The underlying assumption is that if the same author publishes in multiple journals, those journals are likely thematically related.

Similarly, the article "Methods for Analyzing Heterogeneous Data to Construct Social Profiles" [4] explores the identification of implicit relationships for social profile construction, proposing a graph-based approach to uncover such connections.

These studies demonstrate the effectiveness of graph visualization in revealing inter-object relationships. To apply such methods, an analyst must select relevant object attributes and define rules for how entities are linked—essentially specifying the graph construction logic.

Existing approaches to relationship discovery include the co-authorship graph method described in [5], primarily used to assess thematic similarity among journals and conferences. While effective in its domain, this method suffers from limited scalability and narrow applicability.

Another common technique is correlation analysis (as defined in [5]), which measures the statistical relationship between two variables. Although this method does not require a predefined structure, it is restricted to a single data type and cannot capture complex, multi-faceted dependencies.

Papers [6] introduce "Causal Discovery" and "Causal Inference" methods aimed at identifying cause-effect relationships. While powerful for evaluating the impact of one element on another, these methods focus on causality rather than structural relationships, making them less suitable for general systems analysis where the goal is to map connections rather than infer causal mechanisms.

Finally, works [7] and [8] explore semantic analysis and ontology-based modeling. These approaches build domain models that describe properties, internal relationships, and structure, often represented as graphs. While effective for structured knowledge representation, they require prior knowledge of the data schema and are not well-suited for exploratory analysis of unstructured or heterogeneous datasets.

Taken together, these limitations underscore the need for a more adaptable, interactive, and schema-agnostic graph visualization tool capable of revealing implicit relationships across diverse data types.

2. Problem Statement

The methods reviewed above are capable of addressing specific tasks related to identifying relationships between objects; however, they typically require either a predefined data structure or a specific data type. Within the scope of this work, we aim to address the challenge of constructing a graph from input data of heterogeneous structures and types, and of detecting both explicit and implicit relationships between objects— a task that existing methods cannot adequately support.

Based on the literature review, we propose to develop a novel method for identifying explicit and implicit relationships that is independent of the input data's structure or type. In parallel, we will design and implement a flexible visualization tool capable of handling diverse data schemas and enabling interactive exploration of the resulting graphs.

3. Method for Identifying Relationships Between Objects

Let $N = \{obj = (a_1, a_2, \dots, a_M) | a_i - is\ a\ characteristic\ of\ object\ obj, i = 1,2, \dots, M\}$ – be a set of objects, each described by characteristics $a_1, a_2, \dots, a_M$. For each object $obj = (a_1, a_2, \dots, a_M) \in N$ we construct a graph $G =< V, E >$, where:$V = \{v_1, v_2, \dots, v_T\}, T \geq M$, con-

sists of the actual values of the characteristics $a_1, a_2, \dots, a_M$ of obj, and $E = \{(v_i, v_j) \mid \exists\, obj \in N : obj(a_i) = v_i \; and \; obj(a_j) = v_j\}$. Two vertices $v_i, v_j \in V$ are connected by an edge if there exists an object $obj = (a_1, a_2, \dots, a_M) \in N$, such that the value of characteristic a_i of object obj equals v_i and the value of characteristic a_j of object obj equals v_j, where the pair of characteristics $a_i \; and \; a_j$ is specified by the analyst.

Each vertex $v \in V$ is assigned a type corresponding to one of the original characteristics $\{a_1, a_2, \dots, a_M\}$. Consequently, the vertex set can be partitioned as $V = K_1 \cup K_2 \cup \dots \cup K_M$, where $K_i = \{v_j \in V | obj(a_i) = v_j\}$ – is the set of vertices of type a_i, $i = 1,2, \dots, M$. For clarity, a vertex $v_j \in K_i$, is denoted as $v_j^{K_i}$, $j \in \{1,2, \dots, T\}, i \in \{1,2, \dots, M\}$. Thus, the graph $G =< V, E >$ can be viewed as a heterogeneous graph $G =< K_1 \cup \dots \cup K_L, E >$, where $L \in \{1,2, \dots, M\}$ and $v_i, v_j \in K_W$, then $(v_i, v_j) \notin E$.

Let K_L and K_P – denote vertex sets of types a_L and a_P respectively. Based on the foregoing, we define the following:

An Explicit relation exists between vertices $v_i \in K_L$ and $v_j \in K_P$ if and only if $(v_i, v_j) \in E$.

During graph construction, explicit relation are formed between vertices of different types in accordance with the graph-building rule. Consequently, implicit relation can be inferred between vertices of the same type.

An implicit relation between vertices $v_i \in K_L$ and $v_j \in K_L$ exists if and only if there exists a vertex $v_x \in K_P$ such that both $(v_i, v_x) \in E$ and $(v_j, v_x) \in E$, where $L \neq P$.

An explicit relation of order *n* between vertices $v_i \in K_L$ and $v_j \in K_P$ exists if and only if there exists a path $v_i \to v_j$ of length n that does not contain consecutive vertices of the same type, and $n \in \mathbb{N}$ is the smallest natural number for which such a path exists.

An implicit relation of order *n* between vertices $v_i \in K_L$ and $v_j \in K_L$ exists if and only if there exists a vertex $v_x \in K_P$ such that both $(v_i, v_x) \in E$ and $(v_j, v_x) \in E$, are explicit relations, and at least one of these explicit relations is of order $\boldsymbol{n}$.

Obvious implicit relation is implicit relation between vertices $v_i \in V$ and $v_j \in V$, if and only if there exists and object $obj \in N$, which characteristic's value a_i, contains v_i and v_j. In case if there are relations between vertex of same type, obvious implicit relation turns into explicit relation.

It should be noted that these definitions apply to undirected graphs, as directed graphs inherently encode directionality from one vertex to another, thereby eliminating the need for vertex typing.

4. Graph construction algorithm based on different input data structures

Depending on the specific task, the structure of the input data may vary. Individual objects can differ in field names, field types, and nesting depth. In such cases, four types of relationships between fields within any given structure are considered:

- Type 1: Linking fields that do not share a common ancestor;
- Type 2: Linking fields located at the same nesting level and sharing a common ancestor;
- Type 3: Linking fields located at different nesting levels but sharing a common ancestor;
- Type 4: Linking subfields within a single field.

Consider a file with the structure shown in fig 1. To extract fields from such a data structure and establish relationships between them, it is sufficient to reference the field directly. If the field of interest resides at nesting level n, it is assigned an ancestor that represents the path used to access that field. An ancestor is defined as the immediately preceding field in the data hierarchy. When the data structure exhibits deep nesting, the concept of the nearest an-

cestor is introduced. The nearest ancestor is the field through which the target field is accessed. Each field can have only one nearest ancestor; consequently, every field will have a unique path composed of its ancestors and its single nearest ancestor. A nearest ancestor may correspond to multiple fields—for example, the field “affiliation” serves as the nearest ancestor for both “name” and “country,” enabling retrieval of the affiliation’s name and associated country.


```
{
  "Title": "article 1",
  "author": {
    "full_name": "Author's name",
    "affiliation": {
      "name": "National Resaerch Nuclear University MEPhI",
      "country": "Russian Federation"
    },
    "published_year": 2021,
    "keyword": "React"
  }
}
```


Fig. 1. Example of a file with standard nesting

A common ancestor is defined as the field through which one must navigate to retrieve the desired data. It should be noted that the nearest ancestor can also serve as the common ancestor, provided the nesting depth equals one. Otherwise, the common ancestor is considered to be the field at the topmost nesting level, while the nearest ancestor resides at the second-to-last nesting level.

Now consider a file with a nested structure. fig 2 presents an example where accessing a single field yields a list of records containing identical subfields. In this scenario, it is not possible to directly reference the subfields and establish relationships between them, because accessing their ancestor returns a list of values. To handle such cases, the list must first be unpacked, and each list element must be processed individually using the same linking algorithm. This approach effectively implements a Cartesian product of the list elements.


```
{
  "Title": "article_1",
  "author": [
    {
      "full_name": "Author's name",
      "affiliation": [
        {
          "name": "National Research Nuclear University MEPhI",
          "country": "Russian Federation"
        }
      ]
    },
    {
      "full_name": "Another Authors's name",
      "affiliation": [
        {
          "name": "National Research Nuclear University MEPhI",
          "country": "Russian Federation"
        },
        {
          "name": "Moscow State University",
          "country": "Russian Federation"
        }
      ]
    }
  ],
  "published_year": 2021,
  "keyword": [
    "React",
    "Python"
  ]
}
```


Fig. 2. Example of a file with nested data

For such structures, the following examples of field linkage are considered:

Linking fields without common ancestors

Since these fields do not share a common ancestor, linking them requires simply connecting them via the full path to the target field. For the structure shown in Fig. 1, the following field pairs can be linked:

- Title – author.full_name;
- Title – author.affiliation.name;
- Title – author.affiliation.country;
- Title – published_year.

And other combinations where nodes without common ancestors are connected.If one of the selected fields contains a list, a Cartesian product must be performed for linkage. For example, to link authors with keywords, one must first iterate through the list obtained via the "author" key to extract "full_name" values, then iterate through the "keyword" field — effectively performing a Cartesian product between the selected fields. This results in node pairs such as:

- Author's name – React;
- Author's name – Python;
- Another Author's name – React;
- Another Author's name – Python;
- And so on, connecting nodes that lack common ancestors.

Linking fields at the same nesting level sharing a common ancestor

This type enables linking fields located at the same nesting depth—for instance, affiliation name and affiliation country. Consider the structure presented in fig. 2.

To link fields at the same level, one must retrieve data from their shared nearest ancestor and combine them.

This linkage is necessary to connect fields belonging to a specific class of data—such as complete author information or affiliation details. Such linkage ensures connections only within internal data, enabling element-wise joining: i.e., joining based on element position within the data, following the rule [[a],[b]],[[x],[y]] → [[a,b],[x,y]].

In the data structure (fig. 2), the following field pair can be linked:

- author.affiliation.name – author.affiliation.country

Linking fields at different nesting levels but sharing a common ancestor

This type represents a combination of the first and second types. First, data must be retrieved within the scope of a single ancestor (as in the second type). Then, information from the second field is extracted, and a Cartesian product is performed (as in the first type).

Suppose the user wants to link the fields: author.full_name and author.affiliation.country. To achieve this, data must be normalized to a single level before linking. In the data structure (fig. 2), the following field pairs can be linked:

- author.full_name – author.affiliation.country;
- author.full_name – author.affiliation.name.

Linking elements within a single field

This type represents a special case of the first linkage type, where two identical fields are selected. It can be used to link elements within a single record. For example, consider the "full_name" field. Within a single object (in our case, Article Title 1), all authors will be linked to each other. This produces nodes with the following connections:

- Author 1 – Author 1 (for Article 1);
- Author 1 – Author 2 (for Article 2);
- Author 2 – Author 3 (for Article 3).

Based on the linkage types described above, the following variants of field linking can be formed within a single data structure. Each type is denoted by its number: 1 — first linkage type, 2 — second linkage type, 3 — third linkage type, 4 — fourth linkage type (fig. 3).

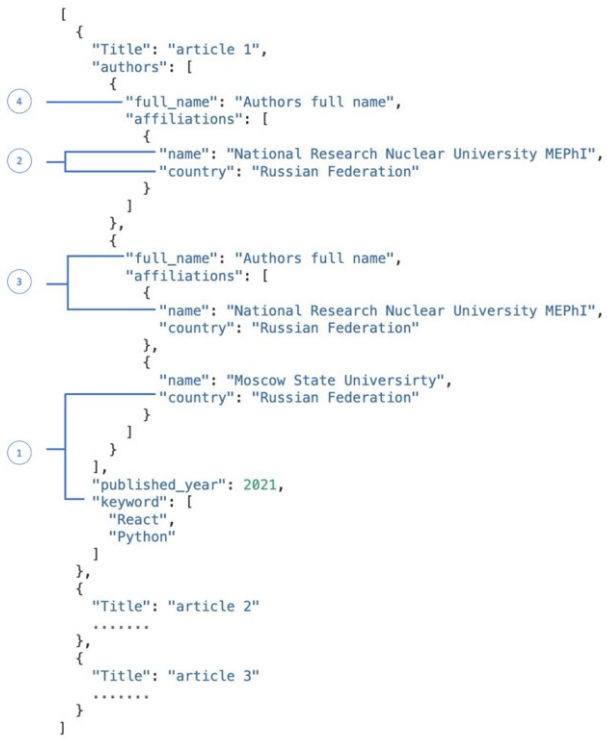


Fig. 3. Example of a data structure

Depending on the type of linkage, the linking algorithm will vary (Fig. 4). For each type, program code has been implemented based on the proposed algorithm, enabling traversal of the entire data structure and retrieval of required field data—preprocessing them into a standardized format suitable for graph construction.

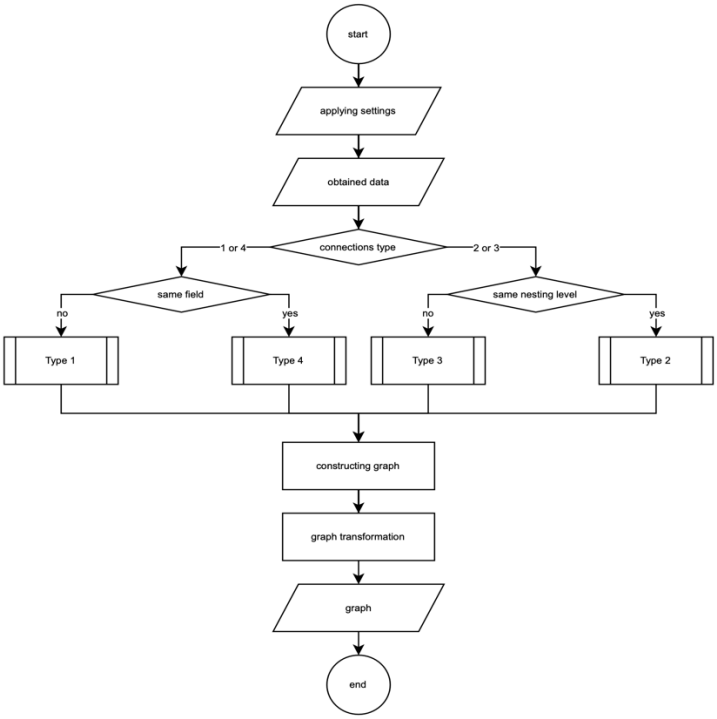


Fig 4. Graph construction algorithm

5. Design and development of a graph visualization tool

During the review of existing graph visualization solutions, a comparative table outlining the advantages and disadvantages of various tools was compiled (Table 1). The following tools were evaluated: Argo Lite [9], Gephi [1], Neo4j [10], and PyVis [11]. The key selection criteria for choosing a visualization tool were:

- Ease of integration
- Customizability of the graphical user interface
- Support for interaction with databases
- Sensitivity to input data structure

Table 1. Graph Visualization Tools

Criterion	Argo Lite	Gephi	Neo4j	PyVis
Open Source	Yes	Free version available	Yes	Yes
Integration Capability	Does not provide integration options	Limited integration capabilities	Provides integration options	Can be integrated with NetworkX for graph creation and visualization
GUI Customization	No interface customization available	Visualization styling can be adjusted, but not the user interface itself	Interface customization requires third-party tools or custom development	Allows graph visualization styling, but limited UI customization
Database Interaction	No database interaction supported	Can interact with graph databases	Has its own native graph database	Requires external libraries for DB interaction
Dependency on Data Structure	Yes	Yes	Yes	Yes

The developed tool employs a client-server architecture. In this setup, the client defines graph configuration parameters, while the server executes queries against the database and transforms the retrieved data into a structure suitable for graph construction.

The tool comprises three main components (Fig. 5): Graphical user interface (GUI) Application Programming Interface (API) and database. The Graphical User Interface, built using ReactJS, enables users to interact with the API by applying specific filters and configuration options.

The Application Programming Interface, implemented in Python, acts as an intermediary between the GUI and the database. It receives user requests, queries the database, processes the raw data, and converts it into a standardized graph-ready format (e.g., nodes and edges with metadata).

For data storage, Elasticsearch was selected due to its scalability, full-text search capabilities, and flexible schema handling — well-suited for complex, nested, or semi-structured data often encountered in graph construction scenarios.

This modular architecture ensures separation of concerns: the frontend provides an intuitive user experience, the backend handles data retrieval and transformation logic, and Elasticsearch serves as a high-performance, scalable data layer.

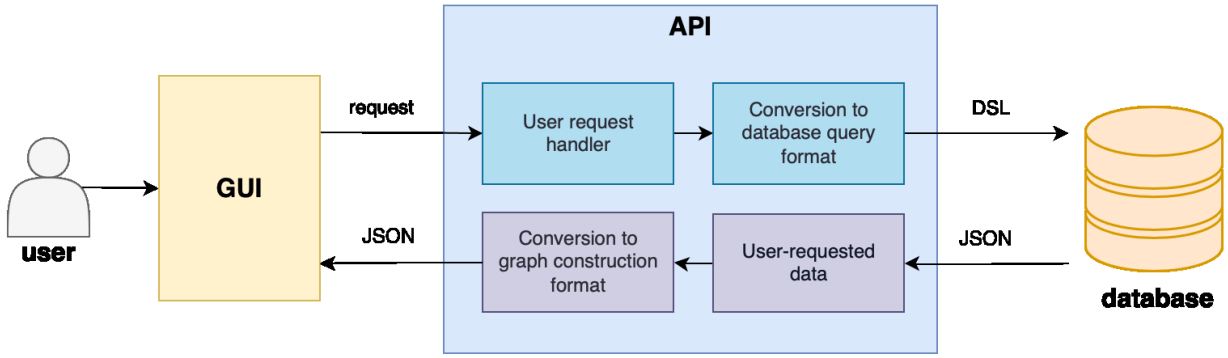


Fig. 5. Architecture of the Graph Construction Tool

6. Tool functionalities

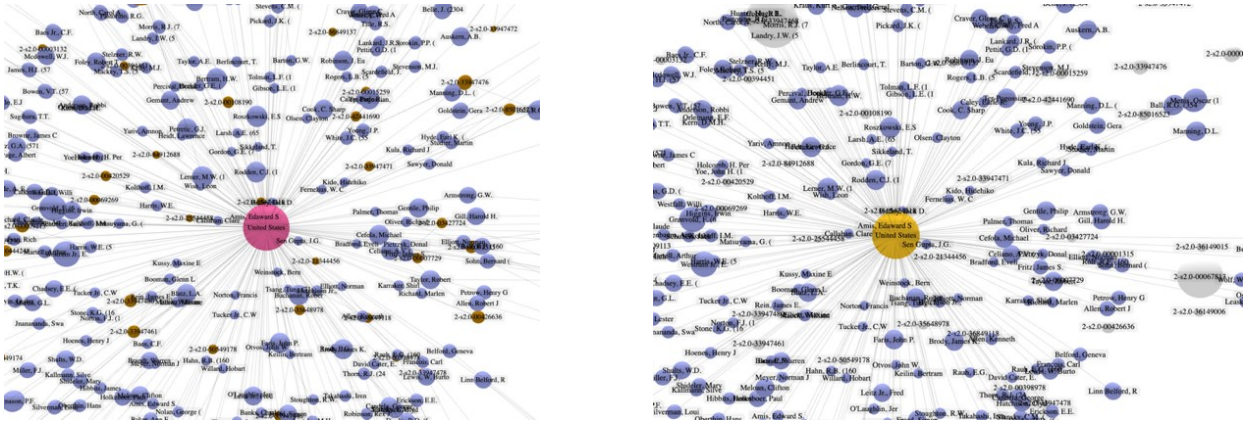
The developed tool provides functional capabilities aimed at supporting the analytical process by directing the user's attention to specific graph segments. The following key mechanisms have been implemented for this purpose:

- Highlighting a node of interest;
- Clearing the current selection;
- Emphasizing of nodes;
- Graph filtering;
- Customization of node.

Together, these mechanisms create a flexible environment in which an analyst can quickly switch between a global overview and detailed exploration of any graph segment.

Highlighting a node of interest

This mechanism is designed to highlight a selected graph node and all its associated connections (Fig. 6). All nodes not connected to the selected one are recolored in gray, allowing the user to focus on the specific segment under examination. When a node is selected, its color changes, keeping the user's attention on the chosen object. This functionality enables the user to filter out all nodes that are irrelevant at the current stage and concentrate on the selected nodes and their relationships with other information entities.



a b

Fig. 6. Graph node selection: (a) graph state before node selection, (b) graph state after the selected node is highlighted

When a node is selected, a summary of its details will be displayed in the additional information panel, including the node's name, weight, and group (Fig. 7).

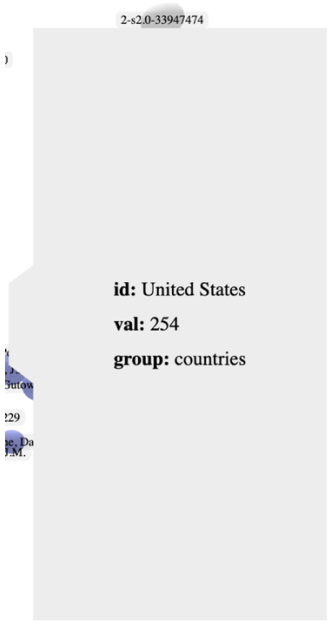


Fig. 7. Example of additional node information display

Clearing the current selection

The ability to clear node selection is necessary to allow the user to choose other graph elements without residual visual effects from the previous selection. When the “Clear Selection” button is clicked, all nodes revert to their original colors, and the additional information panel removes the summary for the previously selected node.

Emphasizing of nodes

To draw the analyst’s attention to specific nodes, a node highlighting feature has been implemented (Fig. 8). This functionality enables emphasis on a particular graph node based on a specified condition, facilitating the tracing of its connections to other objects. Highlighting is achieved using a pulsating (flashing) sphere around the node.

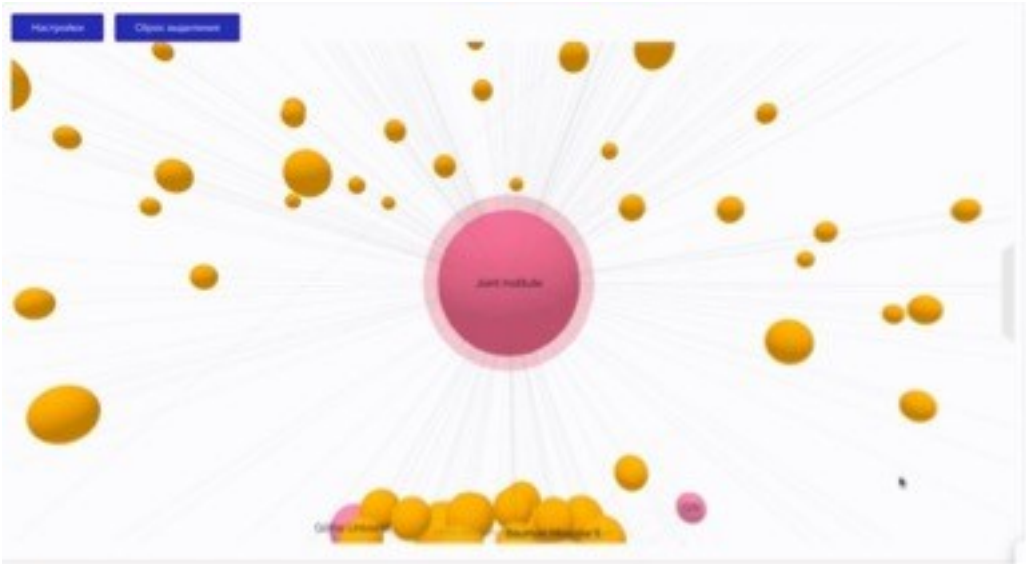


Fig. 8. Sphere for drawing the user’s attention

Customization of node

The user can assign a color and a group to the nodes of interest. The color for each group of nodes is generated automatically (Fig. 9). Additionally, the user must define the type of relationship between groups in the dedicated field labeled “Field Relationships.” Selected fields

will be highlighted using the color of the group to which they belong. If needed, the user can also configure whether nodes in a group display labels and/or a pulsating sphere.

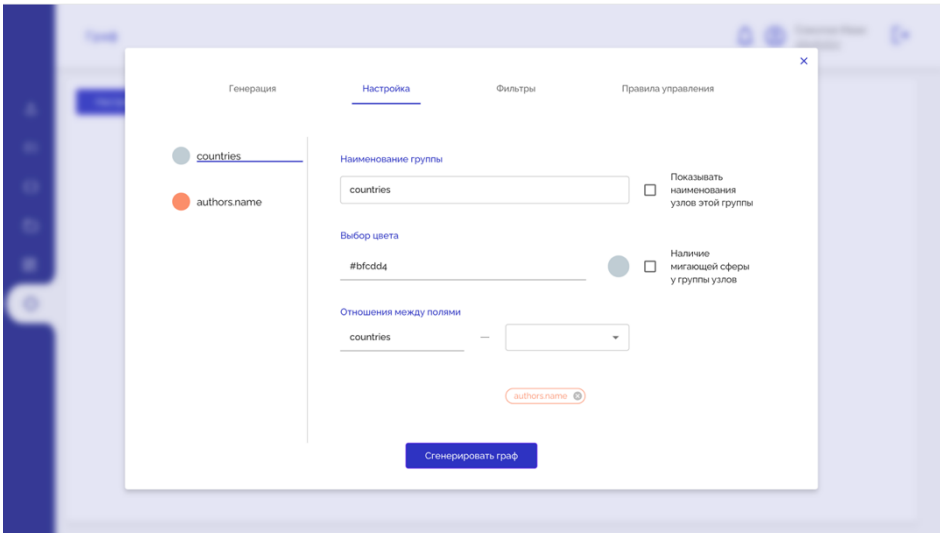


Fig. 9. Configuration of graph nodes based on selected fields

Graph filtering

This mechanism enables graph filtering to reduce the amount of displayed data. Filtering is performed using the Domain Specific Language (DSL) query language based on JSON format. To retrieve a specific document stored in the database, the user must reference a designated field using the query syntax: { "query": { <query body> } }.

Two filtering modes are provided for practical use (Fig. 10): Basic filtering — selecting relevant fields, operators, and filter types via a user interface; Advanced filtering — writing custom queries directly in the Domain Specific Language (DSL)

The first method allows combining selected filters using logical operators (must, should, must_not). Individual filters are joined with a logical AND to ensure that all specified conditions across the selected fields are applied simultaneously. For example, filtering documents by country and filtering documents by specific keywords are combined with a logical AND, meaning that only documents satisfying both conditions will be included in the result.

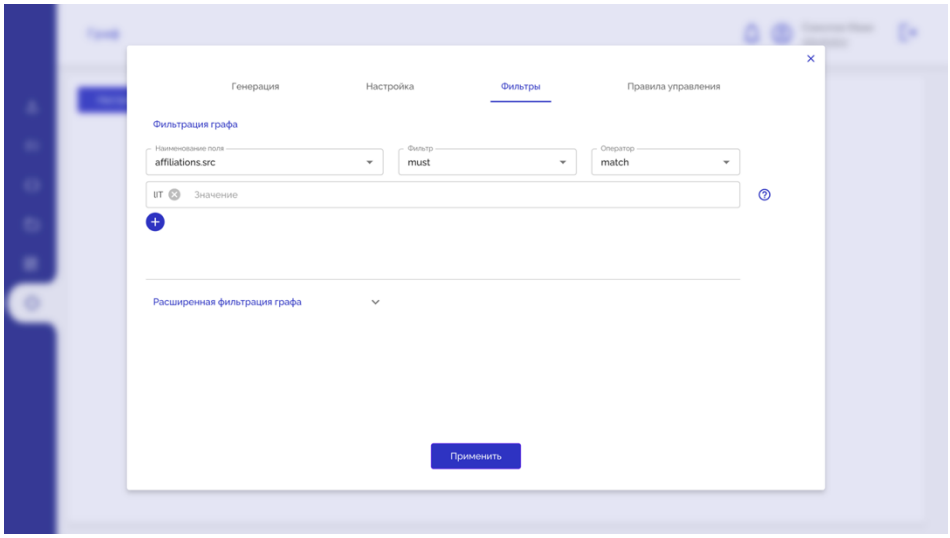


Fig. 10. Graph filtering using an Elasticsearch query

Rules for modifying nodes

To address analytical tasks, a query language (called rules) has been proposed. Consider a scenario where the user needs to select one or more nodes from a specific group that satisfy

certain conditions. To support this, a set of rules has been implemented, allowing the user to modify node properties—such as weight, color, group, label, or highlighting—based on specified criteria. For this purpose, a specialized query language has been developed:

Rule's scheme

field_type : <values> {conditions = conditions} [change_pattern = change_pattern]

where:

- field_type — the target field whose nodes are to be considered;
- <values> — a list of specific node values to which the changes will be applied. Node values are separated by commas;
- {conditions = conditions} — a list of conditions that must be satisfied for the changes to be applied. Conditions are separated by commas and combined with a logical AND (i.e., a document must satisfy all specified conditions);
- [change_pattern = change_pattern] — a list of modifications to apply

The following nodes' values are subject to modification:

- val - node weight (int);
- name - node name (str);
- label - node label (str);
- label_vis - whether the label is visible (boolean);
- color - node color (str);
- highlited - whether a pulsating sphere is enabled (boolean);
- group - node group (str).

To assign a specific data type to a value, the corresponding type constructor must be explicitly specified before that value. The following data types are supported:

int() – for integer (numeric) values;

float() – for decimal or floating-point numbers (e.g., 0.5, 1.0);

bool() – for boolean values (True or False);

str() – for textual values. If a text value contains commas, it must be enclosed in forward slashes: /text, with comma/.

If the field_type field is set to all, then {conditions = conditions} and [change_pattern = change_pattern] will be applied to all fields and all nodes in the graph.

If the <values> field contains <*>, then {conditions = conditions} and [change_pattern = change_pattern] will be applied to all nodes corresponding to the field specified in field_type.

To define {conditions = conditions}, the user must explicitly specify the exact field to be used in the comparison. The following comparison operators are supported: =, >=, >, <=, <.

All conditions within a single rule are combined with a logical AND (i.e., a node must satisfy all listed conditions for the rule to take effect). Example of a condition:

{condition 1 = value 1, condition 2 <= value 2}

where condition 1 / condition 2 are the fields on which the comparison condition is applied, and value 1 / value 2 are the corresponding values to compare against.

To specify [change_pattern = change_pattern] the user must define the exact modifications to be applied to the selected nodes. Example of a change_pattern:

[color = user_color, val = int(user_val)]

where user_color – the color specified by the user; user_val – the weight value assigned by the user.

Example of rules:

author.gender: <female> [color = red];
countries: <US, RU> [color = blue, group = countries];
eid: <*> {probability >= float(0.5)} [val = int(10), highlited = bool(True)]

In the graphical user interface, a dedicated tab has been implemented to allow users to input and apply rules. This tab serves as a rule editor where users can define and execute custom node modification rules (Fig. 11).

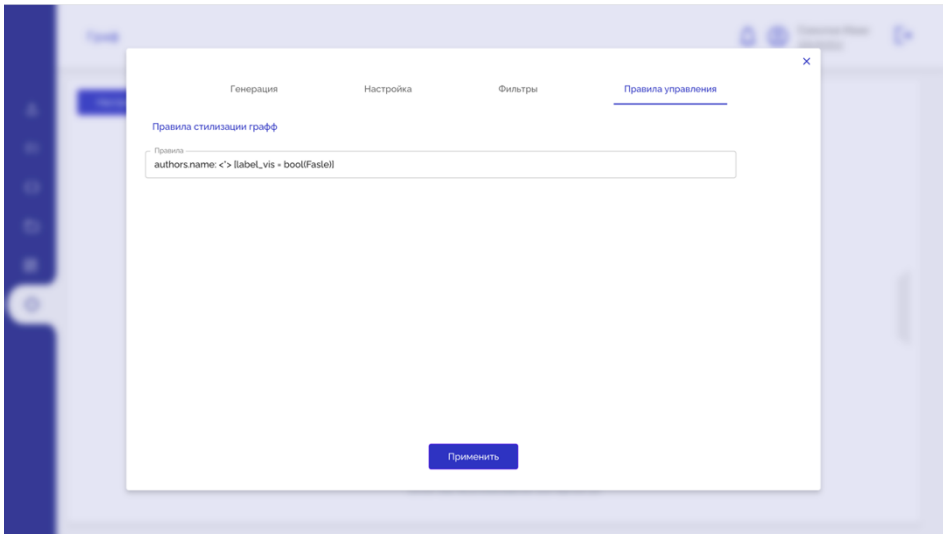


Fig. 11. Example of rule implementation for configuring graph nodes

7. Experimental Results

The developed tool is used to address scientific and technical tasks. As part of its validation, two case studies were examined: Detection of violations of nuclear non-proliferation obligations and Visual analysis of the publication activity of a scientific organization.

7.1. Detection of violations of nuclear non-proliferation obligations

The system validation for this task was conducted on 5,280 publications related to nuclear technologies. The following fields were selected for graph construction: countries — countries (green nodes), authors.full_name — full author name (blue nodes), eid — article identifier (brown nodes). The resulting graph is shown in Fig. 12.

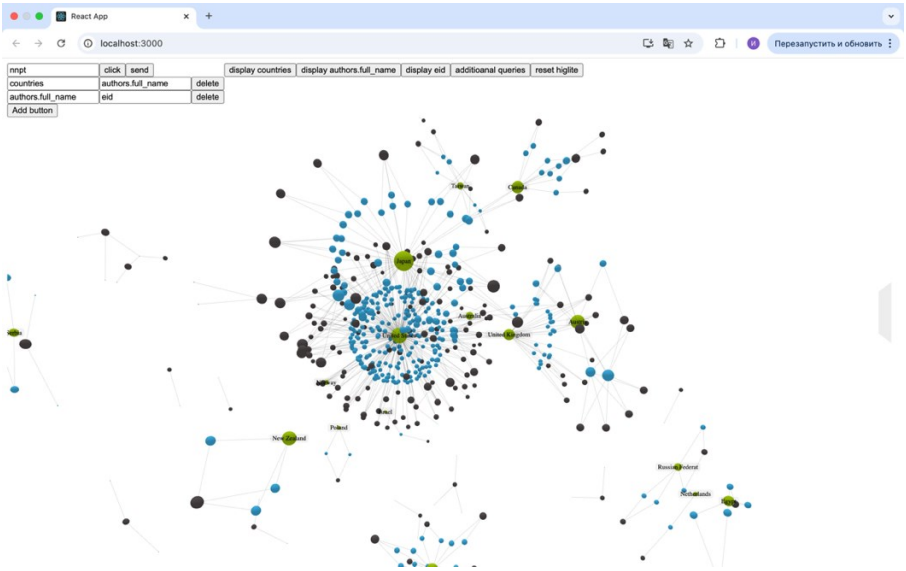


Fig. 12. Graph of authors, countries, and articles

To draw attention to nuclear-armed states, nodes representing the “nuclear five” countries are colored green, while all other countries are displayed in red.

During graph analysis, it was discovered that the United States has co-authored publications with two countries outside the nuclear five (Australia and Norway), prompting targeted

filtering. To reduce the number of articles, a filter was applied to United States-authored papers that must also include either Australia or Norway.

After applying this graph filter, it was revealed that Germany and Denmark also co-authored papers with the United States (Fig. 13). This allows experts to identify collaborative publications of particular interest for further investigation.

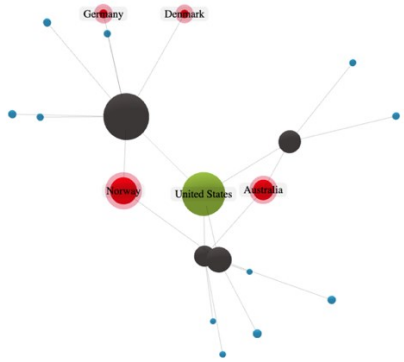


Fig. 13. Graph filtering by the United States

A graph of research topics by country has also been constructed (Fig. 14). From this graph, one can identify intersections of countries that publish on multiple topics (red intersections) and groups of countries that primarily focus on a single research topic (blue intersections).

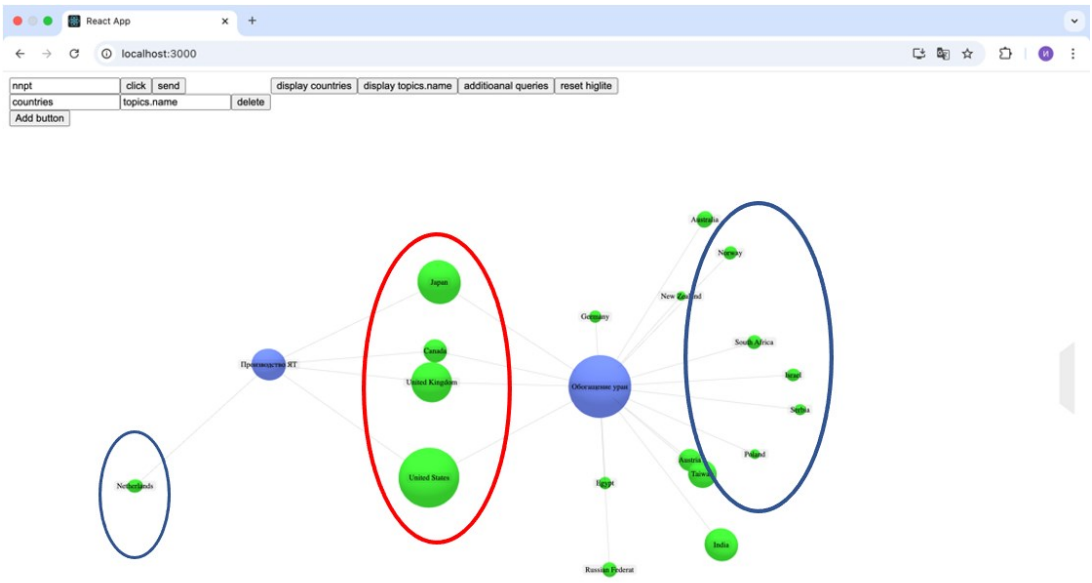


Fig. 14. Graph of countries' research topics

7.2. Visual analysis of the publication activity of a scientific organization

To analyze the structure and thematic focus of the publication activity of the Joint Institute for Nuclear Research (JINR), a graph-based visualization method was employed. Metadata from 36,008 publications were used as input data. The graph was constructed using the following fields: affiliations.name (affiliation name) — red nodes, keyword (keywords) — pink nodes. The resulting graph comprises 1,470 nodes and 5,291 edges (Fig. 15).

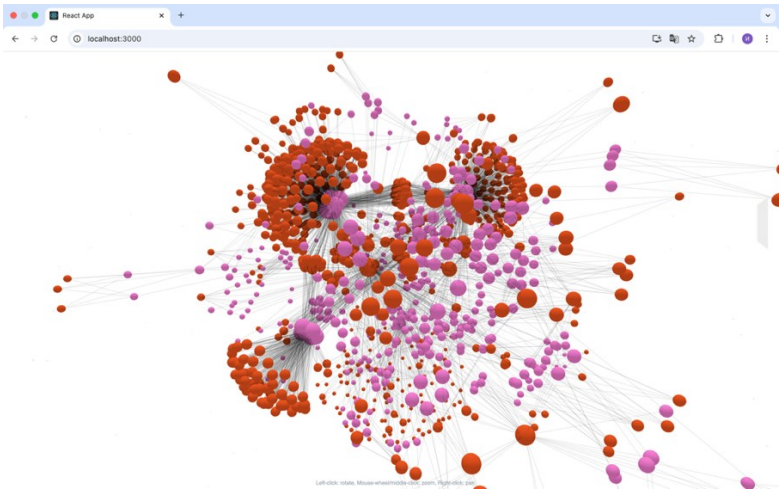


Fig. 15. Graph of JINR's publication activity

The resulting graph is divided into several clusters, each representing collaborations among organizations within specific research topics. Figure 16 presents the thematic areas of JINR's research. By examining the highlighted nodes, one can identify which organizations collaborate with JINR and the specific topics of their joint research.

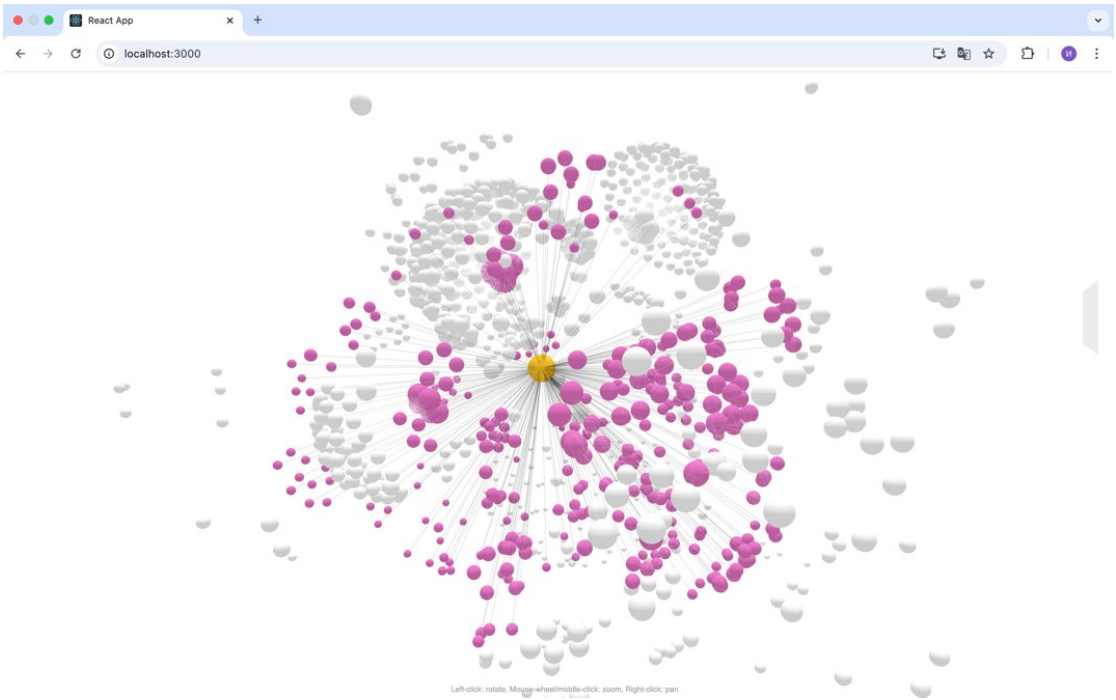


Fig. 16. Highlighting connections of the JINR node

To reduce data volume, filtering was applied to retain only publications involving JINR. As a result, a graph consisting of 1,406 nodes and 5,504 edges was produced (Fig. 17).

Fig. 17. Filtered graph for JINR

In the resulting graph, JINR's connections with external organizations are clearly visible. In particular, the cluster shown in Fig. 18 focuses on particle physics and illustrates the institute's primary collaborations in this field. Applying a keyword filter for "jets" and "quark gluon plasma" enables a more detailed view of these relationships and allows identification of specific partner organizations collaborating with JINR within this research area (Fig. 19).

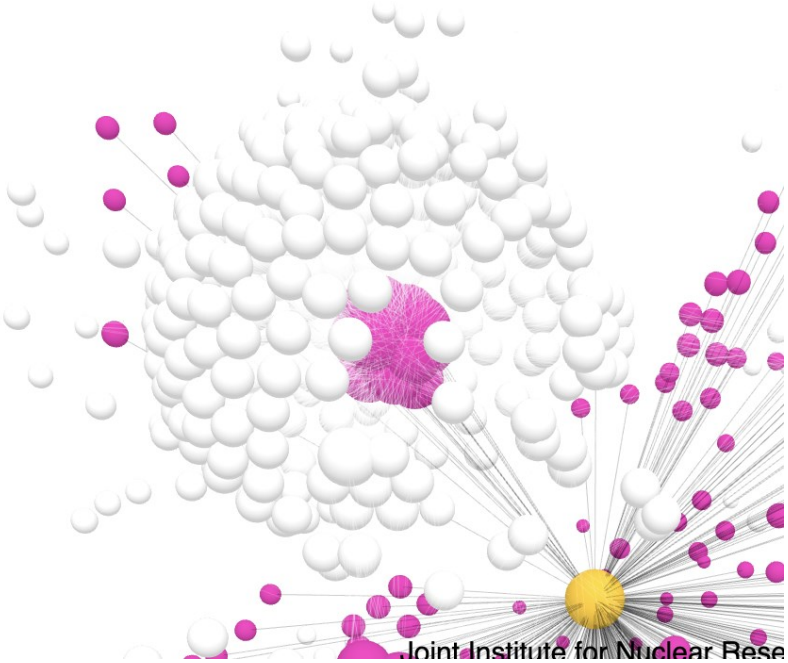


Fig. 18. JINR thematic cluster dedicated to particle physics

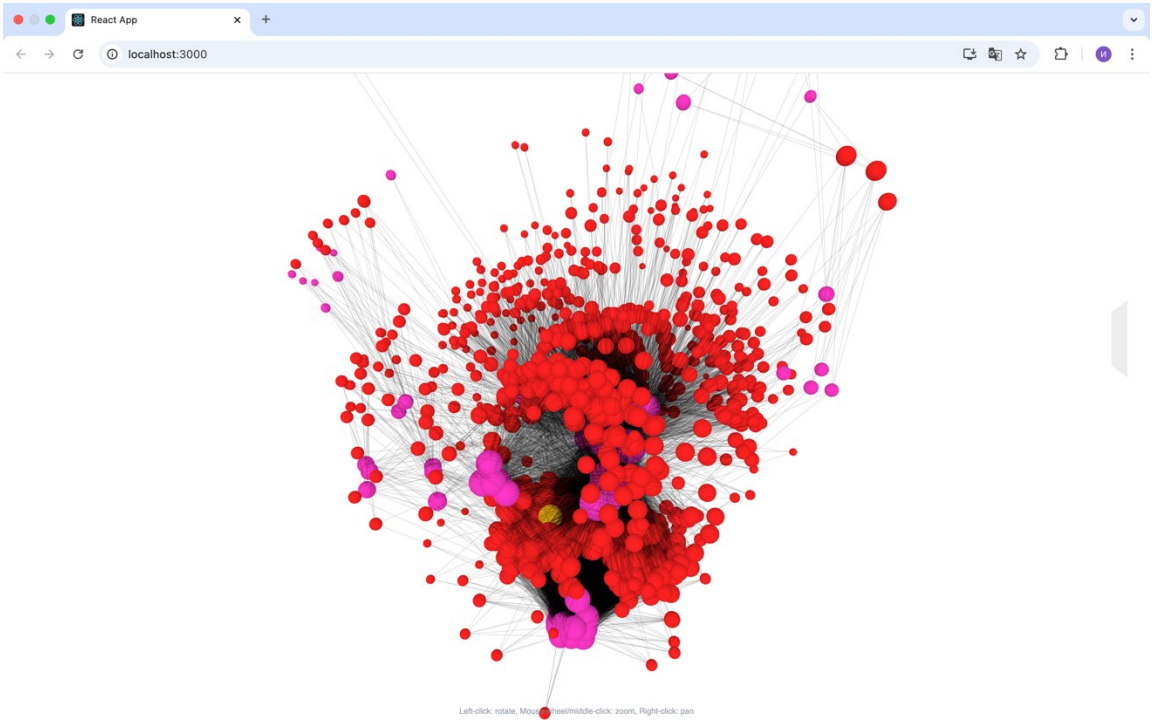


Fig. 19. Graph of organizational relationships within the selected research topic (JINR – yellow node)

8. Discussion of Results

The developed method for identifying explicit and implicit relationships between objects, along with the implemented graph modeling tool, demonstrated high effectiveness in solving analytical tasks of varying complexity. The key advantage of the proposed approach is its independence from the structure of input data, enabling it to process both simple tabular data and complex nested formats. This makes the tool universal and applicable to a wide range of systemic analysis tasks.

The user interface was implemented using ReactJS, while the data processing logic and storage layer were built on Python and Elasticsearch. Support for diverse data types, the ability to dynamically modify the graph, and the availability of interactive filtering mechanisms provide analysts with extensive control over the analytical process. This ensures the system is accessible not only to experienced specialists but also to users with minimal technical expertise.

Thus, the proposed method and graph modeling tool represent a powerful solution for uncovering both explicit and implicit relationships between objects, regardless of data structure or type. They can be applied across various domains—from scientific analysis and bibliometrics to political and economic monitoring—offering analysts a qualitatively new level of insight into complex systems and their internal interconnections.

9. Conclusions

In today's environment, where scientific and technical data volumes are growing rapidly, tools capable of transforming raw datasets into clear, analyzable graph structures are increasingly critical. This work presents a method for constructing graphs from input data of heterogeneous structure and for revealing both explicit and implicit relationships between objects, as well as a graph visualization tool that addresses the challenge of detecting interconnections among information entities. The tool enables analysts to define nodes of interest, specify linkage strategies, and customize both the graph structure and its visual representation.

Validation of the proposed method and tool was carried out through two applied scenarios involving relationship mapping between entities, specifically in the context of identifying re-

search topics by country and organization. Two datasets were used for this purpose: 5,280 documents on nuclear materials and 36,008 scientific publications from the Joint Institute for Nuclear Research (JINR).

References

1. Gephi, https://gephi.org, [last accessed: 15/07/2025].
2. VOSviwer, https://www.vosviewer.com/, [last accessed: 15/07/2025].
3. A.S. Kozitsyn, S.A. Afonin, “Discovering hidden dependencies between objects based on the analysis of large arrays of bibliographic data,” [in Russian], Proceedings of the 6th International Conference Actual Problems of System and Software Engineering, Moscow, 2019.
4. A. Timonin, A. Bozhday, “Methods of heterogeneous data analysis for social profile building,” Russian journal of management, 5(3), 2017, doi: 10.12737/article_59f5d8b75a2392.32622518.
5. D.V. Lindley, Regression and Correlation Analysis, London: Palgrave Macmillan UK, 1990.
6. M. Georgi, «Methods for Mining Causality from Observations in Artificial Intelligence,» Izvestiya SFedU. Engineering Sciences, V. 3(192), pp. 125-134, 2023 doi: 10.18522/2311-3103-2023-3-125-134.
7. S.V. Batishchev, T. V. Iskvarina, and P. O. Skobelev, “Metody i sredstva postroeniya ontologiy dlya intellektualizatsii seti internet,” Izvestiya Samarskogo nauchnogo tsentra RAN, V. 4(1), pp. 91–103, 2002.
8. T.V. Batura, “Metody i sistemy semanticheskogo analiza tekstov,” Software Journal: Theory and Applications, V. 12(4), 2016, doi: 10.15827/2311-6749.16.4.4.
9. Argo Lite, https://poloclub.github.io/argo-graph-lite/, [last accessed: 17/07/2025]
10. Neo4j, https://neo4j.com/, [last accessed: 17/07/2025]
11. PyVis, https://pyvis.readthedocs.io/en/latest/, [last accessed: 14/07/2025]