DiVE - DieLink Visualization Environment

Bachelor Thesis

Kevin Schuff

Supervisor:

Dr. Karsten Tolle
Institut für Informatik
Goethe-Universität Frankfurt am Main

Date of submission: 26.11.2025

Bitte dieses Formular zusammen mit der Abschlussarbeit abgeben!

Erklärung zur Abschlussarbeit

gemäß § 35, Abs. 16 der Ordnung für den Bachelorstudiengang Informatik vom 17. Juni 2019:

Hiermit erkläre ich

_____

(Nachname, Vorname)

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Ich bestätige außerdem, dass die vorliegende Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Zudem versichere ich, dass alle eingereichten schriftlichen gebundenen Versionen meiner vorliegenden Bachelorarbeit mit der digital eingereichten elektronischen Version meiner Bachelorarbeit übereinstimmen.

Frankfurt am Main, den

_____

Unterschrift der/des Studierenden

Abstract

Die studies are a procedure in numismatics where coins are compared and categorized according to whether they were produced using the same set of dies. The results of such analyses can be represented as graphs. The aim of this thesis is to determine whether existing tools or a newly developed standalone solution can support the creation of interactive visualizations for die study categorizations. Based on examinations of existing visualization approaches in numismatics, published die studies and discussions with domain experts, a set of requirements for such a tool was identified. Existing tools were evaluated against these requirements and it was found that none of them fulfilled the requirements completely. This motivated the development of a custom solution named DiVE (DieLink Visualization Environment), which meets most of the defined requirements and provides a tailored solution for supporting die studies. DiVE demonstrates the potential for digital tools to automate the manual steps involved in die study workflows.

Zusammenfassung

Stempelstudien sind ein Verfahren in der Numismatik, bei dem Münzen verglichen und danach kategorisiert werden, ob sie mit denselben Stempeln hergestellt wurden. Die Ergebnisse solcher Analysen können in Form von Graphen dargestellt werden. Das Ziel dieser Arbeit ist es, festzustellen, ob bestehende Tools oder eine neu entwickelte eigenständige Lösung die Erstellung interaktiver Visualisierungen für die Kategorisierung von Stempelstudien unterstützen können. Auf der Grundlage von Untersuchungen bestehender Visualisierungsansätze in der Numismatik, veröffentlichter Stempelstudien und Diskussionen mit Fachexperten wurde eine Reihe von Anforderungen an ein solches Tool ermittelt. Bestehende Tools wurden anhand dieser Anforderungen bewertet, wobei sich herausstellte, dass keines davon die Anforderungen vollständig erfüllte. Dies war der Anlass für die Entwicklung einer maßgeschneiderten Lösung namens DiVE (DieLink Visualization Environment), die die meisten der definierten Anforderungen erfüllt und eine maßgeschneiderte Lösung zur Unterstützung von Stempelstudien bietet. DiVE demonstriert das Potenzial digitaler Tools zur Automatisierung der manuellen Schritte im Arbeitsablauf von Stempelstudien.

Contents

List of Figures

List of Tables

1 Introduction

Die studies are an established method in numismatics for analyzing the production of coins. Their results can be represented as graphs (Yoon, 2025). The graph construction is often done manually with graphics editing software (Talbot, 2015). This part of the workflow is very labor intensive, while also having the potential to be automated.
Therefore the aim of this thesis is to determine the requirements for a tool that would be able to support die studies and evaluate if existing tools or a newly developed solution can fulfill these requirements. These research question will be guiding the approach:

- **RQ1:** What requirements must a software tool fulfill in order to support die studies through the creation of interactive visualization ?
- **RQ2:** To what extent do existing tools satisfy these requirements and what unmet needs remain?
- **RQ3:** How effectively does the developed solution address the identified gaps and other requirements ?

To achieve this, the work proceeds through several stages, beginning by the definition of the initial requirements. It then investigates whether existing visualization tools can meet these requirements and finds that none of them fulfills them completely. This leads to a prototyping phase, during which feedback from domain experts was gathered and additional requirements were identified. Followed by developing an implementation that aims to meet the combined set of requirements.
Afterwards `DiVE` (DieLink Visualization Environment) is presented by giving an Overview of the User interface and the system workflow. This section also includes justifications for design choices and explains important parts of the implementation. It is followed by an evaluation of the implementation, which investigates if it meets the identified requirements. Subsequently, the methodological approach, as well as challenges and limitations of the implementation will be discussed. The thesis concludes with a summary of the process and a reflection on the contributions of this work.

2 Background

During die studies, coins are compared and grouped into similar coins that have been struck with the same set of dies. Numismatics can draw two important conclusions

from grouping the coins. First, they can estimate the relative coin production of each type made. Second, die identifications can be used to derive the order in which coins have been produced. This second conclusion is based on die-links. Coins that share at least one die are considered die-linked. The concept of die-links forms the basis for organizing the coins into a graph. Since die-linked coins were struck around the same time, following these links can reveal a sequence of production (Yoon, 2025). A range of visualization tools is available for representing graph data. Gephi offers powerful network analysis with rich network visualizations and is widely used in academic publications (Bastian et al., 2009). Vistorian supports interactive network visualization and is also web-based, which removes the need for installation (Serrano Molinero et al., 2017). Orange is an open-source machine learning and data visualization tool that, through extensions, also supports graph visualization and customization (Demšar et al., 2013). Cytoscape, an open-source software platform, at first designed for biological research (Shannon et al., 2003).

Besides visualization tools there is also a variety of graph analysis libraries and graph visualization libraries. NetworkX, one of the popular graph analysis libraries, can create, manipulate and analyze graph structures (Hagberg et al., 2008). It can be combined with visualization libraries such as Pyvis, which offers built-in NetworkX integration (Perrone et al., 2020).

Another graph visualization library that integrates with NetworkX is Dash Cytoscape (Plotly, 2024). It combines Cytoscape with Dash, a web application framework that provides user interface elements which can be used to control the network visualization (Parmer et al., 2025).

3 Methodology

This thesis followed an experimental approach and iterative design process. The process began with defining the essential requirements that any potential solution should meet. The term *solution* is used deliberately, since at the initial stage of this work the specific form of the *solution* had not yet been determined. Several possibilities were considered, including developing of a standalone tool, extending an existing visualization tool or defining a workflow, which combines preprocessing the input and working with an existing tool.

Afterwards, a survey of existing tools was conducted to assess whether existing tools

could already meet these requirements. Based on this evaluation, the decision was made to explore the development of a standalone tool, to offer a more tailored solution.
Throughout the prototyping phase, feedback from both the advisor and a numismatic expert Markus Möller was gathered. Their input helped define additional requirements, that were essential for the solution and not apparent from the start.

3.1 Define requirements

The requirements of the potential solution were defined based on three main sources. First, initial discussions with the advisor regarding the input structure and control flow. Second, an inspection of existing implementations of coin visualizations, such as Gortana et al., 2018, which provided a clearer understanding of the functionalities an interactive visualization should provide and how an intuitive user experience can be achieved. Lastly, an examination of a typical die study from Talbot, 2015, which helped clarify what the output of a potential solution should look like.
The goal was to specify the essential requirements a solution should provide, taking into account what is technically feasible within the project's scope and timeframe. For the intended use, it is assumed that the tool receives a list of coins that has been labeled with their corresponding dies. Each row represent one coin and contains all the relevant information the user can provide, including a unique identifier for the coin, the associated dies, URLs to images of the coin and potentially other relevant metadata. An example of such a list of coins show in Table 1. Coin 1 and coin 2 for example would be considered die-linked, since they share the same front die. The graphstructure displaying the die-links would look like Figure 1.

coin id	front die	back die	front img	back img
1	V4	R5	Images/obv/1.jpg	Images/rev/1.jpg
2	V4	R4	Images/obv/2.jpg	Images/rev/2.jpg
3	V1	R4	Images/obv/3.jpg	Images/rev/3.jpg
4	V2	R1	Images/obv/4.jpg	Images/rev/4.jpg
5	V2	R2	Images/obv/5.jpg	Images/obv/5.jpg
6	V4	R2	Images/obv/6.jpg	Images/rev/6.jpg
7	V4	R3	Images/obv/7.jpg	Images/rev/7.jpg
8	V3	R5	Images/obv/8.jpg	Images/rev/8.jpg

Table 1: Example: labeled List of coins

The information from the list of coins would be used to construct graph data structures that represents the relationships between coins and dies. For this purpose, two graph views were considered:

- **coin view:** coins are represented as nodes and edges connect coins that were minted using the same die or die combination.
- **die view:** dies are represented as nodes and edges connect dies that were used together with edge weight proportional to the number of coins

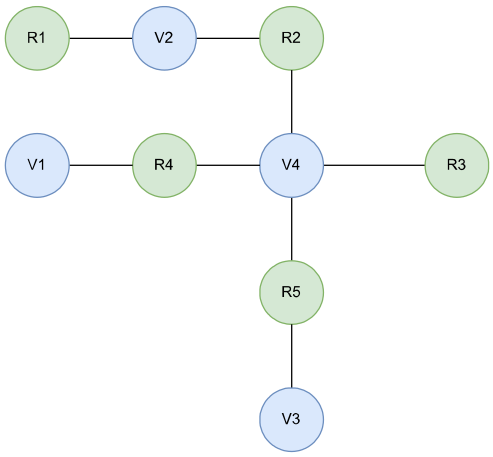

Figure 1: Structure of die study graph

These graph representations serve as input for the visualization component of the solution, which in turn should offer the option to display multiple views. Furthermore, the visualization should support automatic layouts that arrange nodes in a meaningful pattern while still allowing users to freely adjust node positions manually.
In addition, users should be able to adjust and customize the graph. This includes functionalities such as customizing or filtering nodes based on their attributes. When customizing the graph and dragging nodes around, it is useful to display additional information about the node in the form of a tooltip. Included URLs or file paths to coin images in the input, should be used as node images in both views.
Because the intended users are from a non-technical field, it is especially important that the user experience is clear, intuitive and require minimal configuration. Finally, the solution should be able to export either the full graph or a selected section of the visualization as a standard image file (PNG, JPG, or SVG). These requirements are summarized in Table 2 and will be used to evaluate the selected tools in terms of their suitability for addressing the problem.

3.2 Technologies survey

To determine whether existing visualization tools already meet the previously defined requirements, several tools were tested and evaluated. It also should be emphasized

that this evaluation is highly subjective. The intention is not to offer an objective benchmarking of existing tools, nor to diminish any efforts that went into developing these tools. Rather, the goal is to assess their suitability within this specific context.

	Gephi	Cytoscape	Orange3 Network	Vistorian
data transformation	✗	✗	✗	✗
multiple views	✓	✓	✓	✓
freely adjustable Layouts	✓	✓	✗	✗
customizing	✓	✓	✓	✓
filtering	✓	✓	✗	✗
tooltip	✓	✓	✓	✓
node pictures	✗	✓	✗	✗
clear and intuitive UX	✗	✗	✓	✓
export	✓	✓	✓	✓

Table 2: Tool evaluation based on initial requirements

3.2.1 Data transformation

For all evaluated tools, the expected input format, such as the onw shown in Table 1 did not match the input formats required by these tools. Therefore, a small labeled coin dataset was converted into node and edge lists corresponding to the two views described earlier (coin view and die view), since the tools require different data structures and do not provide such a conversion by default.
Therefore, this requirement was not met by any of the evaluated tools initially. However, it could be addressed by adding a preprocessing step in the form of a Python script. While this may present a drawback for a non-technical user base if no easy to use interface is provided, it could again be resolved by implementing simple user interface that controls this preprocessing step.

3.2.2 Multiple views

The solution requires support for multiple views and this requirement is met by all evaluated tools. Gephi and Cytoscape offer multiple visualization tabs between users can easily switch. Orange3 Network displays a graph in a separate pop-up window, allowing multiple windows to be open simultaneously. In the intended use, this may be advantageous, since the two graphs are related in terms of content. This becomes especially relevant when considering more complex features such as linked views, for

example when selection a coin in the coin view results in highlighting the dies connected to the coin in the die view. This would benefit from a side-by-side visualization. Orange3 Network does not provide such linked-view functionality, but its pop-up window approach would for a foundation for implementing it. In Vistorian, which is browser based, multiple views can be opened simply in different tabs.
All tools therefore support multiple views, highlighting Orange3 Networks approach, as it could accommodate future features such as linked views.

3.2.3 Freely adjustable Layouts

The solution should provide at least one layout algorithm that arranges the nodes in a meaningful way, which includes avoid overlap, minimizing edge intersections and ensuring sufficient spacing between components. It would be beneficial to have multiple layouts, since there is no scientific consensus in numismatics on how to present a die study in numismatics on how a die study should be presented. In this regard, Gephi and Cytoscape offer a wide range of layout options. Vistorian also offers a variety of layouts suitable for the intended use and Orange3 Network only offers one layout. In contrast to Gephi and Cytoscape, neither Orange3 Network nor Vistorian allows users to freely adjust the positioning of nodes.

3.2.4 Customizing

Customizing node appearance makes group of nodes with similar attributes visually consistent, allowing each group to be easily distinguished. This can be achieved through multiple approaches, such as border color, size, shape and labels. For the intended use border color appeared to be the most suitable option, since it is the only method besides labels, which should remain reserved for identification, which does not interfere with using images as nodes. It would also be useful to support combinations of attributes, for example when a user wants to highlight coins with a specific combination of die affiliation and condition.
All evaluated tools offer a form of discrete mapping for a chosen attribute, where each attribute value can be assigned a distinct border color (either manually or automatically). These mappings are flexible in Gephi and Cytoscape and can be changed. In Orange3 Network and Vistorian, the user can choose which attribute to map, but cannot control the specific color assignments.
In regards to attribute combinations in Gephi and Cytoscape it is possible, but not

intuitive, to first apply a filter for a specific attribute combination and afterwards only color those specific nodes. Orange3 Network and Vistorian, do not offer support for attribute combinations.

3.2.5 Filtering

In the specific intended use, filtering can be used to create a selection of nodes that meet certain conditions. In addition to customizing these nodes, it should also be possible to hide them or to show only the selected nodes. Gephi has extensive options for filtering a graph. However, only one filter can be active at a time. This is inconvenient, when considering this scenario: A user might first create a time consuming filter to hide a specific set of nodes and later decides to customize nodes based on a combination of attributes. In this case, the user must delete the initial filter and looses the previous work, because filtering is also required for node customization. The key issue is that filtering and hiding are intertwined.
In contrast, Cytoscape handles this differently. The user can first filter and hide nodes and then customize other nodes using the filter feature again. This is possible because Cytoscape's filter feature only selects nodes, while the hiding functionality is independent from filtering.
Orange3 Network has similar functionality, where the user can filter nodes, which creates a highlighted selection, but there is no way to hide this selection. Vistorian does not offer any filtering functionality.

3.2.6 Tooltip

The tooltip should provide additional or complete information about a node when the user either hovers over it or clicks on it. This helps users identify inconsistencies in their data and encourages exploration. Gephi, when used with an official plugin, and Orange3 Network both satisfy this requirement. Cytoscape does not include a built-in tooltip, but clicking a node filters the visible node table to display only that node's information. Vistorian includes an on-hover tooltip, although the displayed information is limited to the selected label field.

3.2.7 Clear and intuitive user experience

Because the potential users come from a non-technical niche, the user interface should be clear and present only the features and information that are relevant to them. Fur-

thermore the user experience should be intuitive, for example, if a user want to hide nodes, it should be immediately obvious which interface element provides this functionality.
In Gephi, the filter feature in particular is not intuitive, as it requires understanding of querying and offers considerably more functionality than is needed for the intended use. Additionally, the way graph customization features are split between the Overview and Preview tabs can be confusing for inexperienced users. Cytoscape's filtering approach, while offering less flexibility, is more intuitive and easier for inexperienced users to understand. This consolidation contributes to a more approachable user experience, even if the overall design remains complex.
With feature rich tools like Gephi and Cytoscape it becomes apparent that keeping the user interface clear and the user experience intuitive is challenging. Because these tools serve a wide range of complex use cases, they present a large number of functionalities on the screen at once. This was considered not ideal, especially for new and non-technical users.
Orange3 Network and Vistorian, while offering less functionality, offer a clear and more intuitive user experience.

3.2.8 Node pictures

The provided URLs to images in the input should be integrated in the graphs as nodes. Cytoscape supports passthrough mapping of attributes, allowing the user to select columns containing the URLs, which are then used as nodes. Gephi, Orange3 Network and Vistorian, on the other hand do not offer any built-in support for this requirement.

3.2.9 Export

The solution should allow the export of graph as image files. All tools meet this requirement, particularly Gephi and Cytoscape, which offer users control over the quality of the exported images.

3.3 Standalone

Since no existing tool fulfilled all requirements, the development of a standalone tool was explored. A custom solution could address the specific needs of the intended use case more effectively than the evaluated tools.

3.3.1 Prototyping

Before selecting technologies for developing a standalone tool, the focus was deliberately limited to a Python based solution. This ensures that future developers, likely other students, can easily understand, maintain or extend the tool, since Python is part of the university's introductory curriculum.
For the graph analysis component, NetworkX was the only candidate examined, due to it's wide adoption in academia and its integration with several graph visualization libraries.
For the graph visualization component, PyVis and Dash Cytoscape were compared. Both support filtering and highlighting, but they differ significantly in how these features are implemented. In PyVis, this functionality is provided through built-in UI elements, which are simple to use. However, these UI elements offer no customization. In contrast Dash Cytoscape allows the interface to be customized to specific needs, which requires more implementation effort, since the developer has to define how each UI element interacts with the visualization. This offers great flexibilty in designing the user experience. Therefore NetworkX combined with Dash Cytoscape was considered the more suitable choice.

3.4 Requirements Extension

During testing out libraries for a standalone tool and further discussions with the advisor and numismatics expert Markus Möller, it became obvious the requirement list had be extended, to accommodate additional functionality ideas.

- **node pictures full size:** Because the current requirement only entails node sized pictures, the tool should also offer a more detailed version for inspection.
- **flexible edge condition:** Since both options of the die assignments are important edge conditions. The tool should offer at least these two, or even be flexible and offer the switch between them.
- **adaptive node images:** Building on the last requirement, the node images should automatically change depending on the edge condition.
- **interlinking views:** In terms of content the two views are already connected. So changes in one view should affect the other.

4 Implementation

This section presents the implementation of the developed tool. It begins by introducing the components of the user interface and providing an overview of the system workflow, before examining the most important features in detail. This includes discussion design choices that were made and the reasoning behind them. Finally, a use case is presented to demonstrate the tool's capabilities.

4.1 Overview of the User Interface

Based on the requirement, the user experience aims to be clear and intuitive. Therefore, the user interface is divided into two components, making it easy to navigate and understand. It displays only one visualization at a time, but the user can switch back and forth via the view-selector at any time. The sidebar contains all controls for manipulating the visualization. These include customizing nodes through coloring, hiding specific nodes and selecting a layout used to arrange the nodes. It also contains the node details box and the export button.

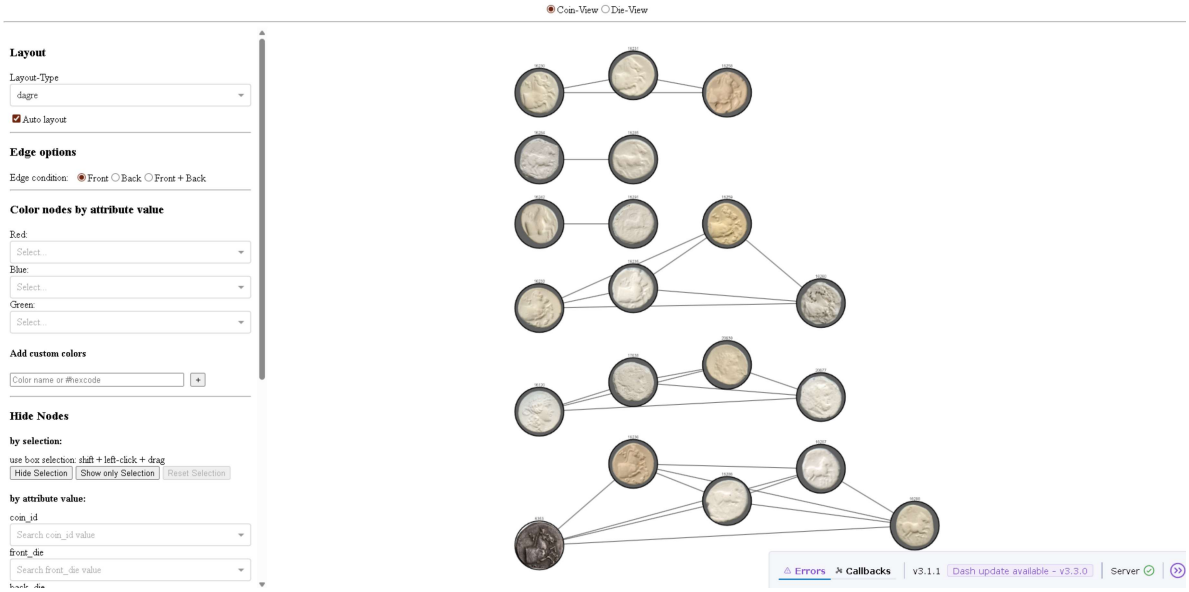

Figure 2: Application's UI (pictures from Corpus Nummorum)

4.2 Overview of the system workflow

Overview of the system workflow diagram depicts the workflow of the application. Upon starting the application, the user is greeted with a start overlay, where they can choose their input file, which should be a labeled list of coins similar to Example: labeled List of coins. They must also specify the input fields containing the die assignments and the paths to the coin images. These will be referred to as:

- **front die**: the die used to strike the front (obverse) side of a coin
- **back die**: the die used to strike the back (reverse) side of a coin
- **front image**: image of front side of coin
- **back image**: image of back side of coin

Once the user has chosen a file, the tool suggests an input limitation for files exceeding the implemented limitation. In either case, the application continues by verifying that the file is not empty and constructing the coin graph structure using NetworkX. It iterates through each line of the input file, creating a node and adding all field values as attributes, with the first value also serving as the node's identifier. The edges will than be added based on the default setting. This means all nodes that have the same value in the front die field, will be connected. The edge label will be the value of the front die field.
Once the coin graph structure has been created, the application utilizes the graph's nodes to build parts of the visualization controls and to construct the die graph structure. Afterwards, both graph structures will be used as input to create their respective visualizations. These visualizations request their images from the internal Flask server. The Flask server downloads the images from the sources and serves them to the visualization.
At this point, the application waits for further interactions from the user. When using the `graph-view selector`, users can switch between the two visualizations, meaning only one of the visualization is visible at a time. Interacting with any of the visualization controls triggers an update to the visualizations. This includes:

- **Layout**: changes how the nodes in the visualizations are arranged

- **Color nodes**: highlights nodes with certain attribute-value combinations with a selected color (only in coin view).
- **Hide nodes**: hide nodes based on a selection or based on attributes
- **Edge options**: In coin view, allows changing the edge condition and in die view allows scaling edge thickness based on edge weight.

The other components of the user interface, grouped under Information and Output, request information from the currently active visualization.

- **statistics**: provides updates on current number of coins, dies and visible components
- **node details**: displays additional information about nodes currently hovered over
- **export**: exports the active visualization

Exporting a visualization typically marks the end of a use case scenario.

4.3 Features

After presenting the overall workflow, the following sections examine each feature in detail. For every feature the potential implementation options are discussed and evaluated based on feasibility, suitability and consultations with the advisor. The implementation is presented throughout, occasionally supported by code excerpts. Code sections marked with "..." indicate omitted parts.

4.3.1 Dash Cytoscape

This is a brief introduction some Dash Cytoscape related terms that are relevant for understanding the implementation. Every Dash Cytoscape component has the key properties: elements, stylesheet, layout.
Elements contains all graph data, for nodes that includes the node attributes. The **Stylesheet** defines the visual appreance of nodes and edges and is completely separate from the actual graph data. It consists of pairs of `selectors`, which target specific elements, and `style` definitions, which specify the visual properties of those elements. The **Layout** determines how the positioning of nodes is computed.

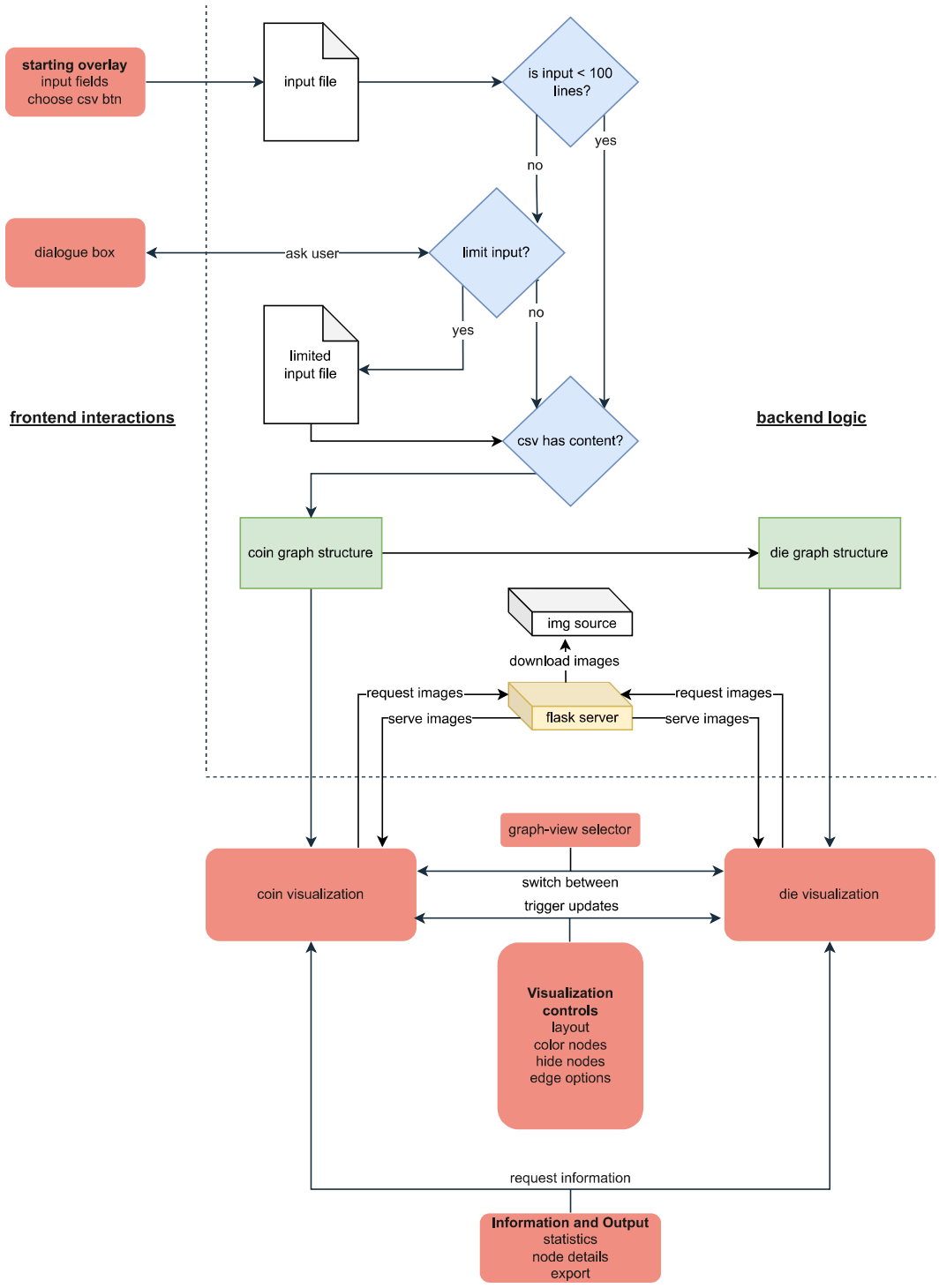

Figure 3: Overview of the system workflow diagram

Dash callbacks are used to react on changes in components such as dropdowns or buttons and to update other components accordingly. In the context of this implementation, callbacks are used to trigger updates to the visualizations.

4.3.2 Data formatting

Converting the list of coins into the input format required by the Cytoscape component involved two considerations. The first was determining how the application should obtain the relevant field names from the list of coins. This could be achieved either by requiring a standardized table format or by allowing the user to specify the relevant field names manually. The latter option was chosen, as it was considered more appropriate for this application.
The second consideration was whether making an intermediate step in the conversion process would be beneficial, providing access to a graph data structure. In this approach, the list of coins is first converted into a NetworkX Graph, which is then transformed into an element list for the Cytoscape component. Although this introduces an additional conversion step, the NetworkX Graph provides access to built-in functionality such manipulation of the graph structure, efficient iteration over graph nodes and statistical calculations. Additionally the node's of the coin graph can be used to efficiently construct the die graph. For these reasons, this option was considered advantageous and was chosen.

```
coin_graph = load_graph_from_csv(decoded)
# build edges according to selected mode
add_edges_by_mode(coin_graph, ...)
# build die-graph with input columns
die_graph, _ = create_dies_graph(coin_graph, ...)
# cytoscape elements for graphs
coins_base_elements = nx_to_elements(coin_graph)
dies_elements = nx_to_elements(die_graph)
```

4.3.3 Interlinking views

As described in Section 3.1, the application should display at least two visualizations.

- **coin view:** coins are represented as nodes and edges connect coins that were minted using the same die or die combination.

- **die view:** dies are represented as nodes and edges connect dies that were used together with edge weight proportional to the number of coins

These two views should be linked together, meaning that interactions performed in one visualization should influence the representation in the other. Several implementation for achieving this behavior were considered:

- **Selections-based linking**, where selecting nodes in one view highlights the associated nodes in the other.
- **Visibility-based linking**, where hiding or showing specific nodes in one view updates the visibility of the corresponding nodes in the other.

While both options were suitable, Visibility-based linking was chosen because it also addressed other issues identified during discussions with the advisor, such as the need for a more convenient way to hide a specific groups of nodes. The specific implementation is explained in the next Section 4.3.4.
Another point of consideration was whether the two views should be displayed simultaneously or whether only one should be visible at a time, with the option to switch between them. Displaying the linked views side by side allows users to immediately observe how interactions in one view effect the other. However, this approach carries the potential downside (especially as the input size increases) of making the user interface visually overloaded. Therefore, displaying only one view at a time with the option to switch between them was chosen.
In the implementation, the application controls the visibility of the views using a Dash radio-button component `id='graph-view-selector'`, which triggers a callback function whenever its value changes.

```
@app.callback(
    Output('cy-coins', 'style'),
    Output('cy-dies', 'style'),
    ...
    Input('graph-view-selector', 'value'),
    )
def toggle_visible_view(view):
    base = {'width': '100%', 'height': '100%'}
    if view == 'dies':
        return {**base, 'display': 'none'}, {**base, 'display': 'block'}, ...
```

```
    else:
        return {**base, 'display': 'block'}, {**base, 'display': 'none'}, ...
```

4.3.4 Hiding Nodes

The application allows users to hide nodes based on their attributes. Two approaches were considered: filtering out all nodes with a specific attribute value or filtering out all nodes that lack the specific attribute value. While both approaches were suitable, the first option was deemed more practical for the intended used of the application.
During discussions with the advisor, it became clear that attribute-based hiding has its uses, but may not be intuitive for all users. Therefore, the application also supports selection-based hiding. Users can create a box selection with holding Shift key, then left-clicking and dragging the cursor. Afterwards, they can use the `Hide Selection` button to hide all nodes within the box selection or the `Show only Selection` Button to hide all nodes outside the box selection. A `Reset Selection` button becomes available once nodes have been hidden with through this feature and allows the user the undo any selection-based hiding.
This feature also enables the visibility-based interlinking between the coin view and the die view. To illustrate how this interlinking operates, the following section examines two cases and the considered strategies to handle them:

The first case concerns hiding a selection of nodes in the coin view and how this affects the die view. The straightforward approach would be to hide all dies associated with the hidden coins. However, this may hide dies even when other visible coins are still linked to them. To avoid this issue, the implementation excludes hidden coins when reconstructing the die graph. As a result, all dies that are associated with at least one visible coin remain visible. This strategy is illustrated in Figure 4.

The second case concerns hiding a selection of nodes in the die view and how it affects the coin view. A possible approach would be to hide a coin if both of its associated dies are hidden. Nevertheless, it was considered more intuitive that hiding a die should also hide all coins associated with it. This process is illustrated in Figure 5.
Because the first case does not use the straightforward approach, the die view's elements must be updated when hiding occurs. Hiding coins may reduce edge weights in the die view, since edge weights correspond to the number of coins associated with a pair of dies. Rather than manipulating the elements list directly it is more efficient to rebuild the graph and apply the necessary adjustments there. In the coin view, we only show

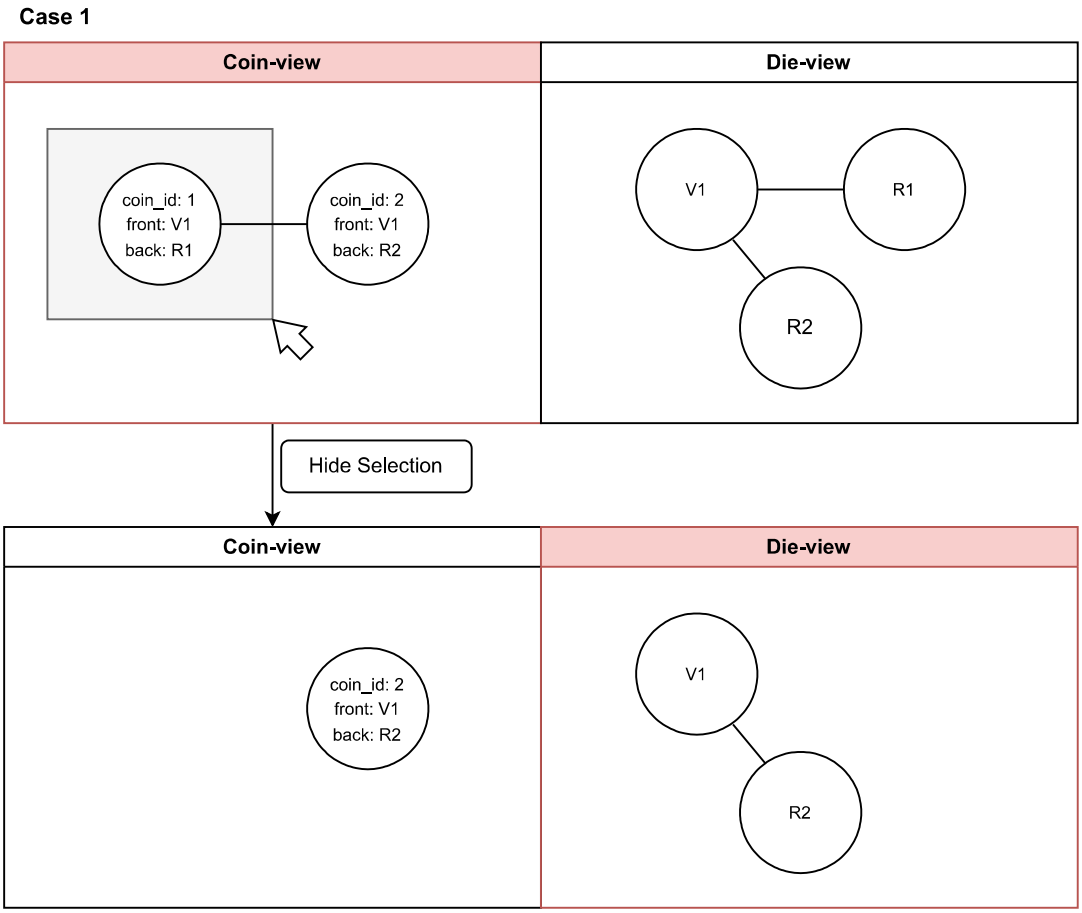

Figure 4: Effects of hiding coins

or hide nodes, so this can achieved through the stylsheet property.

For attribute-based hiding, the user interface provides dropdown menus. Any change triggers the rebuilding of the stylesheet for the coin view. The stylesheet contains, for every attribute value in all dropdowns, a selector, which targets all nodes with this specific attribute value and hides them. Because the views are intended to be interlinked, the application also updates the die view by rebuilding the die graph from the filtered coin graph.

```
for attr, values in filter_store.items():
    for val in values or []:
        hiding_rules.append({
            'selector': f"node[{attr}='{css_escape(val)}']",
```

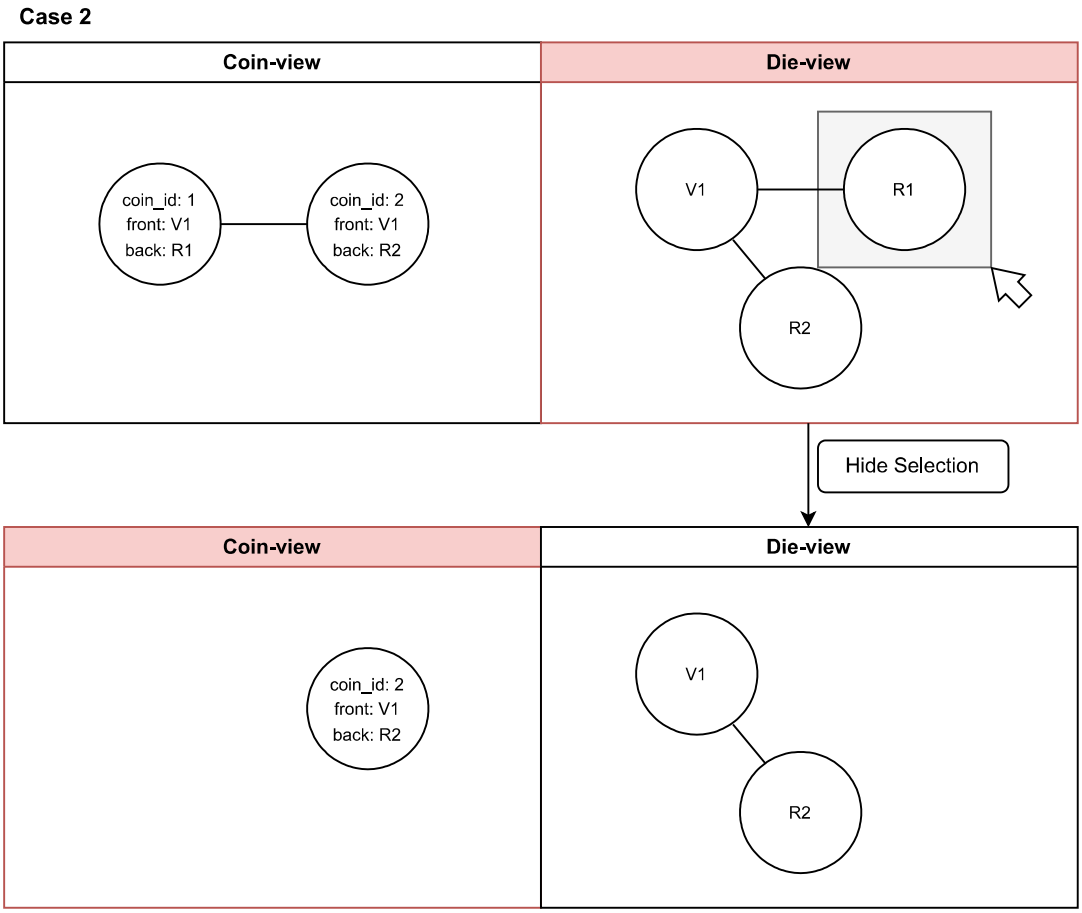

Figure 5: Effects of hiding dies

```
        'style': {'display': 'none'}
    })
```

For selection-based hiding, the implementation takes the filtered coin graph, the hidden store (where the information of nodes from previous selection-based actions are stored) and the currently selected nodes. From this information it rebuilds the filtered die graph, updates the die and coin stylesheet, which are applied to the corresponding views.

When showing only a selection, the logic is similar.Instead of taking the currently selected nodes, the implementation gets relevant information from all nodes and than removes the ones from the selection. Figure 6 provies an overview of the logic behind

the hiding process.

4.3.5 Edge Condition and Mode

The edge condition defines which nodes attributes must match for an edge to be created between two nodes. In the die view, the edge conditions follows the conventions used in die studies: if two dies were used together to mint a coin, then these dies are connected by an edge.
In contrast, the edge condition for the coin view was not predetermined. Therefore, the application provides a flexible approach. Since the purpose of the application is to support die studies, it was the natural conclusion to derive the edge condition from the die assignments associated with each coin. Therefore the application offers users to choose from the following edge conditions:

- **front :** Nodes (Coins) with the same front die assignment are connected by edge.
- **back :** Nodes (Coins) with the same back die assignment are connected by edge.
- **front + back :** Nodes (Coins) with the same front die and back die assignment are connected by edge.

In the implementation, this functionality is realized through a Dash radio-button component `edge-mode`, which triggers a callback function whenever its value changes. The callback loads the stored coin graph, removes all existing edges and then adds new edges based on the selected `edge-mode`. In `add_edges_by_mode()` the implementation iterates over all node pairs and based on the selected `edge-mode`, compares the relevant attributes. If they match, an edge is added.

```
nodes = list(G.nodes(data=True))
# compare every node pair
for i in range(len(nodes) - 1):
    u_id, u_dict = nodes[i]
    # extract relevant attributes from node u
    front_u = str(u_dict.get(front_key, "")).strip()
    back_u = str(u_dict.get(back_key, "")).strip()

    for j in range(i + 1, len(nodes)):
        v_id, v_dict = nodes[j]
        # extract relevant attributes from node v
```

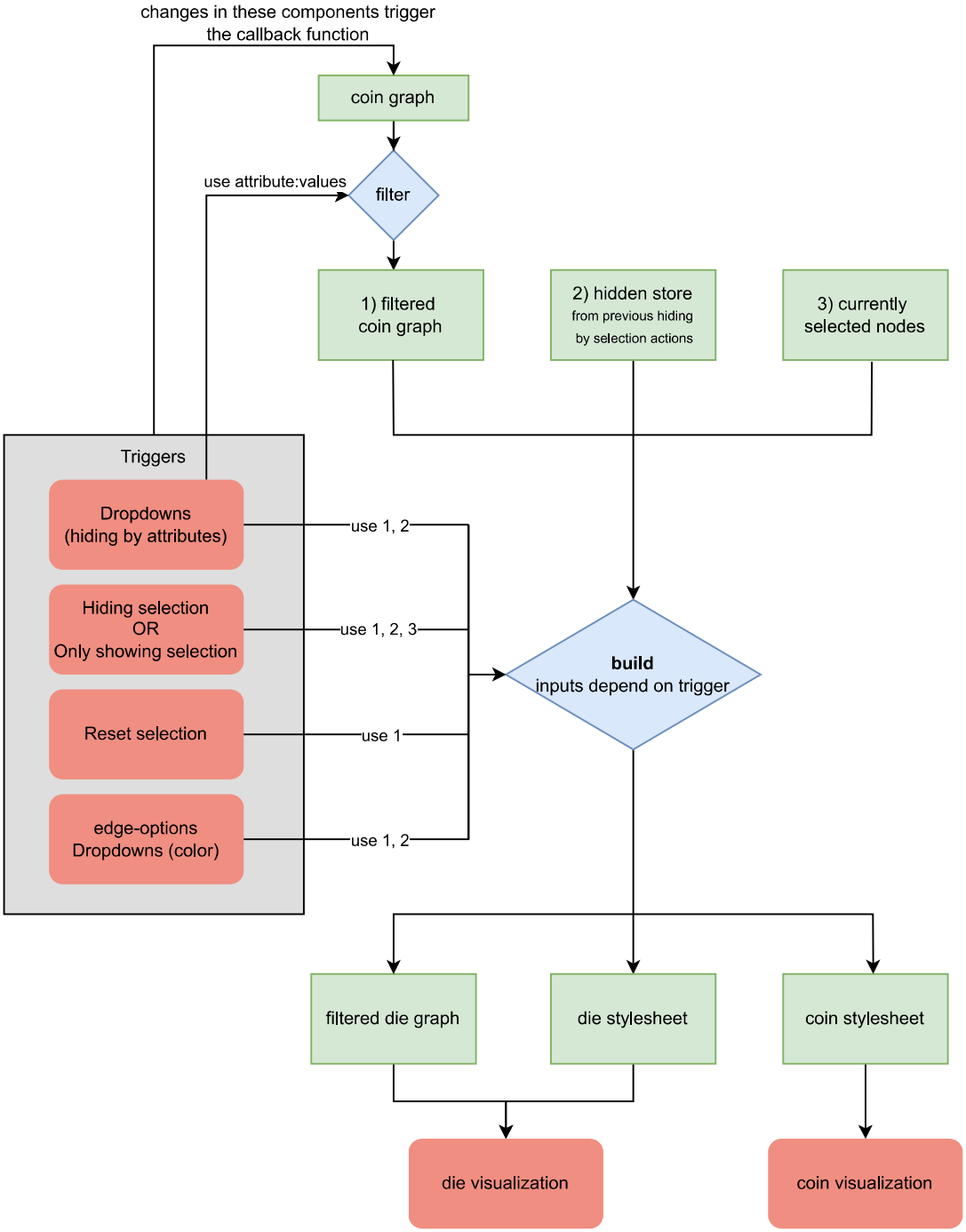

Figure 6: Workflow for hiding nodes

```
front_v = str(v_dict.get(front_key, "")).strip()
back_v = str(v_dict.get(back_key, "")).strip()

# add edge between nodes, if attribute from associated mode matches
if mode == 'front' and front_u and front_u == front_v:
    G.add_edge(u_id, v_id, attr='same_front', label=front_u)
elif mode == 'back' and back_u and back_u == back_v:
    G.add_edge(u_id, v_id, attr='same_back', label=back_u)
elif mode == 'both' and front_u and back_u and front_u == front_v and back_u == back_v:
    G.add_edge(u_id, v_id, attr='same_front_back', label=(front_u + '/' + back_v))
```

4.3.6 Layouts

As evaluated earlier in Section 3.2, Cytoscape's layouts fulfill the layout requirement by offering algorithms that arrange the nodes in a meaningful way, avoiding overlapping, minimizing intersecting edges and provide space between components. The application offers a selection of layouts from Cytoscape's built-in options as well as external layouts from Dash Cytoscape that were considered suitable for die studies.
Each layout comes with its own set of configuration options, which could theoretically be offered to the user. However, this was deliberately avoided, as the potential information overload was deemed to outweigh the potential benefit of increased customization. It would also be possible to have external algorithms calculate the node positioning and use the elements position property to assign the resulting positioning. Since this was not a priority, it was not explored further.
It still had to be decided, when to apply the layouts, considered options were:

- **Dropdown with layouts with attached button**, so when the button is pressed the currently selected layout is applied.
- **Dropdown with layouts without button**, so when the selection in the dropdown changes, the currently selected layout is applied.

Both options were viable, but the latter was selected because it aligned better with subsequent design decisions.
Every Cytoscape component includes an`autoRefreshlayout` property of ,which, if enabled, refreshes the layout on every change in the element list. Due to the implementation structure, such changes occur frequently. While automatic refreshing is useful in most situations, it becomes undesirable once the user wants to manually arrange nodes prepare for export. Therefore, the application provides a toggle button to disable this

behavior.
In the implementation, this callback function primarily controls the application of layouts. This means that you can always apply a layout via the layout dropdown, regardless of the autoRefreshLayout property.

```
@app.callback(
    Output('cy-coins', 'layout'),
    Output('cy-dies', 'layout'),
    Input('layout-selector', 'value'),
    State('graph-view-selector', 'value'),
    State('auto-layout-toggle', 'value'),
    prevent_initial_call=True
)
def set_layout(selected_layout, active_view, auto_layout_toggle):
    auto_enabled = 'on' in (auto_layout_toggle or [])
    layout = build_layout(selected_layout)
    # If auto-layout is off, only change layout on layout-selector
    if not auto_enabled:
        if ctx.triggered_id == 'layout-selector':
                    if active_view == 'coins':
                        return layout, no_update
                    elif active_view == 'dies':
                        return no_update, layout
        else:
            return no_update, no_update
    # Apply layout only to the currently active view
    if active_view == 'coins':
        return layout, no_update
    else:
        return no_update, layout
```

The `dcc.store layout-choices` keeps track of the last selected layout in each view and synchronizes the layout dropdown, when switching the view. This callback function synchronizes the layout dropdown, when switching the view.

```
@app.callback(
    Output('layout-selector', 'value'),
    Input('graph-view-selector', 'value'),
    State('layout-choices', 'data'),
)
def sync_dropdown_to_view(active_view, choices):
    choices = choices or {}
    return choices.get(active_view, 'dagre')
```

4.3.7 Customizing nodes

The first design decision was to choose which visual encoding the application should offer for customize node appearance. These visual encoding options were considered not suitable to our application: *Border thickness and sizing* are only meaningful for attributes with numerical values,and noticeable differences require large numeric ranges. Using *Shapes*, did not align with the intended use, since both coins and dies (represented through either side of a coin) are conventionally depicted as circular nodes.
After ruling out these options, *labeling* proved to be a suitable visual encoding. As additional text next to the identifier can be easily added through the stylesheet.`Border color` was also identified as a viable option. Due to the time constraint, only the border color approach was implemented. It was prioritized because it allows the encoding of attribute combinations, whereas encoding multiple attribute through labeling can become visually overwhelming.
Three possible methods for customizing node appearance were examined. The first relied on the user making a *selection of nodes* (through box selection or multi selection) and then assign a color to the selected set. The second option was *attribute based mapping*, where a single attribute is chosen and each possible value is mapped to a corresponding color. These two approaches are easy to use and understand, but lack in flexibility. Third option is based on *attribute combinations*, allowing the user to define logical conjunctions of attribute value pairs for each color. Nodes matching the conjunction are automatically colored accordingly. Due to its flexibly, the third option was chosen to be implemented. Currently, the Implementation offers this feature only for the coin view. Therefore it is hidden, when switching to die view. The simplified workflow for this feature is displayed in Figure 7. Once the uploaded file has been approved the associated callback function goes through all nodes in the coin graph extracting each unique attribute value pair, that will be used to populate the default color dropdowns. The application offers three color dropdowns on start, but the user can at any time add additional dropdowns, if needed.

```
@app.callback(
    ...
    Output({'type': 'color-dropdown', 'index': 'red'}, 'options'),
    Output({'type': 'color-dropdown', 'index': 'blue'}, 'options'),
    Output({'type': 'color-dropdown', 'index': 'green'}, 'options'),
    Input('csv-approved', 'data'),
    ...
```

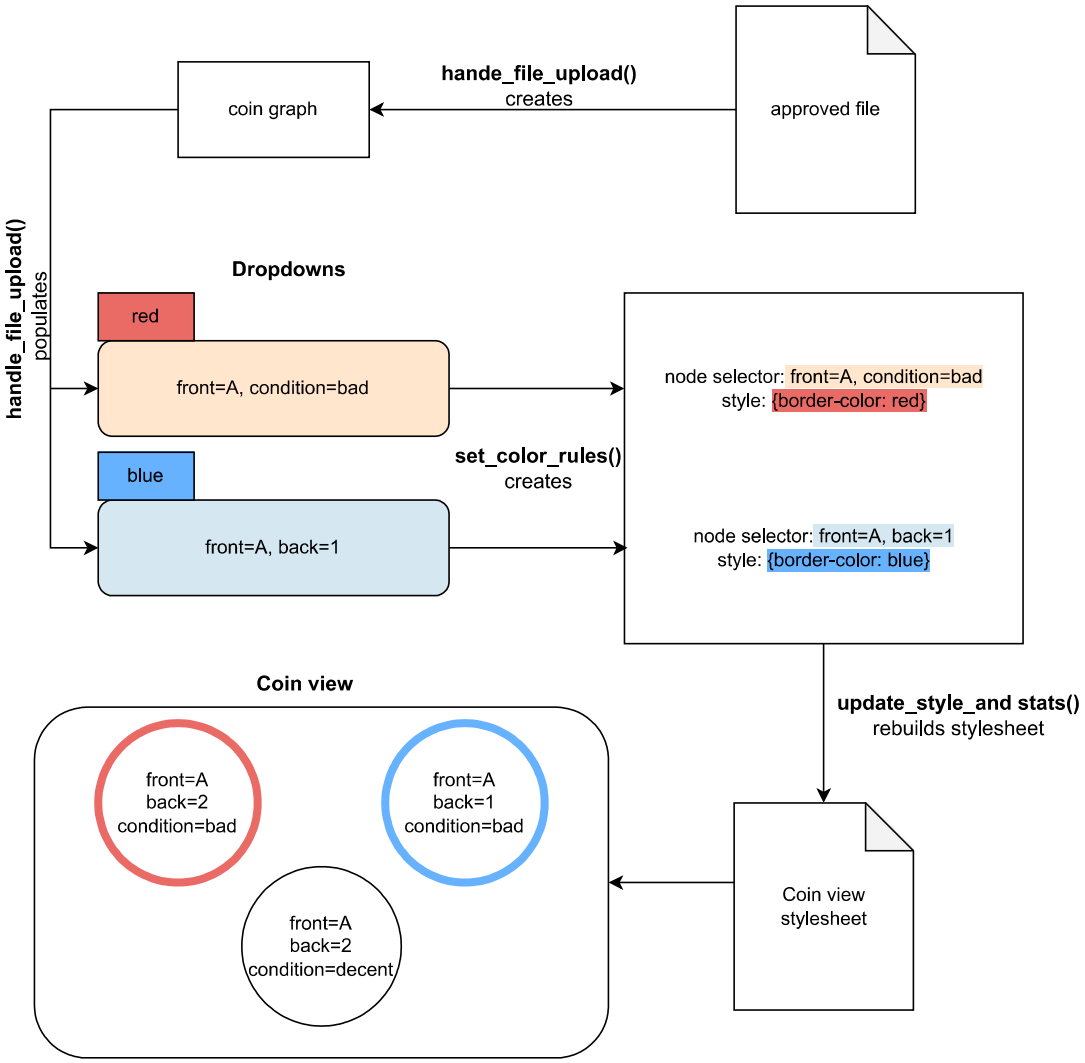

Figure 7: Workflow for coloring nodes feature

```
    )
    def handle_file_upload(...):
        ...
        # set of  all "attribute=value" strings for color dropdown
        combinations = set()
        for _, data in coins_graph.nodes(data=True):
            for attribute, value in data.items():
                if value is not None:
                    attribute_values.setdefault(attribute, set()).add(value)
                    combinations.add(f"{attribute}={value}")
        ...
        # build options for color dropdowns
        # expects [{'label':displayed text, 'value':returned value}]
        options = [{'label': c, 'value': c} for c in sorted(combinations)]
        ...

        return (
            ...
            options,
            options,
            options
        )
```

When a selection in a color dropdown changes, the stylsheet of the coin view is rebuilt. Part of that process is handled by the function shown below, which constructs the stlyesheet rules for this feature.

Each entry in `color_values_list` is a List of `attribute=value` strings. Each entry in `color_ids` is the related colordropdown id, where the intended color is stored under *index* field.

The function loops over these lists in parallel. For each dropdown, it creates a selector that matches nodes fulfilling all specified attribute value pairs and assigns the chosen border color.

```
def set_color_rules(color_values_list, color_ids):

    color_rules = []
    # iterate over both lists in parallel, each pair represents one dropdown
    for color_values, id_ in zip(color_values_list or [], color_ids or []):
        # extract index field, which stores the color
        color = id_.get('index') if isinstance(id_, dict) else None
        if not color or not color_values:
```

```
            continue

        selector = 'node'
        # build up attribute selector by adding each attribute=value pair
        for condition in color_values:
            if isinstance(condition, str) and '=' in condition:
                # split up attribute value pairs
                attr, val = condition.split('=', 1)
                selector += f"[{attr}='{css_escape(val)}']"
        color_rules.append({
            'selector': selector,
            'style': {'border-color': color,}
        })

    return color_rules
```

Cytoscape applies the stylsheet top to bottom. This means if two selectors target the same node and assign a border color, the rule appearing later wins. Consequently, if the attribute combinations in two color dropdowns match the same node, the later one will override the coloring. This is also shown in the example of Figure 7

4.3.8 Node pictures

As explained earlier in Section 3.1, the application should use the URLs or relative file paths associated with each coin, to display them as nodes images. When embedding images from the CN dataset via URLs, the browser produced an error, which led to the MDN Web Docs. The error occurred because the external image server did not send the CORS header in its response to the request like: `Access-Control-Allow-Origin: *`. Without the header, the browser blocks the request.(MDN contributors, 2025a)
This results in Dash Cytoscape being unable to embed the requested image in a canvas. You can circumvent this issue by using `'background-image-crossorigin':'null'` property in the stylesheet, which lets you display the images in the canvas, but in doing so, the canvas becomes 'tainted', so the browser will block the export and throw an exception. A commonly suggested fix is to change the configuration of the server hosting the images(MDN contributors, 2025b). Since this was outside the application's control and expected to happen with other image hosts, because using CORS is a common security measure, a workaround was implemented.
Inside the Dash application is a proxy endpoint, this endpoint fetches the external image and returns it directly to the client. Because the browser sees the image as being

loaded from the same origin, it does not need to apply CORS restrictions. This results in the Dash Cytoscape component to successfully embed the pictures in the canvas, without triggering CORS related errors. The `proxify` function can now be used to route any external image URL through the local proxy endpoint.

```
def proxify(url):
    return f"/img_proxy?url={quote(url, safe=':/%?&=')}"
```

When using relative file paths, the images can be stored in the `assets` folder and be used as node backgrounds.
Having established the procedure for accessing the images, it is necessary to decide which images are to use in each view. In the coin view, we introduced the option to change the edge condition. It comes naturally that for each option we take the associated pictures to use, that means:
When the edge condition is set to `front`, use the front images of each coin as node background. When the edge condition is set to `back` use the back images of each coin as node background.
This leaves the edge condition `front + back`. Based on discussions with the advisor, it was decided to display a merged image in this setting. For this, another endpoint was introduced:

```
@flask_app.get("/merge_split")
    def merge_split_route():
        front = request.args.get("front") or ""
        back  = request.args.get("back") or ""
        w     = request.args.get("w", type=int) or 200
        h     = request.args.get("h", type=int) or 200
        if not front or not back:
            return Response("missing front/back", status=400)
        try:
            data = merge_side_by_side(front, back, w, h)
            resp = Response(data, mimetype="image/png")
            resp.headers["Cache-Control"] = "public, max-age=86400"
            return resp
        except Exception as e:
            return Response(f"merge error: {e}", status=500)
```

this uses `merge_side_by_side`, where the left half of the front side and the right half of the back side of each coin are merged into one picture, to send in the response.

```
def merge_side_by_side(front, back, w = 200, h = 200):
```

```
    # Load raw bytes
    front_bytes = _load_bytes_from_source(front)
    back_bytes = _load_bytes_from_source(back)
    # Decode into OpenCV images
    front_img_decoded = cv2.imdecode(np.frombuffer(front_bytes, np.uint8), cv2.IMREAD_COLOR)
    back_img_decoded = cv2.imdecode(np.frombuffer(back_bytes, np.uint8), cv2.IMREAD_COLOR)
    if front_img_decoded is None or back_img_decoded is None:
        raise ValueError("One of the images could not be decoded")
    # Direct resize to target size (may distort aspect ratio)
    front_img_resized = cv2.resize(front_img_decoded, (w, h), interpolation=cv2.INTER_AREA)
    back_img_resized = cv2.resize(back_img_decoded, (w, h), interpolation=cv2.INTER_AREA)
    # Left half from front, right half from back
    mid = w // 2
    left_half  = front_img_resized[:, :mid]
    right_half = back_img_resized[:, mid:w]
    merged = np.hstack((left_half, right_half))
    # Encode back to PNG
    ok, buf = cv2.imencode(".png", merged)
    if not ok:
        raise ValueError("Encoding merged image failed")
    return buf.tobytes()
```

When the user switches the edge condition, it triggers the `update_styles_and_stats` callback function, which updates the coin view's stylesheet, so the correct images are displayed.

```
if edge_mode == 'front':
    styles.append(_img_rule('bg_front'))
elif edge_mode == 'back':
    styles.append(_img_rule('bg_back'))
else:
    # mode == 'both': lowest → highest priority (later wins)
    styles.append(_img_rule('bg_back'))   # fallback 2
    styles.append(_img_rule('bg_front'))  # fallback 1
    styles.append(_img_rule('bg_split'))  # preferred
```

For the die view, each die can have multiple coin images associated with it, providing several possibilities for representing it. Ideally, the application would allow the user to select the most suitable image from all available options. However, due to the time constrains, the current implementation only offers a simple solution. Only the first available coin image is used to display the die.

4.3.9 Picture overlay

Another feature closely related to the previous one is the option to view the full size image associated with a node. Several approaches were considered:

- Displaying the full size version of the image within a tooltip window, when hovering over the node.
- Opening a separate browser tab with the original image URL upon interacting with the node.
- Displaying the full size image in an overlay that is activated by interacting with the node.

The tooltip-based approach was discarded, since the implementation displayed the additional information a node has in a separate container of the sidebar. Opening the image in a separate tab would have been feasible. However, after discussions with the advisor and Markus, it was concluded that the implementation should support the simultaneous display of multiple images. This made the separate tab option unsuitable. Therefore, the overlay approach was adopted. When a node is clicked, an overlay is shown with the full size version of its image, constrained by the vertical height of the container. In the coin view, when the `front + back` edge condition is selected, where the coin images are merged, both original images are displayed within the overlay.

4.4 Use Case: die study

In this use case scenario, a numismatic expert is conducting a die study on a set of coins. This set is already labeled with the assigned dies used in production. The expert's primary goals are to identify potential inconsistencies in the dataset and to generate visualizations suitable for further use. For demonstration purposes, a small subset of the CN corpus is used to display functionalities. (Peter et al., 2024) After specifying the relevant field names and uploading the dataset, the

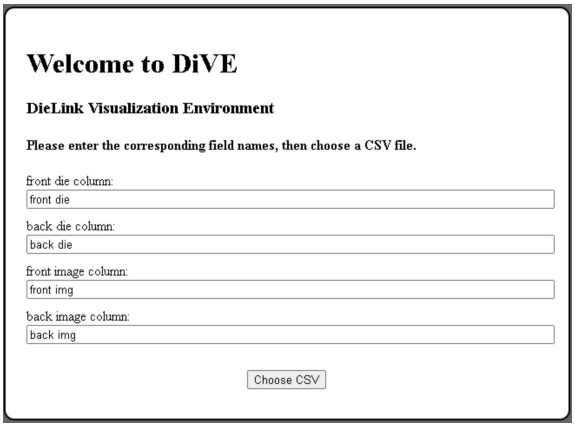

Figure 8: DiVE application start

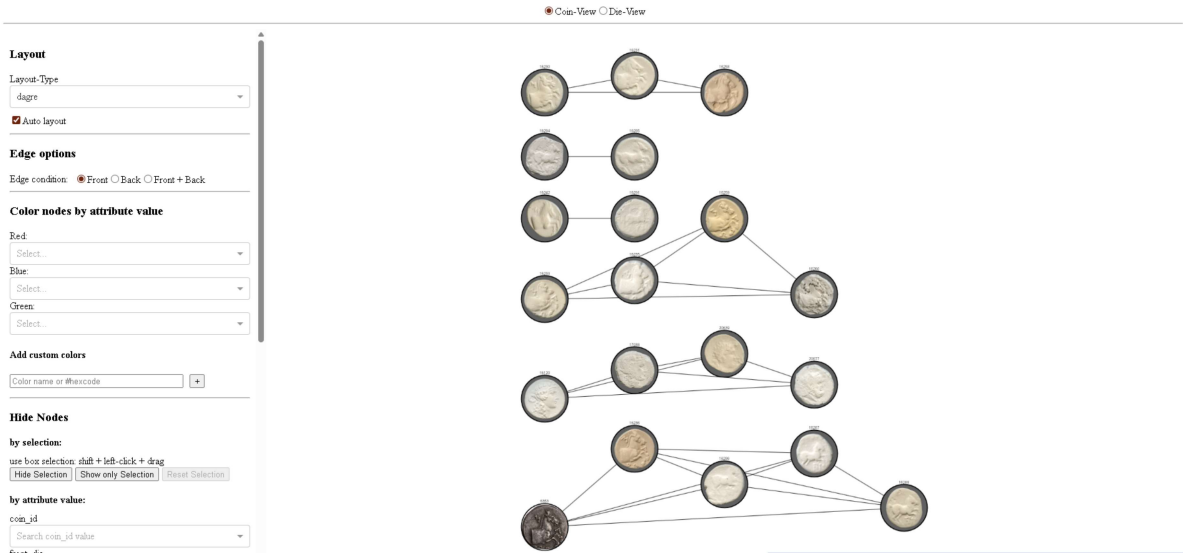

Figure 9: DiVE starting (pictures from Corpus Nummorum)

workflow starts in the coin view (showcased in Figure 9). Each coin is represented as a node and edges denote shared die assignments. The expert first explores the dataset using the available edge-conditions, which allows them to visually verify the die assignments, by comparing the associated coin images. For a more detailed picture of a coin, they can click on its node to display a full size image in an overlay. When a coin image appears to be suspicious, they can temporarily remove the coin from both visualizations. This can be done through the hiding by attribute feature, where the expert uses the node label or hovers over the coin and matches one of its attributes in the corresponding dropdown. Or the expert can draw a box selection and use the associated `Hide selection` button.

Alternatively, the user may choose to highlight coins by assigning coins with a specific attribute combination using the node coloring feature (showcased in Figure 10).
Once inconsistencies have been reviewed, the expert switches to the die view to examine the die links. Here the dies are represented by an image of an associated coin. In this view the expert can use the layout dropdown to choose layouts that display the nodes sequentially such as `Dagre` or `Klay`. For smaller datasets `Circle` and `Cose` can be used to give an overview. To refine the visualization the expert can move nodes, to minimize any overlapping or disable the `scaled edges with weight` option, so have uniform edges.

Once satisfied with the configuration, the expert can export either view, resulting in

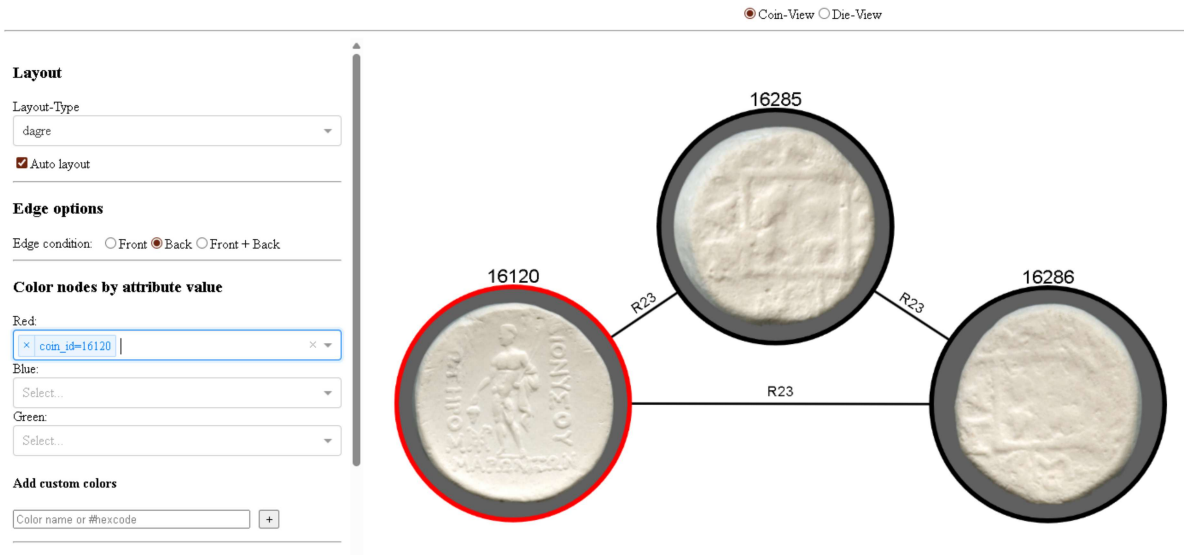

Figure 10: DiVE highlighting nodes (pictures from Corpus Nummorum)

suitable images for publications or continued analysis (Example export Figure 11).

5 Evaluation

After completing the implementation chapter, it should be evaluated if DiVE fulfills the earlier defined requirements and also the extended requirements. The tool sufficiently transforms the expected data format to an input for the visualization component, where both the coin view and die view are available. Furthermore the component offers multiple layouts, that arrange the nodes in a meaningful pattern, while allowing nodes to be freely placed afterwards.

DiVE does offer some customization like coloring nodes in the coin view or allowing edge thickness to be scaled with edge weight in the die view. These functionalities are useful but there should be more options. At least coloring nodes in the die view and adjusting node sizes and label texts would be needed to consider this requirement met.

In contrast the filtering requirement is fulfilled, as there are even multiple ways to filter. The included URLS and file paths to coin images are being used as node images in both coin and die view. Also it is possible to export the visualization as a standard image file.

The extended requirements are fulfilled by DiVE, because it offers a full size version of

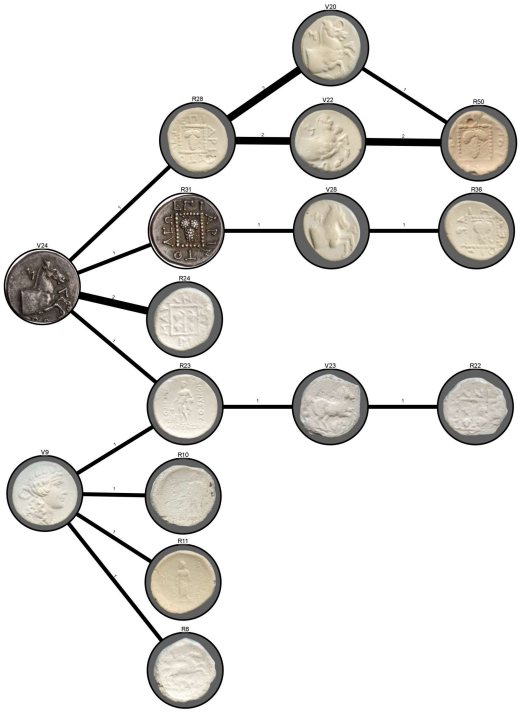

Figure 11: DiVE exported PNG (pictures from Corpus Nummorum)

requirements	DiVE
data transformation	✓
multiple views	✓
freely adjustable layout	✓
customizing	✗
filtering	✓
tooltip	✓
node pictures	✓
clear and intuitive UX	✓
export	✓
node picture full size	✓
flexible edge condition	✓
adaptive node image	✓
interlinking views	✓

Table 3: Overview of DiVE meeting defined requirements.

node images on click and allows for flexible edge conditions in the coin view, where the node images that are being used are adapting to the selected edge condition. Furthermore the interlinking of views is achieved through the visibility-based approach, where hiding nodes in one view affects the visibility of nodes in the other view.

6 Discussion

The methodological approach chosen for this thesis proved appropriate, for the nature of the problem. Once the initial requirements for the tool were defined, potential candidates for the use case were evaluated on the basis of these requirements. The following prototyping phase, together with gathering feedback from the advisor and numismatics expert Markus Möller, helped to reveal additional requirements that were not apparent from the start. So the chosen Methodology helped the refinement of requirements.
One of the most challenging aspects of the implementation was the interlinking of views. This requirement emerged relatively late in the process, which made its integration into the existing system architecture difficult. In retrospect, it would have been advantageous to keep the underlying coin graph and die graph completely separate after initially building them. The nodes of the die graph would then be extended with all associated attributes of the coin nodes. This would enable the views to remain visually interlinked while keeping the underlying logic separate. Such an approach would also improve the maintainability and extensibility of the tool.
Despite these challenges, the tool demonstrates clear strengths. Limiting the functionality to only what is needed for the intended used proved to be essential for creating an intuitive user experience, especially compared to the other evaluated tools that serve a wide range of use cases. Minimal configuration enables the user to quickly detect visual inconsistencies in the dataset, a benefit that was showcased in the use case.
Before visualizations for die studies were created manually. The automated generation of the visualizations and the ability to easily modify them, are likely to accelerate this part of the of the workflow.
Nevertheless, the current implementation also has limitations. Performance may decline when dealing with large inputs, suggesting that future versions of DiVE may benefit from preprocessing the data. This could enable the visualization to load the graph components incrementally rather than everything at once.

7 Conclusion

This thesis set out to identify the requirements a tool must fulfill in order to support die studies through the creation interactive visualizations and evaluated if existing tools or the developed solution `DiVE` satisfy these requirements.
To achieve this, the work proceeded through several stages, beginning by defining the initial requirements based on discussions with the advisor, a review of existing implementations in the space and on published die studies. It then investigated whether existing tools could meet these requirements and found that none fulfilled them completely. This led to an prototyping phase, during which additional requirements were identified. Followed by developing an implementation that meets most of the final requirement set.
The thesis therefore contributes a set of requirements that a digital tool should fulfill to support die studies by creating interactive visualizations. Additionally a tool that meets most of these requirements. Furthermore it speeds up the process of image creation and can help find inconsistencies in the dataset of a die study, through the interactive visualization.
However, several limitations remain. Performance may become an issue when working with large datasets. The customization requirements is not fulfilled. In addition `DiVE` has not yet undergone testing by numismatics experts.
Therefore future work should focus on the unmet requirements of `DiVE`, particularly regarding customization of the visualization. Potential extensions include coloring nodes in the die view, adjusting node sizes and label texts. Additional unrelated features that are worth mentioning are an automatically created legend for the coloring, and a session saving feature. Further steps should involve testing `DiVE` with numismatics experts to gather feedback and ensure its suitability for practical use. If performance does become an issue, preprocessing the data and loading graph components incrementally should also be considered.
In summary, the thesis provides a practical contribution toward the digitalization of the die studies workflow.

References

Bastian, M., Heymann, S., & Jacomy, M. (2009). Gephi: An open source software for exploring and manipulating networks. *Proceedings of the International AAAI Conference on Web and Social Media*, *3*(1), 361–362. https://doi.org/10.1609/icwsm.v3i1.13937

Demšar, J., Curk, T., Erjavec, A., Gorup, Č., Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., & Zupan, B. (2013). Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, *14*, 2349–2353. http://jmlr.org/papers/v14/demsar13a.html

Gortana, F., von Tenspolde, F., Guhlmann, D., & Dörk, M. (2018). Off the grid: Visualizing a numismatic collection as dynamic piles and streams. *Open Library of Humanities*, *4*(2), 30. https://doi.org/10.16995/olh.280

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th python in science conference* (pp. 11–15).

MDN contributors. (2025a, July 4). *Reason: Cors header 'access-control-allow-origin' missing*. Retrieved October 31, 2025, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS/Errors/CORSMissingAllowOrigin

MDN contributors. (2025b, September 18). *Use cross-origin images in a canvas*. Retrieved October 31, 2025, from https://developer.mozilla.org/en-US/docs/Web/HTML/How_to/CORS_enabled_image

Parmer, C., Duval, P., & Johnson, A. (2025, July 31). *A data and analytics web app framework for python, no javascript required.* (Version 3.2.0). https://doi.org/10.5281/zenodo.14182630

Perrone, G., Unpingco, J., & Lu, H.-m. (2020). Network visualizations with pyvis and visjs. https://arxiv.org/abs/2006.04951

Peter, U., Franke, C., Köster, J., Tolle, K., Gampe, S., & Stolba, V. F. (2024, February). Corpus nummorum – a digital research infrastructure for ancient coins. https://doi.org/10.5281/zenodo.10633905

Plotly. (2024). *Dash-cytoscape (version 1.0.1)* [Software]. https://github.com/plotly/dash-cytoscape

Serrano Molinero, V., Bach, B., Plaisant, C., Dufournaud, N., & Fekete, J.-D. (2017). Understanding the Use of The Vistorian: Complementing Logs with Context Mini-Questionnaires. *Visualization for the Digital Humanities*. https://inria.hal.science/hal-01650259

Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., Amin, N., Schwikowski, B., & Ideker, T. (2003). Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, *13*(11), 2498–2504. https://doi.org/10.1101/gr.1239303

Talbot, J. (2015). *What is icenian coinage?* [Doctoral dissertation, University of Oxford]. University of Oxford. Retrieved November 22, 2025, from https://www.academia.edu/42757375/WHAT_IS_ICENIAN_COINAGE

Yoon, D. (2025, February 10). *Die links and sequences* [Accessed: 2025-11-19]. American Numismatic Society. https://numismatics.org/pocketchange/die-links-and-sequences/

Appendix A: Coins Used from *corpus Nummorum*

This thesis makes use of coin images from *Corpus Nummorum* within the developed software application.

All images are provided by *Corpus Nummorum* under the license:
Attribution-NonCommercial-ShareAlike 4.0 International.
General reference to the corpus https://www.corpus-nummorum.eu/

This is a list of all coins used to showcase the application:
coin id: 6353
citation notes: cn coin 6353, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_6353 [Last Download: 2025/11/24]
coin id: 16120
citation notes: cn coin 16120, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16120 [Last Download: 2025/11/24]
coin id: 16230
citation notes: cn coin 16230, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16230 [Last Download: 2025/11/24]
coin id: 16231
citation notes: cn coin 16231, in: Corpus Nummorum, https://www.corpus-nummorum.

eu/CN_16231 [Last Download: 2025/11/24]
coin id: 16233
citation notes: cn coin 16233, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16233 [Last Download: 2025/11/24]
coin id: 16235
citation notes: cn coin 16235, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16235 [Last Download: 2025/11/24]
coin id: 16236
citation notes: cn coin 16236, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16236 [Last Download: 2025/11/24]
coin id: 16242
citation notes: cn coin 16242, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16242 [Last Download: 2025/11/24]
coin id: 16258
citation notes: cn coin 16258, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16258 [Last Download: 2025/11/24]
coin id: 16259
citation notes: cn coin 16259, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16259 [Last Download: 2025/11/24]
coin id: 16260
citation notes: cn coin 16260, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16260 [Last Download: 2025/11/24]
coin id: 16284
citation notes: cn coin 16284, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16284 [Last Download: 2025/11/24]
coin id: 16285
citation notes: cn coin 16285, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16285 [Last Download: 2025/11/24]
coin id: 16286
citation notes: cn coin 16286, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16286 [Last Download: 2025/11/24]
coin id: 16287
citation notes: cn coin 16287, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16287 [Last Download: 2025/11/24]
coin id: 16288

citation notes: cn coin 16288, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16288 [Last Download: 2025/11/24]
coin id: 16291
citation notes: cn coin 16291, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_16291 [Last Download: 2025/11/24]
coin id: 17038
citation notes: cn coin 17038, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_17038 [Last Download: 2025/11/24]
coin id: 20639
citation notes: cn coin 20639, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_20639 [Last Download: 2025/11/24]
coin id: 20677
citation notes: cn coin 20677, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_20677 [Last Download: 2025/11/24]