Design Project - Module 11

BibSTEAK

Bib(La)TeX Streamlined Tool for Editing Academic Knowledge

Maria Sandu	s2964082
Lisa te Braak	s3000885
Edwin Martens	s2996251
Wouter van Gent	s3008290
Max van Zanten	s2844931
Fabian Tabrea	s2995735

University of Twente
Enschede, The Netherlands

November 2025

Abstract

When using LaTeX with bibliography management, often issues relating to completeness and consistency of data arise. Thus, our Design Project was to develop an application for reference management. With BibSTEAK, manually editing your references in .bib files is a thing of the past. It is a portable Python application, available with both a Command Line Interface and a Graphical User Interface, the latter serving as a high-fidelity prototype, proving that the functionalities integrated in the CLI can be transferred to have a graphical visualization as well. The main goal of the system is to aid users, mainly academics, in cleaning and completing their collection of references. In order to reach this goal, we have designed a system that manipulates the data in a .bib file by parsing it into a Python object, applying certain actions on it, and regenerating it back into a .bib file. This allows the system to be extendable. This tool can be deployed by individual LaTeX users, with a focus on power users who are looking for ways to script automatic editing of .bib files.

Contents

1 Introduction

LaTeX, the typesetting software system, is a favourite among academics. With LaTeX, creating scientific texts is made easier, especially for those with many mathematical formulas, graphs, or images. Since scientific articles usually include a bibliography, LaTeX also includes an easy way to cite sources and format them into a bibliography. Typing out references, however, is too effortful, which is why, along with LaTeX, there exists BibTeX. BibTeX is a file format in which one can format references for later citation using LaTeX. So, a .bib file will contain multiple references, each separately formatted into its own data-object, similarly to JSON or XML. An example could be:

```
@book{jansson:mvc,
  author = {Masha Jansson},
  year = {1989},
  title = {Multi Variable Calculus},
  publisher = {Penguin}
}
```

File: references.bib

Multiple of these can go into one file. In LaTeX, entries can be cited using their citekey. Here, that would look like `\cite{jansson:mvc}`. Then, when `\bibliography{references.bib}` gets used in the LaTeX file, all cited references will be pulled from references.bib and formatted into a neat and beautiful bibliography. In theory. The reality is different. To make sure a beautiful and complete bibliography gets realized, the .bib file needs to contain complete and clean data. Common issues are missing fields, different naming conventions, duplicates, etc. These are especially common when pulling references from different sources and/or using multiple different .bib files. But who has time for manual editing of tens of references? BibSTEAK is an attempt to solve this problem. A reference manager, powerful and quick, that will transform your .bib files with minimal effort. In section 3, we will explain our project proposal and extracted requirements for the creation of this product. In the sections after that, more details of the implementation will be uncovered, ending with our testing and results.

2 Research

2.1 Domain exploration

Before making a requirement specification, we wanted to explore what other systems already exist. This to make sure we are not making a carbon copy of another system, but also to see what features they might lack, or features that we feel like we can do better. We have looked at a couple of existing systems, such as:

– Zotero

– JabRef

– Mendeley

2.1.1 Key Similarities in Domain

All three of these systems are tools to assist users in collecting, organizing and citing references. All systems allow the users to import references and generate citations and bibliographies. These systems also have some integration with other applications, such as Word or LibreOffice. Another big similarity between the three is the layout of the application. They are all divided in three sections that contain the explorer, the list of references, and the single selected reference.

2.1.2 Stand out features/issues

JabRef contains a feature that tries to look up your reference on online sources, such as CrossRef to edit and improve your current reference. It also gives the option to select which side of the information you want to keep. This way, you can see if the information is correct first before editing your reference with the new information.

In Zotero, you can attach the actual reference document and store it with the reference so that you can always read it. We thought it was a nice feature and that it makes the workflow of writing a paper a little easier.

In Mendeley, we learned that adding tags to references is a hassle. Mainly because you have to manually add tags to every single reference. There is no batch feature for this. When starting with an empty repository, this would not be much of an issue. But once you have to start with a lot of references, this can become very time-consuming.

Adding to the previous point, it seems that batch editing your references is quite difficult. Thus if you want to change something in all references, the process will take a lot of time and is therefore not user friendly.

All of these systems do not directly edit a bib file. They can import a bib file and have it shown in the own system and then generate a bib file. But all the steps in the middle are executed in their system and not in the bib file. Thus for every change you make you would have to regenerate the bib file every time.

3 Requirements Specification

3.1 Extracted Requirements

To extract requirements for this project, we carefully read the proposal we received and had a start-up meeting with the client. We organized these into functional and non-functional requirements.

Functional

- The app can organize references into collections
- The app can have tagging and filtering for references
- The app can detect duplicates
- The app can merge references using custom rules
- The app can have formatting tools to clean/normalize/transform entries in batches (like refactor)
- All actions can be undo- and redo-able
- The app can support power users by enabling them to script actions The app can enrich entries using the web (finding DOI or links, etc)
- The entries can be visualized as a connected graph
- The app can support offline usage
- The app can minimize and maximize citations
- The app can support offline usage

Non-Functional

- The app must be clean and intuitive
- The app shall not disrupt the workflow

These requirements do not tell us a lot about the end product, though. That is why we decided to sit down and discuss the logical implications of these requirements and what other extensions we could make to the system. We also wrote more concrete descriptions for the requirements.

3.2 Prioritization

Once we had this complete list, we decided to use the MoSCoW method of prioritizing these requirements, meaning that they were separated into must have, should have, could have, and won't/would have. The results of this are listed below.

Musts

- The app must be able to merge references using custom rules
  The app must be able to merge references that are related to the same paper into only one single reference based on custom rules, such as overriding certain bibliographic fields during merge or aggregating them if they are distinct. For example, a DOI and a URL could be merged by only including the DOI based on custom rules set by the user (possibly with defaults).

- The app must have formatting tools to clean/normalize/transform entries in batches (like refactor)
  The app must be able to manipulate entries in such a way that they can be normalized, sanitized, and transformed based on custom rules. For example, deleting the redundant fields, i.e. the author's name is repeated, or there are multiple white spaces in the title.

- The app must be clean and intuitive (non-functional)
  The user should be able to find all the relevant features easily without accessing the help menu too often. The app should follow standard UI layout rules, such as avoiding deeply nested menus, using immediate feedback for every action performed by the user, and having explicit errors and warnings in case some actions are not allowed.

- The app must have all its actions undoable and redoable
  All actions in the app must be directly undoable. When you undo an action, it can easily be redone. The basic implementation of this is a linear action history with the last 10 actions, where you can only undo/redo the last action. With this implementation, you can, for example, undo the last five actions, redo the last two, and perform a new action. At this point, you can undo the newest action, but you have lost the previous three actions that were reversed.

- The app must support the workflow (non-functional)
  For the application to be useful to the user we want to make integration as easy as possible to extend and enhance the workflow of the user. For this the user should not have to repeat a task multiple times (this includes filtering, selecting, and executing commands). A single task should be able to be completed in a short time span.

- The user must be able to manually add single or multiple references
  A user must be able to add a reference or multiple references to each BibTeX file. A user will be able to select the reference type and will be shown the corresponding entry fields. Where the user can see at a glance which ones are required and which ones are optional. Then the user can fill them in and, with the press of a button, add the reference to the selected bib file.

- The app must have a file explorer
  The application will contain a folder with all the bib files stored/saved by the user. This folder will be shown as a section in the application so the user can go through them, select them, and refactor them (an expansion for this would be for the user to be able to create folders within this folder, to sort or order their files).

– The app must support different types of references
  The application will be able to recognize all 14 different BibTeX reference types. The application will be able to show which fields are required and which are optional per reference type. The required/optional entry fields can be changed according to some presets of the most popular editors. But the user is also able to make their own preset.

– The app must be able to organize references into collections
  The application will allow the user to have collections in the form of labels/tags. They will not be able to be nested like folders, but one reference could have multiple labels attached to it. When uploading a reference, the user will be able to click a button to add it to an already-made label or create a new label to add it to. Users can rename, delete, or reorder collections.

– The app must have tagging and filtering for references
  After references have been tagged/labeled, they will be able to be filtered upon using a small categories menu that includes all of the labels that have been created.

– The app must be able to detect duplicates
  First, the application should normalize titles (by using only lowercases, stripping any punctuation out of the title. Secondly, the system should handle variations in author names by normalizing them into a consistent format (by ignoring punctuation and capitalization, matching initials with full names, and comparing them regardless of the order they appear in). For more confidence, the year of the publication can also be used. After detecting duplicates, the application will prompt the user to review them, and only after will it process the removal.

The above requirements are the basic features that constitute an MVP of the system. This means that our key functionalities are:

– Merging

– Cleaning

– Undo/redo

– Adding references

– Organization using tagging/filtering

– Detection of duplicates

Shoulds

– The app should support offline usage
  The app should be able to support offline usage for every action that does not require an internet connection or an API call via the web. Actions such as editing bib files, sorting them based on certain criteria, or viewing them in different formats should be done offline as well. However, actions that require requests via the web will be available to the user only with a stable internet connection. When the user is offline, the buttons of actions that can only be performed online will be greyed out. When hovering over the button the user should be informed that

an internet connection is required for the action. Additionally, when an action fails because of a bad connection, the user will be informed of this with a pop-up.

- The app should minimize and maximize the length of citation descriptions
  The app should be able to shorten or extend bib file entries based on user input. This means that the user will be able to add custom rules and abbreviations, which the system will use to automatically alter bib files. These rules should include a priority, such that the bib file can be gradually shortened or extended. Using this, the user can easily change the size of the references dynamically (for example to make them fit on a page).

- The app should have two possible views: raw BibTeX and BibLaTeX, and a stylized version (which is cleaner to edit)
  A user can click on a button to preview the current bib files compiled as a PDF. This is done by a standard compiler that has default settings that will be unchangeable.

- The app should have standard batch commands available in a simple menu, so no typing is necessary for standard usage
  When a user has selected a collection (or singular) of references, the user should be able to execute commands (such as minimizing, abbreviating, etc.) using a list of standardized commands that would be executed by just pressing the button. These commands will be visible in a simple list.

- The app should have shortcuts for important and frequently used features
  A user will be able to use certain keyboard shortcuts to execute commands on references or use other actions. For example, `ctrl + shift + "-"` would minimize the selected references. The users will have a file where they are able to use keyboard shortcuts to trigger certain actions. The web app will support modifier keys such that users can apply shortcuts specific to their operating systems (CTRL and Command will be interchangeable). A list with shortcuts will be provided so that a power user can learn and use them accordingly.

- The app should support power users by enabling them to script actions
  The users will have a file where they are able to configure/script their own preferred actions and commands. In case of a different configuration for commands, they will override the default shortcuts.

After the MVP is realized, these features are also required. They may be softer requirements, but they are definitely also necessary.
Most of these focus on the user experience. We want the normal user to be able to do most things by simply clicking buttons and letting the tool do the work. However, from the proposal, there was also a requirement to make power users able to do commands by typing and even scripting. Thus, we chose to add requirements for both of these types of user experiences by separating them within the functionalities of the program.
There is also one other key feature that is added onto the requirements: minimizing/maximizing.

Coulds

- The app could have an action history panel to undo and redo actions
  The app could have an action history panel with the last 20(?) actions, such that not only the last action can be undone, but all actions in the list can be selected. The interface should clearly specify what happens when an action is selected. For example, undo actions should look visually different, such that redoing actions still remains intuitive.

- The app could have a text-based command palette that can be used instead of using the clickable UI
  The app could have a standard command palette that can be used to perform multiple actions instead of manually triggering them with the mouse via the interactive UI. In other words, the app could be used solely with a keyboard for users who prefer this way of interaction with the app.

- The entries could be visualised as a connected directed graph in relation to other papers
  All entries could be interpreted as nodes that are connected via edges (citations) to other papers in a directed graph. This way, certain direct and indirect relations between groups of papers or pairs could be inferred visually.

- The app could have capitalization batch commands for all types of tags (e.g., capitalize all author fields or CamelCase all titles)
  The app could have text formatting batch commands in order to preserve consistency, improve readability, and adhere to certain user-defined formatting rules. For example, capitalizing only the initials of the author's names or rendering the name particle of each author in lowercase should be possible to be done globally for all entries: (vincent van GOGH –> Vincent van Gogh)

- The app could be able to compute stats for a paper using the reference graph
  When the visualisation of the related papers has been made, the app can also show stats about the visualisation. For example: the Erdos number, the number of direct citations (in-neighbours), the shortest chain of citations to an arbitrary paper, etc.

- The app could have a feature that proposes citation keys (e.g., the ref "the interpretation of dreams" by Sigmund Freud can have an auto-proposal for the citation key "Freud:dreams" or whatever)
  When a user adds a reference to a bib file, the application will suggest a reference/citation key to have a standardized form of creating the keys. This can be based on the author and the year for example or other entry fields.

- The app could be able to split and merge separate Bib files
  The user could be able to select two (or more) bib files and execute a merge for which the application will combine all selected files. The application might have some filtering or options for the user to select so that the merge doesn't always create duplicates or can leave out certain reference types for example.

- Furthermore, the user can click on a button to create a separate bib file from the selected references.

- The app could be able to update reference information (without having to remove and add the reference again)
  The user will be able to update reference information by selecting a reference and pressing a button ("edit") which will bring up the same form type as adding a new reference but only now filled in with the current information. Then the user can edit that information and save the edited version of the reference to the existing bib file.

- The app could have a section that explains a list of advanced commands to the user
  The application will contain a button (shaped like a question-mark) which opens a little section on the screen that contains a document explaining advanced commands to the user. Which the user in turn can use to optimize his workflow.

- The app could convert between formats (BibTeX, JSON, XML, CSV)
  The type of file will be detected from its extension (.bib/.json/etc). Each supported type will need an individual parser, such that after normalization all of the data is in the same format. The format it will be transformed into would be a json array where each field of a reference is an object. After, the json format can easily be parsed to make a bib reference, given that the fields are in accordance with what a bib reference can have (it needs to go through a check, and if other fields are detected, the user should be warned).

- The app could be able to propose duplicate removal and/or merges automatically
  Given that duplicate removal and merge have been implemented, when started/having had files uploaded, the application could automatically detect duplicates or merges that are available and prompt the user to accept or refuse if they want them to happen. Consequently, 2 buttons in the bar of the application could be dedicated to showing a notification sign for each duplicate and merge suggestion.

- The app could be able to enrich entries using the web (finding DOI or links, etc)
  When the reference has at least a DOI or a link, and the app is online, it will look in standard places (Google Scholar, DBLP, that one website that has all those papers for some reason) to get extra information and autofills them into the fields.

- The app could store default field settings
  When using any command, the user can click a button to set it as a default. It will get saved in a settings.json (or similar) that stores the defaults, which is locally on the user's computer.

- The app could have an option where a user can create templates so they don't have to select all their options every time they wish to edit a Bib file in the same way they have done before
  At any point in the workflow, a user can click to create a template that will store some of the Bibfile-wide commands, like minimize all the way and capitalization commands. These settings will be stored in a separate file locally, so that the app can use the template again later.

The could section, or also the "probably not" section, mostly contains extra features and functionalities that we as a team came up with, inspired by the proposal given to us. Some extra features we thought up were...

- Graph visualization of references
- Creation of templates and storing defaults
- Edit references

Wonts

- The app won't be able to generate citations The app won't be able to generate actual citations according to reference style guides, nor any in-text citations. This means that the style of the citations will be implemented by other applications.
- The app won't be able to dynamically shorten or extend references with a slider The application won't be able to dynamically shorten or extend references. Even not with a slider, this is mainly due to the already complicated task of minimizing and maximizing the references. So doing this dynamically would add a big layer of difficulty.
- The app won't have themes to style the application to the users' liking (or at least a light and dark mode) The application will not allow the user to change the theme to their own liking, but also not to choose from a light or dark theme. It will have one style that users will be bound to, but it will look nice :).
- The app won't integrate easily with the DBLP or BibSLEIGH application for searching papers The app will not be able to have a functionality where, when adding a new reference, you can type into a search box and find papers on DBLP and BibSLEIGH.

These requirements were ideas we came up with that would be a nice to have but are outside of the scope of the project. Furthermore, they would take too much time out of our allocated time to implement the necessary features and have therefore been put to the side.

4 Global Architecture and Design

4.1 System proposal

Our client had some system requirements already in their proposal. It would ideally be 1) no-install, 2) supporting offline usage, and 3) with as few dependencies as possible. In short, the tool should be small, standalone, and it should not disrupt the workflow. To this end, we decided in the initial proposal that:

> "BibSTEAK is mostly an offline web-app, which can be installed using an installer. After installing it, it will be run using a single executable, which will run the front-end and the back-end, and open the front-end for the user to start working immediately. We are planning to program it using React.js, plus Django front-end and back-end frameworks."

Let's discuss the choices made here and how they connect to the three system requirements. (1) and (2) had to be compromised on, since they were incompatible in our view. If we would choose a no-install option, the webapp would be hosted on a web server and thus required to be online. If we would choose to fully support offline usage, this problem presents itself the other way around. Thus, we decided to meet in the middle. The executable allows for the user to simply boot up the app from their desktop, without interrupting the workflow too much. This way, we can also still have both offline and online functionalities, since it is not permanently hosted on a server. Plus, it would still look and act like a webapp, namely, it would be in the browser.
The user story here is that the user installs the webapp, opens the executable (in browser), and inserts their `.bib` file, transforms the `.bib` file using the tool, and then the tool automatically overwrites the file with the changes.

4.2 Changed plan

After delivering our system proposal, our clients had some feedback on the design. They told us that it was not preferable for the application to boot up using an executable, if there was also no command line and/or possibility to use a makefile to run commands. It was also unclear to them what "mostly offline" meant, since they wanted it to be offline by default.
The framework we were set to use, namely React, was also a heavy framework that would have required using many libraries, with a lot of features that were not of use because of the scope of the project. Since it was in the interest of the stakeholders to have as few dependencies as possible, using React.js would have been overkill, and it would have also brought a big learning curve. Because of these reasons, we decided to change our global design. In the following paragraphs, we will explain what changes we made to our architecture and why.

4.2.1 Python

Instead of using Python in combination with React.js and Django, we decided to make the core of the application in pure Python, without a framework. This is because we are all comfortable with coding in Python, meaning we could start implementing our core functionalities straight away. Furthermore, this allowed us

to stick to requirement (3) and keep the amount of installed dependencies low. All the while keeping our happy medium with regards to requirements (1) and (2), meaning that it would still be a desktop program with optional online functionality.

4.2.2 Command-Line Interface

Just like the client suggested, we also created a Command-Line Interface (or CLI). The CLI acts as a single layer of abstraction between the Python code and the user, allowing them to type commands and receive results within the terminal window. This will also allow us later to implement scripting features.

4.2.3 Graphical User Interface

However, we also chose to keep our vision of a Graphical User Interface (or GUI). Beautifying these `.bib` files is a visual process, so it makes sense to have a graphics-based way of communicating with the system too. Especially for less technically inclined users and those who value personal oversight more than automation. Since React.js was quite a heavy framework for this use case, a more lightweight, Python-based framework was chosen, named NiceGUI [1]. This allowed for the creation of a browser-based application that demonstrates how the functionalities present in the CLI translate to a graphical visualization as well.

The application starts with the setup page 1, where the users are prompted to decide what configurations they want to have, which can be later adjusted in the settings page. Then, the user gets directed to the main page 2, where they can see the bib files they have in their directory, as well as the references in each of them, and can perform certain actions.

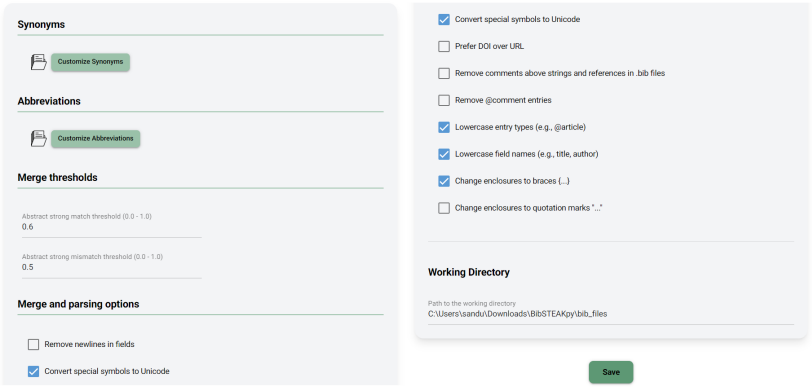

Figure 1: The Setup/Settings page

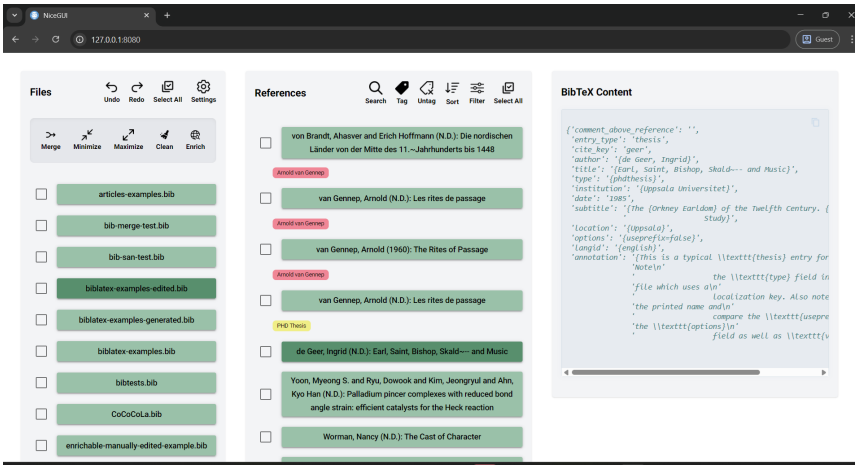

Figure 2: The GUI main page

Thus, the GUI serves as a high-fidelity prototype, integrating most of the functionalities available in the CLI, and it is a proof of concept that it can be extended to incorporate all functionalities in the future.

4.2.4 Complete System

File Organization The structure of our system is divided into key components. The most important of which are listed below:

– JSONS
  – `abbreviations.json`
  – `config.json`
  – `synonyms.json`
  – `tags.json`

  Folder with JSONs for storage of other data outside of .bib files.

– UTILS
  – `abbreviations_exec.py`
  – `batch_editor.py`
  – `cleanup.py`
  – `enrichment.py`
  – `file_generator.py`
  – `file_parser.py`
  – `filtering.py`
  – `json_loader.py`
  – `merge.py`
  – `ordering.py`
  – `sub_bib.py`

- `synonyms.py`
- `tagging.py`
- `view.py`

  Folder containing the code for the commands.
  See section 5.3 for more details.

- TESTING

  Folder for unit tests.

- `CLI.py`

  File that is the Interface between the terminal and the utils.

- `gui_main.py`

  File that is the Interface between the GUI/browser and the utils

- `history_manager.py`

  File that implements the version control for .bib files.

- `interface_manager.py`

  File that manages both the CLI and GUI.

- `objects.py`

  File that contains the BibFile object.
  See section 5.1 for more details.

Overview In conclusion, the system consists of three main parts. The utils, the CLI, and the GUI. Hypothetically, a user can either select to use the CLI or GUI, with the latter being a high-fidelity prototype for now. The relations between the elements can be seen in Diagram 3.

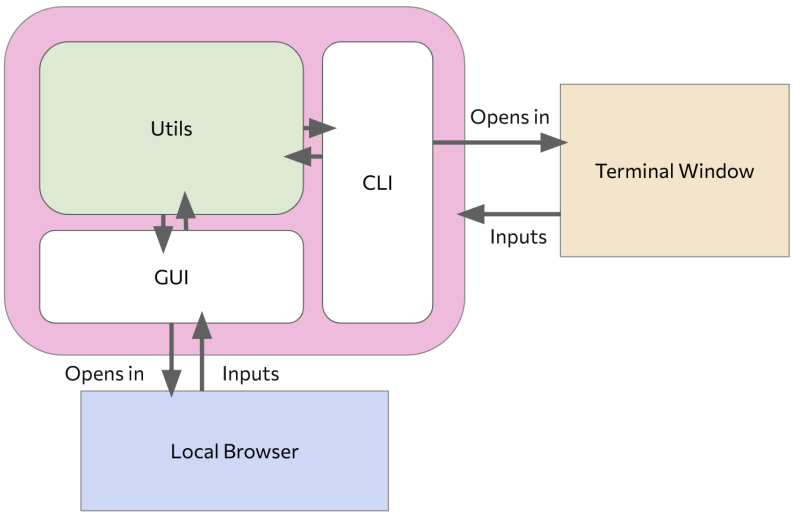

Figure 3: Diagram of global design

5 Detailed Design

5.1 The BibFile Object

To easily edit bib files, we created our own BibFile Python class. This class clearly separates the different types of entries as described in the BibTeX format description [2]. The four types of entries are: string, preamble, comment, and reference. These types are added to the content of the BibFile object as a list, preserving the order. In addition to the content, the object also stores the original file path when parsed.

An important design decision of the BibFile object was how to store (or ignore) text that is not part of an entry. In the .bib files provided by our supervisors, most of this text relates to the reference or string below the text. When storing these comments separately, they become much harder to track when changing the order of the content in the BibFile object. Because of this, we decided to add text above references and strings as part of the entries with the comment_above_reference and comment_above_string fields. More details on how different types of comments are parsed can be found in 5.2.

The string and reference entry types also contain different fields to allow for easy access to specific parts of the entry. The String object contains the abbreviation, the long_form, and the enclosure type separately.[1] The Reference object contains separate fields for the entry_type and cite_key. Currently there is no specific handling based on the entry_type, but this can easily be added by adding child classes based on the entry_type, which inherit from the Reference class. All fields[2] are added to the Reference object as attribute, in the order of appearance.

In these different classes the __eq__ method is also implemented to ensure that different BibFile objects with the same content are still recognized as equal. This is used in the history implementation, see 5.4.

5.2 Parsing and Generation

The many ways to edit .bib files with the methods in 5.3 all use the previously mentioned BibFile object. A quick overview of how this works can be seen in 4. Because of this, the conversion between the object and a .bib file is a crucial part of the application. To do this conversion, we have made a parser and a generator.

[1] Note that in the BibTeX format description[2] strings are only given in the format abb = "abbreviation". However, we found many examples of the format abb = {abbreviation} being used, so this is also supported.

[2] Officially called tags[2]

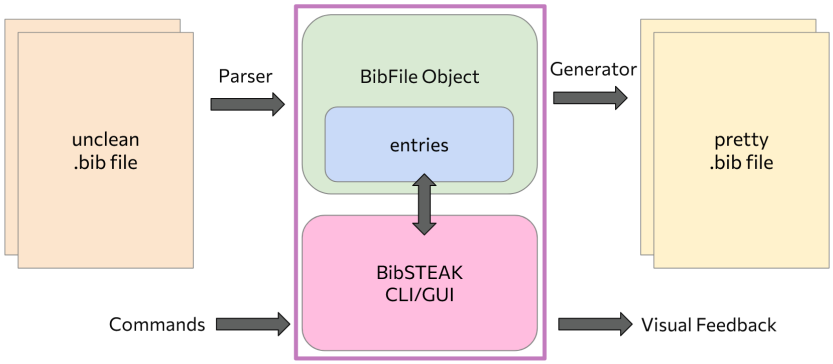

Figure 4: General overview of how functions interact with the BibFile object.

The implementation of the parser uses an Enum as a state and handles the characters of a .bib file in order from top to bottom, based on that state. In 5, you can see a quick overview of how the states change in the parser. Note that this is a simplified diagram, which only shows the states of the Enum. The parser keeps track of some more specific details. For example, the level of braces is also stored, for when there are braces inside of braces.

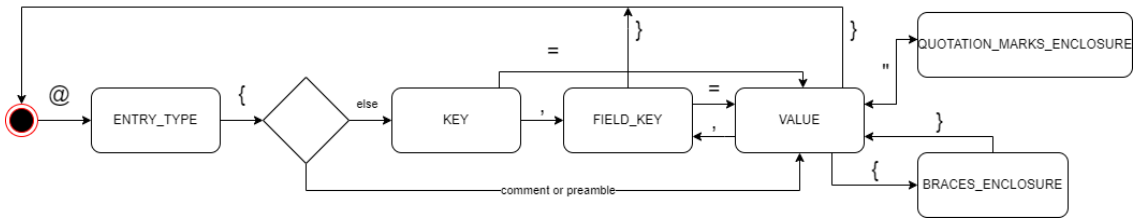

Figure 5: State changes during parsing.

Although the parser tries to keep most of the content of the .bib file original in the BibFile object, there are some cases where the content is slightly altered. Moreover, there are also cases where an error is thrown because of invalid content, rather than deleting the invalid content. This was done to notify the user, so they can change the invalid content manually. In practice, comments in unsupported places make up most of this invalid content. Many other instances of content that other parsers[3] mark as invalid are considered valid by our parser.

Below follows an example bib file to illustrate the behavior of the parser. Content that causes a ValueError in the parser is marked as such.

```
ValueError. @preamble{preamble}
ValueError. @comment{comment}
comment_above_string
@string{abbreviation = "long_form"} added to the reference below
comment_above_reference
@entry_type (can contain spaces){cite_key (can contain spaces), ValueError.
  field key = field value, ValueError.
  field key = field value, <- removed, since key and value match.
  field key = "field value", <- ValueError (duplicate field keys, different
contents).
```

[3]See examples in 6

```
10      } @{,=,} @{,}
11      Any remaining text that cannot be parsed as part of an entry will be added as a
         str.
12
```

As you can see in this example, comments are only allowed in specific locations: above a string or reference and at the end of the file. Comment entries are, of course, allowed everywhere except inside another entry. Comments using % are not parsed differently: they are also only allowed in the specified positions. The advantage of this is that the comments are clearly connected to the Reference object after the parsing process, which means that they will be kept together even after changing the order of references. Also interesting to note are the empty references near the bottom of the file. These are perhaps surprisingly also parsed as references, since there is no requirement for non-empty values in these spots. Similarly, there is also nothing that stops you from adding spaces in fields or citation keys. Because of this, commented-out fields with a % symbol will also be parsed as part of the reference, although they will remain commented out in LaTeX. Furthermore, newlines are ignored in the default parsing process. Only when using the config option remove_newlines_in_fields (or manually calling the parser with that argument) does the parser handle newlines differently. In that case the newlines and additional indentation inside field values are replaced by just a single space. After removing all the ValueErrors from this file, it gets parsed and generated as shown below.

```
1       @preamble{preamble}
2       @comment{comment}
3       comment_above_string
4       @string{abbreviation = "long_form"}
5
6       added to the reference below
7       comment_above_reference
8       @entry_type (can contain spaces){cite_key (can contain spaces),
9         field key          = field value,
10      }
11
12      @{,
13                           = ,
14      }
15
16      @{,
17      }
18      Any remaining text that cannot be parsed as part of an entry will be added as a
         str.
19
```

It should be noted that, on top of the removed duplicate field, the whitespace in the file has also been changed significantly. This comes from our bib file generator. The generator is way simpler than the parser, just looping over the elements in the content of the BibFile object and handling the different entry types accordingly. The generator makes two additional changes[4].

First is the alignment of fields and strings. This works by searching for the latest spot of the = symbol, either in a string or in a field, and then aligning all the other strings and fields with that position. In the given example, the '=' symbol is the longest in the string, so the fields will be aligned based on that. A man-

[4]Depending on the function arguments or (if none were given) the config file.

ual align_fields_position can be given to the function to overwrite the automatic alignment.

Second is the addition of newlines inside fields with long field values. This is set with a constant inside the file_generator, with a default value of 100. When keeping the default values of the generate_bib method, this is dependent on the remove_newlines_in_fields option in the config: if the original newlines are removed they will be automatically added.

5.3 Utils

5.3.1 Batch replace

One of the simplest use cases of the BibFile object is a more complete version of replacing all occurrences in the file. Our version of this only replaces these occurrences inside of field values, with the added option of specifying the field keys. Additionally it also changes the long_form in strings and ensures that the abbreviation of a string is not changed. This allows you to replace data in certain fields without having to worry about other unrelated occurrences changing. This function is also used internally in the implementation described in 5.3.2.

5.3.2 Abbreviations

Manually replacing abbreviations is, of course, possible using batch_replace. However, if you want to replace the same abbreviations at a later point in time, this takes the same amount of effort again. To make this process easier we have added the abbreviations.json file, which can be used to store your own custom abbreviations. Using the col or exp commands you can then easily expand or collapse the abbreviations in a specific bib file. An example of some abbreviations in abbreviations.json is shown below:

```
1        {
2          "ACM": ["Association for Computing Machinery", ["publisher"]],
3          "LNCS": ["Lecture Notes in Computer Science", []]
4        }
5
```

File: abbreviations.json

Similar to batch_replace, you can also add field keys in this file. To replace the abbreviations in every field you can keep the list empty. In addition to this basic replacement, you can also automatically convert the abbreviations to strings and add them to the file using the add_abbreviations_as_strings config setting.

5.3.3 Clean

During the development of this tool we came across many small personal preferences for bib files. Instead of adding a new command for each of these preferences, we decided to add a single clean function that uses preferences from the config file. To keep the format simple, all of these options don't change anything if set to false

or an empty list. Below, an example of a config is shown.

```
        "convert_special_symbols_to_unicode": true,
        "prefer_DOI_over_URL": true,
        "remove_comments": false,
        "remove_comment_entries": false,
        "lowercase_entry_types": true,
        "lowercase_fields": true,
        "change_enclosures_to_braces": true,
        "change_enclosures_to_quotation_marks": false,
        "preferred_field_order": [
            "author", "title", "year", "journal", "booktitle", "publisher",
            "volume", "number", "pages", "DOI", "URL", abstract"],
        "unnecessary_fields": [
            "ee", "venue", "month"]

```

File: config.json

Starting with the conversion of special characters. These are converted from the escape sequences to the actual character. While looking through some examples of these escape sequences, it became clear that many characters and formats are possible. The first implementation of the conversion was using a giant dictionary with all these special kinds of formats. However, to keep things organized we have switched to use the Unicode Combining Diacritical Marks [3] where possible. This reduces the dictionary size significantly and ensures that new characters can easily be added. You can easily add other special characters either by adding the entire escape sequence in special_latex_to_unicode or adding them directly to the latex_to_unicode dictionary. The current values are listed below.

```
        special_latex_to_unicode = {
            '{\\ss}': 'ß', '\\ss': 'ß',
            '{\\o}': 'ø', '\\o': 'ø',
            '{\\O}': 'Ø', '\\O': 'Ø',
            '{\\ae}': 'æ', '\\ae': 'æ',
            '{\\AE}': 'Æ', '\\AE': 'Æ',
            '{\\l}': 'ł', '\\l': 'ł',
            '{\\L}': 'Ł', '\\L': 'Ł'
        }
        latex_to_unicode = {
            "\\`": ('\u0300', ['a', 'e', 'i', 'o', 'u']),  # Grave accent
            "\\'": ('\u0301', ['a', 'e', 'i', 'o', 'u', 'y', 'c']),  # Acute accent
            "\\^": ('\u0302', ['a', 'e', 'i', 'o', 'u']),  # Circumflex accent
            "\\~": ('\u0303', ['a', 'n', 'o']),  # Tilde
            "\\=": ('\u0304', ['a', 'e', 'i', 'o', 'u', 'p']),  # Macron
            "\\u": ('\u0306', ['a', 'e', 'i', 'o', 'u']),  # Breve
            "\\.": ('\u0307', ['o']),  # Dot above
            '\\"': ('\u0308', ['a', 'e', 'i', 'o', 'u']),  # Diaeresis
            "\\a": ('\u030A', ['a']), "\\r": ('\u030A', ['a']),  # Ring above
            "\\H": ('\u030B', ['o', 'u']),  # Double acute accent
            "\\v": ('\u030C', ['c', 'r', 's', 'z']),  # Caron
            "\\c": ('\u0327', ['c', 's']),  # Cedilla
            "\\k": ('\u030C', ['a', 'e', 'i', 'o', 'u']),  # Ogonek
        }

```

File: cleanup.py

The characters in latex_to_unicode are automatically replaced in different formats. Take, for example, an "i" with an acute accent. This will be replaced in the following formats:

```
            {\'{\i}}, \'{\i}, {\'{i}}, \'{i}, {\'\i}, {\'i}, {\' i}

```

Note that just \'i is not replaced, as it could cause potential unintended replacements.
The config option prefer_DOI_over_URL allows you to remove the URL if both DOI and URL exist in the file. The remove_comments setting will remove the comments above references/strings, whereas remove_comment_entries will remove @comment entries. lowercase_entry_types and lowercase_fields make all entry types and field keys lowercase, respectively. The change_enclosures_to_braces and change_enclosures_to_quotation_marks settings will change the enclosure, removing the previous enclosure or enclosures. Note that capitalization is not kept in LaTeX if brace enclosures are removed. The preferred_field_order will be used in the cleanup process to order the fields, keeping the order for field keys that are not in the preferred order. Lastly, the unnecessary_fields setting is used to remove specific field keys from all entries. This can be especially useful in combination with the enrichment feature, since you might not want to add specific fields.

5.3.4 Enrichment

The use of the enrichment utility is to find additional information about the reference that the user has not yet put into the reference. This is done by looking up the reference in 4 different API's, namely: CrossRef.org, DBLP, DataCite, and OpenAlex. Let us take a deeper dive into how this is achieved with reliable and correct results.

To enrich a file, we need a BibFile object that was created by the parser explained in 5.2. First, we check if the user has an internet connection with the following method:

```
1         def checkInternet() -> bool:
2             return (lambda a:
3             True if 0 == a.system('ping 8.8.8.8 -n 3 -l 32 -w 3 > clear')
4             else False)(__import__('os'))
5
```

This function tries to ping google.com and check if there is a valid response. We only establish the existence of this connection once when executing the enrichment function. When the connection breaks down during the execution, we cancel the command and stop the whole function. This means that no changes will be made if the command failed.

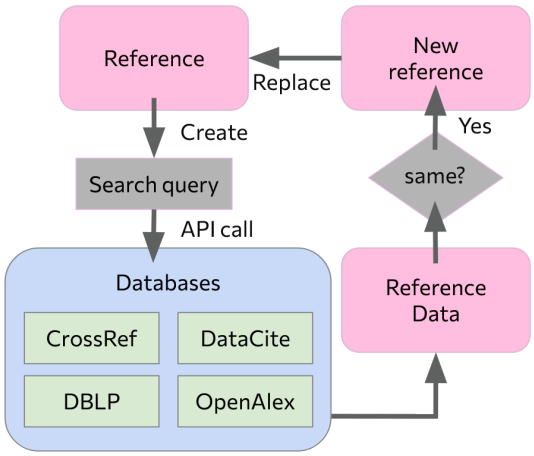

Figure 6: Process of enrichment

Above you can see the process of enrichment for a reference. First we take a reference object from a BibFile object and create a search query. This query is created by using the fields available in the reference but mainly uses the title and author. This search query is then used to look up the reference on the databases. This is done as follows:

To search a reference on DBLP, we call the following method:

```
        def _search_dblp(session: requests.Session, query: str, timeout: float) ->
    Dict[str, Any] | None:
            URL = "https://dblp.org/search/publ/api"
            params = {'q': query, 'h': 1, 'format': 'json'}
            response = session.get(URL, params=params, timeout=timeout)
            response.raise_for_status()
            data = response.json().get("result", {}).get("hits", {}).get("hit", [])
     or []
            if isinstance(data, dict):
                data = [data]
            if not data:
                return None
            data_info = data[0].get("info", {})
            return _map_dblp_to_bib_dict(data_info)

```

File: enrichment.py

The method takes a session created for all the API requests (this session is for letting the endpoints know that these requests are coming from this application, and it reduces the risk of them being flagged or blocked), a search query, and a timeout to define how long the application should wait for a response. Then, the method makes the request and checks if there are any valid responses and hits for the search query. If not, it will return None. If the response contains a hit the method will map the correct dictionary by parsing the dictionary with information towards the `_map_dblp_to_bib_dict()` method. This method takes all the information in the response and makes it into a usable dictionary which has normalized entries. An example includes the field "year" which is done as follows: `if data.get("year"): result["year"] = data["year"]`. Once all the information has been stored in the dictionary we check if the reference found at the endpoint is the same as the reference we are looking for. We do this by checking if the title and author are the same, (or at least 80% equal).

This is done by normalizing the authors and the title for both the original and the retrieved information, and checking if there are equal. The title has a weight of 80% because that is usually more important than the author. If the score comes out equal to or close to 80%, the application deems the information correct and corresponding.

After all the gathered information has been collected and stored we merge the different responses together by order of trust. It will take CrossRef as the main source (if any information is found). Any other information from the other endpoints will be added if that information wasn't already available from CrossRef. This way, we can control which endpoint is most trusted and should be used as the best source of information.

Lastly, the function will merge all the found information with the existing information. Any field that already exists in the original .bib file will not be touched by the function. It will only add information and not update it.

5.3.5 Querying

Filter and Search Filter/search is a utils that can search for keywords in references in your BibFile. Either using one search term ("search") or a (field, value) pair ("filter"). The methods iteratively search through the references in the file. The search term and the references are normalized to lowercase to match more accurately.

Tagging The tagging feature enables the user to group references by giving them a tag. The tag feature uses the previous querying command. So, the subset of references that returns from a query can be tagged using `tag <tag> <query>`.
We chose to store the tagged references in a JSON file. We only store the citekey of the reference, and not the entire reference, for easier access later in the program. For example, if there are 2 references returning from querying for "Kant", they will be stored as follows:

```
1        {
2        "kant": ["kant:kpv","kant:ku"]
3        }
4
```

File: tags.json

Subfiles The reason filtering, searching, and tagging are features is that one can create a "sub file" with a subset of the references by using the `sub` command. This file can then be cleaned, etc, by using the other commands as well.

5.3.6 Merging

The goal of merging is to combine two BibTeX files into one file by merging entries when they clearly refer to the same work. It will resolve field conflicts when merging two references with defaults and user input. To run a merge, we need two BibFile objects that are produced by the parser and specified by the user in the CLI. The merging process works as follows:

Preambles All preambles from file 1 are copied first, and then any unique preambles from file 2 are added

String definitions Next, it merges @string definitions with collision handling. If an abbreviation exists only in file 1, then we keep it. If both files define the same abbreviation with identical expansion, then we keep one. If both define the same abbreviation with different expansions, then we prompt which long from to keep and, if needed, interactively rename the abbreviation across one file to resolve the conflict.

References After Preambles and strings are handled, the references are merged. The tool will compare every reference from one file with every reference from the other file to determine whether they refer to the same work. Identity is determined using progressively weaker matching signals in this order:

1. DOI match. If both references contain a DOI and the normalized DOI matches, then they are the same work and will be merged automatically.

2. Author and Title signature match. If a DOI does not exist, we compute normalized author and title signatures. If they match, we then look at abstract similarity. Abstract similarity is computed with SequenceMatcher, and thresholds come from user configuration with a default. Users can set the strong match percentage and the weak match percentage. If two abstracts are above the strong match threshold, then they are automatically merged. If they are below the weak match threshold, they are automatically not merged. If they fall in between, the user will be asked if they want to merge the references.
3. URL match. If DOI and author/title do not identify the reference, we then use URLs and only consider URLs from a trusted scholarly domain list (ACM, IEEE, Springer, Nature, PubMed, arXiv, OpenReview). If those match, we will prompt the user to confirm the merge.

If none of the signals match, the references are not merged.

An addition that could be made is the use of synonyms. There is a synonym file that records synonyms that could be regarded as equal during the merging process. This allows the user to match more references based on their preferences e.g. using different ways author names can be written. You can easily add new synonyms to the synonyms.json file by using the functions in synonyms.py.

Field-level merge When two references are merged, each field is compared individually. If two field values are the same after normalization, then it is automatically resolved. If they differ meaningfully, the user is prompted: keep left, keep right, or manually enter a new value.

5.4 Version Control

5.4.1 History Management

Version control functionalities similar to Git have been developed in order to facilitate a modern workflow, implicit back-ups, and redo/undo commands. This has been done in an effort to allow the user to retrieve at any point any historic version of a file altered using the BibSTEAK tool.

Version Hash Tree In order to achieve this, a version tree that stores a unique hash ID for each versioned file is built and tracked locally. The hash is generated for each non-trivial modification of the file done with a BibSTEAK command. We call this a commit to the hash tree. A non-trivial modification is one that actually changes the content of the file, for example, running the same command twice will only generate at most one hash. Furthermore, since it is possible to check out historic versions of a file, the HEAD pointer will move along the tree, as it is depicted in the figure below:

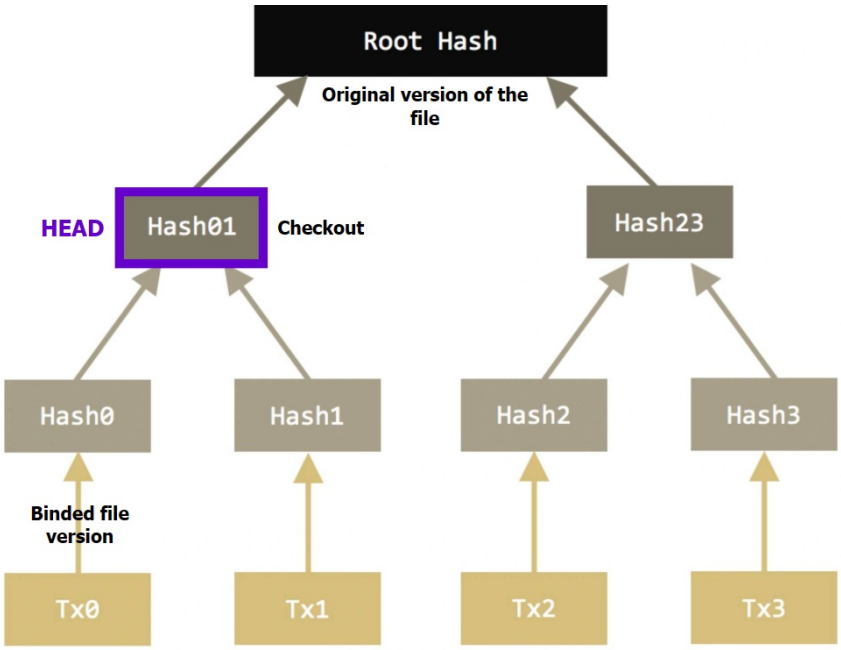

Figure 7: Example of Hash Tree with a detached HEAD - The same structure is used to manage the local history of each bib file loaded and altered with BibSTEAK. Adapted from [4]

5.4.2 Checkout

Since every version of a bib file is tracked with a unique hash, the user is always able to use the checkout command to load any historic version of the file at any point in time using its unique identification hash.

5.4.3 Undo

The undo command loads the previous version of the file (if it exists, i.e., the current version of the file is not the root of the version tree)

5.4.4 Redo

The redo command loads the succeeding version of the file from the version tree (if it exists, i.e, the current version of the file is not a leaf of the version tree)

5.4.5 History Command

The history command prints the version graph structure and the timestamp of each commit in order to allow the user to understand the historical modifications and the branching of a file better.

5.4.6 Delete History

Deletes the history tree from the tracker file and all the historic versions stored for a particular file.

5.4.7 Comment

The comment command allows the user to attach a comment to any commit in order to label that version of the file for identification of other specific purposes.

5.5 Graph Visualization

The graph visualization functionality allows the user to see the connections between the papers in a bib file and k neighbours (cited papers) up to degree 3. In other words, the user can generate a parameterized k-regular citation graph. In this way, the user is able to see all the citations between the k most relevant papers (sorted in terms of their citation count in descending order) to detect citation patterns or other relevant works in the network topology. Picking a higher k parameter will lead to higher loading times.

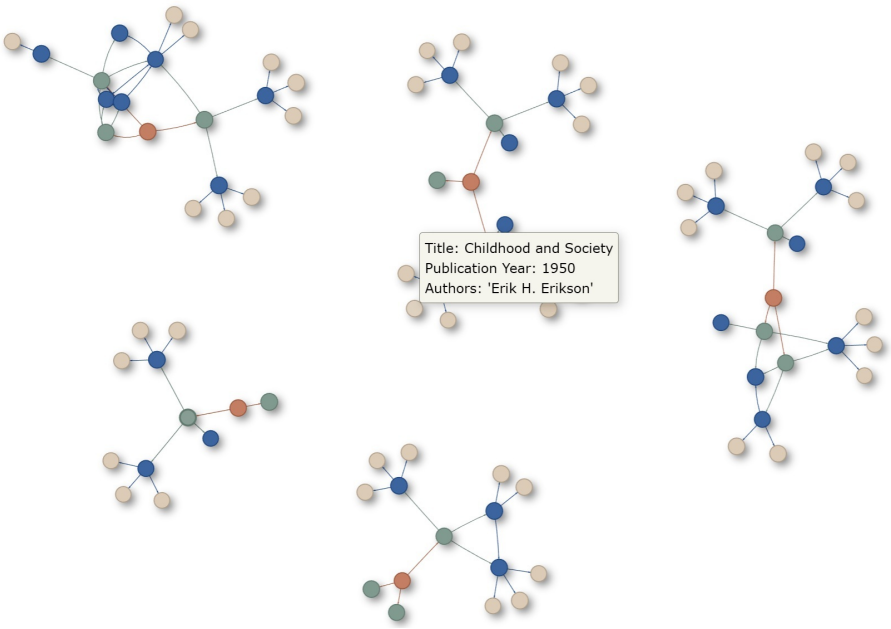

Figure 8: Example of graph visualization based on 5 entries in a bib file (the nodes corresponding to the local entries are highlighted in orange)

The graph visualization tool works depends on the OpenAlex [5] citation database. Thus, an internet connection is required for this functionality. The query is done based on the title and the publication year of a paper. In the end, the best query match for each paper is retrieved. The visualization has been made possible using the *pyalex* package for web API calls and the *vis-network* JavaScript package for physics and graph generation. Moreover, the visualization is done in a web browser by initializing a session with the NiceGUI framework.

5.6 Scripting

All commands and functionalities of BibSTEAK can be expanded and enhanced via Python scripting. This can be easily done since every command has been documented, and it lives in the utils directory. This way, the user is able to construct custom loops or conditional logic in order to extend the sequential behaviour of BibSTEAK.

```
import argparse
import history_manager
from utils import file_parser, file_generator, enrichment, cleanup,
    abbreviations_exec


def enr_clean_col(absolute_path):
    bib_file = file_parser.parse_bib(absolute_path)
    history_manager.initialise_history(bib_file)

    cleanup.cleanup(bib_file)
    enrichment.sanitize_bib_file(bib_file)
    cleanup.cleanup(bib_file)
    abbreviations_exec.execute_abbreviations(bib_file, True, 10000)

    file_generator.generate_bib(bib_file, absolute_path)
    history_manager.commit(bib_file)

```

```
18
19 if __name__ == '__main__':
20     # You can run this script using 'python example_script.py -path=<absolute\path
       >'
21     parser = argparse.ArgumentParser(
22         description="Script that enriches, cleans and collapses original file,
       while saving history."
23     )
24     parser.add_argument("-path", required=True, type=str)
25     args = parser.parse_args()
26
27     enr_clean_col(args.path)
```

File: examplescript1.py

Another way of scripting is using the API object from the BibSTEAK package. In this object, most commands are exposed and documented. An example of such usage can be seen in the code snippet below.

```
1 from BibSTEAK import api
2
3 WD = "C:\\Users\\tabre\\Desktop\\storage"
4 TARGET_FILE = "example2.bib"
5 TARGET_FILES = ["example.bib", "example2.bib", "example3.bib"]
6
7 api.set_wd(WD)
8 api.list_names() # Retrieve all file names from the set working directory.
9 api.exp(TARGET_FILE)
10
11 # Example of looping and conditional behaviour
12 for filename in TARGET_FILES:
13     if api.search(filename, "John Lenonx"):
14         api.ord(filename) # Sort in ascending order
15     else:
16         api.ord(filename, True) # Sord in descending order
```

File: examplescript2.py

5.7 Graphical User Interface

5.7.1 Threading and Synchronization

The GUI has a main thread where everything from the rendering of interface components to the processing of user interactions (for example, button clicks) and the update of visual components happens. Running heavy processes like merging, where user input is needed, requires the creation of new threads to run in parallel because blocking the thread on which the GUI is running would prevent NiceGUI from processing further events, ultimately making the page unresponsive and the browser client disconnect.

The merge logic in the CLI relies on standard console input-output, such as `input("Choose which to keep:")` or `print("Merging...")`, but the GUI does not use this. Calls to input() inside the GUI thread would block the interface, so they had to be converted into pop-up dialogues that wait for the user's response. Also, calls to print() were captured and displayed in the GUI instead of in the terminal. So, the merging process runs inside a background thread, also further referred to as a worker thread, and merge.py uses interface_handler.py for all prompts and messages. When using the GUI, the interface_handler assigns to a Merge object that provides print_hook and input_hook. While input_hook makes sure prompt requests wait in the queue

until it is notified by the event of each request, when the user inputs are detected, print_hook() stores the printed lines in a buffer.
When the merge begins, the GUI calls `merge.start(selected_files_list, files, merge_files_fn)`, which displays the progress dialog for the merge and starts a background thread through:

```
        t = threading.Thread(target=self._worker, daemon=True)
        t.start()

```

File: merge_ui.py

Thus, new dialogues or GUI components don't change. All prompts are requested by placing them in a queue through `self._prompt_q.put(req)`. Then, the GUI uses timers (ui.timer()) in order to check of new prompts have been requested and whether the merge has finished. When a prompt is requested and is ready to be shown, a dialog is created by the main thread using components from NiceGUI through `self._prompt_dialog.open()`. Each of these prompt requests includes a threading.Event object that the worker thread waits on `req['event'].wait()`, and when the user makes a choice, the even gets released `req['event'].set()`. When the merge is finished, the worker thread sends a completion signal by setting `self._done_event.set()`, the progress dialog is closed, and the results proceed.

5.7.2 Integration

In the GUI, the following functionalities are integrated:

- loads configurations through json_loader.py, both on the setup and the settings pages.
- uses the configured working directory from the config.json to display the files, with respective references and bib contents columns.
- from the toolbar in the files column, users can undo or redo (through use of history_manager.py) actions by selecting one file by either using the checkbox next to the desired file, or by clicking on the desired file. The user can also run merge (through use of merge.py), minimize - collapse abbreviations and maximize - extend abbreviations (through use of abbreviations_exec.py), cleanup (through use of cleanup.py), enrich (through use of enrichment.py) on selected files by selecting the checkboxes for the desired files, or using the select all button - it deselects on second click. In order to change configurations, the user can use the settings button, which will redirect them to the settings page.
- the toolbar in the references column provides functionalities such as searching for terms, tagging with a custom name and color, untagging (when a reference has been selected, by ticking its checkbox, then all of the tags for that reference will automatically be removed. If no reference has been selected, the user will be asked to input the name of the tag they want to remove), sorting (through use of ordering.py), filtering (through use of filtering.py) - consists of a 2 page dialog, first the user selects if they want to filter on a certain field, or also on a certain value for that field. Then, the corresponding next page is displayed.

6 Testing and Results

6.1 Test Plan

6.1.1 Scope and Objective

To test our application, we first define the scope and the objective of the tests. We want to make sure that the user can load, view, edit, enrich, and organize single or multiple bib files safely and consistently. The tests need to ensure that data integrity is upheld, so no loss of fields or data occurs from doing any operations on the files. The test also need to confirm the non-functional requirements such as performance on large files, Unicode/LaTeX handling, and undo/redo history.

6.1.2 Test Environment

To ensure that the application is usable in all operating systems we have tested the application in the following systems: Windows 11, macOS 14, and Ubuntu 22.04. On all these systems, the test used Python v3.10-3.12. So any version of these should be fine for running the program. These tests (except for the enrichment) have been run with both the network on and off. Lastly, the test data for the application can be found in the bib-files folder, which contains the following files:

– CoCoCola.bib

– bib-merge-test.bib

– bib-san-test.bib

– biblatex-examples.bib

– bibtests.bib

– enrichable-manually-edited-example.bib

– manuall-edited-example.bib

– sanitized-bib-test.bib

6.1.3 CLI Test Cases

Workspace and Files		
Test case	**Action**	**Expected Result**
Set working directory	execute "cwd <path>"	path persisted in config, check with "pwd"
load bib file	execute "load <path/to/*.bib>"	file added to working directory, check with "list"
view file	execute "view <file.bib>"	Full content printed, Unicode correctly shown

Abbreviations and Expansion		
Test case	**Action**	**Expected Result**
collapse names	execute "col <file.bib>"	Target fields replaced using abbreviations.json, no other fields have changed.
expand back	execute "exp <file.bib>"	Command returns file to original, diff = 0.

Cleanup and Normalization		
Test case	**Action**	**Expected Result**
clean bib file	execute "clean <file.bib>"	Escape sequences are changed to Unicode.
check clean consistency	execute "clean <file.bib>" twice	Outputs of both executions are identical

Ordering, Search and Filter		
Test case	**Action**	**Expected Result**
check ordering function	execute "ord <file.bib>"	Order of reference fields follow "preferred_field_order" value from config.
check search function	execute "search <file.bib> 'graph neural'"	List of matching entries from (title/abstract/keywords)
check filter function	execute "filter <file.bib> author 'Smith, John'"	Only entries satisfying search query

Merge and De-duplication		
Test case	**Action**	**Expected Result**
two file merge	execute "mer <a.bib><b.bib> <merged.bib>"	Union of entries; Duplicates resolved with user input.
Merge all in directory	execute "mer -all <all.bib>"	Same as two file merge; Execution <1 second per entry if no duplicates.

Enrichment		
Test case	**Action**	**Expected Result**
Enrich missing DOI/URL	execute "enr <file.bib>"	Network calls made; DOI/URL filled if found.
Offline execution of enrichment	execute "enr <file.bib" without network	User is shown no network error.

History Control		
Test case	**Action**	**Expected Result**
Undo/redo	execute "undo <file.bib>; redo <file.bib>;"	Content jumps to previous/next versions. Doing both command gives diff = 0.
Checkout function	execute "checkout <file.bib> <commit_hash>"	Expect historical version restored.
History delete function	execute "h_del <file.bib>"	History folder removed for that file only; File contents remain intact.

Other		
Test case	**Action**	**Expected Result**
Graph generator	execute "graph"	Generates artifact and opens browser to show graph.
Config print	execute "config"	Should print config keys with current values.

6.1.4 Automated testing

For the parser and the generator, we also have some automated testing in test.py and test_parser.py. The test_files function in test.py works as follows: it parses all the bib files in the working directory and generates the file again from the parsed object. At this point, it checks if the original file and the generated file have the same content. As is explained in 5.2, the generator does change some content, so some changes are ignored: spaces, commas, and capitalization. Moreover, it also checks if the generated file is parsed as the same object as the original file. Furthermore, the files are also converted if they cannot be correctly decoded. If you run test.py with your working directory set, all files encoded with Windows-1252 will automatically be converted to UTF-8.

The compare_parsing function in test_parser can be used to compare the parser with another bib file parser library[6]. This requires "pip install bibtexparser". The function will compare the number of parsed references and the number of fields in each parsed reference. A limitation here is that a file with duplicate cite keys cannot be correctly compared.

6.2 Results

6.2.1 CLI Test Results

Test case	Result	Notes
Set working directory	PASS	-
Load bib file	PASS	-
View file	PASS	-
Collapse names	PASS	-
Expand back	PASS	-
Clean bib file	PASS	-
Check clean consistency	PASS	-
Check ordering function	PASS	-
Check search function	PASS	-
Check filter function	PASS	-
Two file merge	PASS	-
Merge all in directory	PASS	Could use a progress bar
Enrich missing DOI/URL	PASS	Could use a progress bar; Misses logging of references that could not be found online.
Offline execution of enrichment	PASS	Could use better/faster system to check connection.
Undo/redo	PASS	Should not parse document on commit.
Checkout function	PASS	-
History delete function	PASS	-
Graph generator	PASS	Could use a progress bar.
Config print	PASS	Should print config keys with current values.

6.2.2 Automated tests results

In our set of test .bib files, the automated tests find a couple of files that cannot be parsed, but all of these files contain duplicate fields with different contents. Additionally, the automated tests also find some differences between the original and the generated file. These differences are caused by duplicate fields with equal values, which are removed during the parsing of the file.

When using the compare_parsing function with the same files, there are also some inconsistencies found. These are mainly differences in the number of references caused by the difference in design decisions between the parsers: our parser often parses more entries, since it does not care about missing fields or commented-out references. It just parses everything as a reference. Furthermore, there are also some .bib files with duplicate citation keys, which means that they cannot be properly compared with the current compare_parsing implementation.

The main takeaway of this automated testing is that there will always be some files that cannot be parsed or are parsed differently from other parsers. However, these remain a small portion of all of the tested files, the overwhelming majority of the tested files being parsed similarly to the bibtexparser library[6].

7 Appendix

7.1 Special Thanks

We would like to thank Vadim Zaytsev and Nhat Bui for their supervision, valuable feedback, and continuous support.

7.2 Release

BibSTEAK is available in our GitHub repository, which can be found here.

7.3 Individual Contributions

Edwin

– Presentations

– Enrichment

– Abbreviations

– Installation

Fabian

– Version Control

– Graph Visualization

– Scripting via API object

– CLI basic backbone

Lisa

– Filtering

– Tagging

– Searching

– Cleaning CLI

– Error handling CLI + misc. files

Maria

– Design GUI

– Implementation GUI

– Integration GUI

– Poster

Max

– Design merge protocol

– Implementation Merging

– Error Handling

– Tab Completion

Wouter

– BibFile Object

– Parser

– Generator

– Cleanup

– Batch replace

– Abbreviations

References

[1] [Online]. Available: https://nicegui.io/

[2] A. Feder. (2006). [Online]. Available: https://www.bibtex.org/Format/

[3] [Online]. Available: https://www.unicode.org/charts/PDF/U0300.pdf

[4] [Online]. Available: https://iq.wiki/wiki/merkle-tree

[5] [Online]. Available: https://openalex.org/

[6] [Online]. Available: https://pypi.org/project/bibtexparser/