ForVis - Doc3 - Visualization of Logical Formulas

Micek

1 Introduction

The aim of this paper is to present the current state and development possibilities of a system for visualizing logical formulas using new techniques, including graphs, diagrams, and charts. Additionally, theoretical methods of selected algorithms are presented, as well as a practical comparison of existing functionalities with newly implemented ones for the purposes of this study. The system for visualizing logical formulas, called ForVis, is very useful in solving so-called satisfiability problems. By visualizing the problem and the logical formula, a programmer can identify and utilize common features of formulas, improving their program.

2 Satisfiability Problem

The satisfiability problem is one of the greatest challenges in mathematics and theoretical computer science. Solving it can resolve many other problems. It is defined as checking whether there exists an assignment of logical values (true/false) to variables in a given formula that satisfies it. The satisfiability problem (abbreviated as SAT from the English 'propositional satisfiability problem') is an NP-complete problem.

3 SAT Solvers

Algorithms that solve the satisfiability problem are called SAT solvers. All of them have exponential complexity in the worst case. Despite this, SAT-solving programs are frequently used in various fields of science. Examples include:

- automation of electronic design,
- FPGA routing,
- planning and scheduling problem-solving,
- automatic test pattern generation,
- probability reduction of problems,
- providing information about common set features,

- broad applications in inference systems.

4 Logical Formula Visualization

To improve the efficiency of algorithms solving the SAT problem, effective visualization of the existing problem can be useful. The ForVis system currently includes many built-in visualization methods, but further development and improvement may significantly enhance its performance. The most important of these are the SAT Interaction Graph, SAT Factor Graph, and SAT Resolution Graph.

4.1 SAT Interaction Graph

The SAT Interaction Graph is the simplest visualization implemented in the ForVis application. The program's output is a visualized formula in graph form, where the vertices represent literals provided as input, and edges exist between two variables only if they belong to the same clause.

4.2 SAT Factor Graph

Another visualization is the SAT Factor Graph. The output of this method is a graph whose vertices represent both clauses (numbered C_1, C_2, C_3, etc.) and literals provided as input. Edges exist when a literal belongs to a given clause.

4.3 SAT Resolution Graph

The SAT Resolution Graph differs significantly from previous visualizations as its vertices represent only formulas, numbered C_1, C_2, C_3, etc. A connection between formulas exists only if one clause contains at least one literal X and another clause contains its negation $\neg X$.

4.4 Matrix Visualization

The ForVis system also offers matrix visualization. It works in a simple way: if two literals X_A and X_B appear in at least one disjunctive formula, the corresponding cell in the table is set to X or 1.

4.5 Distribution Chart

The final described functionality is a statistic that shows the number of occurrences of literals in the entire input formula. The number of negated variables is marked in red, while the number of non-negated variables is marked in light blue.

5 Methods and Techniques Used to Improve Visualization of Logical Formulas in the System

5.1 System Architecture

5.1.1 Architecture

The system, which will be improved and extended in this study in terms of functionality (Figure 10), consists of several interdependent components. Each component can operate independently but communicates with others through various protocols. These components are:

- The web interface (frontend) is developed using Angular. It is tightly integrated with the backend server, which handles all business logic, including user management and visualizations.
- The backend is written in Python using the popular Django framework. A SQL database is used for data storage.
- The PostgreSQL database manages data and stores input objects for the ForVis system as well as visualization results.
- Nginx enables multiple server processes to execute backend code and automatically distribute traffic evenly among them.
- To execute visualizations in parallel, the Django server uses the message broker RabbitMQ.

The modified system's deployment is simplified by Docker, which automatically builds and runs the appropriate images in containers based on configuration files. An image is a set of file system layers and Docker metadata. In simple terms, it can be considered a mini-system where a given application is run. A container allows running a process (program) in an isolated environment containing all dependencies required for its execution.

A user who wants to visualize a given logical formula must upload the selected input file to the server via a web browser. The backend server (Django) then stores it in the database. To process the visualization, it sends a message to the message broker, which asynchronously executes the selected visualization algorithm. The result is stored as a database object in PostgreSQL, allowing the user to review all previous visualizations.

Since visualizations may consist of many variables and clauses, processing time can be long. To optimize performance, a task queue functionality was added using the Celery framework (Diagram 1). The user selects a file and a visualization algorithm on the web interface. The system assigns the task to Celery, and once the visualization is complete, the user is notified via the interface. The process is depicted in Diagram 1.

5.1.2 Docker

ForVis is a complex system comprising multiple modules. To efficiently manage and deploy them, the project incorporates Docker and Docker Compose. These tools enable virtualization at the OS level by creating containers that encapsulate applications. Containers are defined using text files called Dockerfiles, which specify how to build and configure the container.

In the original system version, Docker was included but misconfigured. The backend used the python-igraph library, which requires cmake and libigraph0-dev for compilation. To resolve build issues, developers had to manually install these dependencies inside the container. To automate this, the configuration was modified as follows:

```
RUN apt-get update && apt-get install -y libigraph0-dev
```

With this update, running the command ‘docker-compose up‘ from the project root directory correctly initializes all containers.

5.1.3 Python

The ForVis backend is implemented in Python using the Django web framework. Django provides essential functionality out-of-the-box, allowing developers to focus on application development. Features include an Object-Relational Mapper (ORM) for database structure management and an Admin tool for easy administration.

5.1.4 PostgreSQL

All server data in ForVis is stored in the PostgreSQL database, a widely used object-relational database system. Django supports SQL queries, allowing developers to work with database objects in Python without writing raw SQL commands.

The database schema consists of three tables and two enumerations. The primary tables are:

- **Profile**: Linked to the Django User table (not shown in Diagram 2), enabling authentication.
- **TextFile**: Represents an uploaded input file with fields for name, content, and a flag indicating compression status.
- **JsonFile**: Represents a visualized file with fields indicating visualization type, status (Empty, Pending, Done), and task progress.

Diagram 2 illustrates the database schema for ForVis.

5.1.5 Angular

The web interface (frontend) of the application is built using Angular, a popular TypeScript-based framework. Angular provides:

- A component-based structure for web applications,
- Libraries for routing, form handling, and client-server communication,
- Development tools for building, testing, and updating code efficiently.

5.1.6 Celery / RabbitMQ

Some logical formula files can be large. To optimize system performance, the project incorporates Celery, an open-source task queue system. RabbitMQ is used as a message broker to manage and distribute queued tasks asynchronously.

5.2 System Errors

An analysis of the initial ForVis system identified several errors and usability issues. Although they may seem minor, they significantly impact user experience. Resolving these issues required substantial programming effort and expertise.

5.2.1 File Management Issues in the Database

The original system stored each uploaded file as a FileField model entry. Files were then compressed using SatELite_v1.0_linux to save storage space. However, the compression tool could not access newly stored files due to incorrect permissions, resulting in a 'Permission Denied' error.

To fix this, the following setting was added to Django's 'settings.py':

```
FILE\_UPLOAD\_PERMISSIONS = 0o777
```

Without this change, files were marked as 'EMPTY' (Figures 11 and 12), indicating they were not processed correctly.

5.3 Email Notification Functionality

5.3.1 Error Notifications

The original version of the system did not utilize email functionality. In the new version, email notifications have been integrated to confirm task execution, notify users of system errors, and provide updates on system activity. This integration was achieved using Django's built-in email module. While Python provides an interface via the 'smtplib' module, Django offers a more user-friendly approach that facilitates faster email delivery, easier testing, and compatibility with non-SMTP platforms.

In the initial version, when an unhandled exception occurred, the system returned an ‘Internal Server Error‘ without notifying administrators. To address this, a logger object named ‘email_on_exception_logger‘ was added using Django’s ‘logging‘ package. First, formatters were configured to ensure that log records could be properly rendered as text before being stored in the log file (Figure 14). These formatters define the exact text structure of log records (Listing 6).

Handlers were then configured to determine how the logging system processes messages (Listing 5). Handlers can display messages on the screen, write them to a file, or send them to an external output, such as a debugging port. Each handler has a log level that determines the importance of messages. If the message level is lower than the handler’s threshold, it is ignored. The standard Python logging levels are:

- DEBUG: Low-level debugging information.
- INFO: General system information.
- WARNING: Minor issues that occurred.
- ERROR: Serious errors that occurred.
- CRITICAL: Critical system failures.

Listing 7 shows the configured ‘mail_admins‘ handler, based on Django’s ‘AdminEmailHandler‘. It sends an email to system administrators whenever an error message is logged. This follows two principles:

- If the message contains request attributes, they are included in the email.
- If the log record contains stack trace information, it is also included.

To further enhance logging functionality, a dedicated ‘email_on_exception_logger‘ was configured. Any ‘ERROR‘ or ‘EXCEPTION‘ log entry triggers an automatic email notification to administrators (Listing 8).

5.3.2 Inbox and Administrator Configuration

For email notifications to function correctly, the system’s email inbox needed proper configuration. This included setting up a Gmail account and defining email server settings (Figure 14). Additionally, system administrators were configured via the ‘ADMINS‘ list in Django’s ‘settings.py‘ file (Figure 13).

5.3.3 Email Confirmation of Visualization Completion

A new feature allows users to receive an email notification upon completion of their visualization task. This was implemented through a simple service called ‘EmailService‘ (Figure 16), which includes a method named ‘send_email‘. This method allows sending an email with a given message as a parameter. Although simple, this class significantly enhances the usability of the ForVis application (Figures 17 and 18).

6 Visualizations

The primary goal of the system is to visualize logical formulas graphically. This can be achieved using various techniques, such as graphs, charts, binary trees, or even two-dimensional tables. The system includes multiple visualization methods, such as 'sat_vis_factor', 'sat_vis_interaction', and others. Each method presents the formula differently.

Visualization tasks are initiated by calling the 'start_task' endpoint, passing the formula file as a URL parameter. The corresponding method creates database records and launches an asynchronous task. Once the task is completed, the results are stored in the database and can be viewed via the web interface. Users can also download visualizations as PDFs or save them as JPG images.

6.1 Existing Visualizations

The system currently supports DIMACS CNF file input, which is transformed into an appropriate graph representation. CNF (Conjunctive Normal Form) is defined as a formula composed of clauses, where each clause is a disjunction of literals:

$$(x \vee \neg y) \wedge (z \vee y \vee \neg x) \tag{1}$$

DIMACS CNF files are ASCII text files compatible with all operating systems. The file consists of natural numbers representing variables, with optional negation indicated by a preceding minus sign. Each clause ends with a zero. Example content is shown in Figure 19.

6.2 New Visualizations

A new visualization algorithm has been implemented to detect 'communities' in graphs, representing subgraphs with dense internal connections and sparse external connections. The CNM algorithm (Convolutional Neural Network) was used for this purpose, as described in the research paper 'Mining Communities in Networks'. The algorithm calculates modularity to measure the quality of community partitions. The implementation is optimized using binary trees and max heaps, achieving computational complexity of $O(m * d * log(n))$.

7 Presentation of the Modified Version of ForVis System

7.1 Community Visualization

The visualization of the community detection algorithm in the new version of ForVis differs slightly from existing visualization methods. Each method takes a logical formula file as input, while the CNM algorithm requires a graph. A

‘communities‘ button has been added to the visualization menu to allow users to execute the algorithm on a selected graph. For visualizations that do not use graphs (e.g., tables), the button is replaced with the text ‘Does not support‘. Figures 24-28 show selected graphs and their detected communities.

8 User Guide

8.1 Client User Guide

ForVis is an intuitive system. Users must first create an account using the ‘Sign Up‘ button in the top navigation bar. If an account exists, users can log in via the ‘Login‘ button. To visualize a formula, users must upload a file containing the formula. In the Sat panel (Figure 31), the ‘Browse‘ option allows file selection, initiating the upload process.

The ‘Upload all‘ button stores files in the database, ‘Remove all‘ deletes all files, and ‘Cancel all‘ removes files uploaded to the browser but not yet saved (Figure 32). After uploading a file, users can request visualization using the ‘Visualize‘ option. A window will appear allowing users to choose a visualization method (Figure 33).

To analyze visualization results, users can select ‘Visualizations‘ from the navigation bar. A table with previous visualizations will be displayed (Figure 34). Clicking ‘Visualize‘ opens a selected result.

8.2 Administrator User Guide

The administrator panel is accessible at ‘/admin‘. Administrators can log in using predefined credentials. A sidebar on the right lists database tables (Figure 35). Selecting a table displays objects that can be viewed and modified (Figure 36). Administrators can add, delete, or modify objects, and create new user accounts in the ‘User‘ table.

9 System Performance Evaluation

To assess system performance, three visualization methods (Interaction Graph, Factor Graph, Resolution Graph) were tested on different input files:

- File 1: 118,700 variables, 3,480,580 clauses, 2 variables per clause.
- File 2: 841 variables, 120,147 clauses, 3 variables per clause.
- File 3: 94,998 variables, 302,862 clauses, 3 variables per clause.

Performance results are shown in Figure 36. The computational complexity of the algorithms is:

- Sat Interaction Graph: $O(k * n^2)$

- Sat Factor Graph: $O(k * 3 * n)$
- Sat Resolution Graph: $O(k * n + lz * kz)$
- Communities Graph: $O(m * d * \log(n))$

System memory usage is critical when processing large datasets. If multiple visualizations with hundreds of thousands of variables and clauses are run simultaneously, the system may exceed RAM limits and terminate processes. This occurs because visualizations are stored in memory as Python objects.

10 Conclusion

The original ForVis system was unreliable, with no administrator control over its stability. The new version introduced significant improvements, including an email service for error reporting, bug fixes, and enhanced visualization capabilities. These updates improve the system's reliability and usability, making ForVis a more effective tool for analyzing logical formulas.