1 Introduction

The goal of this thesis was to provide a detailed discussion on the current capabilities and future development of the **ForVis** system (http://forvis.agh.edu.pl), designed for the visualization of logical formulas. The project incorporated modern visualization techniques, such as graphs, charts, and figures, to enhance the presentation of logical data.

The first part of the thesis discusses the basic visualization methods available in the ForVis system. The system's backend is implemented in `Python`, while the user interface is developed in `Angular`. The development project for the ForVis system, aimed at graphical representation of logical formulas, was divided into four phases:

1. **Hypergraphs** – Logical formulas were represented using hypergraphs to better reflect the complexity of logical structures.

2. **Graph embeddings** – Various graph embedding mechanisms were analyzed and implemented to provide a more intuitive representation of logical relationships.

3. **Community detection** – The Louvain algorithm was visualized, enabling the detection of communities in graph structures.

4. **CDCL heuristics** – Heuristic visualization for the CDCL algorithm, which plays a crucial role in solving SAT problems, was implemented.

Additionally, a *SAT solver* based on the CDCL algorithm was implemented within the project.

1.1 System Architecture

The *ForVis* system was designed as a comprehensive solution for visualizing complex logical formulas. The architecture follows a modular approach, allowing each component to function independently while collaborating with others via various communication protocols. Below is a detailed description of the system's architecture components.

The *ForVis* system consists of three main layers:

- **Frontend (presentation layer)** – Responsible for user interaction.

- **Backend (business logic layer)** – Handles data processing and communication with the database.

- **Database (data storage layer)** – Ensures persistent storage of input files, visualization results, and user data.

Additionally, the system supports:

- **Asynchronous task processing** using *Celery* and *RabbitMQ*.
- **Containerization** with *Docker* and *Docker-Compose*.
- **Network traffic management** through the *Nginx* proxy server.

1.1.1 Frontend

The frontend is implemented in *Angular 5*, using *TypeScript*. The component-based structure facilitates a modular approach to application development, enhancing readability and maintainability.

Key features of the frontend:

- **Views and components:** Each component has separate files for the view (HTML), logic (TypeScript), styles (CSS/SCSS), and unit tests (*Karma*).
- **Communication with backend:** Handled via HTTP protocol (POST method), using dedicated Angular services.
- **User functionalities:**
  - Uploading input files.
  - Selecting visualization algorithms.
  - Displaying and managing visualization results.

1.1.2 Backend

The backend is implemented in *Python* using *Django Rest Framework*, enabling rapid and efficient web application development.

Key backend components:

- **Views:** Implemented as *APIView* classes, handling various HTTP requests.
- **Models:** Defined in `profiles/models.py`, allowing the creation of database structures and migrations.
- **Routing:** Views are mapped to URLs in `profiles/urls.py`.
- **Asynchronous processing:** Resource-intensive tasks (e.g., visualization generation) are executed asynchronously using *Celery* and *RabbitMQ*.

1.1.3 Database

The system uses *PostgreSQL* as the relational database management system.

Key database features:

- **Data models:** Store input files, visualization results, and processing status.
- **Integration with Django ORM:** Allows working with Python objects instead of direct SQL queries.
- **Data models:**
  - The **TextFile** table stores input files.
  - The **JsonFile** table stores visualization results (in JSON format) and processing status.
- **Integration with Django ORM:** Allows working with Python objects instead of writing direct SQL queries.
- **User data:** Stored and managed using Django's authentication mechanisms.

1.1.4 Containerization

The system has been containerized using *Docker*, which allows for easy management and deployment of applications.

Containerization framework:

- Containers for individual components, such as backend, frontend, RabbitMQ, PostgreSQL, and Nginx.
- `Dockerfile` files define how containers are built and how dependencies are installed.
- *Docker-Compose* makes it easy to run a multi-container system by centrally managing configuration.

1.1.5 Nginx

Nginx acts as a proxy server that manages traffic between the frontend and the backend.

Nginx Features:

- Load balancing between backend processes.

- Data compression to optimize performance.
- Handling multiple user requests at the same time.

1.1.6 Asynchronous task processing

Some input files may require large computational resources, so the system uses *Celery* to handle asynchronous tasks.

Working Mechanism:

1. Tasks are passed to *Celery* via *RabbitMQ*.
2. Processing is done in parallel, making the system scalable and efficient.
3. Processing results are saved in the database, and the user is notified when the task is completed.

1.1.7 System operation process

1. User uploads input file via frontend.
2. Backend saves file in database and passes visualization task to *Celery*.
3. *Celery* processes task asynchronously using *RabbitMQ*.
4. Visualization result is saved in *PostgreSQL* and available in user interface.
5. User can browse and manage his visualizations.

1.2 Visualization methods currently existing in the system

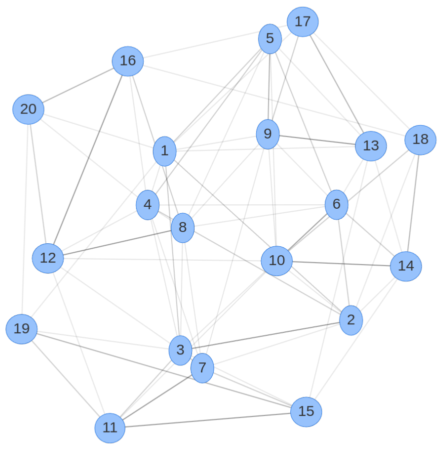
To present all types of visualizations, except for Tree Visualization and 2-Clauses Interaction Graph, the `example_20_40.cnf` file was used, containing 20 variables and 40 clauses. Its content is presented in listing 1. On the other hand, the `example_20_38.cnf` file was used for Tree Visualization and 2-Clauses Interaction Graph visualization, which contains 20 variables and 38 clauses. Its content is in listing 2.

1.2.1 Interaction Graph

Description: The Interaction Graph is a simple graph in which vertices correspond to literals, and edges represent the co-occurrence of literals in the same clause.

Implementation:

- Nodes represent literals.
- Edges are added between literals that co-occur in a clause.
- Duplicate edges are eliminated.
- The algorithm iterates over all clauses and merges pairs of literals, creating a graph structure that highlights the interactions of variables.



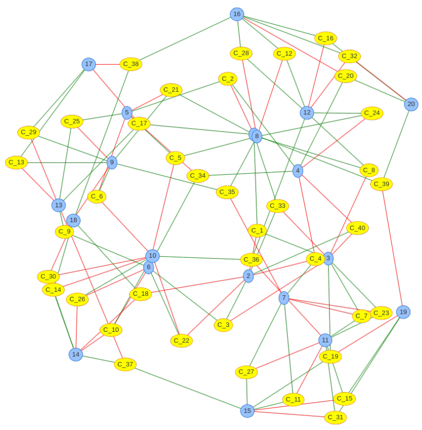
Rysunek 1: SAT Interaction Graph `example_20_40.cnf`

1.2.2 Factor Graph

Description: The Factor Graph extends the Interaction Graph by adding clause nodes. Edges connect literals to the clauses they appear in, providing a richer representation.

Implementation:

- Nodes include literals and clauses.
- Edges are drawn between literals and their corresponding clauses.
- Colored edges indicate the presence of positive (green) or negative (red) literals.
- The algorithm builds a graph by analyzing clauses and assigning literals to the appropriate clause nodes.



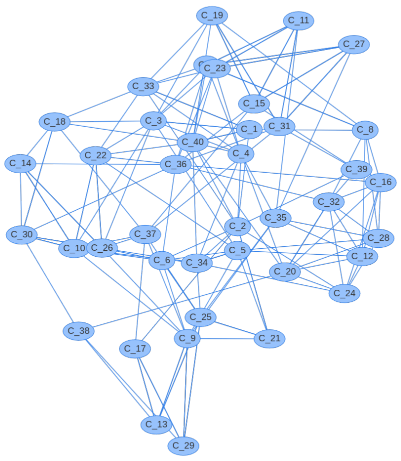
Rysunek 2: SAT Factor Graph `example_20_40.cnf`

1.2.3 Resolution Graph

Description: The Resolution Graph represents the relationships between clauses. Nodes correspond to clauses, and edges are drawn when clauses share complementary literals.

Implementation:

- Nodes represent clauses.
- Edges are added when a literal in one clause has its negation in another.
- The graph highlights logical dependencies and resolution steps.
- Redundant edges are eliminated for clarity.



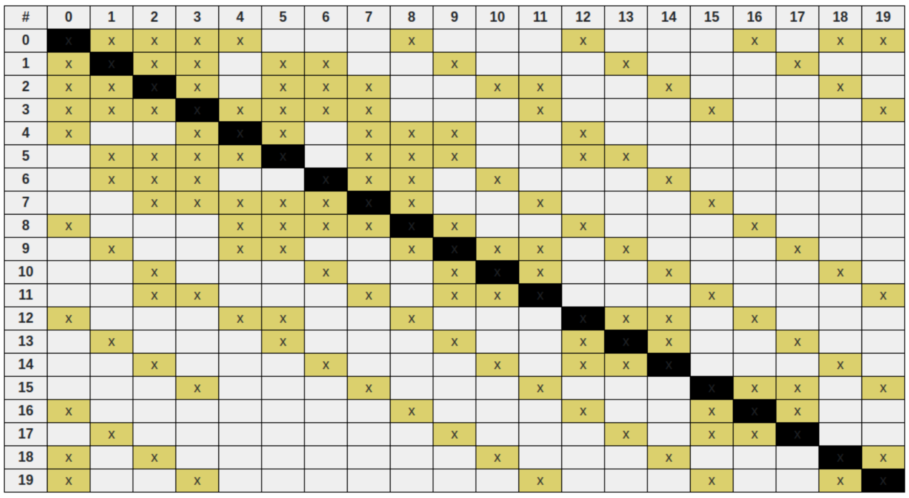
Rysunek 3: SAT Resolution Graph `example_20_40.cnf`

1.2.4 Matrix Visualization

Description: The Matrix Visualization provides a tabular view of the relationships between literals. Each cell indicates the co-occurrence of literals in the clause.

Implementation:

- Rows and columns represent literals.
- Cells contain values indicating positive, negative, or no co-occurrence.
- Data is structured in a scalable way, allowing analysis of large formulas.



#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	x	x	x	x	x				x				x				x		x	x
1	x	x	x	x		x	x			x				x				x		
2	x	x	x	x		x	x	x			x	x			x				x	
3	x	x	x	x	x	x	x	x				x				x				x
4	x			x	x	x		x	x	x			x							
5		x	x	x	x	x		x	x	x			x	x						
6		x	x	x			x	x	x		x				x					
7			x	x	x	x	x	x	x			x				x				
8	x				x	x	x	x	x	x			x				x			
9		x			x	x			x	x	x	x		x				x		
10			x				x			x	x	x			x				x	
11			x	x				x		x	x	x				x				x
12	x				x	x			x				x	x	x		x			
13		x				x				x			x	x	x			x		
14			x				x				x		x	x	x				x	
15				x				x				x				x	x	x		x
16	x								x				x			x	x	x		
17		x								x				x		x	x	x		
18	x		x								x				x				x	x
19	x			x								x				x			x	x

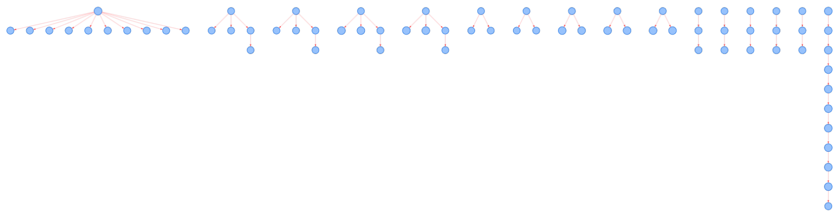
Rysunek 4: SAT Matrix Visualization `example_20_40.cnf`

1.2.5 Tree Visualization

Description: Tree Visualization presents a hierarchical structure of formulas, showing dependencies between clauses and variables.

Implementation:

- Nodes represent clauses, and edges represent logical dependencies.
- Hierarchical layout organizes the tree for better readability.
- The algorithm analyzes clauses and builds a tree structure based on dependencies.

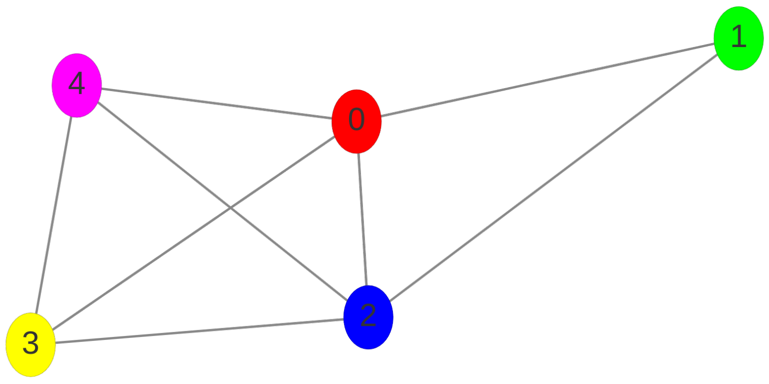

Rysunek 5: SAT Tree Visualization `example_20_38.cnf`

1.2.6 Cluster Visualization

Description: Cluster Visualization groups related literals and clauses into clusters, representing modular structures in a formula.

Implementation:

- Community detection algorithms identify clusters.
- Nodes are colored by cluster.
- Edges between clusters are minimized to emphasize modularity.



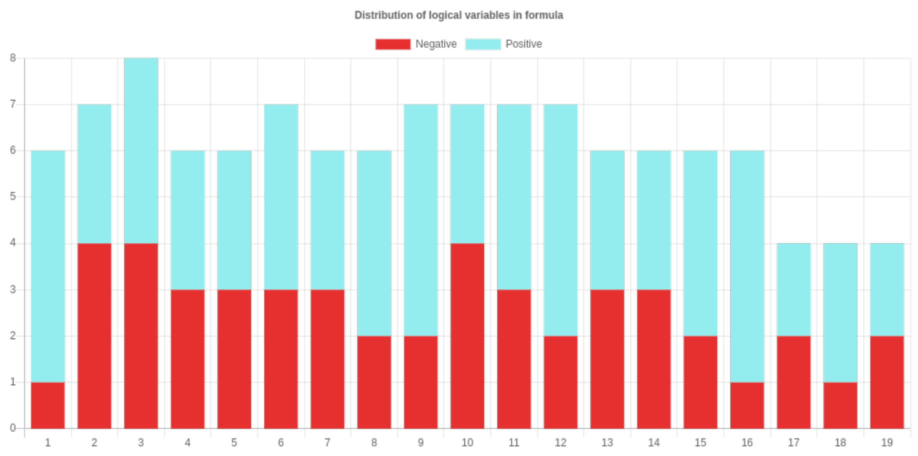
Rysunek 6: SAT Cluster Visualization `example_20_40.cnf`

1.2.7 Distribution Chart

Description: The Distribution Chart visualizes the frequency of literal occurrences in a formula, distinguishing between positive and negative occurrences.

Implementation:

- The bar chart displays the frequencies of literals.
- Positive and negative occurrences are shown in different colors.
- The chart summarizes the distribution of variables in a formula.



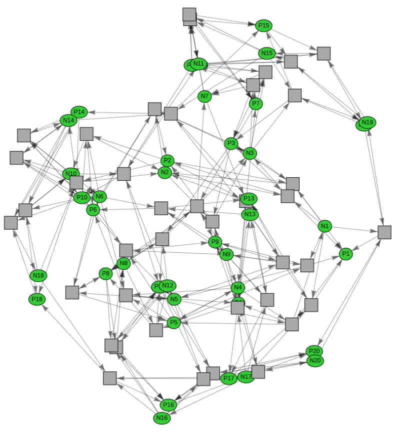
Rysunek 7: SAT Distribution Chart `example_20_40.cnf`

1.2.8 Direct Graphical Model

Description: The Graphical Model visualizes clauses and literals as connected nodes with directed edges indicating logical implications.

Implementation:

- Nodes represent literals and clauses.
- Directed edges indicate logical implications or dependencies.
- The algorithm uses physics-based systems for clarity.



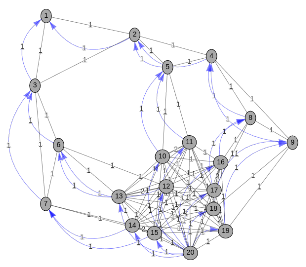
Rysunek 8: SAT Direct Graphical Model `example_20_40.cnf`

1.2.9 2-Clauses Interaction Graph

Description: This visualization focuses on relationships in CNF formulas consisting of two-element clauses, emphasizing interactions in minimal clause structures.

Implementation:

- Nodes represent literals.
- Edges connect literals in the same two-element clause.
- Edge styles and colors distinguish interaction types (positive-positive, negative-negative, mixed).



Rysunek 9: SAT 2-Clauses Interaction Graph `example_20_38.cnf`

1.2.10 DPLL Solver Visualization

Description: This visualization shows the decision-making process of the DPLL algorithm, including variable assignments and conflicts.

Implementation:

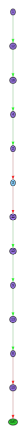
- Nodes represent decisions, conflicts, or end states (SAT/UNSAT).
- Directed edges trace the course of the decision tree.
- Heuristics such as DLIS, Jeroslow-Wang, and MOMS guide decision nodes, reflecting different visual layers.
- Colors distinguish node types and states.

Heuristics	Mechanism of action	Advantages	Disadvantages
MOMS	Selects the variable most frequently occurring in the smallest clauses	Fast elimination of short clauses	Does not take into account the structure of the entire formula
DLIS	Selects the dominant variable in unsatisfied clauses	Dynamically adapts to the current state	High computational cost
Jeroslow-Wang	Selects the variable that maximizes the sum of clause weights	Takes into account the weight clauses, often more effective	High computational costs

Tabela 1: Comparison of heuristics used in DPLL algorithm

Rysunek 10: SAT DPLL Solver Visualization MOMS `example_20_38.cnf`

Rysunek 11: SAT DPLL Solver Visualization DLIS `example_20_38.cnf`



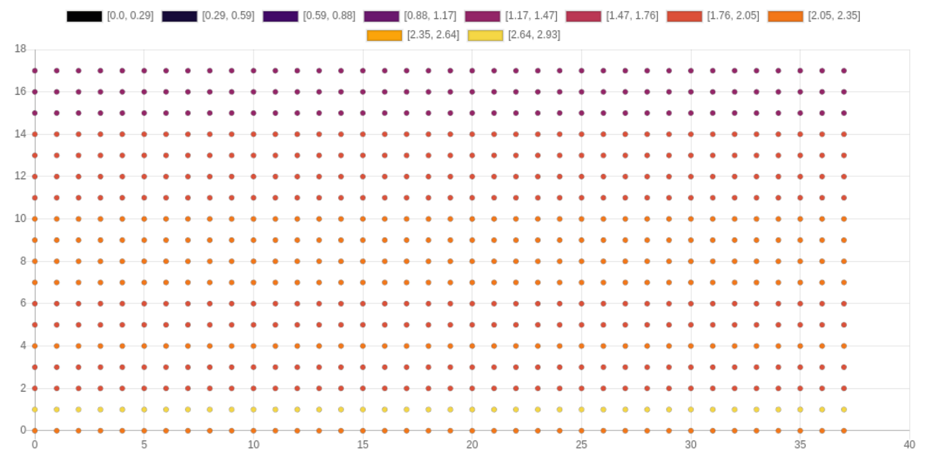
Rysunek 12: SAT DPLL Solver Visualization Jeroslow-Wang `example_20_38.cnf`

1.2.11 Heatmap

Description: Heat Maps provide a visual representation of the relationships between clauses and variables, using color intensity to show frequency and importance.

Implementation:

- The matrix maps clauses (rows) and literals (columns).
- The intensity of the color reflects the frequency of variables in the clauses.
- The data is scaled for clarity, and outliers are highlighted in other colors.
- The algorithms process and scale large data sets, enabling powerful visualization.



Rysunek 13: SAT Heatmap Visualization `example_20_38.cnf`

1.2.12 Raw File Representation

Description: The raw content of a DIMACS CNF file is presented without additional processing, providing a plain text view.

Implementation:

- The file content is read and displayed in its original form.
- No transformations or visualizations are applied.

```
p cnf 20 40
1 -2 3 0
-1 4 5 0
2 -3 6 0
-2 -4 7 0
5 -6 8 0
-5 9 -10 0
3 -7 11 0
-3 8 12 0
6 -9 13 0
-6 10 -14 0
7 -11 15 0
-8 12 16 0
9 -13 17 0
-10 14 18 0
11 -15 19 0
-12 16 20 0
```

```
13 -17 1 0
-14 18 -2 0
15 -19 3 0
-16 20 4 0
1 -5 9 0
-2 6 -10 0
3 -7 11 0
-4 8 12 0
5 -9 13 0
-6 10 -14 0
7 -11 15 0
-8 12 16 0
9 -13 17 0
-10 14 -18 0
11 -15 19 0
-12 16 -20 0
1 2 -3 0
4 -5 6 0
-7 8 9 0
10 -11 12 0
-13 14 15 0
16 -17 18 0
-19 20 1 0
2 -3 -4 0
```

Listing 1: File `example_20_40.cnf`, containing 20 variables and 40 clauses

```
p cnf 20 38
1-2 0
1-3 0
2 -4 0
2 -5 0
3 -6 0
3 -7 0
4 -8 0
4 -9 0
5 -10 0
5 -11 0
6 -12 0
```

```
6 -13 0
7 -14 0
7 -15 0
8 -16 0
8-17 0
9 -18 0
9 -19 0
10 -20 0
11 -20 0
12 -20 0
13 -20 0
14 -20 0
15 -20 0
16 -20 0
17 -20 0
18 -20 0
19 -20 0
-1 2 3 0
-2 4 5 0
-3 6 7 0
-4 8 9 0
-5 10 11 0
-6 12 13 0
-7 14 15 0
-8 16 17 0
-9 18 19 0
-10 -11 -12 -13 -14 -15 -16 -17 -18 -19 20 0
```

Listing 2: File `example_20_38.cnf`, containing 20 variables and 38 clauses

2 TODO: New Ways to Visualize Logical Formulas

2.1 TODO: Graph Embeddings Visualization

2.1.1 Introduction to Graph Embeddings Theory

2.1.1.1 Basic Concepts Graph embeddings (*Graph Embeddings*) is a method of representing vertices (or entire graphs) in vector form. Its main goal is to preserve both structural and semantic information in a possibly low-dimensional space.

Vertices and Edges A graph G is defined as a pair (V, E), where V denotes a set of vertices (*vertices*) and E a set of edges (*edges*). Each edge $e \in E$ connects some pair of vertices (v_i, v_j), which can be written as $e = (v_i, v_j)$.

Graph representation One of the most popular ways of representing graphs is:

- **Adjacency matrix** $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge between } v_i \text{ and } v_j, \\ 0 & \text{otherwise.} \end{cases}$$

- **Adjacency list** L, which for each vertex enumerates the set of all its neighbors.

2.1.1.2 Graph Embeddings Graph Embeddings techniques allow for transferring (embedding) vertices into vector space in such a way that the dimensions of these vectors (embeddings) reflect the similarities and relational properties of the vertices in the graph. In practice, this means that:

- **We minimize the dimensionality of the data**, which allows for faster processing.
- **We preserve information about the graph structure**, i.e. vertices that are similar in the original graph (e.g. have common neighbors or are located in a similar topological area) will be located close to each other in the new vector space.
- **Simplified analysis and visualization**, because a small number of dimensions can be visualized much more easily (usually reduced to 2D or 3D).

Why is embedding better than adjacency matrix? The adjacency matrix grows quadratically with the number of vertices, which makes it very large and unwieldy for many applications. Additionally, it does not directly carry information about the features of the vertices (unless it is extended in a very complex way). Embeddings, on the other hand, can compactly store vertex properties in the form of vector components in addition to structural relations.

Graph Embeddings Algorithms There are many methods for embedding graphs, including algorithms such as *DeepWalk*, *Node2Vec*, *GraphSAGE*, and *LINE*. In our research, we focus on one of the most popular methods:

- **Node2Vec**: A method extending the *DeepWalk* approach, which uses *random walks* with parameters p and q controlling the way the graph is explored. The main steps of Node2Vec are:

  1. Initializing random walks (e.g., for each vertex we perform several dozen independent walks).
  2. During the walk, we move to the next vertex with probabilities that modify the number of returns (similar to DFS) or long jumps (similar to BFS).
  3. Recording the visited sequence of vertices in a way similar to a sequence of words in a sentence (an approach similar to *Word2Vec* in learning text representations).
  4. Train a *skip-gram* (or *CBOW*) model to create vectors representing vertices.

  This ensures that vertices that frequently appear together in similar contexts (i.e. have similar neighborhood structures) will obtain similar vector representations.

Loss function In *skip-gram*-based approaches (such as *Node2Vec* or *DeepWalk*), the most common loss function to minimize is the sum of the negative logarithms of the vertex probabilities in the context window. This can be written as follows:

L = - $\sum_{v\in V}\sum_{u\in\mathcal{N}(v)}\log\Pr(u \mid v)$, where $\mathcal{N}(v)$ is the set of vertices in the "neighborhood"(context) of vertex v. The model learns vertex representations such that the probability $\Pr(u \mid v)$ is large for vertices that often appear together in paths.

2.1.1.3 Description of the whole method To illustrate the approach in practice, we will use an example of a SAT-type problem (in *DIMACS CNF* format). Each clause contains a number of logical variables, and our goal is:

1. Create a graph whose vertices are *variables* appearing in clauses.
2. Create edges between vertices that co-occur in the same clause (a relationship in the form of "we appear together").
3. Perform an initial visualization (e.g. in the form of a simple graph).
4. Perform **embedding of the graph** in vector space using `node2vec` (using ready-made libraries in Python).

5. Perform dimension reduction, e.g. using PCA (ang. *Principal Component Analysis*), to obtain a two-dimensional representation from the received embeddings.

6. Perform clustering (e.g. `KMeans`) and evaluate it with the *silhouette score* indicator to determine the most "natural"number of clusters.

7. Finally, present the visualization results in the form of a static image (points of embedded vertices with labels).

This approach allows us to see whether variables that often appear together are actually closer together in the embedding space and naturally form clusters.

2.1.2 Implementation

2.2 TODO: Hypergraph Visualization

In this example, we are not constructing a representation in a **low-dimensional vector space** (i.e., we are not doing `Node2Vec`-style embedding here), but rather visualizing the structures of a **hypergraph** and a "classical"graph for the SAT problem. The code draws variables on a circle, connects them (using lines) within clauses, and then creates a hypergraph visualization using the `hypernetx` library.

2.2.1 Introduction to Hypergraph Theory

A hypergraph is a structure that extends the concept of a graph by allowing edges to connect more than two vertices. Unlike graphs, where edges connect only two vertices, hypergraphs allow any number of vertices to be connected. In a hypergraph, edges are called hyperedges. A hyperedge is a set of vertices that are connected to each other. For example, an edge connecting three vertices is a triple.

As with graphs, hypergraphs can be represented using an adjacency matrix, where rows correspond to vertices and columns to hyperedges. Another popular representation is a list of hyperedges, where each hyperedge has an associated list of vertices it connects.

2.2.2 Why a (hyper)graph?

Hypergraphs generalize graphs, allowing the representation of more complex relationships between vertices. In graphs, edges are always two-element, while hypergraphs allow for more flexible relationships.

Representing the SAT problem as a *hypergraph* can be convenient when we want to show that one edge (a so-called hyperedge) connects multiple variables simultaneously (in a clause). In a standard graph, we would have to break this down into multiple pairwise edges connecting variables that co-occur in the clause.

2.3 TODO: CommunityDetectionLouvain

2.3.1 Implementation

- **Loading a DIMACS CNF file:** The `read_dimacs_cnf` function reads a DIMACS CNF file, parses its content, and returns the number of variables and a list of clauses.

- **Louvain community detection:** The `louvain_community_detection` function implements the Louvain algorithm for detecting communities in a graph constructed from logical clauses. The algorithm maximizes modularity by iteratively improving the partitioning of vertices into communities.

- **Community visualization:** Communities are visualized in the graph using different colors, allowing easy identification of groups of logical variables.

2.3.1.1 `read_dimacs_cnf(filename)`

A similar implementation to the one from 2.1.2.1.

2.3.1.2 `louvain_community_detection(num_vars, clauses, threshold=0.0000001)`

- **Description:** Implements the Louvain algorithm for detecting communities in a graph created from logical clauses.
- **Parameters:**
  - `num_vars` - Number of variables.
  - `clauses` - List of clauses.
  - `threshold` - Modularity improvement threshold, default is `0.0000001`.
- **Returns:** Community graph and dictionary.

```python
def louvain_community_detection(num_vars, clauses, threshold
    =0.0000001):
    G = nx.Graph()
    for clause in clauses:
        for literal in clause:
            G.add_node(abs(literal))

    for clause in clauses:
        if len(clause) > 1:
            for i in range(len(clause)):
                for j in range(i+1, len(clause)):
                    G.add_edge(abs(clause[i]), abs(clause[j]))

    partition = community.best_partition(G, resolution=1)
    prev_modularity = community.modularity(partition, G)

    while True:
        improvement = False
        for node in G.nodes:
            for neighbor in G.neighbors(node):
                partition_copy = partition.copy()
                partition_copy[node] = partition_copy[neighbor]
                new_modularity = community.modularity(
    partition_copy, G)
                if new_modularity - prev_modularity > threshold:
                    partition = partition_copy
                    prev_modularity = new_modularity
                    improvement = True
        if not improvement:
            break

    communities = defaultdict(list)
    for node, comm_id in partition.items():
        communities[comm_id].append(node)

    return G, communities
```

2.3.1.3 Community Visualization

- **Description:** Visualizes communities in a graph using different colors, allowing easy identification of groups of logical variables.

```
num_vars, clauses = read_dimacs_cnf("DIMACS_files/turbo_easy/
    example_2.cnf")
G, communities = louvain_community_detection(num_vars, clauses)
print(communities)

pos = nx.spring_layout(G)
plt.figure(figsize=(10, 6))

colors = plt.cm.tab10.colors

for comm_id, nodes in communities.items():
 nx.draw_networkx_nodes(G, pos, nodelist=nodes, node_color=[
    colors[comm_id]], node_size=300, label=f"Community {comm_id}"
    )
```

```
nx.draw_networkx_labels(G, pos, labels={node: str(node) for node
    in nodes}, font_color='black')

nx.draw_networkx_edges(G, pos, alpha=0.5)
plt.title("Louvain Community Detection")
plt.legend()
plt.show()
```

2.4 TODO: CDCL_VSIDS.py

2.4.1 Visualization of VSIDS heuristics in CDCL

Heuristics used in CDCL (Conflict-Driven Clause Learning) algorithms are a key element influencing their efficiency. When dealing with visualization of heuristics, one can focus on several important aspects, such as VSIDS (Variable State Independent Decaying Sum) and clause structure. Here are some visualization and analysis suggestions that might be helpful:

1. VSIDS visualization

    (a) VSIDS value change graph for variables
    We can create a line graph that shows how VSIDS values for individual variables change over time. Each variable will be represented by a separate line on the graph.

    (b) VSIDS value histogram
    A histogram can show the distribution of VSIDS values for all variables at a given point in time. This will help us understand which variables are most preferred for a decision.

2. Visualizing the structure of clauses

    (a) Clause connection graph
    A graph can be created where nodes represent clauses and edges represent shared literals between clauses. Nodes can be colored by the number of connections, which shows which clauses are more "central".

    (b) Connection matrix
    A matrix where cells represent the number of shared literals between clauses can be used to visualize connections. Such a matrix can be represented as a heatmap.

3. Clause connection metrics

    (a) Connection density
    This metric can calculate the average number of connections (shared literals) per clause. This can be used to identify clauses that are more central to the structure of the problem.

   (b) Association coefficient
       You can calculate the association coefficient for pairs of clauses, which shows how strongly they are related through their literals.

4. Variable selection strategies

   (a) Variable selection frequency histograms
       Histograms can show how often each variable was selected for substitution throughout the solution process. This can help you identify which variables were key.

   (b) Solution step analysis
       You can visualize how variables were selected in each step of the algorithm, allowing you to see patterns and possible changes in strategy as the problem is solved.

3 TODO: Summary