Dokumentacja projektu Logic Formulas

Jakub Banach, Karol Błaszczak

January 26, 2024

Contents

1 Wstęp

Projekt Logic Formulas ma na celu rozwój projektu http://forvis.agh.edu.pl/ służącego do reprezentacji formuł logicznych. Nasz projekt został podzielony na 3 etapy. Pierwszym z nich było stworzenie reprezentacji formuł logicznych w formie hipergrafów, drugim była eksploracja pojęcia osadzeń grafowych oraz zaimplementowanie go w naszym środowisku. Jako ostatni zaimplementowaliśmy SAT Solver przy użyciu algorytmu CDCL. Projekt został w całości wykonany w języku programowania Python

1.1 Struktura projektu

Projekt składa się z 3 plików Python: `embeddings.py`, `hypergraphs.py` oraz `CDCLSolver.py`, 3 plików `.ipynb` o tych samych nazwach, zawierających kod z plików `.py`, oraz 26 przykładowych plików `CNF` znajdujących się w folderze `./DIMACS_files` i podzielonych pod kątem trudności.

1.2 Użyte biblioteki

- **networkx**
- **matplotlib**
- **scikit-learn**
- **node2vec**
- **argparse**
- **ipywidgets**
- **plotly**
- **dash**

2 embeddings.py

Program odpowiada za generowanie i wizualizowanie osadzeń węzłów na podstawie formuł logicznych.

2.1 Wprowadzenie do teorii osadzeń grafowych

2.1.1 Podstawowe pojęcia

2.1.1.1 Wierzchołki i krawędzie Graf składa się z wierzchołków V i krawędzi E, gdzie V to zbiór wierzchołków, a E to zbiór krawędzi, które łączą te wierzchołki.

2.1.1.2 Reprezentacja grafu Tradycyjna reprezentacja grafu może być macierzą sąsiedztwa A lub listą sąsiedztwa L. Macierz sąsiedztwa informuje, czy między wierzchołkami istnieje krawędź, a lista sąsiedztwa zawiera sąsiadujące wierzchołki dla każdego wierzchołka.

2.1.2 Graph Embeddings

Po co to w ogóle stosujemy?

Graph Embeddings, czyli osadzanie grafu, to technika reprezentacji wierzchołków grafu w przestrzeni. Są lepsze od macierzy sąsiedztwa, ponieważ dodatkowo upakowują właściwości każdego wierzchołka w wektorze o niższym wymiarze.

Graph Embeddings zakłada przypisanie każdemu wierzchołkowi wektorowej reprezentacji, zwanej embeddingiem. Ta reprezentacja ma za zadanie zachować strukturalne właściwości grafu, takie jak bliskość wierzchołków o podobnym znaczeniu.

2.1.2.1 Algorytmy Graph Embeddings Istnieje wiele algorytmów do osadzania grafów, w naszym programie wykorzystujemy jeden z nich:

- **Node2Vec**: Algorytm generowania embeddingów wierzchołków, uwzględniający strukturę sąsiedztwa. Polega na wykonywaniu określonej ilości losowych spacerów (tzw. random walk) o określonej długości, które dla każdego wierzchołka przechodzą po dowolnych połączonych krawędziach do kolejnych wierzchołków (z jakimś konkretnym prawdopodobieństwem). W jednym random walku dany wierzchołek może być odwiedzony kilka razy!

2.1.2.2 Funkcja straty Podczas treningu modelu Graph Embeddings stosuje się funkcję straty, która mierzy odległość między rzeczywistymi relacjami w grafie a predykcjami modelu.

2.1.3 Opis całej metody

Na samym początku na podstawie każdej linijki z pliku tworzymy połączenia pomiędzy każdą zmienną z danej podformuły (są one od siebie w pewien sposób zależne). Następnie robimy wizualizację tych połączeń i dopiero tutaj wchodzi naszenie osadzenie grafów. Realizujemy je przy przy pomocy metody node2vec (korzystamy z gotowej biblioteki w Pythonie: Node2Vec). Następnie konwertujemy to na model word2vec, ponieważ osadzenia węzłów są wektorami numerycznymi w wielowymiarowej przestrzeni, co czyni je trudnymi do bezpośredniej wizualizacji. Na koniec prezentujemy to w postaci konkretnej statycznej i dynamicznej wizualizacji (odpowiednio plot i dash z bibliotek w Pythonie).

2.2 Implementacja

Po wczytaniu pliku (read_dimacs_cnf()) tworzymy przy pomocy node2vec osadzenia grafowe (korzystamy z gotowej biblioteki w Pythonie: Node2Vec). Następnie konwertujemy to na model word2vec, ponieważ osadzenia węzłów są wektorami numerycznymi w wielowymiarowej przestrzeni, co czyni je trudnymi do bezpośredniej wizualizacji. Później przy pomocy PCA robimy redukcję wymiarów do 2, w celu przejrzystszej reprezentacji. Na sam koniec przy użyciu algorytmu KMEANS dzielimy graf na określoną liczbę podgrup wg zasady najbliższego sąsiedztwa

2.2.1 `read_dimacs_cnf(filename)`

- **Opis:** Odczytuje plik DIMACS CNF i ekstrahuje liczbę zmiennych oraz klauzul.
- **Parametry:** `filename` - Ścieżka do pliku DIMACS CNF.
- **Zwraca:** Krotka zawierająca liczbę zmiennych i listę klauzul.

```
def read_dimacs_cnf(filename):
    clauses = []
    with open(filename, 'r') as file:
        for line in file:
            if line.startswith("c"):
                continue
            if line.startswith("p cnf"):
                num_vars, num_clauses = map(int, line.strip().split()[2:])
            else:
                clause = list(map(int, line.strip().split()[:-1]))
                clauses.append(clause)
    return num_vars, clauses
```

2.2.2 `adding_to_graph(clauses)`

- **Opis:** Buduje graf na podstawie podanych klauzul, gdzie wierzchołki to zmienne logiczne, a krawędzie reprezentują wystąpienia tych zmiennych w jednej klauzuli.
- **Parametry:** `clauses` - Lista klauzul.
- **Zwraca:** Graf reprezentujący formułę logiczną w sieci NetworkX.

```
def adding_to_graph(clauses):
    G = nx.Graph()
    for clause in clauses:
        for i in range(len(clause)):
            for j in range(i+1, len(clause)):
                G.add_edge(abs(clause[i]), abs(clause[j]))
    return G
```

2.2.3 `generate_node_embeddings(G)`

- **Opis:** Generuje osadzenia węzłów za pomocą algorytmu Node2Vec.
- **Parametry:** `G` - Graf w formie NetworkX.
- **Zwraca:** Model Node2Vec z osadzeniami węzłów.

```
def generate_node_embeddings(G):
    node2vec = Node2Vec(G, dimensions=64, walk_length=40, num_walks=40, workers
    =4)
    model = node2vec.fit(window=10, min_count=1, batch_words=4)
    return model
```

2.2.4 `operations(model, graph, num_clusters=3)`

- **Opis:** Generuje osadzenia węzłów za pomocą algorytmu Node2Vec.
- **Parametry:** `model` - Model Node2Vec, `graph` - Graf w formie NetworkX, `num_clusters` - Liczba grup (domyślnie 3)
- **Zwraca:** Parametry modeli .

```
def operations(model, graph, num_clusters=3):
    node_ids = list(graph.nodes)
    node_embeddings = [model.wv[str(node_id)] for node_id in node_ids]
    pca = PCA(n_components=2)
    embeddings_2d = pca.fit_transform(node_embeddings)
    kmeans = KMeans(n_clusters=num_clusters, random_state=42)
    cluster_labels = kmeans.fit_predict(embeddings_2d)

    return node_ids, embeddings_2d, cluster_labels
```

2.2.5 `visualize_cluster_node_embeddings(model, graph, num_clusters=3)`

- **Opis:** Dokonuje grupowania osadzeń węzłów przy użyciu algorytmu k-means i wizualizuje grupy. Przypisanie danego klastra do danej grupy jest tylko na podstawie odległości. Poszczególne klastry nie mają wypisanych cech poza sąsiedztwem.
- **Parametry:** `model` - Model Node2Vec, `graph` - Graf w formie NetworkX, `num_clusters` - Liczba grup (domyślnie 3).

```
def visualize_cluster_node_embeddings(model, graph, num_clusters=3):
    node_ids, embeddings_2d, cluster_labels = operations(model, graph,
    num_clusters)
    plt.figure(figsize=(10, 8))
    for i in range(num_clusters):
        cluster_points = embeddings_2d[cluster_labels == i]
        plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label=f'Cluster
     {i+1}')

    for i, txt in enumerate(node_ids):
        plt.annotate(txt, (embeddings_2d[i, 0], embeddings_2d[i, 1]), xytext
    =(2, 2), textcoords='offset points', fontsize=7)
    plt.legend()
    plt.show()
```

2.2.6 `visualize_interactive(model, graph, num_clusters=3)`

- **Opis:** Dokonuje grupowania osadzeń węzłów przy użyciu algorytmu k-means i wizualizuje grupy w sposób interaktywny w postaci serwera przy użyciu biblioteki dash.
- **Parametry:** `model` - Model Node2Vec, `graph` - Graf w formie NetworkX, `num_clusters` - Liczba grup (domyślnie 3).

```
def visualize_interactive(model, graph, num_clusters=3, selected = None):
    window_width = 1000
    window_height = 800
    node_ids, embeddings_2d, cluster_labels = operations(model, graph,
    num_clusters)

    df = pd.DataFrame({'X': embeddings_2d[:, 0], 'Y': embeddings_2d[:, 1], '
    Node': node_ids, 'Cluster': cluster_labels})

    app = dash.Dash(__name__)

```

```
app.layout = html.Div([
    dcc.Graph(id='scatter-plot'),
    dcc.Dropdown(
        id='cluster-dropdown',
        options=[{'label': f'Cluster {i}', 'value': i} for i in range(num_clusters)],
        value=None,
        placeholder="Select a cluster"
    )
], style={'width': f'{window_width}px', 'height': f'{window_height}px'})

@app.callback(
    Output('scatter-plot', 'figure'),
    [Input('cluster-dropdown', 'value')]
)
def update_scatter_plot(selected_cluster):
    fig = go.Figure()

    for i in range(num_clusters):
        visible_points = [True if label == i else False for label in df['Cluster']]
        if selected_cluster is None or i == selected_cluster:
            fig.add_trace(go.Scatter(
                x=df[visible_points]['X'],
                y=df[visible_points]['Y'],
                mode='markers+text',
                marker=dict(size=8),
                text=df[visible_points]['Node'],
                textposition='top right',
                name=f'Cluster {i + 1}'
            ))

    fig.update_layout(title=f'Node Embeddings Clustering (K-means, {num_clusters} clusters)',
                      xaxis=dict(title='Component 1'),
                      yaxis=dict(title='Component 2'))

    return fig

app.run_server(debug=True)
```

3 hypergraphs.py

Program odpowiada za generowanie i wizualizację hipergrafów na podstawie formuł logicznych.

3.1 Wprowadzenie do teorii hipergrafów

Hipergraf to struktura rozszerzająca koncepcję grafu poprzez umożliwienie istnienia krawędzi łączących więcej niż dwa wierzchołki. W przeciwieństwie do grafów, gdzie krawędzie łączą jedynie dwa wierzchołki, hipergrafy pozwalają na łączenie dowolnej liczby wierzchołków.

3.1.1 Podstawowe pojęcia

3.1.1.1 Hiperkrawędzie W hipergrafie, krawędzie są nazywane hiperkrawędziami. Hiperkrawędź to zbiór wierzchołków, które są ze sobą połączone. Dla przykładu, krawędź łącząca trzy wierzchołki to trójka.

3.1.1.2 Reprezentacja hipergrafu Podobnie jak w przypadku grafów, hipergrafy mogą być reprezentowane za pomocą macierzy sąsiedztwa, gdzie wiersze odpowiadają wierzchołkom, a kolumny hiperkrawędziom. Inna popularna reprezentacja to lista hiperkrawędzi, gdzie każda hiperkrawędź ma przypisaną listę wierzchołków, które łączy.

3.1.2 Hipergrafy a Grafy

Hipergrafy generalizują grafy, umożliwiając reprezentację bardziej złożonych relacji między wierzchołkami. W grafach, krawędzie są zawsze dwuelementowe, podczas gdy hipergrafy pozwalają na bardziej elastyczne relacje.

3.1.3 Przykłady zastosowań

Hipergrafy znajdują zastosowanie w wielu dziedzinach, takich jak analiza sieci społecznych, modelowanie relacji w bazach danych, czy reprezentacja danych w uczeniu maszynowym.

3.1.4 Opis Metody

Metoda rozpoczyna się od wczytania pliku DIMACS CNF za pomocą funkcji `read_dimacs_cnf`, która analizuje plik i zbiera informacje o liczbie zmiennych, klauzulach i samych klauzulach. Następnie, na podstawie każdej linijki z pliku, tworzone są połączenia pomiędzy zmiennymi danej podformuły. Te połączenia są następnie wizualizowane przy użyciu grafu, co umożliwia ich lepszą reprezentację w przestrzeni wielowymiarowej. Ostatecznie, hipergrafy są prezentowane w postaci statycznej i dynamicznej wizualizacji przy użyciu biblioteki `plotly`, `matplotlib` oraz `NetworkX`.

3.2 Implementacja

- **Wczytywanie pliku DIMACS CNF:** Funkcja `read_dimacs_cnf` wczytuje plik DIMACS CNF, analizuje jego zawartość i zwraca liczbę zmiennych oraz listę klauzul.
- **Generowanie hipergrafu:** Funkcja `generate_hypergraph` na podstawie wczytanych danych tworzy hipergraf, gdzie zmienne są węzłami, a połączenia między nimi reprezentują zależności pomiędzy zmiennymi w klauzulach.
- **Rysowanie hipergrafu:** Funkcja `draw_hypergraph` generuje statyczną wizualizację hipergrafu przy użyciu biblioteki `NetworkX` i `matplotlib`. Wizualizacja uwzględnia zależności między zmiennymi, a kolory węzłów reprezentują przynależność do klauzuli.
- **Rysowanie interaktywnego hipergrafu:** Funkcja `draw_interactive_hypergraph` tworzy interaktywną wizualizację hipergrafu przy użyciu biblioteki `plotly`. Każdy węzeł reprezentuje zmienną, a połączenia między nimi są dynamicznie prezentowane. Dodatkowo, wizualizacja uwzględnia stopień węzłów.
- **Aktualizacja progu (threshold):** Funkcja `update_threshold` reaguje na zmiany wartości progu, czyli minimalnej liczby połączeń wymaganej do utrzymania węzła. Aktualizuje statyczną i dynamiczną wizualizację, co umożliwia interaktywne eksplorowanie grafu.

Cały kod wykorzystuje biblioteki takie jak `NetworkX`, `matplotlib` oraz `plotly`, co pozwala na efektywne generowanie i prezentację hipergrafu w kontekście analizy formuł logicznych. Dodatkowo, korzysta z interaktywnych elementów interfejsu użytkownika, takich jak suwak (`FloatSlider`), umożliwiających dostosowanie widoczności hipergrafu w zależności od wartości progu.

3.2.1 `read_dimacs_cnf(filename)`

- **Opis:** Odczytuje plik DIMACS CNF i ekstrahuje liczbę zmiennych oraz klauzul.
- **Parametry:** `filename` - Ścieżka do pliku DIMACS CNF.
- **Zwraca:** Krotka zawierająca liczbę zmiennych i listę klauzul.

```
def read_dimacs_cnf(filename):
    clauses = []
    with open(filename, 'r') as file:
        for line in file:
            if line.startswith("c"):
                continue
            if line.startswith("p cnf"):
                num_vars, num_clauses = map(int, line.strip().split()[2:])
            else:
                clause = list(map(int, line.strip().split()[:-1]))
                clauses.append(clause)
    return num_vars, clauses
```

3.2.2 `generate_hypergraph(num_vars, clauses, threshold)`

- **Opis:** Generuje hipergraf na podstawie podanych klauzul, gdzie wierzchołki to zmienne logiczne, a hiperkrawędzie reprezentują klauzule zawierające co najmniej dwie zmienne.
- **Parametry:** `num_vars` - Liczba zmiennych, `clauses` - Lista klauzul, `threshold` - Próg stopnia wierzchołka (domyślnie 0).
- **Zwraca:** Hipergraf reprezentujący formułę logiczną w sieci NetworkX.

```
def generate_hypergraph(num_vars, clauses, threshold):
    G = nx.Graph()

    for var in range(1, num_vars + 1):
        G.add_node(var)
```

```
    for i, clause in enumerate(clauses):
        G.add_nodes_from(map(abs, clause))
        for pair in itertools.combinations(map(abs, clause), 2):
            G.add_edge(*pair, clause=i + 1)

    nodes_to_remove = [node for node, degree in dict(G.degree()).items() if degree <= threshold]
    G.remove_nodes_from(nodes_to_remove)

    return G
```

3.2.3 `draw_hypergraph(num_vars, clauses, threshold=0)`

- **Opis:** Wizualizuje hipergraf za pomocą Matplotlib.
- **Parametry:** `num_vars` - Liczba zmiennych, `clauses` - Lista klauzul, `threshold` - Próg stopnia wierzchołka (domyślnie 0).

```
def draw_hypergraph(num_vars, clauses, threshold=0):
    G = generate_hypergraph(num_vars, clauses, threshold)
    pos = nx.spring_layout(G, seed=42)

    node_colors = [0] * (num_vars + 1)
    for node in G.nodes():
        for i, clause in enumerate(clauses):
            if abs(node) in map(abs, clause):
                node_colors[node] = i + 1
                break

    colors = [node_colors[node] for node in G.nodes()]

    fig, ax = plt.subplots(figsize=(8, 8))
    nx.draw(G, pos, with_labels=True, font_size=10, node_color=colors, cmap=plt.cm.Blues, node_size=500, ax=ax)
    plt.show()
```

3.2.4 `draw_interactive_hypergraph(num_vars, clauses, threshold=0)`

- **Opis:** Tworzy interaktywną wizualizację hipergrafu za pomocą Plotly.
- **Parametry:** `num_vars` - Liczba zmiennych, `clauses` - Lista klauzul, `threshold` - Próg stopnia wierzchołka (domyślnie 0).

```
def draw_interactive_hypergraph(num_vars, clauses, threshold=0):
    G = generate_hypergraph(num_vars, clauses, threshold)
    pos = nx.spring_layout(G, seed=42)

    edge_trace = go.Scatter(
        x=[],
        y=[],
        line=dict(width=0.5, color='#888'),
        hoverinfo='none',
        mode='lines')

    for edge in G.edges():
        x0, y0 = pos[edge[0]]
        x1, y1 = pos[edge[1]]
        edge_trace['x'] += (x0, x1, None)
        edge_trace['y'] += (y0, y1, None)

    node_trace = go.Scatter(
```

```
        x=[],
        y=[],
        text=[],
        mode='markers+text',
        hoverinfo='text',
        marker=dict(
            colorscale='Blues',
            reversescale=False,
            color=[],
            size=10,
            colorbar=dict(
                thickness=15,
                title='Node Degree',
                xanchor='left',
                titleside='right'
            ),
            line=dict(width=2)))

    for node in G.nodes():
        x, y = pos[node]
        node_trace['x'] += (x,)
        node_trace['y'] += (y,)
        node_trace['text'] += ('Var: ' + str(node),)

    for node, adjacencies in enumerate(G.adjacency()):
        node_trace['marker']['color'] += (len(adjacencies[1]),)

    layout = go.Layout(
        titlefont=dict(size=16),
        showlegend=False,
        hovermode='closest',
        margin=dict(b=5, l=5, r=5, t=5),
        annotations=[dict(
            text="",
            showarrow=False,
            xref="paper", yref="paper")],
        xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),
        yaxis=dict(showgrid=False, zeroline=False, showticklabels=False))

    fig = go.Figure(data=[edge_trace, node_trace], layout=layout)
    fig.show()
```

3.2.5 `update_threshold(threshold_value)`

- **Opis:** Aktualizuje wizualizację hipergrafu na podstawie wartości suwaka progowego.
- **Parametry:** `threshold_value` - Nowa wartość progu.

```
def update_threshold(threshold_value):
    clear_output(wait=True)
    draw_hypergraph(num_vars, clauses, threshold=threshold_value)
    draw_interactive_hypergraph(num_vars, clauses, threshold=threshold_value)
```

4 CDCLSolver.py

Program ten zawiera Solver CDCL do rozwiązywania formuł logicznych.

4.1 Algorytm CDCL

Algorytm Conflict-Driven Clause Learning (CDCL) to zaawansowany algorytm rozwiązujący problem SAT (Boolean Satisfiability Problem). Poniżej przedstawiono główne kroki algorytmu CDCL.

Algorithm 1 CDCL Algorithm

Wejście: Formuła w postaci Koniunkcyjnej Normalnej (CNF)
Wyjście: "SAT" jeśli jest spełnialna, "UNSAT" jeśli jest niespełnialna
Zainicjuj przypisanie, stos decyzji i klauzule nauczone
while true **do**
    $conflict\_clause \leftarrow UnitPropagation()$
    if $conflict\_clause \neq None$ **then**
        if $Not\ ResolveConflict(conflict\_clause)$ **then**
            return "UNSAT"
        end if
    else if $AllVariablesAssigned()$ **then**
        return "SAT"
    else
        $var\_to\_assign \leftarrow ChooseVariable()$
        $AssignVariable(var\_to\_assign)$
    end if
end while

4.1.1 Podfunkcje

Algorithm 2 Unit Propagation

while true **do**
    $unit\_clause \leftarrow FindUnitClause()$
    if $unit\_clause \neq None$ **then**
        $PropagateUnitAssignment(unit\_clause)$
    else
        return *None*
    end if
end while

Algorithm 3 Find Unit Clause

for each *clause* in *clauses* **do**
    $unassigned\_literals \leftarrow UnassignedLiterals(clause)$
    if $Length(unassigned\_literals) = 1$ **then**
        return *clause*
    end if
end for
return *None*

Algorithm 4 Resolve Conflict

$conflict\_var \leftarrow Abs(conflict\_clause[0])$
$conflict\_level \leftarrow GetDecisionLevel(conflict\_var)$
$LearnedClauses.Append(conflict\_clause)$
while $Length(decision\_stack) > 0$ *and* $GetDecisionLevel(decision\_stack[-1]) > conflict\_level$ **do**
    $Backtrack()$
end while
return *True*

4.2 Implementacja

W implementacji klasy CDCLSolver rozwiązujemy problem SAT przy użyciu algorytmu CDCL (Conflict-Driven Clause Learning). Inicjalizujemy solver z daną liczbą zmiennych i klauzulami. Następnie iteracyjnie wykonujemy kroki rozwiązującego algorytmu CDCL.

- **Unit Propagation:** Wykonujemy unit propagation, czyli propagację wartości jednostkowych, dopóki jest to możliwe. Jeśli natrafimy na konflikt, przechodzimy do kroku rozwiązania konfliktu.
- **Konflikt:** W przypadku konfliktu, podejmujemy decyzję dotyczącą strategii rozwiązania konfliktu. Jeśli nie można go rozwiązać, zwracamy "UNSAT" (niezadowalający). W przeciwnym razie, przechodzimy do kroku powrotu (backtrackingu).
- **Backtracking:** Powracamy do poprzednich decyzji, cofając się do punktu, w którym konflikt został rozwiązany.
- **Rozwiązanie:** Jeśli wszystkie zmienne zostały przypisane, zwracamy "SAT" (zadowalający).

Metoda solve() wykonuje te kroki iteracyjnie aż do osiągnięcia rozwiązania lub stwierdzenia niespełnialności. W rezultacie, na końcu wydrukowany zostanie wynik działania solvera dla danej formuły CNF.

Ostatnia część kodu wczytuje plik DIMACS CNF, tworzy instancję solvera i wywołuje metodę solve(), a następnie drukuje wynik.

4.2.1 Klasa `CDCLSolver`

- **Opis:** Implementuje solver CDCL (Conflict-Driven Clause Learning) do rozwiązywania formuł logicznych.
- **Atrybuty:** `num_vars` - Liczba zmiennych, `clauses` - Lista klauzul, `assignment` - Obecne przypisanie zmiennych, `decision_stack` - Stos decyzji, `learned_clauses` - Lista klauzul nauczenia się.
- **Metody:**
  - `solve()`: Próbuje rozwiązać formułę logiczną za pomocą CDCL.
  - `unit_propagation()`: Implementuje krok propagacji jednostkowej.
  - `find_unit_clause()`: Znajduje klauzulę jednostkową podczas propagacji jednostkowej.
  - `propagate_unit_assignment(unit_clause)`: Propaguje przypisanie klauzuli jednostkowej.
  - `choose_variable()`: Wybiera nieprzypisaną zmienną.
  - `assign_variable(var)`: Przypisuje zmienną.
  - `resolve_conflict(conflict_clause)`: Rozwiązuje konflikt podczas CDCL.
  - `backtrack()`: Powraca do poprzedniego poziomu decyzji.
  - `get_decision_level(var)`: Pobiera poziom decyzji zmiennej.

```
class CDCLSolver:
    def __init__(self, num_vars, clauses):
        self.num_vars = num_vars
        self.clauses = clauses
        self.assignment = [None] * (num_vars + 1)
        self.decision_stack = []
```

```
        self.learned_clauses = []

    def solve(self):
        while True:
            conflict_clause = self.unit_propagation()
            if conflict_clause is not None:
                if not self.resolve_conflict(conflict_clause):
                    return "UNSAT"
            elif all(self.assignment[1:]):
                return "SAT"
            else:
                var_to_assign = self.choose_variable()
                self.assign_variable(var_to_assign)

    def unit_propagation(self):
        while True:
            unit_clause = self.find_unit_clause()
            if unit_clause is not None:
                self.propagate_unit_assignment(unit_clause)
            else:
                return None

    def find_unit_clause(self):
        for clause in self.clauses:
            unassigned_literals = [lit for lit in clause if self.assignment[abs
(lit)] is None]
            if len(unassigned_literals) == 1:
                return clause
        return None

    def propagate_unit_assignment(self, unit_clause):
        lit = [lit for lit in unit_clause if self.assignment[abs(lit)] is None
][0]
        self.assignment[abs(lit)] = lit
        self.decision_stack.append(lit)

    def choose_variable(self):
        for var in range(1, self.num_vars + 1):
            if self.assignment[var] is None:
                return var

    def assign_variable(self, var):
        self.assignment[var] = var
        self.decision_stack.append(var)

    def resolve_conflict(self, conflict_clause):
        if len(self.decision_stack) == 0:
            return False  # UNSAT

        conflict_var = abs(conflict_clause[0])
        conflict_level = self.get_decision_level(conflict_var)

        self.learned_clauses.append(conflict_clause)
        while len(self.decision_stack) > 0 and self.get_decision_level(self.
decision_stack[-1]) > conflict_level:
            self.backtrack()

        return True

    def backtrack(self):
        last_decision = self.decision_stack.pop()
        self.assignment[abs(last_decision)] = None
```

```
67      def get_decision_level(self, var):
68          return len([lit for lit in self.decision_stack if abs(lit) == var])
```

4.2.2 funkcja `read_dimacs_cnf(filename)`

- **Opis:** Odczytuje plik DIMACS CNF i ekstrahuje liczbę zmiennych oraz klauzul.
- **Parametry:** `filename` - Ścieżka do pliku DIMACS CNF.
- **Zwraca:** Krotka zawierająca liczbę zmiennych i listę klauzul.

```
1  def read_dimacs_cnf(filename):
2      clauses = []
3      with open(filename, 'r') as file:
4          for line in file:
5              if line.startswith("c"):
6                  continue
7              if line.startswith("p cnf"):
8                  num_vars, num_clauses = map(int, line.strip().split()[2:])
9              else:
10                 clause = list(map(int, line.strip().split()[:-1]))
11                 clauses.append(clause)
12     return num_vars, clauses
```

5 Podsumowanie

Wyniki raportu dla wykonania poszczególnych funkcjonalności:

5.1 Osadzenia grafowe

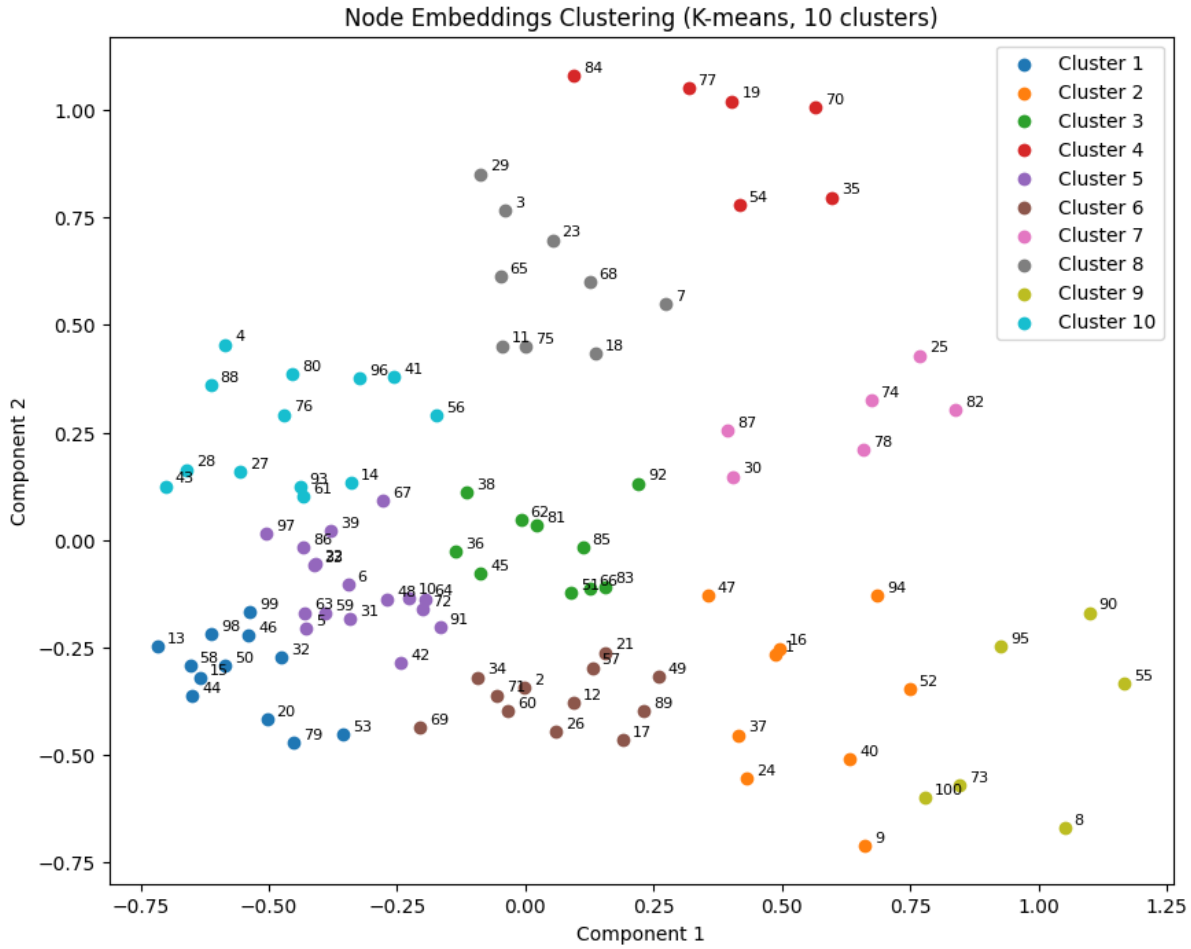

Figure 1: Grupowanie dla 100 zmiennych

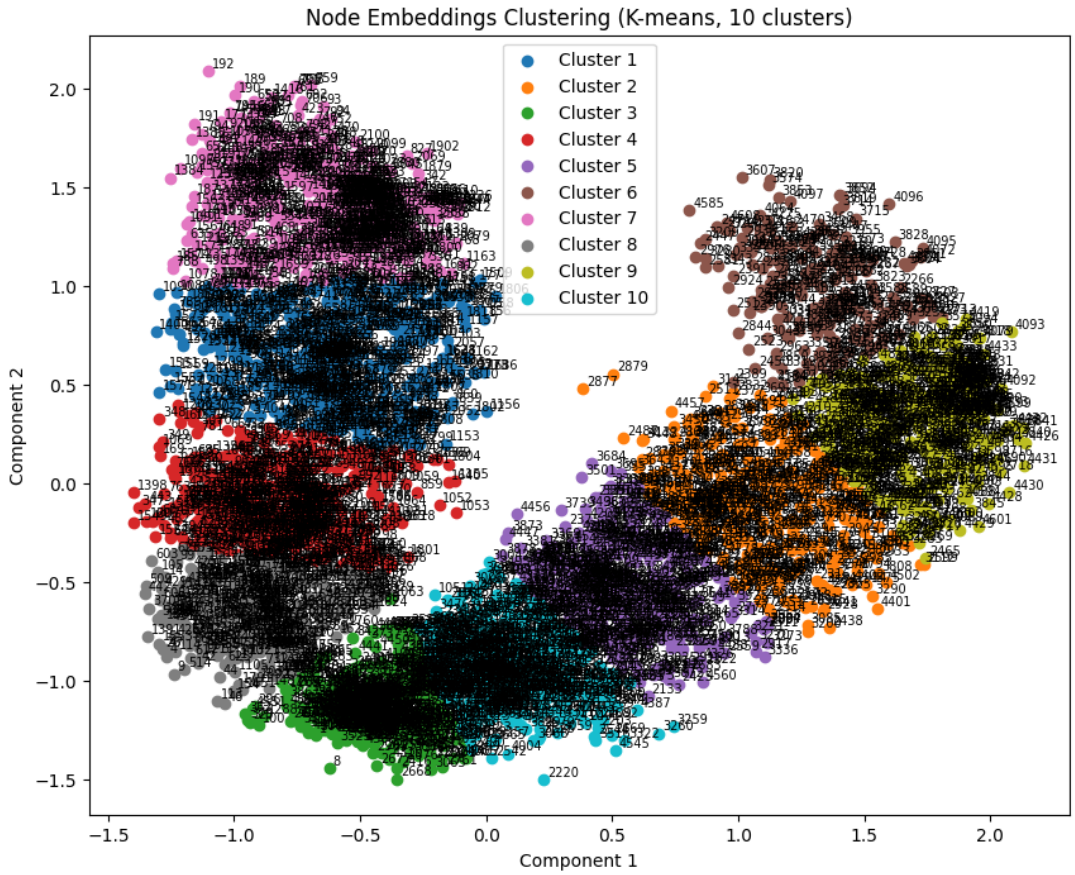

Figure 2: Grupowanie dla 4000 zmiennych

Figure 3: Jeden klaster z powyższego wykresu (inne kolory)

5.2 Hipergrafy

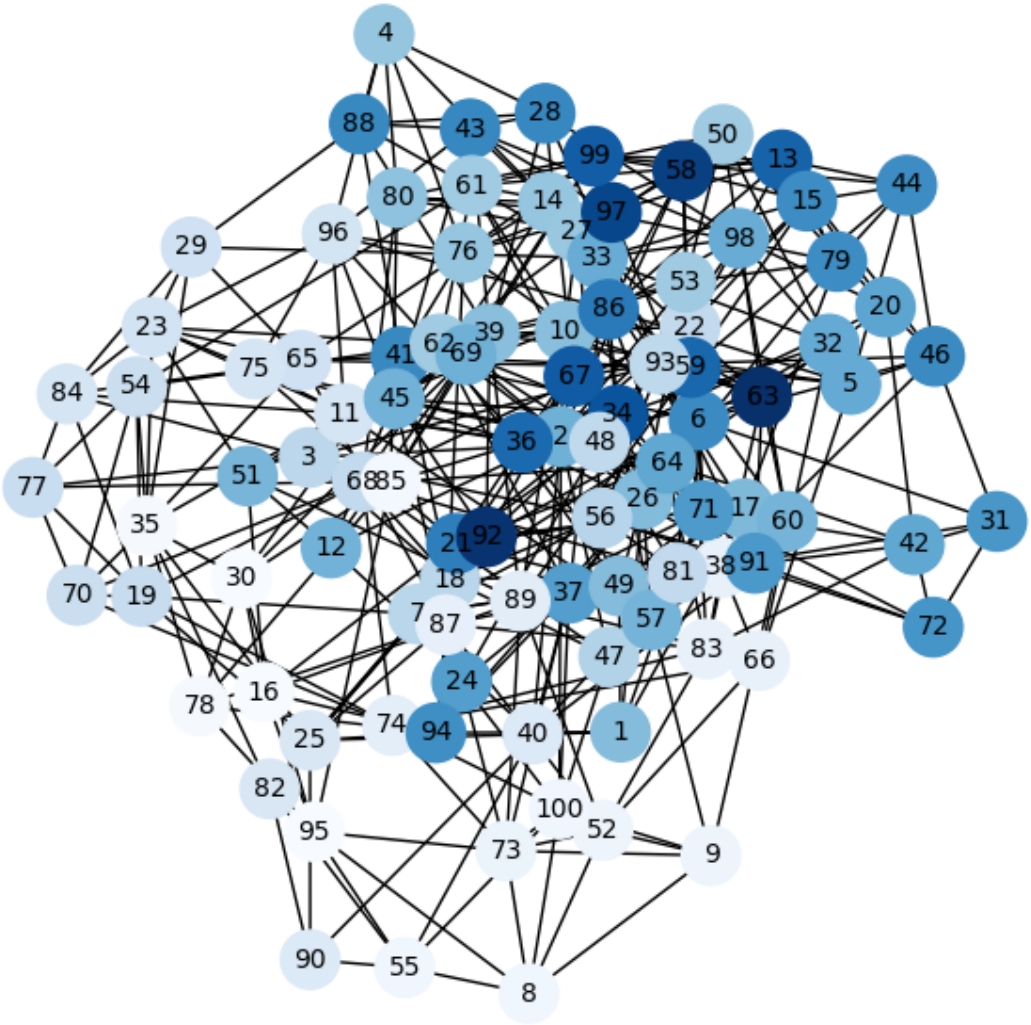

Figure 4: Hipergraf (100 zmiennych)

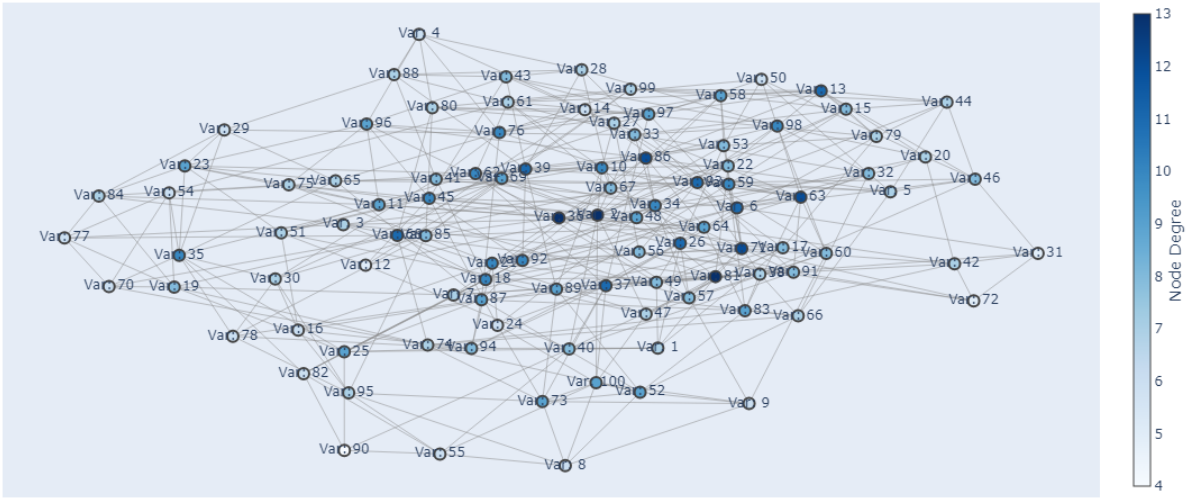

Figure 5: Hipergraf interaktywny (100 zmiennych)