Implementacja transformacji Tseitina w języku Python3

Daniel Rubak, Jakub Semczyszyn

Informatyka EAIiIB 2020

Spis treści

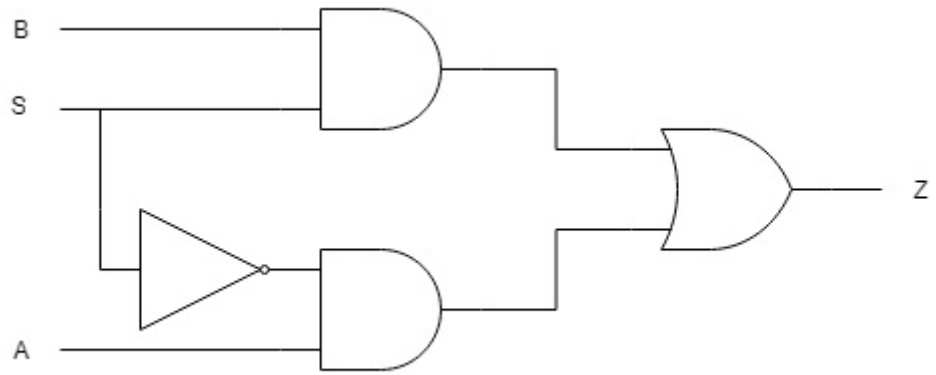
1 Transformacja Tseitina

Transformacja ta pozwala na sprowadzenie dowolnej formuły logicznej (lub też odpowiadającego jej układu kombinacyjnego) do postaci 3-CNF. Długość (ilość węzłów w drzewie składniowym) formuły po transformacji jest liniowo proporcjonalna do długości początkowej i ograniczona - zależnie od akceptowanych węzłów (np. czy XOR jest traktowany jako jeden węzeł czy złożenie kilku) i sposobu zapisu (m.in. formy reprezentacji negacji) różne źródła [1,2] podają wzrost 3-5 krotny. Ilość różnych przypisań zmiennych, które spełniają formułę wyjściową jest równa ilości wektorów wejściowych układu, dla których układ zwraca wartość 1.

Formuła wejściowa jest równospełnialna (*equisatisfiable*) z formułą wyjściową. Algorytm ma złożoność liniową, zależną od długości formuły wejściowej [3].

[2] Żeby przeprowadzić transformację należy przedstawić układ za pomocą formuły logicznej. Następnie dla każdej podformuły nie będącej atomową (jak również całej formuły) wprowadzić równoważną jej zmienną. Utworzyć koniunkcję tych równoważności oraz zmiennej reprezentującej całą formułę i ostatecznie każdy człon tej koniunkcji przetransformować na CNF używając klasycznych przekształceń logicznych, opartych na prawach De Morgana lub też odnosząc się do, utworzonych tym sposobem, gotowych wzorów (Rysunek 1).

Przykład



Powyższy układ możemy przedstawić przy pomocy formuły:

$$Z = (B \wedge S) \vee (A \wedge \neg(S))$$

Tworzymy równoważności:

$$X_1 \Leftrightarrow B \wedge S$$
$$X_2 \Leftrightarrow \neg S$$
$$X_3 \Leftrightarrow A \wedge X_2$$
$$X_4 \Leftrightarrow X_1 \vee X_3$$

CNF:

$$T(Z) = (X_1 \Leftrightarrow B \wedge S) \wedge (X_2 \Leftrightarrow \neg S) \wedge (X_3 \Leftrightarrow A \wedge X_2) \wedge (X_4 \Leftrightarrow X_1 \vee X_3) \wedge X_4$$

Transformacja równoważności na CNF:

$$(X_1 \Leftrightarrow B \wedge S) \equiv ((X_1 \Rightarrow B \wedge S) \wedge (B \wedge S \Rightarrow X_1))$$
$$\equiv ((\neg X_1 \vee (B \wedge S)) \wedge (\neg(B \wedge S) \vee X_1))$$
$$\equiv ((\neg X_1 \vee B) \wedge (\neg X_1 \vee S) \wedge (X_1 \vee \neg B \vee \neg S))$$

$$(X_2 \Leftrightarrow \neg S) \equiv ((X_2 \Rightarrow \neg S) \wedge (\neg S \Rightarrow X_2))$$
$$\equiv ((\neg X_2 \vee \neg S) \wedge (X_2 \vee S))$$

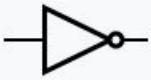
$$(X_3 \Leftrightarrow A \wedge X_2) \equiv ((X_3 \Rightarrow A \wedge X_2) \wedge (A \wedge X_2 \Rightarrow X_3))$$
$$\equiv ((\neg X_3 \vee (A \wedge X_2)) \wedge (\neg(A \wedge X_2) \vee X_3))$$
$$\equiv ((\neg X_3 \vee A) \wedge (\neg X_3 \vee X_2) \wedge (X_3 \vee \neg A \vee \neg X_2))$$

$$X_4 \Leftrightarrow X_1 \vee X_3 \equiv ((X_4 \Rightarrow X_1 \vee X_3) \wedge (X_1 \vee X_3 \Rightarrow X_4))$$
$$\equiv ((\neg X_4 \vee X_1 \vee X_3) \wedge ((\neg X_1 \wedge \neg X_3) \vee X_4))$$
$$\equiv ((\neg X_4 \vee X_1 \vee X_3) \wedge (X_4 \vee \neg X_1) \wedge (X_4 \vee \neg X_3))$$

Podstawienie do CNF:

$$T(Z) = ((\neg X_1 \vee B) \wedge (\neg X_1 \vee S) \wedge (X_1 \vee \neg B \vee \neg S)) \wedge ((\neg X_2 \vee \neg S) \wedge (X_2 \vee S)) \wedge$$
$$\wedge((\neg X_3 \vee A) \wedge (\neg X_3 \vee X_2) \wedge (X_3 \vee \neg A \vee \neg X_2)) \wedge ((\neg X_4 \vee X_1 \vee X_3) \wedge (X_4 \vee \neg X_1) \wedge (X_4 \vee \neg X_3)) \wedge X_4$$

Bramki logiczne jako podformuły w formie CNF

Type	Operation	CNF Sub-expression
 AND	$C = A \cdot B$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
 NAND	$C = \overline{A \cdot B}$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee C) \wedge (B \vee C)$
 OR	$C = A + B$	$(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$
 NOR	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$
 NOT	$C = \overline{A}$	$(\overline{A} \vee \overline{C}) \wedge (A \vee C)$
 XOR	$C = A \oplus B$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$
 XNOR	$C = \overline{A \oplus B}$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee B \vee \overline{C})$

Rysunek 1: Źródło: https://en.wikipedia.org/wiki/Tseytin_transformation

2 Opis funkcjonalności

Stworzony w ramach projektu program pozwala na przetransformowanie wprowadzonej do niego formuły do postaci CNF przy pomocy transformacji Tseitina oraz wygenerowanie wszystkich przypisań zmiennych spełniających tę formułę. Formułę można wprowadzić w formie stringa (również w pliku .txt) lub pliku DNF o strukturze zgodnej z formatem DIMACS CNF, dokładne wymagania określające formę zapisu operatorów i strukturę formuł opisane są w punkcie ”format wejściowy”. Transformacja i generowanie przypisań są osobnymi funkcjami, co pozwala również na skorzystanie, w miarę potrzeby, z tylko jednej z tych operacji. Funkcja odpowiedzialna za transformację pozwala na zapis przerobionej formuły do pliku oraz udostępnia dwie metody jej zapisu: czytelny dla człowieka oraz DIMACS CNF. Funkcja odpowiedzialna za generowanie przypisań wykorzystuje moduł PySAT. Moduł ten to API udostępniające warstwę pomiędzy kodem i danymi użytkownika, a SAT solverami. Umożliwia przekształcenia między różnymi formami zapisu formuł logicznych oraz prostą interakcję z solverami poprzez funkcje realizujące wyspecjalizowane funkcjonalności. Dostępne w nim SAT solvery (w liczbie 10), przyjmują na wejście między innymi format DIMACS CNF. Funkcja udostępniona w projekcie zwraca wszystkie możliwe przypisania zmiennych, dla których formuła wejściowa jest prawdziwa w formie słownika mapującego zmienną na wartość ze zbioru {0,1}, poprzez wywołanie analogicznej funkcji z PySAT (odpowiednio dostosowującej się do wybranego solvera) i przekonwertowanie jej wyników na pożądany format. Dodatkowo, możliwe jest włączenie funkcjonalności, która generuje raport czasów i statystyk transformacji oraz zapisuje go do pliku CSV (więcej na ten temat w punkcie ”Wydajność algorytmu”).

2.1 Opis użycia

```python
from bparser.tseitin_generator import TseitinFormula

def simple_tests():
    formulas = {
        '(a || b) && c || !(d && e)': 'string',
        '(!(p && (q || !r)))': 'string',
        '(a && b) || (a && !c)': 'string',
        '(a && b) or ((c || d) and e)': 'string',
        'src/data/simple_dnf_0.dnf': 'file',
        'src/data/formula.txt': 'file',
        'a and !a': 'string',
        '!x1 and x2 or x1 and !x2 or !x2 and x3': 'string',
        '!a and a': 'string'
    }

    for test_id, (formula_value, formula_format) in enumerate(formulas.items()):
        print("\n=============== TEST %d ===============\n" % (test_id))
        formula = TseitinFormula(
            formula=formula_value, formula_format=formula_format, export_to_cnf_file=True, debug=True,
            interrupt_time=4, return_all_assignments=True)
```

Rysunek 2: Przykładowe użycie programu

Celem użycia programu należy zaimportować funkcję **TseitinFormula** z pakietu *bparser*, modułu *tseitin_generator*. Jest to konstruktor, który wywołuje metody odpowiedzialne za przetwarzanie oraz konwersję podanych formuł na postać akceptowalną przez większość obecnych na rynku SAT Solverów. Formuły można podać w wybranej przez siebie notacji. Przykłady pokazano na rysunku powyżej, w kluczach słownika **formulas**.

Aby przekształcić dowolną formułę do postaci CNF niezbędne jest zapisanie jej do postaci akceptowalnej przez aplikację. W tym celu należy stworzyć obiekt *TseitinFormula*, którego konstruktor przyjmuje parametry:

- *formula* - formułę logiczną w formie zmiennej typu string lub ścieżki do pliku,

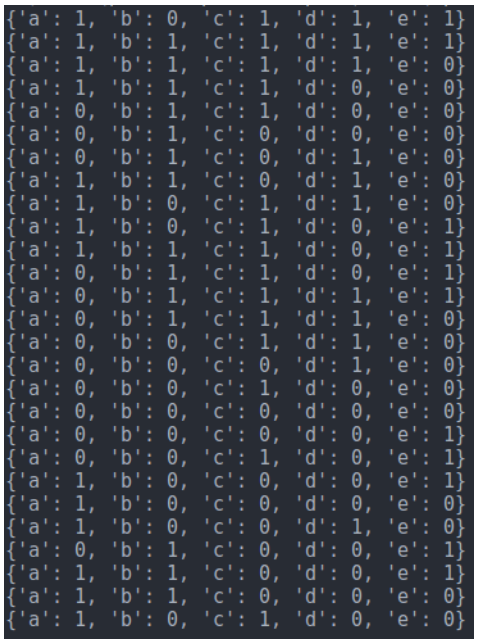
- *formula_format* - jeden z dwóch akceptowanych formatów formuły (string, file - odpowiednio dla formuły zapisanej przy pomocy zmiennej typu string w kodzie oraz takiej, która jest zapisana w pliku .txt lub .cnf), (domyślnie 'string').
- *export_to_cnf_file* - parametr typu bool, określający, czy formuła ma zostać zapisana do pliku w formacie CNF (domyślnie False).
- *debug* - parametr typu bool, określający, czy program ma wypisywać do konsoli, na jakim etapie działania się znajduje (domyślnie False).
- *use_solver* - parametr typu bool, określający, czy program ma zaaplikować formułę do solvera, czy jedynie ją przetransformować (domyślnie True).
- *solver_name* - parametr typu string, określający, jaki solver ma zostać użyty. Jest to skrótowa nazwa, którą można znaleźć w dokumentacji PySAT lub w pliku README projektu. Jeśli parametr *use_solver* jest ustawiony na False, to ten parametr jest ignorowany. (domyślnie 'm22' - MiniSat 2.2)
- *return_all_asignments* parametr typu bool - ustawienie go na True powoduje, że zwracane są wszystkie możliwe przypisania spełniające formułę, a na False, że zwrócone zostaje pierwsze znalezione przypisanie spełniające formułę. Jeśli parametr *use_solver* jest ustawiony na False, to ten parametr jest ignorowany. (domyślnie False).
- *use_timer* - parametr typu bool, określający, czy solver ma mierzyć czas poszukiwania przypisań (domyślnie True).
- *interrupt_time* - parametr typu int, określający, po ilu sekundach solver ma przerwać poszukiwanie rozwiązań; jeśli jest ustawiony na None, to przerwanie nie nastąpi - solver będzie szukał do skutku, niezależnie od czasu trwania (domyślnie None). UWAGA - ta funkcjonalność działa tylko z solverami typu MiniSat.

W przypadku gdy formuła jest zapisana w pliku, program wywoła odpowiednią konwersję, zamieniając jego treść na string.

Do pozyskania przypisań z klasy służy metoda *getTermsAssignment*, która zwraca je w formie tablicy słowników. Przykładowe przypisanie dla formuły

$$(a \vee b) \wedge c \vee \neg(d \wedge e)$$

pokazano na rysunku poniżej.



Rysunek 3: Przykładowe przypisanie wartości dla poszczególnych zmiennych logicznych

W celach diagnostycznych zaimplementowano również metodę *toString*, która przekształca drzewo semantyczne na gotową formułę logiczną uzyskaną po transformacji Tseitina. Dodatkowo, poszczególne klauzule wypisywane są w osobnych liniach w celu zwiększenia czytelności. Aby wyłączyć tą funkcję wystarczy ustawić parametr *split* na False. Przedstawiona wcześniej formuła po transformacji wygląda następująco:

```
(a or b or !phi0) and
(!a or phi0) and
(!b or phi0) and
(!phi0 or !c or phi1) and
(phi0 or !phi1) and
(c or !phi1) and
(!d or !e or !phi2) and
(d or phi2) and
(e or phi2) and
(phi1 or phi2 or !phi3) and
(!phi1 or phi3) and
(!phi2 or phi3) and
(phi3)
```

Rysunek 4: Przykładowa postać Tseitina

W tekście opisane są jedynie metody przeznaczone dla użytkownika. Diagram klas projektu oraz krótki opis wszystkich stworzonych funkcji znajdują się na końcu dokumentu. Diagram w czytelnej postaci (o większych wymiarach) jest również dołączony do plików projektu.

3 Format wejściowy

Program pobiera od użytkownika formułę w postaci napisu typu String. Użytkownik dostaje dowolność w wyborze zapisu formuły. Program przyjmuje następujące symbole:

- jako wartości logiczne - cyfry 1 i 0 oraz napisy ”False”, ”false”, ”True”, ”true”

- jako zmienne logiczne - dowolne* ciągi małych i wielkich liter oraz znaku ”_”, mogące być zakończone liczbą
- jako symbol równości - ”==”
- jako symbol nierówności - ”∼=” i ”!=”
- nawiasy ”(” i ”)”
- jako symbol koniunkcji - ”AND”, ”and”, ”&&”
- jako symbol dysjunkcji - ”OR”, ”or”, ”||”
- jako symbol negacji - ”!”, ”∼”, ”NOT”, ”not”

*Niedozwolone są nazwy zmiennych w formacie ”phiN”, gdzie N to liczba, ponieważ są one zarezerwowane dla zmiennych wyprowadzanych w trakcie transformacji. Oczywistym jest również, że słowo ”dowolne” w powyższym stwierdzeniu nie obejmuje ciągów opisanych w innych punktach tego wyliczenia.

Choć nie jest to z naukowego punktu widzenia zalecane, możliwe w programie jest mieszanie różnych zapisów.

Przykładowe formuły akceptowane przez program

– ’(a || b) && c || !(d && e)’,

– ’(!(p && (q || !r)))’,

– ’(a && b) || (a && ∼c)’,

– ’(a && b) or ((c || d) and e)’

– ’not (var_1 and (var_2 or var_3)) and not var_4 or var_5 == false’

3.1 Pliki DNF

Program udostępnia również możliwość konwersji pliku DNF, o składni analogicznej do formatu DIMACS CNF, do postaci formuły typu String akceptowanej przez *Tokenizer*. Służy do tego funkcja *getFormulaFromDIMAC*, będąca metodą klasy TseitinFormula i wywoływana w jej konstruktorze, która jako parametr przyjmuje ścieżkę do pliku, a zwraca formułę przekonwertowaną na String. Plik musi spełniać następujące zasady:

- Pierwsze linie służą do wpisywania komentarzy, muszą zaczynać się od litery ’c’. Zwyczajowo w pliku DIMACS pierwsza linia zawiera następnie spację i nazwę pliku, a ostatnia linia komentarza jest pusta (zawiera tylko literę ’c’), jednak nie jest to wymagane.
- Kolejna linia, po liniach komentarza, to linia opisująca problem. Składa się kolejno z litery ’p’, spacji, ciagu ”dnf”, spacji, liczby zmiennych, spacji i liczby klauzul.
- Następnie występują linie zawierające klauzule. Każda taka linia składa się z numerów zmiennych, które mogą być poprzedzone minusem i muszą być rozdzielone pojedynczymi spacjami. Na końcu, po kolejnej spacji, musi wystąpić 0, będące znakiem zakończenia linii. Interpretacja symboli wygląda następująco: poszczególne linie są połączone dysjunkcją (można więc stwierdzić, że 0 tłumaczone jest na $\vee$); spacje między numerami zmiennych interpretowane są jako koniunkcja (nie licząc spacji przed 0); liczba interpretowana jest jako zmienna o takim numerze, a minus przed liczbą jako zanegowanie tej zmiennej (czyli w praktyce jako słowo ” not ”).

Przykładowo akceptowany jest plik postaci:

```
c   simple_dnf.cnf
c
p dnf 3 2
1 -3 0
2 3 -1 0
```

Zostanie on zamieniony na String postaci: ” userdef1 and not userdef3 or userdef2 and userdef3 and not useredf1 ”.

4 Parsowanie formuł i budowa drzewa semantycznego

4.1 Skanowanie

Rolę skanera pełni klasa *Tokenizer*. Wykorzystuje ona obsługę wyrażeń regularnych udostępnioną przez moduł *re* do podzielenia formuły na tokeny i zapisania ich w formie listy. Pozwala również na sprawdzenie jakiego typu jest dany token (jaki symbol lub wyrażenie reprezentuje) oraz udostępnia podstawowe funkcje do obsługi listy tokenów.

Metoda odpowiedzialna za główny etap skanowania, *tokenize*, kompiluje najpierw wzorzec ze zbiorem wszystkich akceptowanych ciągów, za wyjątkiem nazw zmiennych i wartości logicznych (ponieważ te nie są ujednolicone), a następnie dzieli przy jego pomocy wyrażenie wejściowe, zapisując wynik do listy tokenów. W następnym kroku następuje scalenie znaków negacji ze znakiem równości, jeśli taki po nich występuje, jest to konieczne z uwagi na fakt, że znak negacji może być zarówno osobnym tokenem (typu NOT) jak i częścią tokenu nierówności (typu NEQ). Kolejnym etapem jest oczyszczenie tokenów z nadmiarowych białych znaków oraz usunięcie ewentualnych pustych tokenów. Ostatecznie następuje ujednolicenie zapisu wartości logicznych (do wartości typu bool) oraz budowana jest lista typów tokenów, której elementy odpowiadają elementom listy tokenów, co pozwala na przyspieszenie operacji w trakcie parsowania. Rozróżniane typy tokenów to:

- VAL - wartość logiczna
- VAR - zmienna logiczna
- EQ - token równości
- NEQ - token nierówności
- LP - token lewego nawiasu
- RP - token prawego nawiasu
- AND - token koniunkcji
- OR - token dysjunkcji
- NOT - token negacji

4.2 Parsowanie

Klasa *BooleanParser* przechodzi kolejno po wszystkich tokenach w liście Tokenizera i na ich podstawie buduje drzewo składniowe. Grupy tokenów klasyfikowane są jako następujące części:

- Expression (zanegowane lub nie) - składające się z dowolnej ilości AndTerm rozdzielonych tokenem dysjunkcji
- AndTerm - składający się z dowolnej ilości Condition rozdzielonych tokenem koniunkcji
- Condition - będące dwoma (zanegowanymi bądź nie) Terminalami rozdzielonymi tokenem równości bądź nierówności LUB pojedynczym (zanegowanym lub nie) Terminalem LUB (zanegowanym lub nie) Expression
- Terminal - będący zmienną lub wartością logiczną

Każda taka część jest parsowana sobie odpowiednią funkcją, wywołującą, zgodnie z powyższym opisem, funkcje obsługujące jej części składowe. Każda funkcja tworzy odpowiedni dla danej części węzeł w drzewie. Symbole negacji, jak i nawiasy, zapamiętywane są w odpowiednich węzłach, w liście, co umożliwia umieszczenie ich we właściwych miejscach przy powrotnym generowaniu formuły na podstawie drzewa (wiąże się to z tym, że symbole te są "ruchome" - znaczeniowo odnoszą się do korzenia wyrażenia, ale w zapisie stoją przed całym wyrażeniem, a nie korzeniem). Stos parsera został więc zaimplementowany pośrednio, poprzez stos wywołań odpowiednich metod.

Działanie poszczególnych metod:

Metoda parse : sprawdzenie, czy cała formuła jest zanegowana, jeśli tak by było to negacja zostałaby zapamiętana w liście "ruchomych" symboli korzenia drzewa; do korzenia zostaje przypisany wynik wywołania metody *parseExpression*.

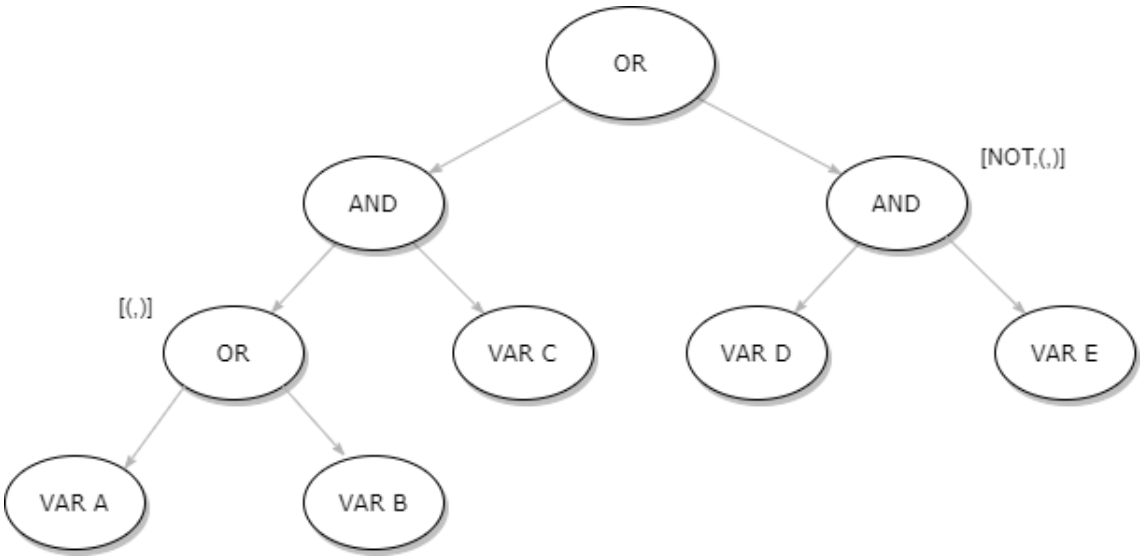
Metoda parseExpression : wywołanie metody *parseAndTerm* i przypisanie jej wyniku do jednej zmiennej; tak długo, jak po AndTerm występuje token dysjunkcji, wczytywany jest następny token, wywoływana metoda *parseAndTerm*, a jej wynik zapisywany w kolejnej zmiennej; tworzony jest węzeł z tokenem OR, którego lewym potomkiem jest pierwsza zmienna, a prawym druga i jest on następnie przypisywany do pierwszej zmiennej; na końcu zwracana jest pierwsza zmienna, w wyniku czego, jeśli był jedynie jeden AndTerm to zostanie zwrócone poddrzewo zawierające go jako korzeń, w przeciwnym wypadku zostanie zwrócone lewostronne drzewo złożone z poddrzew poszczególnych AndTermów, połączonych węzłami OR.

Metoda parseAndTerm : wywołanie metody *parseCondition* i przypisanie jej wyniku do jednej zmiennej; tak długo, jak po Condition występuje token koniunkcji, wczytywany jest następny token, wywoływana metoda *parseCondition*, a jej wynik zapisywany w kolejnej zmiennej; tworzony jest węzeł z tokenem AND, którego lewym potomkiem jest pierwsza zmienna, a prawym druga i jest on następnie przypisywany do pierwszej zmiennej; na końcu zwracana jest pierwsza zmienna, w wyniku czego, jeśli był jedynie jeden Condition to zostanie zwrócone poddrzewo zawierające go jako korzeń, w przeciwnym wypadku zostanie zwrócone lewostronne drzewo złożone z poddrzew poszczególnych Condition, połączonych węzłami AND.

Metoda parseCondition : sprawdzane jest, czy całe Condition jest zanegowane, jeśli tak, token negacji zostaje pominięty i zostaje ustawiona świadcząca o tym flaga; następnie sprawdzane jest, czy Condition rozpoczyna się otwierającym nawiasem (będzie to oznaczać, że, jeśli znaleziony zostanie również zamykający nawias, Condition musi zostać przeparsowane jako kolejne, zagnieżdżone Expression), jeśli tak, to wywoływana jest metoda *parseExpression*, a jej wynik zapisywany jest do zmiennej, sprawdzana jest flaga negacji i jeśli była ustawiona to wartość wyrażenia zostaje zanegowana, do listy "ruchomych" symboli wpisywany jest NOT token, a flaga zostaje wyzerowana, następnie do listy "ruchomych" symboli wpisywane są tokeny lewego i prawego nawiasu i sprawdzane jest, czy wyrażenie kończy się nawiasem zamykającym, jeśli tak to wynik parseExpression zostaje zwrócony, jeśli nie, to rzucony zostaje wyjątek; Jeśli Condition nie rozpoczyna się od nawiasu, to będzie zaczynało się od Terminala, więc wywołana zostanie funkcja *parseTerminal*, a jej wynik zostanie zapisany w pierwszej zmiennej; jeśli była ustawiona flaga negacji, to wartość Terminala zostaje zanegowana, a flaga wyzerowana; W kolejnym kroku sprawdzone zostanie z jaką sytuacją parser ma do czynienia - czy terminal stoi przed porównaniem (EQ lub NEQ), czy funkcjonuje samodzielnie (to znaczy jest przed OR, AND, nawiasem zamykającym lub na końcu całej formuły), czy jest jeszcze inaczej, co oznacza błąd. Jeśli terminal stoi przed porównaniem, to utworzony zostanie węzeł z odpowiednim tokenem (typu EQ lub NEQ), następnie na kolejnym tokenie zostanie wywołana metoda *parseTerminal* i wynik pierwszego parsowania zostanie przyłączony jako lewy potomek nowego węzła, a wynik tego parsowania (z analogiczną jak w przypadku pierwszego obsługą negacji) jako prawy potomek, po czym węzeł zostanie zwrócony. Jeśli terminal funkcjonuje samodzielnie to zwyczajnie zostanie zwrócony jego węzeł. Jeśli po pierwszym terminalu występuje kolejny token i nie należy on do zbioru {EQ, NEQ, OR, AND, RP} to rzucony zostaje wyjątek.

Metoda parseTerminal : Jeśli token jest zmienną to tworzony jest węzeł drzewa typu VAR, zawierający nazwę zmiennej, jeśli wartością logiczną to węzeł typu VAL, zawierający tę wartość. Tak utworzony węzeł zostaje zwrócony.

Przykładowo formuła $(A\ or\ B)\ and\ C\ or\ not\ (D\ and\ E)$ zostanie zamieniona w drzewo:



Proces będzie przebiegał następująco:

1. Lista tokenów będzie zawierać [”(”, ”A”, ”or”, ”B”, ”)”, ”and”, ”C”, ”or”, ”not”, ”(”, ”D”, ”and”, ”E”, ”)”], a lista typów tokenów [LP, VAR, OR, VAR, RP, AND, VAR, OR, NOT, LP, VAR, AND, VAR, RP].
2. Metoda *parse* nie wykryje negacji na początku i wpisze do korzenia drzewa wynik metody *parseExpression*.
3. W metodzie *parseExpression* zostanie wywołana metoda *parseAndTerm*.
4. W metodzie *parseAndTerm* zostanie wywołana metoda *parseCondition*.
5. W metodzie *parseCondition* zostanie rozpoznany token nawiasu otwierającego. Parser przejdzie do następnego tokenu. Celem przeparsowania wyrażenia w nawiasie, wywołane zostanie *parseExpression*.
6. W metodzie *parseExpression* zostanie wywołana metoda *parseAndTerm*.
7. W metodzie *parseAndTerm* zostanie wywołana metoda *parseCondition*.
8. W metodzie *parseCondition* nie zostanie rozpoznana negacja ani nawias, więc wywołana zostanie metoda *parseTerminal*.
9. W metodzie *parseTerminal* zostanie rozpoznany token zmiennej. Zostanie utworzony węzeł typu VAR, z wartością ”A” (zaznaczony na rysunku jako VAR A). Parser przejdzie do następnego tokenu. Węzeł zostanie zwrócony do metody z punktu 8.
10. Następny token zostanie rozpoznany jako OR, co znaczy, że terminal był samodzielny, więc zostanie zwrócony wyżej, do metody z punktu 7.
11. Jako że token to OR, a nie AND, to węzeł z terminalem ponownie zostanie zwrócony wyżej, do metody z punktu 6.
12. W metodzie *parseExpression* z punktu 6 zostanie przeparsowany kolejny token zmiennej, ciągiem wywołań analogicznym do punktów 7-11. Zostanie utworzony węzeł typu OR, jako jego lewy potomek przyłączony zostanie węzeł VAR A, a jako prawy VAR B. Jako, że kolejny token to nawias zamykający, a nie następny OR, to całe poddrzewo zostanie zwrócone do metody z punktu 5.

13. W metodzie *parseCondition* z punktu 5 zapamiętane w liście węzła OR zostaną nawias otwierający i zamykający, a następnie parser upewni się, że nawias zamykający faktycznie tam jest, przejdzie do kolejnego tokenu i zwróci poddrzewo do metody z punktu 4.

14. W metodzie *parseAndTerm* z punktu 4 rozpoznany zostanie token AND, więc zostanie wywołane *parseCondition*, analogicznie jak w punktach 7-10, w wyniku czego powstanie węzeł VAR C. Utworzony zostanie węzeł AND, jako jego lewy potomek przyłączony zostanie węzeł OR (wraz z całym poddrzewem), a jako prawy węzeł VAR C. Kolejny token to OR, a nie AND, więc powiększone poddrzewo zostanie zwrócone wyżej, do metody z punktu 3.

15. W metodzie *parseExpression* z punktu 3 rozpoznany zostanie token OR, więc zostanie wywołane *parseAndTerm*. W wyniku dalszych wywołań powstanie prawe poddrzewo: Węzeł AND z lewym potomkiem VAR D i prawym potomkiem VAR E. Nowością względem poprzednich punktów jest fakt, że przed całym AndTermem występowała negacja, a więc token NOT został zapamiętany w liście węzła AND razem z tokenami nawiasów. Utworzony zostanie węzeł OR. Dotychczasowe poddrzewo będzie jego lewym poddrzewem, nowe poddrzewo będzie jego prawym poddrzewem, a jako że wszystkie tokeny zostały przeparsowane, całość zostanie zwrócona do metody z punktu 2 (nowopowstały węzeł OR będzie korzeniem całego drzewa).

5 Zastosowanie transformacji Tseitina

Aby przekonwertować drzewo semantyczne na formułę CNF, napisano algorytm, wykorzystujący stos przejścia drzewa w kolejności postorder, który przetwarza poszczególne węzły drzewa na formuły tymczasowe a następnie, na ich podstawie, generuje docelową postać Tseitina. W niniejszej sekcji poszczególne kroki algorytmu zostaną opisane bazując na przykładzie z poprzedniego rozdziału.

Mając utworzone drzewo semantyczne, generacja formuł tymczasowych musi zostać rozpoczęta poczynając od węzłów położonych na końcu poszczególnych gałęzi. Każda z formuł tymczasowych składa się z czterech elementów:

- pierwszy terminal lub indeks klauzuli, której terminal wynikowy zostanie wykorzystany do dalszego parsowania
- rodzaj operandu
- drugi terminal lub indeks klauzuli, której terminal wynikowy zostanie wykorzystany do dalszego parsowania
- wartość logiczna definiująca, czy dana klauzula jest zanegowana czy nie, przyjmuje wartości True lub False

Dla drzewa semantycznego z poprzedniego rozdziału formuły tymczasowe wyglądałyby następująco (FT -skrót od *Formuła Tymczasowa*):

```
['a', 'OR', 'b', False]      (FT1)
[0, 'AND', 'c', False]       (FT2)
['d', 'AND', 'e', True]      (FT3)
[1, 'OR', 2, False]          (FT4)
```

Następnie, formuły tymczasowe mapowane są do nowych zmiennych logicznych, które wymagane są do dalszego przetwarzania. Dla rozpatrywanego przypadku mapa ta wygląda następująco:

```
{
    'phi0': {'first_term': 'a', 'second_term': 'b', 'operator': 'OR'},
    'phi1': {'first_term': 'phi0', 'second_term': 'c', 'operator': 'AND'},
    'phi2': {'first_term': 'd', 'second_term': 'e', 'operator': 'NAND'},
    'phi3': {'first_term': 'phi1', 'second_term': 'phi2', 'operator': 'OR'}
}
```

Bazując na powyższych listingach można zauważyć, że dla *FT1* została stworzona zmienna *phi0*. W związku z tym, iż jest to pierwsza klauzula (została zapisana do listy klauzul pod indeksem 0), zmienna ta została wykorzystana w *FT2* przy podstawieniu do pierwszego terminala. Zgodnie z powyższym, zmienna *phi1* składa się z iloczynu zmiennych *phi0* oraz *c*. Zgodnie z czwartym parametrem *FT3*, formuła ta powinna być zanegowana, stąd w mapie znalazł się operator *NAND* zamiast *AND*.

Ostatnim krokiem algorytmu jest wygenerowanie ostatecznej listy klauzul oraz termianali, które uzyskano przy użyciu transformaty Tseitina. Następuje to poprzez odwołanie do funkcji z modułu pomocniczego *tseitin_convertions*, w których wykorzystane zostały wzory przedstawione na Rysunku 1. Bazując na wyprowadzonych wcześniej formułach tymczasowych uzyskano następujące, finalne formuły:

```
[
    ['a', 'b', -1, 'phi0'],
    [-1, 'a', 'phi0'],
    [-1, 'b', 'phi0'],
    [-1, 'phi0', -1, 'c', 'phi1'],
    ['phi0', -1, 'phi1'],
    ['c', -1, 'phi1'],
    [-1, 'd', -1, 'e', -1, 'phi2'],
    ['d', 'phi2'],
    ['e', 'phi2'],
    ['phi1', 'phi2', -1, 'phi3'],
    [-1, 'phi1', 'phi3'],
    [-1, 'phi2', 'phi3'],
    ['phi3']
]
```

Powyższa struktura danych to wektor klauzul, które po zdekodowaniu przyjmują następującą postać:

```
[
    a or b or !phi0,
    !a or phi0,
    !b or phi0,
    !phi0 or !c or phi1,
    phi0 or !phi1,
    c or !phi1,
    !d or !e or !phi2
    d or phi2,
    e or phi2,
    phi1 or phi2 or !phi3,
    !phi1 or phi3,
    !phi2 or phi3,
    phi3
]
```

Lista terminali dla powyższej struktury:

```
['a', 'b', 'phi0', 'c', 'phi1', 'd', 'e', 'phi2', 'phi3']
```

6 Konwersja na format DIMACS

Format ten jest wykorzystywany przez większość dostępnych na rynku solverów. W związku z tym, program stworzony na potrzeby niniejszej pracy, może zostać wykorzystany jako generator plików DIMACS CNF, które następnie zostaną przekazane do dowolnie wybranego przez użytkownika solvera. Aby uzyskać wspomniany plik wy-

starczy ustawić parametr *export_to_file* na True w trakcie tworzenia obiektu *TseitinFormula*. Plik zostanie wygenerowany do folderu data w katalogu głównym projektu. Przykładowa struktura zapisana do pliku została przedstawiona poniżej, jest to struktura dla przykładu z poprzedniego rozdziału.

```
c   simple_cnf_0.cnf
c
p cnf 9 13
1 2 -3 0
-1 3 0
-2 3 0
-3 -4 5 0
3 -5 0
4 -5 0
-6 -7 -8 0
6 8 0
7 8 0
5 8 -9 0
-5 9 0
-8 9 0
9 0
```

Opis poszczególnych linii:

- linie zaczynające się od *c* to komentarze, ignorowane przez solver
- linia zaczynająca się od *p* to definicja formuły, pierwsza cyfra to liczba zmiennych logicznych (w tym przypadku 9), druga cyfra to liczba klauzul (w tym przypadku 13).
- pozostałe linie to definicje poszczególnych klauzul, gdzie liczby dodatnie oznaczają zmienną logiczną, natomiast liczby ujemne ich negację. Cyfra 0 oznacza koniec linii.

7 Aplikacja formuły do solvera i pobranie wyników

Wywołanie funkcji *solve* (następujące, gdy parametr *use_solver* w konstruktorze *TseitinFormula* był ustawiony na True) powoduje zaaplikowanie formuł w formacie DIMACS do konstruktora klasy Solver udostępnionej przez PySAT (dokumentacja: str. 32), który domyślnie działa na solverze Minisat 2.2, aby to zmienić, należy jako parametr *solver_name* podać skrót odpowiadający pożądanemu solverowi (lista skrótów dostępna jest w dokumentacji biblioteki oraz w pliku README.md). Następnie na utworzonym obiekcie Solver wywoływana jest metoda (również będąca częścią API PySAT) *enum_models()*, która zwraca wszystkie spełniające formułę przypisania w formacie składającym się z kolejnych liczb całkowitych, ujemnych lub dodatnich. Należałoby go odczytywać tak, że liczba oznacza numer zmiennej a znak minusa wartość False (analogicznie niezanegowana liczba oznacza, że zmienna ma wartość True), jednak nie jest to format czytelny dla użytkownika, toteż program konwertuje go na format, w którym poszczególne zmienne mają przypisaną wartość 0 lub 1. Tak przygotowany wynik zapisywany jest w liście, w obiekcie TseitinFormula i można go pobrać przy pomocy funkcji *getTermsAssignment*, która jako parametr przyjmuje wartość logiczną *only_original*, domyślnie ustawioną na True, która określa, czy zwrócona ma zostać wartość przypisań tylko dla zmiennych z oryginalnej, wejściowej formuły, czy również zmiennych podstawieniowych z transformacji Tseitina. Ta funkcja jest również wywoływana w ramach funkcji pobierających raport, a więc można te przypisania zapisać do pliku CSV.

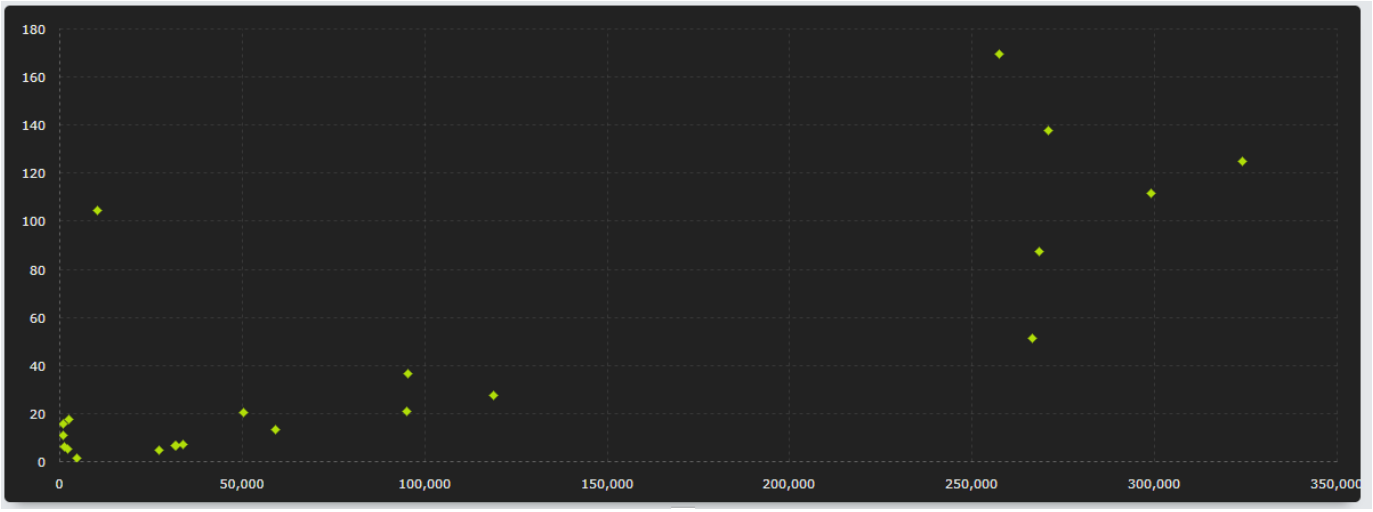
7.1 Diagnostyka - generowanie raportów

W klasie *TseitinFormula* dostępne są funkcje *getSolverReport* oraz *exportReport2CSV*, które użyte po zastosowaniu solvera, zwracają odpowiednio sformatowane statystyki: formułę wejściową, formułę po transformacji Tseitina,

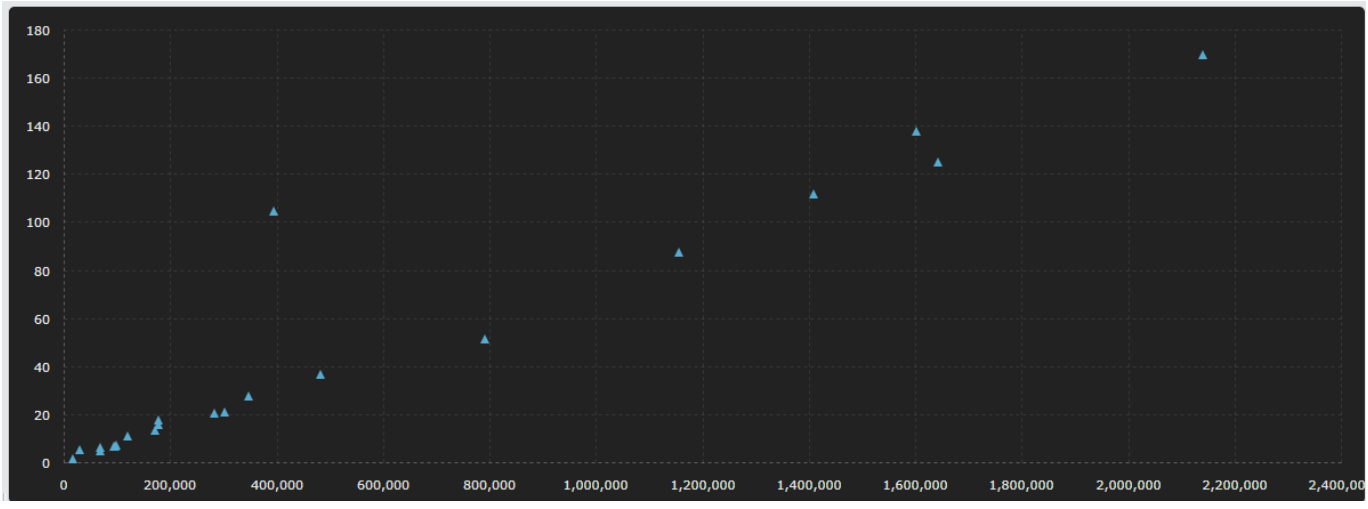
ilość termów przed i po transformacji, liczbę klauzul, czas wykonania oraz przypisania zmiennych spełniające formułę. Pierwsza z nich zwraca wynik w formie stringa, druga zapisuje go do pliku report.csv.

8 Wydajność algorytmu

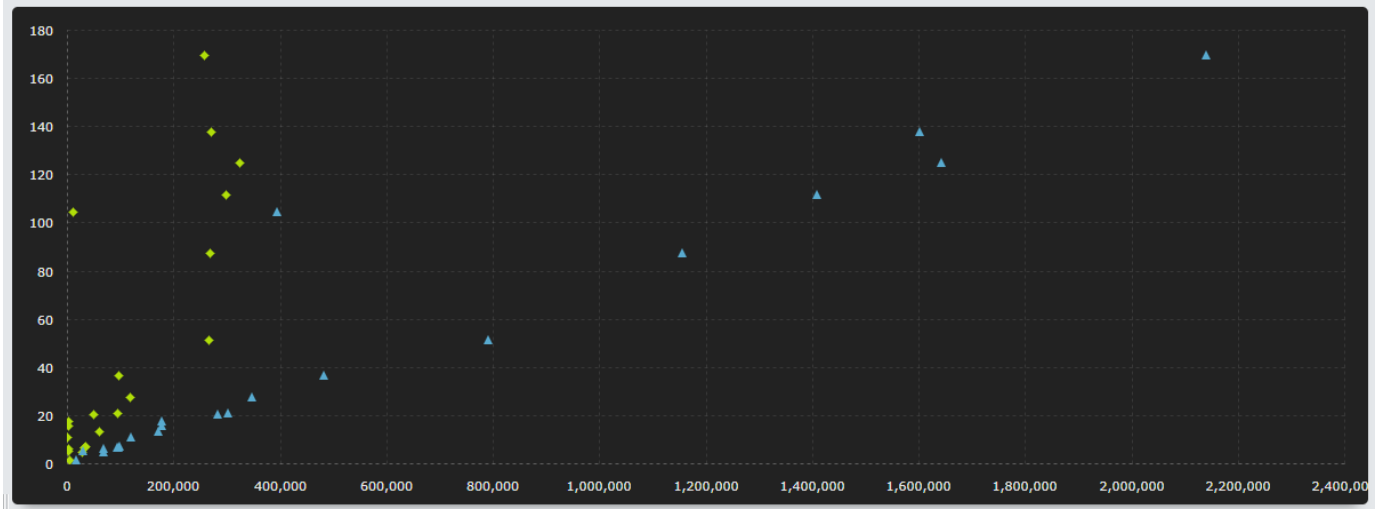
Program został poddany testom obciążeniowym, a ich wyniki, przeanalizowane pod kątem czasu wykonania. Największa ilość zmiennych w plikach testowych wynosiła ponad 324 tysiące, a największa ilość klauzul powyżej 2 milionów. Dokładne wyniki pomiarów są dostępne w pliku dołączonym do projektu na githubie. Pomiary te dokonane zostały na komputerze z dyskiem HDD, a maksymalny czas transformacji wyniósł w nich niecałe 3 minuty, próby na komputerze z dyskiem SSD w większości przypadków nie przekraczały 1 sekundy. Do analizy wyników mogą posłużyć poniższe wykresy:



Rysunek 5: Wykres zależności czasu transformacji w sekundach od ilości zmiennych



Rysunek 6: Wykres zależności czasu transformacji w sekundach od ilości klauzul



Rysunek 7: Złożenie wykresów

Na podstawie pierwszego wykresu można określić, że czas wykonania rośnie wraz ze wzrostem ilości zmiennych i wzrost ten ma trend liniowy, jednak odchylenie standardowe jest bardzo duże, co wskazuje, że trzeba w analizie uwzględnić inne czynniki. Takim czynnikiem może być ilość klauzul w formule, co pokazuje drugi wykres. Poza pojedynczym przypadkiem widać na nim liniowy wzrost czasu w zależności od ilości klauzul, jednak tego pojedynczego przypadku nie należy traktować tu jako błędu pomiarowego. Trzeci wykres, będący złożeniem dwóch poprzednich pokazuje, że ten szczególny przypadek miał relatywnie duży stosunek ilości klauzul do ilości zmiennych. Mała ilość zmiennych sprawia, że aby uzyskać dużą ilość unikatowych klauzul, klauzule te muszą być dłuższe, bardziej złożone, czyli mieć więcej węzłów. Ten fakt oraz liniowy wzrost czasu w zależności od ilości klauzul mogą potwierdzać, że algorytm, tak jak zakładał wstęp teoretyczny, działa ze złożonością liniową (czyli ma liniowy przyrost ilości węzłów względem ich ilości w formule wejściowej oraz na wszystkich zmiennych operacje wykonuje najwyżej z liniową złożonością). Dodatkowym argumentem potwierdzającym tę tezę jest niejednoznaczny przyrost czasu wykonania wraz ze wzrostem ilości zmiennych w większości przypadków - wskazuje to, że sam wzrost czasu może wynikać z korelacji ilości zmiennych z ilością klauzul (poza szczególnymi przypadkami, jak ten omówiony wcześniej) co również pokrywa się z faktem, że w ogólnych przypadkach większa liczba zmiennych nie oznacza znacząco większej ilości węzłów w wejściowej formule. Podsumowując: czas transformacji zależy liniowo od ilości węzłów w drzewie składniowym, co za tym idzie, jest zależny od ilości zmiennych wejściowej formuły w stopniu niewielkim, choć znaczącym oraz od ilości klauzul w tej formule w dużym stopniu.

Analizy czasowe pokazują dodatkowo, że program nie ma problemu z przetransformowaniem nawet największych klauzul, choć trzeba tu zaznaczyć, że z uwagi na rekurencyjną implementację parsera, dałoby się stworzyć taką formułę, która doprowadziłaby do przekroczenia maksymalnej głębokości rekurencji, wymagałoby to jednak celowego planowania i jest praktycznie niemożliwe do osiągnięcia przy pomocy formuły DNF, a jedynie przy pewnych szczególnych przypadkach formuł CNF, które nie są głównym targetem algorytmu. Można również zauważyć, że budowanie drzewa składniowego zajmuje około 40% czasu działania całego programu i jest to drugi najdłuższy czas, zaraz po samej transformacji. Wniosek płynący z tych dwóch obserwacji jest taki, że, choć obecnie program jest przystosowany do sprawnego wykonywania swoich głównych zadań, to w przyszłości można go usprawnić poprzez ulepszenie algorytmu parsowania formuły, na przykład zastępując rekurencyjne wywołania parserem ze stosem, zbliżonym do parserów LL.

W kwestii wynikowej formuły można zauważyć, że wzrost ilości klauzul jest w przybliżeniu liniowy, a drobne odchylenia od tej normy są zależne od wzrostu ilości zmiennych, co wynika z faktu, że nowe klauzule są generowane na podstawie podstawień nowych zmiennych. Z teorii wiadomo, że ilość nowych zmiennych jest równa ilości węzłów w oryginalnym drzewie składniowym, nie licząc liści.

9 Pozyskanie i instalacja programu

Do poprawnego działania programu wymagane są:

- System operacyjny Linux
- Interpreter Pythona, przynajmniej w wersji 3.8
- Zainstalowane pakiety Pythona: pip oraz venv

Kod źródłowy projektu można pobrać stąd. Następnie otworzyć terminal, przejść do katalogu, w którym znajduje się repozytorium i wywołać następujące polecenia:

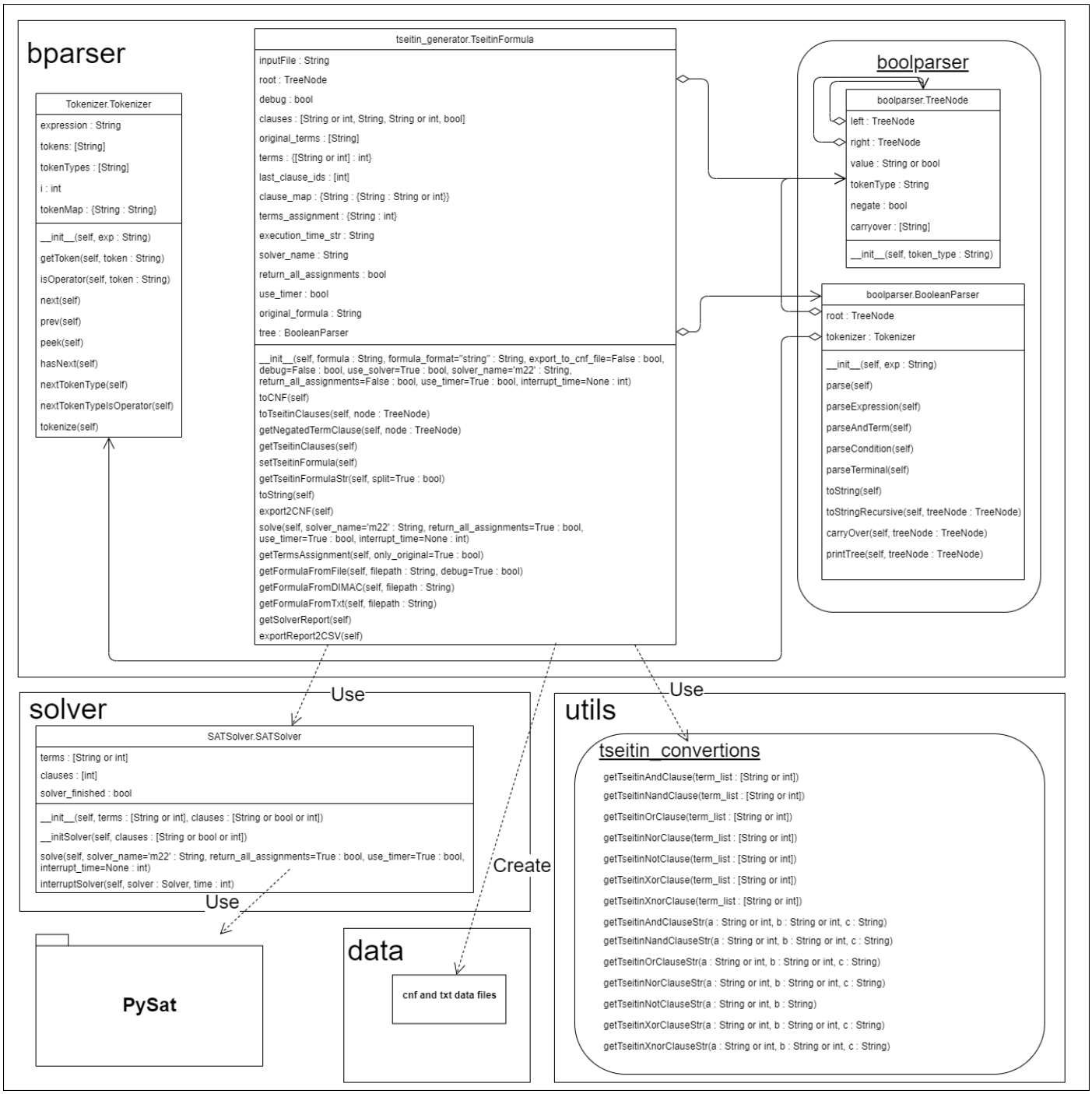
```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Powyższe komendy aktywują wirtualne środowisko, w którym zainstalowane zostaną wszystkie niezbędne pakiety potrzebne do uruchomienia projektu. Co więcej, użycie środowiska wirtualnego nie wpłynie negatywnie na działanie innych projektów gdyż wszystkie pakiety zostaną zainstalowane tylko w obrębie wspomnianego środowiska.

Aby sprawdzić poprawność konfiguracji można uruchomić skrypt zamieszczony w pliku *main.py*.

10 Spis wszystkich funkcji i pól

10.1 Diagram klas



10.2 Spis funkcji i pól

UWAGA - funkcje przeznaczone dla użytkownika zostały opisane w tekście. Celem tego spisu jest ułatwienie dostosowania programu do własnych potrzeb i ma on charakter pomocniczny.

10.2.1 Klasa Tokenizer

- *expression* - przechowuje formułę wejściową w postaci napisu
- *tokens* - lista przechowująca formułę podzieloną na tokeny
- *tokenTypes* - lista przechowująca typy poszczególnych tokenów; indeksami odpowiada liście *tokens*
- *i* - indeks służący do przeglądania list
- *tokenMap* - stały słownik mapujący tekst tokenu na typ tokenu

– *Tokenizer(exp)* - konstruktor klasy

  * *exp* - string zawierający formułę wejściową

– *getToken(token)* - zwraca typ tokenu odczytany z mapy

  * *token* - token, którego typ chce się odczytać

– *isOperator(token)* - zwraca wartość bool, określającą, czy dany token jest dopuszczalny i jest typu AND, OR lub NOT

  * *token* - token, któy chce się sprawdzić

– *next()* - zwraca token z listy *tokens* znajdujący się pod bierzącym indeksem *i* oraz inkrementuje *i*

– *prev()* - dekrementuje *i* oraz zwraca token z listy *tokens* znajdujący się pod bierzącym indeksem *i*

– *peek()* - zwraca token z listy *tokens* znajdujący się pod bierzącym indeksem *i*; jeśli indeks jest poza zakresem listy, zwraca napis ”formula end”

– *hasNext()* - zwraca wartość bool, określającą, czy indeks *i* nie wyszedł poza zakres listy *tokens*

– *nextTokenType()* - zwraca typ tokenu z listy *tokenTypes* znajdujący się pod bierzącym indeksem *i*

– *nextTokenTypeIsOperator()* - zwraca wartość bool, określającą, czy typ tokenu z listy *tokenTypes* znajdujący się pod bierzącym indeksem *i* to EQ lub NEQ

– *tokenize()* - zamienia formułę wejściową na listę tokenów i typów tokenów [dokładne działanie opisano w punkcie 4.1]

10.2.2 Klasa TreeNode

- *left* - lewy potomek węzła w drzewie
- *right* - prawy potomek węzła w drzewie
- *value* - wartość węzła; dotyczy zmiennych i wartości logicznych
- *tokenType* - typ tokenu, który reprezentuje węzeł
- *negate* - informacja o tym, czy wartość tokenu ma zostać zanegowana; dotyczy zmiennych
- *carryover* - lista ”ruchomych” tokenów oddziałujących na ten węzeł

– *TreeNode(token_type)* - konstruktor klasy

  * *token_type* - typ reprezentowanego przez węzeł tokenu

10.2.3 Klasa BooleanParser

- *root* - korzeń drzewa składniowego
- *tokenizer* - Tokenizer, który przetwarza formułę wejściową, czyli przechowuje informacje o jej tokenach

– *BooleanParser(exp)* - konstruktor klasy

  * *exp* - formuła wejściowa

– *parse()* - odpowiedzialna za rozpoczęcie parsowania; więcej informacji w sekcji 4.2

– *parseExpression()* - odpowiedzialna za parsowanie elementu typu Expression; więcej informacji w sekcji 4.2

– *parseAndTerm()* - odpowiedzialna parsowanie elementu typu AndTerm; więcej informacji w sekcji 4.2

– *parseCondition()* - odpowiedzialna za parsowanie elementu typu Condition; więcej informacji w sekcji 4.2

– *parseTerminal()* - odpowiedzialna za parsowanie elementu typu Terminal; więcej informacji w sekcji 4.2

– *toString()* - wywołuje funkcję *toStringRecursive* na korzeniu drzewa

– *toStringRecursive(treeNode)* - generuje string reprezentujący formułę na podstawie drzewa składniowego; w obecnej postaci służy głównie do diagnostyki parsera, jednak stanowi punkt wyjścia do dalszego jego rozwoju

  * *treeNode* - węzeł, od którego rozpocznie się przetwarzanie drzewa (zostanie on potraktowany jako korzeń)

– *carryOver(treeNode)* - przenosi ”ruchome” tokeny na potomków węzła; UWAGA w obecnej postaci nie usuwa tokenów z wyzszych węzłów, jedynie je propaguje, a więc wywołanie tej funkcji zmienia postać drzewa

  * *treeNode* - węzeł, na którym wykonywana jest operacja

– *printTree(treeNode)* - rekurencyjnie wypisuje informacje diagnostyczne o drzewie

  * *treeNode* - węzeł, od którego rozpocznie się przetwarzanie drzewa (zostanie on potraktowany jako korzeń)

10.2.4 Klasa TseitinFormula

- *inputFile* - nazwa pliku wejściowego
- *root* - korzeń drzewa składniowego przeparsowanej oryginalnej formuły
- *debug* - określa, czy do konsoli mają być wypisywane informacje o etapie działania programu
- *clauses* - zakodowane węzły drzewa w postaci [wartość lewego wyrażenia (lub numer z odniesieniem), rodzaj tokenu, wartość prawego wyrażenia (lub numer z odniesieniem), informacja o negacji]
- *original_terms* - lista termów w oryginalnej formule
- *terms* - słownik służący przypisaniu indeksów termom
- *last_clause_ids* - lista służąca do zapamiętywania, z jakiej klauzuli program przeszedł do przetwarzania kolejnej
- *clause_map* - słownik przypisujący opisowe etykiety poszczególnym elementom klauzul
- *terms_assignment* - służy do przechowywania wyników działania solvera
- *execution_time* - służy do przechowywania czasu działania solvera
- *solver_name* - skrótowa nazwa solvera

- *return_all_assignments* - określa, czy zwrócone mają zostać wszystkie przypisania, czy jedynie pierwsze znalezione
- *use_timer* - określa, czy solver ma mierzyć czas wykonania
- *original_formula* - przechowuje w formie stringa wejściową formułę
- *tree* - drzewo składniowe przechowywane jako obiekt klasy BooleanParser, która go wygenerowałav

– *TseitinFormula(formula, formula_format='string', export_to_cnf_file=False, debug=False, use_solver=True, solver_name='m22', return_all_assignments=False, use_timer=True, interrupt_time=None)* - konstruktor klasy

  * *formula* - formułę logiczną w formie zmiennej typu string lub ścieżki do pliku,
  * *formula_format* - jeden z dwóch akceptowanych formatów formuły (string, file - odpowiednio dla formuły zapisanej przy pomocy zmiennej typu string w kodzie oraz takiej, która jest zapisana w pliku .txt lub .cnf), (domyślnie 'string').
  * *export_to_cnf_file* - parametr typu bool, określający, czy formuła ma zostać zapisana do pliku w formacie CNF (domyślnie False).
  * *debug* - parametr typu bool, określający, czy program ma wypisywać do konsoli, na jakim etapie działania się znajduje (domyślnie False).
  * *use_solver* - parametr typu bool, określający, czy program ma zaaplikować formułę do solvera, czy jedynie ją przetransformować (domyślnie True).
  * *solver_name* - parametr typu string, określający, jaki solver ma zostać użyty. Jest to skrótowa nazwa, którą można znaleźć w dokumentacji PySAT lub w pliku README projektu. Jeśli parametr *use_solver* jest ustawiony na False, to ten parametr jest ignorowany. (domyślnie 'm22' - MiniSat 2.2)
  * *return_all_asignments* parametr typu bool - ustawienie go na True powoduje, że zwracane są wszystkie możliwe przypisania spełniające formułę, a na False, że zwrócone zostaje pierwsze znalezione przypisanie spełniające formułę. Jeśli parametr *use_solver* jest ustawiony na False, to ten parametr jest ignorowany. (domyślnie False).
  * *use_timer* - parametr typu bool, określający, czy solver ma mierzyć czas poszukiwania przypisań (domyślnie True).
  * *interrupt_time* - parametr typu int, określający, po ilu sekundach solver ma przerwać poszukiwanie rozwiązań; jeśli jest ustawiony na None, to przerwanie nie nastąpi - solver będzie szukał do skutku, niezależnie od czasu trwania (domyślnie None). UWAGA - ta funkcjonalność działa tylko z solverami typu MiniSat.

– *toCNF()* - przeprowadza transformację Tseitina poprzez wywołanie funkcji *toTseitinClauses, getTseitinClauses* i *setTseitinFormula*

– *toTseitinClauses(node)* - przechodzi drzewo w kolejności postorder przy pomocy stosu i buduje na jego podstawie klauzule odpowiadające poszczególnym węzłom, relacje między klauzulami opisuje przy pomocy indeksów będących odniesieniami, a więc klauzule są w formie gotowej do wprowadzenia zmiennych pomocniczych

– *getNegatedTermClause(node)* - generuje pośrednią klauzulę odpowiadającą negacji zmiennej

– *getTseitinClauses()* - wprowadza podstawienia zmiennych pomocniczych, zamienia zanegowane OR i AND na NOR i NAND, a następnie tak przetworzoną klauzulę dodaje do mapy klauzul

– *setTseitinFormula()* - dla każdej klauzuli w mapie wywołuje odpowiednią funkcję realizującą zamianę na CNF zgodnie z właściwym wzorem

– *getTseitinFormulaStr(split=True)* - zwraca przetransformowaną formułę w formie stringa

    * *split* - określa, czy poszczególne klauzule mają zaczynać się w nowych liniach

- *toString()* - wywołuje *getTseitinFormulaStr(split=False)*
- *export2CNF()* - zapisuje przetransformowaną formułę do pliku w formacie DIMACS CNF
- *solve(solver_name='m22', return_all_assignments)=True, use_timer=True, interrupt_time=None* - wywołuje metodę *solve* z klasy SATSolver, przekazując jej parametry
- *getTermsAssignment(only_original=True)* - zwraca przypisania zmiennych wygenerowane przez solver
    * *only_original* - deycyduje, czy mają zostać zwrócone przypisania jedynie dla wejściowych zmiennych, czy również dla zmiennych pomocniczych
- *getFormulaFromFile(filepath, debug=True)* - sprawdza rozszerzenie pliku wejściowego i na jego podstawie wywołuje odpowiednią funkcję
    * *filepath* - ścieżka do pliku
    * *debug* - określa, czy mają być wypisywane do konsoli informacje o stanie wczytywania pliku
- *getFormulaFromDIMAC(filepath)* - wczytuje plik w formacie DIMACS, traktując go jako DNF i zwraca formułę w postaci stringa
    * *filepath* - ścieżka do pliku
- *getFormulaFromTxt(filepath)* - wczytuje formułę w formie napisu z pliku txt i zwraca ją w postaci stringa
    * *filepath* - ścieżka do pliku
- *getSolverReport()* - zwraca raport z działania programu jako string
- *exportReport2CSV()* - zapisuje raport z działania programu do pliku CSV

10.2.5 Klasa SATSolver

- *terms* - lista termów przetransformowanej formuły
- *clauses* - lista klauzul w formie dostosowanej do solverów
- *solver_finished* - określa, czy solver zakończył poszukiwanie przypisań, czy zostało ono przerwane

- *SATSolver(terms, clauses)* - konstruktor klasy, przyjmuje termy i klauzule z przetransformowanej formuły, wywołuje *__initSolver*
- *__initSolver(clauses)* - koryguje zapis negacji w klauzulach, aby przystosować go do solvera
- *solve(solver_name='m22', return_all_assignments=True, use_timer=True, interrupt_time=None)* - uruchamia funkcję *enum_models* z API PySAT, jeśli podany jest czas przerwania, to przed poszukiwaniem rozwiązań, sprawdza, czy formuła jest spełnialna w rządanym czasie, przy pomocy funkcji *solve_limited* z PySAT; zwraca dane solvera
- *interruptSolver(solver, time)* - handler przerwania działania solvera

10.2.6 Plik tseitin_convertions

Wszystkie funkcje w tym pliku odpowiadają wzorom na przekształcenie klauzul do postaci CNF i występują w 2 wersjach: zwracającej wynik w formie stringa i zwracającej wynik w formie klauzuli zapisanej w liście