

**Rozwiązywanie równań i układów równań nad
skończonymi strukturami algebraicznymi z
wykorzystaniem SAT-solvera.**

Patryk Popek

Praca Licencjacka

Uniwersytet Marie Curie Skłodowskiej w Lublinie

Instytut Informatyki

2023

Spis treści

1	Opis Teoretyczny Problemu	3
1.1	Równania i ich rozwiązywalność	3
1.2	Sat i spełnialność	4
2	Redukcja problemu	5
2.1	Pliki .cnf	5
2.2	Tworzenie zmiennych	6
2.3	Klauzule użyte w redukcji	6
3	Aplikacja i algorytmy w niej zastosowane	8
3.1	Użyte technologie i interfejs	8
3.2	Klasy użyte w programie	9
3.3	Wczytanie algebry do programu	10
3.3.1	Pliki .ua	10
3.3.2	Algorytm w programie	11
3.4	Format i sprawdzenie poprawności równania	11
3.4.1	Wprowadzenie równań do programu	12
3.4.2	Algorytm sprawdzania poprawności wyrażenia w notacji polskiej . .	12
3.5	Krok pomocniczy: zbudowanie listy równań	13
3.6	Tworzenie redukcji	15
3.6.1	Wymuszenie pojedynczej wartości zmiennej	15
3.6.2	Rekurencyjne znajdowanie wartości	16
3.6.3	Porównywanie stron	16
3.6.4	Ogólny algorytm tworzenia formuły CNF	17
3.7	Uruchomienie Sat-Solvera	18
4	Testy aplikacji	19
4.1	Wczytanie algebry	19
4.2	Obsługa równań	20

Wstęp

Równanie matematyczne towarzyszą człowiekowi przez całe życie. W szkole dzieci uczą się rozwiązywać proste problemy za pomocą równań, poznając przy tym proste operacje matematyczne, potrzebne w późniejszym życiu. Na uniwersytetach zaś, wykształceni matematycy próbują udowadniać co raz to trudniejsze twierdzenia dotyczące równań nad coraz skomplikowanymi strukturami algebraicznymi, na przykład problem spełnialności równań (*POL-EQ*) [1] nad grupami skończonymi.

Logika jest działem matematyki, bez której nie istniałaby informatyka. Reprezentatywnym problemem pochodzącym z tej dziedziny jest Sat, należący do klasy problemów NP. Jest to na tyle popularny problem, że co roku organizowany jest konkurs [2] mający na celu wyłonienie najlepszego programu (*Sat-Solver*), dedykowanego do rozwiązywania wyżej wspomniany problem.

Celem tej pracy jest napisanie aplikacji, która rozwiązywałaby równanie matematyczne za pomocą Sat-Solvera, poprzez zredukowanie równania do problemu reprezentowanego poprzez *Sat*. Cała operacja wykonywana by była nad algebrą reprezentowaną w formacie *uaCalc* używanym przez teoretycznych algebraików.

1 Opis Teoretyczny Problemu

Poniżej przedstawione są pojęcia związane z pracą. Są one zgodne z książką Stanley’a Burrisa ”A Course in Universal Algebra” [3]

1.1 Równania i ich rozwiązywalność

Podstawa tej pracy, bez której nie mogłaby być ona wykonana, to algebra. **Strukturą algebraiczną** nazywamy dwójkę $\{A, B\}$ gdzie A jest dowolnym zbiorem, posiadającym dane stałe. Zbiór A nazywany jest dziedziną algebry. B jest zbiorem funkcji nad zbiorem A przekształcającym $\{A^k \rightarrow A\}$. Litera k w powyższym zapisie oznacza **Arność** i jest to liczba argumentów przyjmowana przez funkcję. Liczba elementów z zbiorze A to **liczność**. Format algebry *uaCalc*, który używany jest w tej pracy, dopuszcza tylko i wyłącznie liczby ze zbiory liczb naturalnych, także liczność podana w tym formacie jednoznacznie wskazuje na elementy zbioru tam się znajdujących. **Funkcja** jest operacją przekształcającą zbiór $\{X \rightarrow Y\}$. Innymi słowami, każdemu elementowi $x \in X$ przypisuje dokładnie jeden element $y \in Y$. Zbiór X nazywamy dziedziną funkcji, a zbiór Y przeciwdziedziną. Jeżeli oba zbiory są identycznie to przekształcenie następuje z $\{X \rightarrow X\}$. **Symbol funkcyjny** to znak reprezentujący funkcje w zapisie równania.

Wyrażenie powstałe poprzez złożenie elementów ze zbioru A i funkcji ze zbioru B to **wielomian**. Wyrażenie postaci $t_1 = t_2$, gdzie t_1, t_2 są dowolnymi wielomianami powstałymi nad daną algebrą to **równanie**. Równania dzielą się na dwa typy: **Równanie Sprzeczne** to równanie nieposiadające rozwiązania, czyli za zmienne równania nie da się podstawić żadnej wartości należącej do dziedziny algebry tak, aby po podstawieniu $t_1 = t_2$. **Równanie Spełnialne** to równanie, które posiada co najmniej jedno rozwiązanie, czyli istnieją takie wartości zmiennych, należące do dziedziny algebry, dla których po podstawieniu ich za niewiadome, wyrażenie $t_1 = t_2$ jest spełnione. Metoda zapisu równania, w której najpierw zapisywany jest symbol funkcyjny, a po nim argumenty funkcji nazywany jest **Notacją Polską**. Przykładowo zapis $/ + xy - ab$ jest równoważny zapisowi $(x + y)/(a + b)$. Notacja Polska charakteryzuje się bez nawiasową formą zapisu podanego wyrażenia

1.2 Sat i spełnialność

Aby móc zdefiniować rzeczy złożone takie jak *Sat*, to trzeba zdefiniować najpierw operację elementarne. **Zmienna zdaniowa** jest najmniejszym możliwym elementem logiki rachunków zdań. Przypisuje się jej wartość 0 (fałsz) lub 1 (prawda). **Negacja** jest to zaprzeczenie zmiennej zdaniowej. Oznaczana jest ona za pomocą znaku \neg . **Literał** jest to zmienna logiczna lub jej zaprzeczenie. Jest to podstawowy składnik służący do budowania coraz to bardziej skomplikowanych formuł logicznych. **Wartościowanie** to proces podstawienia wartości logicznej za każdy możliwy literał znajdujący się w formule. Jeżeli po powyższym procesie formuła logiczna zwraca wartość logiczną *Prawda* to znaczy że formuła jest **spełnialna**.

Podstawowymi funkcjami logicznymi, bez których definicji nie można zdefiniować rzeczy złożonych to **koniunkcja**, **alternatywa** i **implikacja**. **Koniunkcja** to funkcja logiczna postaci $x \wedge y$. Jest spełnialna tylko i wyłącznie wtedy, gdy oba literały mają wartość logiczną *prawda*. **Alternatywa** to funkcja logiczna postaci $x \vee y$. Jest nie-spełnialna tylko i wyłącznie wtedy, gdy oba literały mają wartość logiczną *fałsz*. **Implikacja** jest to funkcja logiczna postaci $x \rightarrow y$ gdzie x to poprzednik, a y to następnik implikacji. Funkcja jest nie-spełnialna wtedy i tylko wtedy, gdy poprzednik jest prawdziwy, a następnik fałszywy. **Klauzula** jest alternatywą literałów dowolnej długości. Jeżeli w klauzuli występuje literał oraz jego zaprzeczenie, to klauzula jest spełniona niezależnie od wartości pozostałych literałów. Klauzula posiadająca co najwyżej tylko jeden literał niezanegowany nazywana jest **klauzulą Horna**. Powstała ona z przekształcenia implikacji, w której poprzednikiem jest koniunkcja niezanegowanych literałów dowolnej długości, a następnikiem pojedyncza zmienna zdaniowa.

Sat: Jest to problem decyzyjny należący do klasy NP dotyczący formuł logicznych. Odpowiada on na pytanie, czy dana formuła logiczna jest spełnialna. Jeżeli jest, to zwraca przykładowe wartościowanie spełniające. Popularnym rodzajem formuł, dla których sprawdzany jest powyższy problem to **Koniunkcyjna postać normalna (CNF)**, czyli formuła logiczna będąca koniunkcją klauzul dowolnej długości.

2 Redukcja problemu

Głównym problemem związanym z moją pracą, był sposób przedstawienia równania matematycznego za pomocą formuły logicznej zdolnej do przetworzenia przez Sat-Solver. Finalna formuła, którą chciałem uzyskać, musiała być w postaci CNF.

2.1 Pliki .cnf

Wykonana przeze mnie redukcja nie tylko musiała być przedstawiona w odpowiednim typie formuły logicznej, ale też musiała mieć format akceptowany przez używany Sat-Solver. Takim formatem są pliki z rozszerzeniem .CNF [4] .

```
c 1=x_0 2=x_1 3=&0_0 4=&0_1
c 5=y_0 6=y_1 7=&1_0 8=&1_1
p cnf 8 16
1 2 0
-1 -2 0
3 4 0
-3 -4 0
-1 4 0
-2 3 0
5 6 0
-5 -6 0
7 8 0
-7 -8 0
-5 7 0
-6 8 0
-3 7 0
3 -7 0
-4 8 0
4 -8 0
```

Rysunek 1: Przykładowy plik reprezentujący redukcję CNF

Powyższy plik powstał poprzez redukcję wyrażenia $xor(1, x) = xor(y, 0)$. Pierwsze dwie linie, zaczynające się od znaku 'c' są komentarzami i nie są rozpatrywane przez program. W tym przypadku są one legendą informującą o indeksach użytych zmiennych w czasie redukcji. Linia trzecia, czyli nie licząc komentarzy pierwsza, informuje o strukturze całego pliku. 'p cnf' oznacza, że plik przedstawia formułę w postaci CNF. Kolejne dwie

liczby oznaczają odpowiednio ilość zmiennych oraz liczbę klauzul rozpatrywanej formuły. Dalsza część pliku jest zarezerwowana dla klauzul. '0' jest specjalnym symbolem oznaczającym koniec pojedynczej klauzuli, dlatego wszystkie zmienne indeksujemy od jedynki. Jeżeli indeks reprezentujący zmienną jest ujemny, to oznacza że ta zmienna w klauzuli jest zanegowana.

2.2 Tworzenie zmiennych

Na potrzeby redukcji musiałem stworzyć zmienne, na których podstawie mógłbym wyciągnąć wnioski po zakończeniu pracy Solvera. Zmienne zdaniowe mają postać x_n lub $W_{m,n}$, gdzie $n \in \{0, 1, \dots, C - 1\}$ i $m \in \{0, 1, \dots, k - 1\}$ gdzie k jest liczbą pojedynczych funkcji użytych w całym równaniu. x_n reprezentuje dowolną zmienną równania. Literały tego typu tworzone są dla każdej pojedynczej zmiennej zadeklarowanej w równaniu. $W_{m,n}$ jest symbolem wynikowym dla pojedynczej funkcji. Wartość true zmiennej x_n oznacza, że dla wartości liczbowej n zmiennej x równanie jest spełnione. Dlatego musimy wymusić, aby tylko jedna zmienna z zakresu x_0 do x_n mogła mieć wartość true, ponieważ pojedyncza zmienna nie może przyjmować więcej niż jednej wartości jednocześnie.

2.3 Klauzule użyte w redukcji

1. Klauzula zapewniająca co najmniej jedną zmienną:

Pierwszym krokiem jest zapewnienie, aby pojedyncza zmienna na pewno miała jakąkolwiek wartość. Efekt ten uzyskujemy poprzez klauzule: $(x_0 \vee x_1 \vee \dots \vee x_{C-1})$ gdzie x to dowolna niewiadoma równania. Aby powyższa formuła była spełniona, co najmniej jedna zmienna musi być prawdziwa.

2. Klauzula zapewniająca co najwyżej jedną zmienną:

Punkt pierwszy nie zapewniał limitu górnego aktywnym zmiennych. W tym celu wprowadzamy dodatkowe klauzule: $(\neg x_a \vee \neg x_b)$ gdzie $a \neq b$ oraz $a, b \in \{0, 1, \dots, C - 1\}$. Przez co, jeżeli dwie zmienne będą miały wartość true to klauzula będzie nie spełniona czyli też cały Sat. Dlatego wymusza to pojedynczą wartość true dla całego zbioru zmiennych.

3. Główna część redukcji:

Posiadając odpowiednią liczbę zmiennych możemy przejść do głównej części redukcji. Polega ona na szukaniu wartości działania w tablicy wynikowej operatora wczytanej wraz z algebrą.

x/y	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Tabela 1: Tabela reprezentująca dodawanie %4

Dla równania reprezentowanego poprzez powyższą tabelę przy przykładowych zmiennych $x = 3 \wedge y = 2$ nasza klauzula będzie wyglądała następująco: $(\neg x_3 \vee \neg y_2 \vee \&_{0,0})$. Jest to klauzula horna powstała z następującej implikacji: $((x_3 \wedge y_2) \rightarrow \&_{0,0})$. Możemy tak zrobić, ponieważ znamy całą tabelę, więc wiemy które wartości zmiennych dają konkretne wyniki. Postać ogólna to : $(\neg x_a \vee \dots \vee \neg y_b \vee \&_{m,n})$ gdzie $a, b, n \in \{0, 1, \dots, C-1\}$ i $m \in \{0, 1, \dots, k-1\}$. W powyższej formule ... oznacza wszystkie zmienne rozpatrywanej funkcji. Jeżeli równanie posiada stałe, to nie zamieszczamy ich w klauzuli. Klauzule są tworzone rekurencyjnie. Ilość klauzul stworzonych na podstawie jednej operacji to d^C gdzie d to ilość niewiadomych w funkcji. Jeżeli funkcja ma same stałe to wtedy wylicza się jej wartość i ma ona postać $\&_{x,y}$ gdzie x to indeks zmiennej a y to wyliczona wartość

4. Porównanie stron

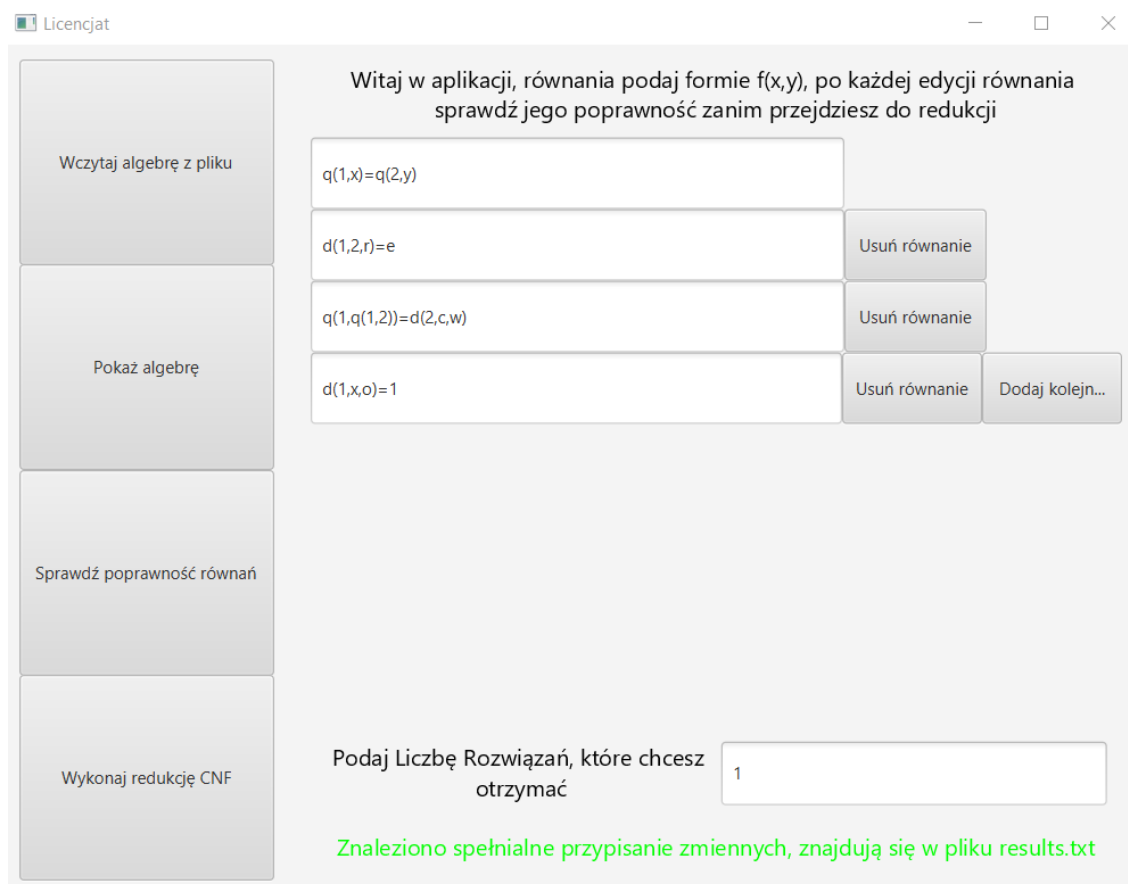
Równanie dzielimy na dwie części. Osobno rozpatrujemy lewą i prawą część. Dlatego jako ostatni krok musimy rozpatrzyć równowartość obu tych stron. Klauzula będzie miała postać $((\&_{a,n} \vee \neg \&_{b,n}) \wedge (\neg \&_{a,n} \vee \&_{b,n}))$ gdzie a, b są indeksami najbardziej zewnętrznych funkcji obu stron równań i $n \in \{0, 1, \dots, C-1\}$. Klauzule te zastosowujemy również w przypadku, gdy strona równania składa się tylko z pojedynczej niewiadomej. Wtedy przyrównujemy tą zmienną do symbolu wynikowego tej strony równania

Jeżeli dowolna niewiadoma występuje w układzie więcej niż raz, to traktujemy ją jako tą samą zmienną. Wymuszenie pojedynczej wartości true przeprowadzamy na niej tylko raz. Redukcja ta działa identycznie zarówno dla zwykłego równania jak i dla układu. Dla układów występuje więcej klauzul typu czwartego

3 Aplikacja i algorytmy w niej zastosowane

Aplikacja, którą napisałem na potrzeby tej pracy podzielona została na kilka kroków. Etapy te zostały ułożone liniowo i każdy kolejny można wykonać dopiero po ukończeniu wszystkich wcześniejszych. Wszystkie etapy są reprezentowane w interfejsie aplikacji za pomocą przycisków.

3.1 Użyte technologie i interfejs



Rysunek 2: interfejs Aplikacji

Do napisania aplikacji zdecydowałem się użyć języka Java w wersji 11 [5], javafx w wersji 17.0.2 [6] do napisania interfejsu oraz biblioteki sat4f w wersji 2.3.0 [7] używającej mi sat-solver, którym rozwiązuje powstały CNF. Wybrałem te konkretne technologie z dwóch powodów:

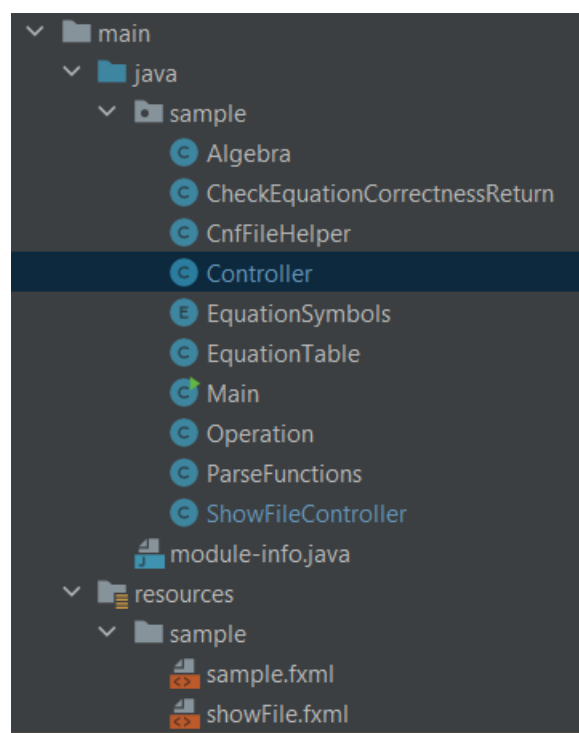
W javiefx bardzo łatwo napisać interfejs aplikacji, dzięki czemu miałem więcej czasu na skupieniu się na logice programu.

Java posiada bibliotekę DocumentBuilder [8], która służy parsowaniu plików xml, czyli dokładnie tego formatu, w którym zapisywana jest algebra.

Interfejs składa się z kilku przycisków, wiadomości powitalnej, miejsca do wprowadzenia równań, liczby interesujących nas wyników oraz miejsca dla wiadomości zwrotnej z programu. Każdy przycisk odpowiada za konkretną funkcję aplikacji, które zostaną opisane niżej

3.2 Klasy użyte w programie

Poniżej przedstawie główne klasy użyte w aplikacji:



Rysunek 3: interfejs Aplikacji

1. **Algebra:** Przechowuje informacje o algebrze. Posiada cardinality algebry, listę operacji oraz informację, czy została już wczytana.
2. **Operation:** Przechowuje informacje o pojedynczej operacji algebry. Posiada swoją nazwę, arność oraz listę wynikową dla każdej kombinacji zmiennych.
3. **CnfFileHelper:** Na Podstawie tej klasy tworzony jest plik wynikowy .cnf, posiada 3 listy, które zostały opisane w dalszej części programu.

4. **Controller** Jest to klasa powiązana bezpośrednio z interfejsem, obsługuje zadeklarowane przyciski.
5. **EquationTable**: Przechowuje informacje o pojedynczej funkcji w równaniu, klasa ta została opisana w dalszej części pracy.
6. **ParseFunctions**: Znajdują się tu wszystkie funkcje związane z redukcją równania do formuły logicznej CNF.

3.3 Wczytanie algebry do programu

3.3.1 Pliki .ua

```
<?xml version='1.0' encoding='utf-8'?>
<algebra>
  <basicAlgebra>
    <algName>P4</algName>
    <cardinality>4</cardinality>
    <operations>
      <op>
        <opSymbol>
          <opName>d</opName>
          <arity>2</arity>
        </opSymbol>
        <opTable>
          <intArray>
            <row r="[0]">0,1,2,3</row>
            <row r="[1]">1,0,3,2</row>
            <row r="[2]">2,3,0,1</row>
            <row r="[3]">3,2,1,0</row>
          </intArray>
        </opTable>
      </op>
      <op>
        <opSymbol>
          <opName>m</opName>
          <arity>2</arity>
        </opSymbol>
        <opTable>
          <intArray>
            <row r="[0]">0,0,0,0</row>
            <row r="[1]">0,1,2,3</row>
            <row r="[2]">0,2,0,2</row>
            <row r="[3]">0,3,2,1</row>
          </intArray>
        </opTable>
      </op>
    </operations>
  </basicAlgebra>
</algebra>
```

Rysunek 4: Przykładowy plik reprezentujący algebrę uniwersalną

Algebry, które są wykorzystywane w mojej aplikacji są reprezentowane poprzez pliki z rozszerzeniem "ua". Skrót ten pochodzi od wyrażenia "universal algebra". Są to pliki w formacie XML.

Powyższe zdjęcie przedstawia wygląd pliku opisującego algebrę. Sekcjami interesującymi nas najbardziej w tym pliku są *cardinality* oraz *operations*. *Cardinality* informuje o dziedzinie całej struktury. Jest ona niezmienna i ograniczona do zakresu $x \in \{0, 1, \dots, \text{cardinality} - 1\} \wedge x \in \mathbb{N}$

Operations opisuje wszystkie dostępne operacje na strukturze. Sekcja ta jest podzielona na podsekcje op poświęcone poszczególnym działaniom. *OpName* informuje o nazwie działania, jest to identyfikator, po którym będzie można znaleźć działanie w programie. *Arity* określa arność funkcji reprezentowanej przez to działanie. Najważniejszą częścią opisu działania jest *opTable*. Jest to struktura przedstawiająca wyniki działania opisywanej funkcji na wszystkich możliwych wariacjach z powtórzeniami o długości arity dziedziny algebry. Tabele te są główną osią, dzięki której powstaje finalna formuła CNF używana w Sat-Solverze. Indeks zmiennej w tabeli jest wartością w systemie dziesiętnym liczby zapisanej w systemie liczbowym o podłódze *cardinality* o długości arity. Własność ta jest wykorzystywana w czasie redukcji. Algebra pokazana stroną wyżej posiada dwa działania: Działanie *d*, które jest dwuargumentowym dodawaniem modulo 4 oraz działanie *m*, które jest dwuargumentowym mnożeniem modulo 4.

3.3.2 Algorytm w programie

1. Wczytaj plik do programu.
2. Zapisz w programie nazwę oraz dziedzinę algebry.
3. Pobierz listę operacji.
4. Przejdź do kolejnej niezapisanej operacji.
5. Zapisz arność i nazwę operacji.
6. Zapisz tablicę wartości funkcji.
7. Jeżeli pozostała jakaś niezapisana operacja wróć do punktu 4.

3.4 Format i sprawdzenie poprawności równania

Równania, na których pracuje moja aplikacja są przyjmowane w formie funkcyjnej czyli $f_{(x,y)}$. Dzięki temu łatwo można zamienić powyższy zapis na notację polską, dla której

znany algorytm sprawdzania poprawności podanego wyrażenia, dla określonej dziedziny. Zamiana ta polega na usunięciu nawiasów i przecinków z podanego stringa. Równanie dzielone jest po znaku równości, przez co poprawność jest osobnie rozpatrywana dla lewej oraz prawej strony. Poprawność jest sprawdzana tylko względem poprawności nazw zmiennych oraz arności operacji. Nie sprawdzana jest poprawność nawiasowania. Jest to spowodowane faktem, iż dla uproszczenia dzieła string reprezentujący równanie metodą split po znakach "(", ";", ")", ";", ","

3.4.1 Wprowadzenie równań do programu

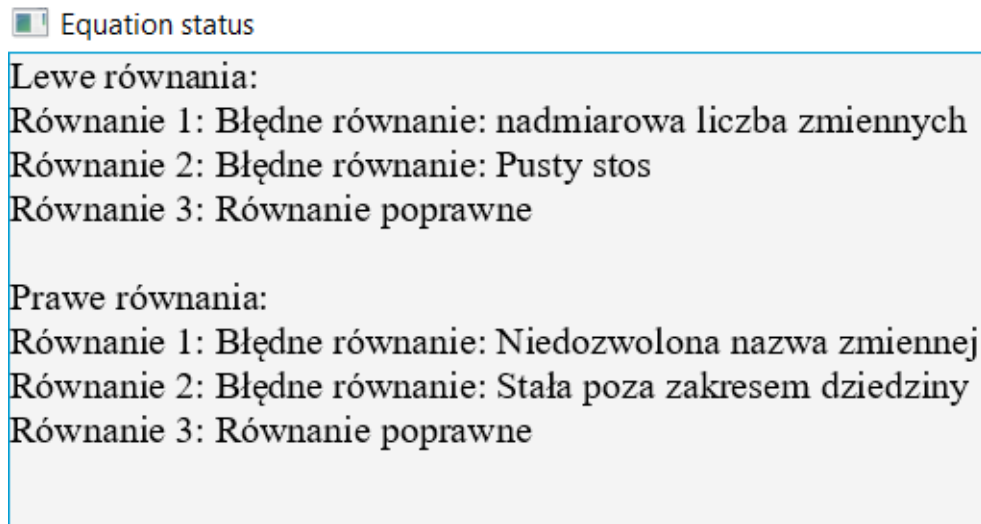
Aby móc wprowadzać równania do programu zadeklarowałem na interfejsie strukturę VBox. Znajdują się w niej kolejno ułożone Hboxy, a w nich pole do wprowadzenia równania oraz przycisk usuwający pole oraz dodający pole. Odnosiniki do tych pól przechowuje w listach, do których się odwołuje, gdy trzeba dodać, usunąć lub odczytać wartość. Poniżej tej struktury znajduje się miejsce do podania ilości interesujących użytkownika rozwiązań. Sat-Solver domyślnie zwraca jedno wartościowanie, które jest przykładowym. Można jednak wymusić na solverze więcej rozwiązań poprzez dopisanie znalezionej wartościowania w formie zanegowanej na koniec pliku i uruchomienie programu ponownie. Dlatego możemy dostać więcej niż jedno rozwiązanie. Jeżeli użytkownik poda więcej rozwiązań do znalezienia niż fizycznie istnieje, solver znajdzie wszystkie rozwiązania i skończy działanie.

3.4.2 Algorytm sprawdzania poprawności wyrażenia w notacji polskiej

Przebieg algorytmu:

1. Przejdź do ostatniego znaku równania.
2. Pobierz kolejny znak równania, jeżeli jest dostępny.
3. Sprawdź, czy rozpatrywany znak jest nazwą dozwolonej operacji.
4. Jeżeli nie, sprawdź czy jest dozwoloną stałą lub zmienną.
5. Jeżeli nie przejdź do kroku 11.
6. jeżeli znak został określony jako operacja ściągnij ze stosu ilość znaków równej arności operacji i dodaj na stos znak wynikowy.

7. Jeżeli na stosie zabrakło symboli lub nastąpił ich nadmiar przejdź do kroku 11.
8. Jeżeli znak został określony jako stała lub zmienna dodaj go do stosu.
9. Jeżeli pozostały nierozpatrzone znaki przejdź do kroku 2
10. Jeżeli nie ma już nierozpatrzonych znaków równania, zakończ działanie i zwróć prawdę
11. Zwróć fałsz



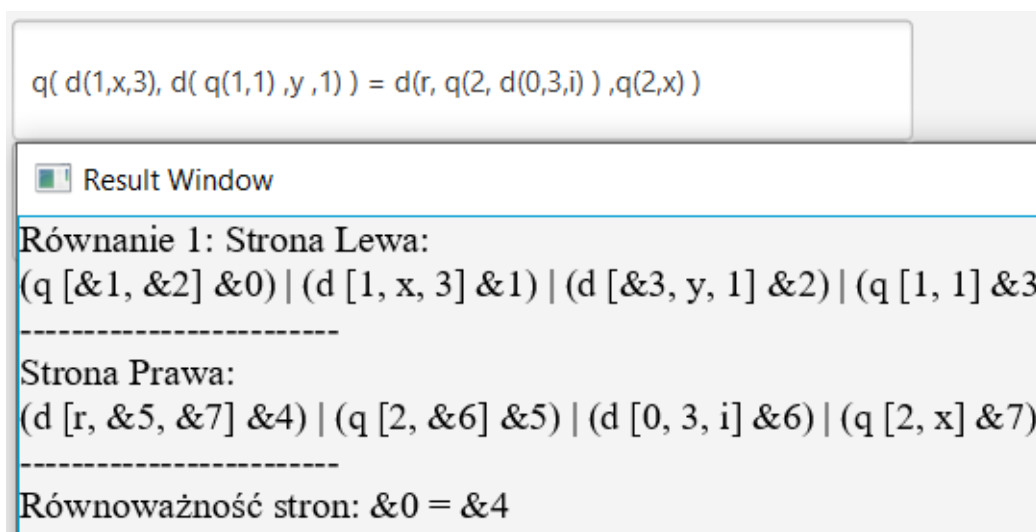
Rysunek 5: Przykładowe wyjście algorytmu

Algorytm sprawdzający zwraca 5 różnych wynikowych komunikatów dotyczących poprawności równania względem wczytanej algebry. Pierwszy komunikat od góry występuje, gdy w momencie napotkania operacji ilość elementów na stosie przekracza jej arność. Mamy wtedy za dużo argumentów do obsłużenia dlatego równanie jest niepoprawne. Komunikat drugi występuje w sytuacji odwrotnej do poprzedniej, czyli na stosie mamy niewystarczającą ilość elementów do zastosowania napotkanej operacji. Komunikat trzeci dotyczy sytuacji gdy zmienna którą napotkamy jest niezgodna z dziedziną. Spowodowane jest to faktem, iż dopuszczamy możliwość nazwy operacji, która jest dłuższa niż jeden znak. Ostatni komunikat sprawdza poprawność podanych stałych do programu. Zapewniamy, aby na pewno znajdowały się w dziedzinie algebry.

3.5 Krok pomocniczy: zbudowanie listy równań

Aplikacja dopuszcza możliwość, w której działanie algebry jest argumentem innego działania. Dzięki temu istnieje możliwość stworzenia nieskończonego drzewa zagnieżdżeń

funkcji. Z perspektywy programu zapis $f(1, x)$ oraz $f(1, f(1, x))$ to ten sam zapis gdyż funkcja najbardziej zewnętrzna ma te same argumenty: stałą 1 na pierwszym miejscu i niewiadomą na drugim. Chcielibyśmy jednak móc rozróżnić powyższe równania. Biorąc pod uwagę powyższy problem stworzyłem dodatkową strukturę o nazwie EquationTable reprezentowaną poprzez trójkę (opName, variables, result), gdzie opName oznacza nazwę operacji, variables jest listą argumentów operacji, a result to symbol reprezentujący wynik działania. Każdemu równaniu tworzę dedykowaną listę z powyższymi elementami. W ten sposób uzyskujemy dostęp do zagnieżdżonych operacji poprzez ich symbol wynikowy. Symbol ten traktowany jest jako zwykła zmienna przez funkcję wyżej



Rysunek 6: Przykładowe wyjście algorytmu

Przebieg algorytmu:

1. Sprawdź, czy równanie składa się z pojedynczego znaku.
2. Stwórz specjalną iterację EquationTable dla pojedynczej zmiennej.
3. Weź kolejny symbol rozpatrywanego równania.
4. Sprawdź, czy jest operatorem.
5. Jeżeli jest, to stwórz nowy obiekt EquationTable w liście równania.
6. Jeżeli jest to funkcja zagnieżdżona, znajdź pierwszy element EquationTable, dla którego liczba znalezionych zmiennych jest mniejsza od arności operacji i dodaj do listy symbol wynikowy reprezentujący wynik znalezionej operacji, po czym zwiększ indeks.

7. jeżeli symbol jest zmienną, znajdź pierwszy element EquationTable, dla którego liczba znalezionych zmiennych jest mniejsza od arności operacji i dodaj do listy tą zmienną.
8. Jeżeli zostały jeszcze symbole wróć do punktu 3

3.6 Tworzenie redukcji

Cała redukcja opiera się na liście EquationTable stworzonej podczas poprzedniego kroku i polega ona na odpowiednim prze-indeksowaniu literałów powstałych z zmiennych i symboli wynikowych. W tym celu stworzyłem kilka funkcji, które zostały opisane poniżej oraz klasę CnfFileHelper. Klasa składa się z trzech list: Lista usedVariables przechowuje informację o zmiennych, które zostały użyte w równaniu. Dzięki temu można uniknąć sytuacji, w której dwa razy wymuszam wartość pojedynczej zmiennej. Lista variableCode przechowuje stworzone literały, na podstawie której indeksuje zmienne w wynikowym pliku. Lista line przechowuje prze-indeksowane linie gotowe do zapisu do finalnego pliku .cnf

3.6.1 Wymuszenie pojedynczej wartości zmiennej

1. Stwórz literały w liście variableCode od x_0 do x_{C-1} , poprzez dopisanie ich na koniec listy.
2. Na podstawie indeksów stworzonych właśnie literałów dodaj do listy line prze-indeksowany wiersz. Odpowiada to pierwszemu typowi klauzuli.
3. Stwórz dwie pętle: pierwsza w zakresie $\{0, 1, \dots, C-1\}$ iterowana po zmiennej z i druga w zakresie $\{z+1, z+2, \dots, C-1\}$ iterowana po zmiennej k .
4. W ciele pętli zapisuj do listy line na podstawie stworzonych indeksów wiersze równoważne klauzuli $(\neg x_z \vee \neg x_c)$

Operacja ta jest wykonywana dla każdej zmiennej i każdego symbolu wynikowego, nawet jeżeli operacja nie posiada zmiennych. Spowodowane jest to faktem, iż gdy całe równanie posiadało tylko operacje z stałymi, to Sat-Solver wstawiał za wszystkie zmienne true, aby spełnić równanie.

3.6.2 Rekurencyjne znajdowanie wartości

Na podstawie listy zmiennych rozpatrywanej operacji tworzę listę Integerów, w której zmienne zastępuję liczbą -1, a stałe przepisuję bez zmian. Na podstawie nowej listy wykonywana jest redukcja mająca na celu wygenerowanie klauzul typu trzeciego.

Przebieg algorytmu

1. Sprawdź czy lista posiada wartości -1.
2. Jeżeli nie, to znajdź miejsca w których oryginalnie znajdowała się zmienna.
3. Dopisz do wiersza indeks zmiennej powstałej z nazwy zmiennej oraz numeru znajdującego się na miejscu zmiennej w nowej liście.
4. Wylicz indeks w tablicy operacji, reprezentowany przez tablice intów.
5. Dopisz do wiersza literal reprezentowany przez symbol wynikowy równania i znaną wartość w tablicy i dodaj wiersz do listy line.
6. Jeżeli lista posiada wartości -1 zamień kolejną -1 na wartość wskazaną przez pętlę.
7. Wywołaj funkcję rekurencyjnie, po czym przywróć wartość -1 i przerwij działanie

Jeżeli operacja posiada same wartości stałe to wylicz indeks w tablicy operacji, reprezentowany przez tablice intów i dodaj do listy line wiersz składający się z pojedynczego literalu reprezentowanego przez symbol wynikowy równania i znaną wartość w tablicy.

3.6.3 Porównywanie stron

1. Znajdź w liście variableCode indeks a, który wskazuje na pierwsze miejsce występowania interesującej nas zmiennej.
2. Ponów krok dla drugiej zmiennej, aby znaleźć indeks b.
3. Dopisuj, w pętli o długości $\{0, 1, \dots, C-1\}$ iterowanej po k, do listy line prze-indeksowane zmienne odpowiadające klauzulom $((\neg x_{a+k+1} \vee x_{b+k+1}) \wedge (x_{a+k+1} \vee \neg x_{b+k+1}))$

Krok ten jest wykonywany najczęściej, gdy porównujemy lewą stronę z prawą stroną równania. Robimy to dla każdego równania. W przypadku gdy strona równania składa się z pojedynczej zmiennej np. $\text{xor}(1,1)=x$ wtedy używamy tego typu klauzul aby powiązać

zmienną z symbolem wynikowym równania, gdyż obie zmienne znaczą wtedy to samo, są sobie równoważne.

3.6.4 Ogólny algorytm tworzenia formuły CNF

Algorytm tworzący formułę CNF składa się z wszystkich algorytmów opisanych wyżej w tym pod rozdziale. Przedstawiony on został poniżej:

1. Weź kolejną operację równania opisaną w liście EquationTable.
2. Sprawdź, czy rozpatrywana operacja jest pojedynczym znakiem.
3. Jeżeli jest, wymuś pojedynczą wartość true dla symbolu wynikowego tej operacji.
4. Jeżeli znak jest zmienną, wykonaj porównanie stron dla tej zmiennej i symbolu wynikowego.
5. Jeżeli znak jest stałą wpisz do listy line literał reprezentowany przez symbol wynikowy i wartość stałej.
6. Jeżeli operacja posiada argumenty, stwórz listę intów.
7. Weź kolejny symbol operacji.
8. Jeżeli jest to wartość stała, przepisz ją bez zmian do listy.
9. Jeżeli jest to zmienna, wpisz do listy wartość -1 i wymuś na niej wartość true jeżeli wystąpiła pierwszy raz.
10. Wróć do punktu 7.
11. Jeżeli wszystkie symbole operacji zostały rozpatrzone wymuś wartość true dla jej symbolu wynikowego.
12. Jeżeli funkcja posiadała niewiadome wykonaj rekurencje
13. Jeżeli nie, wylicz indeks literału w tablicy operacji.
14. Wróć do punktu 1.
15. Jeżeli wszystkie funkcje obu stron równań zostały rozpatrzone wykonaj porównanie stron dla symbolu wynikowego lewej strony i prawej strony równania.

Algorytm ten jest wywoływany osobno dla każdego pojedynczego równania. Najpierw wykonuje się lewa strona, później prawa, a na końcu następuje porównanie. Mogę tak zrobić, ponieważ cały algorytm opiera się na tej samej instancji klasy CNFfileHelper

3.7 Uruchomienie Sat-Solvera

Po ukończeniu redukcji następuje zapis formuły do pliku. Wykorzystuje do tego klasę FileWriter. W pliku znajduje się formuła CNF dotychczas przechowywana w liście line klasy CNFfileHelper oraz za-komentowana linia, która przedstawia indeksy użytych literałów. Po zakończeniu zapisu pliku następuje tworzenie wyjścia programu. Wyjście to składa się z dwóch części: z list EquationTable, które mają służyć za legendę oraz z faktycznego rozwiązania. Wyświetlane są także wartości wynikowe funkcji zagnieżdżonych.

```

Result Window
Równanie 1: Strona Lewa:
(q [1, o] &0) |
Strona Prawa:
(&& [x] &1) |
Równoważność stron: &0 = &1
-----
Równanie 2: Strona Lewa:
(q [1, &3] &2) | (d [1, 2, x] &3) |
Strona Prawa:
(q [0, r] &4) |
Równoważność stron: &2 = &4
-----
Rozwiązanie 1
o=1| x=0| r=0|
&0=0| &1=0| &3=1| &2=0| &4=0|
-----
Rozwiązanie 2
o=3| x=2| r=3|
&0=2| &1=2| &3=2| &2=3| &4=3|
-----
Rozwiązanie 3
o=2| x=3| r=3|
&0=3| &1=3| &3=2| &2=3| &4=3|

```

Rysunek 7: Przykładowe wyjście algorytmu

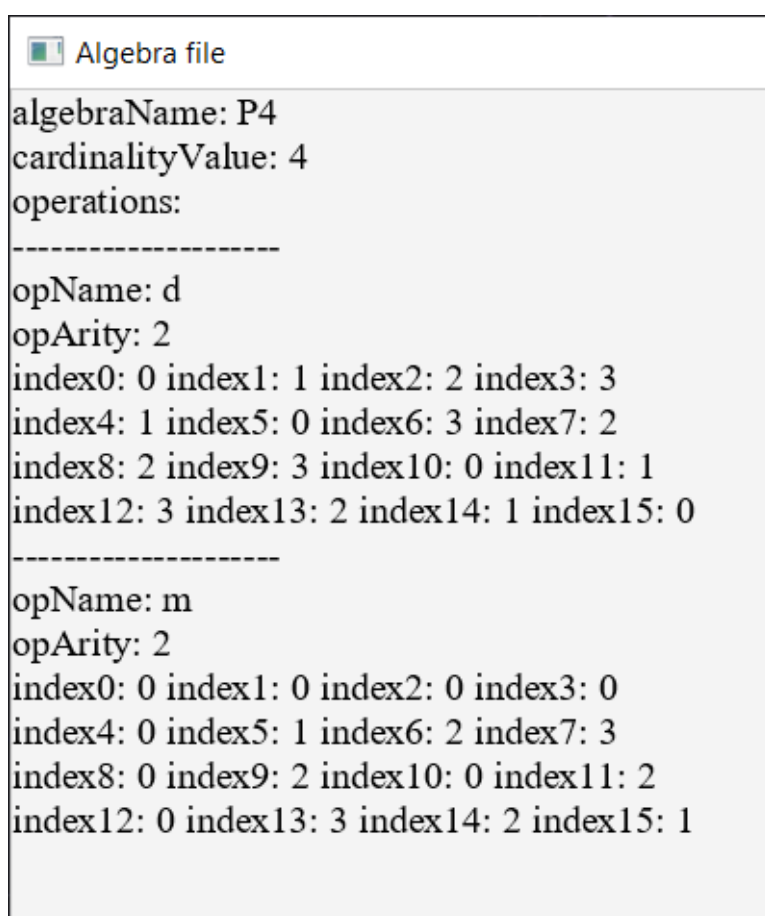
Wyrażenie "&&" w miejscu przeznaczonym na nazwę operacji oznacza, że dana strona równania składa się z pojedynczego znaku. Dzięki temu algorytm potrafi rozpoznać ten szczególny rodzaj równania.

4 Testy aplikacji

W czasie pisania i tworzenia logiki programu powstało kilka problemów, które trzeba było rozwiązać i przetestować, poniżej przedstawiam kilka z nich:

4.1 Wczytanie algebry

Cała aplikacja opiera się na wczytanym ciele algebry. Potrzebowałem więc sposobu, aby móc zweryfikować, czy proces przebiega poprawnie. W tym celu zadeklarowałem osobne okno, do wyświetlenia algebry poprzez wbudowane metody listy oraz gettery klas .



```
Algebra file
algebraName: P4
cardinalityValue: 4
operations:
-----
opName: d
opArity: 2
index0: 0 index1: 1 index2: 2 index3: 3
index4: 1 index5: 0 index6: 3 index7: 2
index8: 2 index9: 3 index10: 0 index11: 1
index12: 3 index13: 2 index14: 1 index15: 0
-----
opName: m
opArity: 2
index0: 0 index1: 0 index2: 0 index3: 0
index4: 0 index5: 1 index6: 2 index7: 3
index8: 0 index9: 2 index10: 0 index11: 2
index12: 0 index13: 3 index14: 2 index15: 1
```

Rysunek 8: Przykładowe wyjście algorytmu

Funkcja ta była na tyle przydatna, że zostawiłem ją na stałe w programie, aby użytkownik mógł sobie przypomnieć, jaka algebra została wczytana do programu.

4.2 Obsługa równań

Musiałem ustalić taką formę wprowadzania równań, aby była łatwa do obróbki. Dlatego postawiłem na formę funkcyjną. Za pierwszym razem badałem string char po charze, niestety rozwiązanie to było podatne na znaki białe i było nieoptymalnie obliczeniowo. Dlatego finalnie postanowiłem usuwać znaki białym funkcją `trim()` oraz dzielić równanie funkcją `split()` po znakach `"(),"`.

Tak jak wspomniałem wyżej redukcja jest tworzona na podstawie listy `EquationTable`. Lista ta, w pierwszej iteracji programu, nie była przystosowana do radzenia sobie z równaniem składającym się z pojedynczego znaku. Dlatego równanie typu `xor(1,1)=x` blokowało program. Musiałem stworzyć specjalną funkcję `if` badającą tego typu specjalne przypadki. Za nazwę funkcji instrukcja wstawiała `&&` i do tablicy argumentów zapisywała ten znak. Dalsza część algorytmu rozpatrywała przypadek właśnie po specjalnej nazwie.

Po uporaniu się z formatem równań należało podłączyć Sat-Solver i sprawdzić czy zwraca dobre wyniki. Na początku podawałem pojedyncze równanie mające tę samą funkcję po obu stronach, na przykład `xor(1,1)=xor(1,1)`. W momencie upewnienia się, że Sat-Solver zwraca poprawne wyniki stworzyłem specjalną algebrę mającą na celu sprawdzenie nie-spełnialności. Algebra ta posiadała `Cardinality=4` oraz pięć operacji: 4 operacje stałe zwracające zawsze odpowiednio: 0,1,2,3 oraz dodawanie `%4`. Algebra taka idealnie nadawała się do testów solvera

Podsumowanie

Literatura

- [1] <https://www.researchgate.net/publication/220169590> 'The Complexity of Checking Identities over Finite Groups
- [2] <http://www.satcompetition.org/>
- [3] <https://math.hawaii.edu/~ralph/Classes/619/univ-algebra.pdf>
- [4] <https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html>
- [5] <https://docs.oracle.com/en/java/javase/11/>
- [6] <https://openjfx.io/javadoc/17/javafx.graphics/javafx/application/Application.html>
- [7] <https://www.sat4j.org/>
- [8] <https://docs.oracle.com/javase/8/docs/api/javax/xml/parsers/DocumentBuilder.html>