WOJSKOWA AKADEMIA TECHNICZNA
WYDZIAŁ CYBERNETYKI

Synteza logiczna funkcji boolowskich szczególnej postaci

mgr inż. Tomasz Mazurkiewicz

Promotor:
prof. dr hab. inż. Tadeusz Łuba

Warszawa 2021

WOJSKOWA AKADEMIA TECHNICZNA
WYDZIAŁ CYBERNETYKI

Streszczenie

mgr inż. Tomasz Mazurkiewicz

Synteza logiczna od wielu lat jest przedmiotem intensywnych badań. W ostatnich latach uwagę naukowców przyciągnęły metody minimizalizacji funkcji boolowskich szczególnej postaci, zwanych funkcjami generowania indeksów. Wynika to z ich licznych zastosowań m.in. w telekomunikacji i cyberbezpieczeństwie. Funkcje te charakteryzują się dużą liczbą zmiennych wejściowych oraz stosunkowo mało licznym zbiorem wektorów w tablicy prawdy. W związku z tym, mogą być one efektywnie minimalizowane z wykorzystaniem metod syntezy logicznej.

Głównym problemem badawczym rozprawy jest realizacja wspomnianych funkcji pozwalająca na ograniczenie wykorzystania pamięci. W celu jego rozwiązania zaproponowano zastosowanie następujących metod:

- dekompozycję liniową z wykorzystaniem zbiorów niezgodności,
- realizację z wykorzystaniem dedykowanej architektury sprzętowej i struktur probabilistycznych,
- dekompozycję funkcjonalną z wykorzystaniem algorytmów teorii grafów oraz problemu spełnialności modulo teorie (SMT).

Uzyskane wyniki potwierdzają efektywność zaproponowanych rozwiązań.

Spis treści

Streszczenie **i**

Spis treści **ii**

Spis rysunków **iv**

Spis tabel **v**

1 Wprowadzenie **1**
1.1 Motywacja i charakterystyka prowadzonych badań 1
1.2 Teza 5
1.3 Struktura rozprawy 6

2 Podstawowe pojęcia i definicje **7**
2.1 Funkcje generowania indeksów 7
2.2 Rachunek podziałów 11
2.3 Elementy teorii grafów 13

3 Dekompozycja liniowa **16**
3.1 Model dekompozycji 16
3.2 Dekompozycja wykorzystująca zbiór niezgodności 17
3.2.1 Metody wyboru dekompozycji 22
3.2.2 Uzyskane wyniki 26
3.2.3 Wnioski z przeprowadzonych badań 34

4 Realizacja generatorów indeksów **36**
4.1 Generatory indeksów 36
4.2 Generatory wykorzystujące struktury probabilistyczne 40
4.2.1 Filtr Blooma 41
4.2.2 Filtr Blooma z pojedynczą funkcją skrótu 44
4.2.3 Filtr Cuckoo 46
4.2.4 Uzyskane wyniki 49
4.2.5 Wnioski z przeprowadzonych badań 61

5 Dekompozycja funkcjonalna **63**
5.1 Model dekompozycji 63
5.2 Dekompozycja wykorzystująca rachunek podziałów 65
5.2.1 Metoda heurystyczna 67
5.2.2 Metoda dokładna 72
5.2.3 Uzyskane wyniki 75
5.2.4 Wnioski z przeprowadzonych badań 83

6 Podsumowanie pracy i dalsze kierunki badań **84**

Bibliografia **87**

Spis rysunków

2.1 Przykładowy graf G . . . 14
2.2 Przykładowe pokolorowanie grafu G . . . 15

3.1 Schemat dekompozycji liniowej . . . 16
3.2 Średni czas obliczeń dla funkcji *1 z N* . . . 28
3.3 Czas obliczeń w zależności od wartości K . . . 31
3.4 Czas obliczeń w porównaniu z wcześniejszymi pracami . . . 31

4.1 Realizacja funkcji generowania indeksów z wykorzystaniem kaskady elementów pamiętających . . . 37
4.2 Realizacja funkcji generowania indeksów z wykorzystaniem wielopoziomowej kaskady elementów pamiętających . . . 38
4.3 Architektura IGU (na podstawie [Sas20]) . . . 38
4.4 Rozmiar pamięci dodatkowej w IGU ($N = 32$) [Maz19a] . . . 40
4.5 Idea działania filtru Blooma [Maz19b] . . . 41
4.6 Idea działania filtru Blooma z pojedynczą funkcją skrótu [Maz19b] . . . 44
4.7 Liczba bitów na element [Maz19a] . . . 49
4.8 Generator indeksów wykorzystujący strukturę probabilistyczną . . . 50
4.9 Wykorzystanie pamięci w zależności od wartości P . . . 51
4.10 Rozmiar pamięci dodatkowej w IGU w porównaniu do rozmiaru struktur probabilistycznych, w zależności od wartości ϵ dla koderów *M z 20* . . . 52
4.11 Prawdopodobieństwa graniczne dla koderów *M z 20* w zależności od wartości P . . . 53
4.12 Prawdopodobieństwa graniczne dla kodera *1 z 20* w zależności od wartości P 54
4.13 Rozmiar pamięci dodatkowej w zależności od liczby wektorów (na podstawie [MBŁ18]) . . . 55
4.14 Wartość najmniejszego modulnika w zależności od liczby wektorów rejestrowanych [Maz19a] . . . 57
4.15 Układ do realizacji pojedynczej operacji modulo [Maz19a] (na podstawie [BS11]) . . . 58
4.16 Liczba iteracji w zależności od wartości modulnika [Maz19a] . . . 59
4.17 Wykorzystanie zasobów logicznych w zależności od wartości ϵ [Maz19a] . 61

5.1 Schemat (nierozłącznej) dekompozycji funkcjonalnej . . . 64
5.2 Postać grafu Γ w dwóch iteracjach algorytmu [Maz20a] . . . 69
5.3 Średni czas obliczeń dla różnych wartości K . . . 78
5.4 Uzyskany graf Γ w zależności od postaci zbioru W . . . 80
5.5 Realizacja funkcji *2 z 20* . . . 80

Spis tabel

2.1 Przykładowa funkcja generowania indeksów 8
2.2 Przykładowa funkcja generowania indeksów ($K = 16, N = 15$) 9
2.3 Przykładowa funkcja generowania indeksów po zastosowaniu algorytmu dekompozycji liniowej 10

3.1 Funkcja generowania indeksów ($N = 5$, $K = 4$) 21
3.2 Postaci zbiorów $C_{p,q}$ 21
3.3 Macierz rozróżnialności 21
3.4 Funkcja po zastosowaniu algorytmu dekompozycji liniowej 22
3.5 Funkcja *1 z 7* 23
3.6 Macierz rozróżnialności dla zredukowanej funkcji *1 z 7* 24
3.7 Macierz rozróżnialności w drugiej iteracji algorytmu 25
3.8 Reprezentacja funkcji *1 z 7* po zastosowaniu proponowanych podejść . . . 26
3.9 Wyniki uzyskane dla funkcji *1 z N* 27
3.10 Efektywność algorytmów dekompozycji liniowej dla funkcji *M z 16* 29
3.11 Efektywność algorytmów dekompozycji liniowej dla funkcji *M z 20* 29
3.12 Liczność K zbioru wektorów rejestrowanych dla funkcji *M z N* 30
3.13 Stopień złożoności reprezentacji dla funkcji *M z 20* [MŁ19a] 32
3.14 Wyniki dla metody wielomianowej (M.w.) oraz metod First-Fit (FF) i MinR, dla losowych funkcji generowania indeksów 33
3.15 Wyniki dla losowych funkcji generowania indeksów 34

4.1 Parametry dla filtru Blooma 43
4.2 Optymalna liczba bitów na element w stukturze 49
4.3 Wartości $\widehat{\epsilon_b}$ oraz $\widehat{\epsilon_c}$ dla funkcji *M z 20* 52
4.4 Wartości poszczególnych m_i dla $\epsilon = 1\%$ [Maz19b] 56
4.5 Wartości poszczególnych m_i dla $\epsilon = 0,5\%$ [Maz19a] 56
4.6 Wyniki sprzętowej implementacji warstwy operacji modulo [Maz19a] . . . 60

5.1 Przykładowa funkcja generowania indeksów 66
5.2 Postać funkcji po dekompozycji 70
5.3 Wyniki uzyskane po zastosowaniu algorytmu dekompozycji liniowej 76
5.4 Wyniki uzyskane dla różnych wartości K [Maz20a] 77
5.5 Liczba funkcji zminimalizowana z wykorzystaniem dekompozycji funkcjonalnej [Maz20a] 78
5.6 Wyniki dla koderów *M z 20* [Maz20a] 79
5.7 Porównanie efektywności metody heurystycznej i dokładnej [Maz20b] . . . 81
5.8 Otrzymane wyniki dla funkcji *M z N* [Maz20b; Maz20a] 82
5.9 Wyniki dla funkcji nieredukowalnych 83

Rozdział 1

Wprowadzenie

1.1 Motywacja i charakterystyka prowadzonych badań

Ostatnie lata to okres gwałtownego rozwoju szeroko rozumianych technologii informacyjnych, w tym telekomunikacyjnych. Rozwój ten sprawia, że ilość przetwarzanych danych jest kilkadziesiąt razy większa niż jeszcze dekadę temu, co wynika m.in. z popularyzacji rozwiązań takich jak Internet Rzeczy (ang. Internet of Things) czy wiedza o danych (ang. Data Science). Wymusza to poszukiwanie nowych metod, które odpowiadać będą współczesnym wyzwaniom związanym z potrzebą klasyfikacji i identyfikacji danych. Problem ten jest szczególnie istotny w przypadku realizacji cyfrowych układów sprzętowych, które przetwarzają informacje i sygnały. Coraz powszechniej układy te projektowane są z wykorzystaniem układów programowalnych FPGA (ang. Field-Programmable Gate Array), które wyposażone są we wbudowane struktury pamięciowe. Z tego względu istotnym zagadnieniem stało się opracowanie metod syntezy logicznej w strukturach wykorzystujących pamięci RAM oraz ROM (ang. Memory-based logic synthesis).

Jedną z podstawowych operacji realizowanych przez układy cyfrowe jest wyszukiwanie danych reprezentujących różnego rodzaju informacje w dużym zbiorze. Mogą być to zarówno informacje statyczne, jak i dynamiczne. Operacja ta stosowana jest m.in. w telekomunikacji i cyberbezpieczeństwie - przy dystrybucji adresów IP, do konwersji kodów, w bazach danych czy do wykrywania danych niepożądanych, np. wirusów.

Co istotne, dane przetwarzane przez układy cyfrowe reprezentowane są zazwyczaj jako długie ciągi binarne. Na przykład, dla protokołu IPv6 adresy IP mają długość 128 bitów. Router podczas działania przechowuje ich kilkadziesiąt tysięcy i sprawdza czy adres, który pojawił się na wejściu odpowiada jednemu z przechowywanych. W przypadku układów cyfrowych realizację takiej funkcjonalności gwarantuje pamięć CAM (ang. Content Addressable Memory). Jej koszt oraz zużycie prądu jest wysokie. W związku z tym,

w literaturze poszukuje się alternatywnych metod pozwalających zapewnić tę funkcjonalność.

Funkcje generowania indeksów [Sas11b] pozwalają na realizację funkcjonalności pamięci CAM przy wykorzystaniu pamięci RAM, a ich idea uznawana jest za kluczową w urządzeniach sieciowych [Sas20]. Funkcje te są funkcjami boolowskimi szczególnej postaci - przypisują one N-bitowym ciągom binarnym unikalne wartości całkowite ze zbioru $[0, K]$, przy czym wartość 0 oznacza, że analizowane dane nie występują w określonym zbiorze wektorów rejestrowanych. Ze względu na wysoki stopień nieokreśloności, synteza logiczna funkcji generowania indeksów wymaga zastosowania nowych metod, opartych przede wszystkim na redukcji argumentów oraz dekompozycji. Pozwala to usunąć istotną barierę, jaką jest duży rozmiar pamięci RAM/ROM potrzebny do bezpośredniej realizacji sprzętowej tych funkcji.

Dzięki swoim właściwościom, większość funkcji generowania indeksów może być poddana skutecznej redukcji liczby zmiennych. Proces ten polega na poszukiwaniu najmniej licznego zbioru zmiennych wejściowych, dla którego dane pozostają spójne. W literaturze zaproponowano metody redukcji argumentów [BŁ14; ŁB15; Bor18], które gwarantują uzyskanie bardzo dobrych wyników. Istnieją jednak funkcje, dla których metody te okazują się całkowicie nieefektywne. Przykładem takich funkcji są kodery M *z* N. Tablica prawdy zawiera wtedy wszystkie możliwe N-bitowe wektory o wadze Hamminga równej M. Metody redukcji pozwalają na znalezienie reprezentacji, która ma zaledwie jedną zmienną wejściową mniej. W związku z tym, konieczne było zaproponowanie nowych metod syntezy funkcji generowania indeksów. Kodery M *z* N wykorzystywane są jako standardowe wektory testowe dla tych metod.

Celem niniejszej pracy jest systematyczne przedstawienie innych, wybranych metod efektywnej realizacji funkcji generowania indeksów pod względem wykorzystania pamięci.

Szczególną uwagę badaczy przyciągnęły algorytmy dekompozycji liniowej [Sas08; Sas11b; SZP12; SFI15; Ast+16; Sas17; HPC19]. Pozwalają one na realizację funkcji boolowskiej jako złożenia funkcji liniowej oraz funkcji ogólnej. Pierwsza z nich wykorzystuje bramki XOR do łączenia zmiennych wejściowych i wprowadza je do funkcji ogólnej, która realizowana jest z wykorzystaniem pamięci RAM lub ROM. Dzięki zastosowaniu warstwy liniowej, rozmiar pamięci może ulec znacznemu zmniejszeniu poprzez ograniczenie liczby wejść do pamięci. Co więcej, warstwa liniowa może być efektywnie realizowana w układach FPGA. Kluczowym zagadnieniem jest zatem zaproponowanie metody, która zagwarantuje jak najlepszą kompresję zmiennych.

Cechą charakterystyczną pierwszych prac dotyczących dekompozycji liniowej było zastosowanie tablic dekompozycji (ang. Decomposition Charts) [Sas11b]. Jednakże, w opinii

autora podejście wykorzystujące zbiory niezgodności [ŁPZ16; ŁBJ16] posiadało potencjał do zapewnienia większej efektywności. W związku z tym, w ramach przeprowadzonych badań zaproponowano i zaimplementowano autorskie modyfikacje algorytmu dekompozycji liniowej wykorzystującego zbiory niezgodności [MŁ17; MŁ18; MŁ19a]. Obejmowały one m.in.:

- zmianę metody wyboru funkcji rozdzielającej w poszczególnych iteracjach algorytmu,
- zmodyfikowanie procedury generowania macierzy rozróżnialności, m.in. poprzez usuwanie z niej powtarzających się wartości.

Efektywność zaproponowanych rozwiązań pod względem jakości generowanego rozwiązania zweryfikowano z wykorzystaniem funkcji *M z N* oraz losowo wygenerowanych funkcji generowania indeksów. Otrzymane wyniki porównane zostały z rezultatami dostępnymi w literaturze. Wskazano konkurencyjność proponowanego podejścia względem dotychczasowych propozycji. W szczególności, dla funkcji *M z 16* oraz *M z 20* uzyskiwane wyniki są identyczne z najlepszymi wynikami dostępnymi w literaturze. Co więcej, zaproponowane modyfikacje pozwoliły uzyskać znaczący wzrost wydajności w porównaniu do istniejącego wcześniej oprogramowania [Kow17] implementującego algorytm dekompozycji liniowej wykorzystujący zbiory niezgodności.

Innym istotnym zagadnieniem w literaturze jest zaproponowanie architektury sprzętowej umożliwiającej efektywną realizację funkcji generowania indeksów. Taki układ cyfrowy nazywamy generatorem indeksów. Największą popularność zdobyła architektura zaproponowana przez prof. Sasao [Sas06; Sas11b; Sas20] (por. rysunek 4.3), oznaczana w skrócie IGU (ang. Index Generation Unit). Składa się ona z dwóch pamięci (głównej oraz pomocniczej), komparatora oraz bramki AND. Pamięć główna przechowuje wartości indeksów dla wektorów, dla których $F(X) \neq 0$. Pamięć pomocnicza oraz komparatora wykorzystywane są z kolei do określenia czy wektor wejściowy istotnie należy do analizowanego zbioru wektorów. Co istotne, architektura ta jest podstawowym elementem także innych rozwiązań [Sas20]. Należy jednak zwrócić uwagę, że posiada ona istotną wadę - im mniejsza liczba zmiennych do reprezentacji funkcji po transformacji liniowej, tym rozmiar pamięci pomocniczej jest większy. Prowadzi to do sytuacji, gdzie dla niektórych funkcji rozmiar pamięci pomocniczej przekracza rozmiar pamięci głównej. W ramach badań zaproponowano autorską architekturę wykorzystującą struktury probabilistyczne [MBŁ18], która nie ma takiej wady. Przeanalizowano zastosowanie filtru Blooma oraz filtru Cuckoo. Działanie tych struktur polega na stwierdzeniu, przy wykorzystaniu funkcji skrótu, czy dany wektor wejściowy jest elementem określonego zbioru czy nie. W związku z tym, wspomniane filtry mogą być wykorzystane zamiast pamięci pomocniczej w generatorze indeksów. Zajętość pamięci wykorzystywanej do realizacji

tych struktur w zaproponowanej architekturze nie jest zależna od liczby zmiennych, jednak wprowadza niewielkie prawdopodobieństwo wyniku fałszywie pozytywnego.

W przypadku filtru Blooma istotnym zagadnieniem jest również dobór stosowanych funkcji skrótu. Jednak nie jest to zadanie proste. W związku z tym, w ramach badań zaproponowano [Maz19a] wykorzystanie filtru Blooma z pojedynczą funkcją skrótu celem dalszej redukcji wykorzystania pamięci przy sprzętowej realizacji generatora indeksów. Przedstawiono wyniki dowodzące, że filtr ten może być efektywnie implementowany w strukturach programowalnych.

Mimo efektywności algorytmów dekompozycji liniowej pod względem jakości uzyskiwanego rozwiązania, istnieją funkcje dla których nie istnieje dekompozycja optymalna. Dla niektórych funkcji, np. *2 z 20*, udowodniono to z wykorzystaniem metody dokładnej [Sas17]. Co więcej, w literaturze wprowadzono pojęcie niedekomponowalnej funkcji generowania indeksów [Sas20]. Jest to funkcja, dla której nie istnieje dekompozycja liniowa oraz rodzina zbiorów niezgodności zawiera wszystkie możliwe wektory niezerowe. W związku z tym w literaturze [SMI16; SMI17] zaproponowano wykorzystanie dekompozycji funkcjonalnej do dalszego zmniejszenia wykorzystania pamięci przy realizacji takich funkcji. Metoda ta polega na reprezentacji funkcji boolowskiej jako złożenie dwóch funkcji, oznaczanych przez G i H, które posiadają mniejszą liczbę zmiennych.

Dostępne prace skupiają się na wykorzystaniu dekompozycji rozłącznej do redukcji pamięci przy realizacji funkcji generowania indeksów. W opinii autora niniejszej rozprawy, podejście takie jest niewystarczające. Istnieją bowiem funkcje, dla których możliwe jest znalezienie reprezentacji wykorzystującej mniej pamięci przy zastosowaniu podejścia nierozłącznego. W związku z tym, ostatnia część badań obejmowała zaproponowanie algorytmów dekompozycji funkcjonalnej funkcji generowania indeksów. W tym celu zaproponowano i zaimplementowano dwie metody, które pozwalają na znalezienie nierozłącznej dekompozycji funkcjonalnej:

1. heurystyczną [MŁ19b] - wykorzystującą algorytmy teorii grafów,
2. dokładną [Maz20b] - wykorzystującą problem SMT (ang. Satisfiability Modulo Theories).

Metody te wykorzystują rachunek podziałów oraz pojęcie r-przydatności do znalezienia podziału zbioru zmiennych wejściowych na dwa rozłączne podzbiory, które stanowić będą wejścia do funkcji G i H. Algorytmy teorii grafów oraz problem SMT zostały z kolei wykorzystane do znalezienia podzbioru zmiennych wejściowych, oznaczanego przez W, który umożliwia reprezentację funkcji z wykorzystaniem dekompozycji nierozłącznej. Co istotne, zaproponowane metody poszukują postaci zbioru W w taki sposób, aby zbiór ten miał jak najmniejszą liczność.

Należy podkreślić, że wcześniejsze prace [Raw+97] wykorzystywały algorytm kolorowania grafu do znajdowania dekompozycji nieliniowej. Jednakże, algorytm ten wykorzystywany był jedynie raz, do wyznaczenia liczby chromatycznej. Na jej podstawie określana była liczba wyjść z funkcji G. W ramach niniejszej rozprawy zaproponowano iteracyjne podejście, które zakłada dodawanie zmiennych do zbioru W dopóki nie zostanie otrzymana z góry określona wartość liczby chromatycznej, zależna od wartości r. Dzięki temu bardzo często uzyskuje się reprezentację, w której funkcja H posiada tyle samo wejść i wyjść. Prowadzi to do znacznej redukcji wykorzystywanej pamięci.

W ramach niniejszej rozprawy, przeanalizowano [Maz20a] również wpływ:

- wybranego algorytmu kolorowania grafu,
- metody wyboru zmiennej do zbioru W

na efektywność czasową i jakość uzyskiwanego rozwiązania dla metody heurystycznej. Zgodnie z oczekiwaniami, wybór heurystycznego algorytmu kolorowania grafu prowadzi do znacznie lepszej efektywności czasowej, kosztem jakości uzyskiwanego rozwiązania. Z drugiej strony, wpływ metody wyboru zmiennej do zbioru W jest znacznie mniejszy.

Efektywność zaproponowanych algorytmów nierozłącznej dekompozycji funkcjonalnej pod względem jakości generowanego rozwiązania zweryfikowana została poprzez znalezienie dekompozycji dla funkcji, które nie podlegają optymalnej dekompozycji liniowej. W tym celu dla analizowanych funkcji generowania indeksów poszukiwano w pierwszej kolejności dekompozycji liniowej, a następnie funkcjonalnej. Otrzymane wyniki udowadniają, że nierozłączna dekompozycja pozwala na dalszą redukcję zajętości pamięci w stosunku do zastosowania jedynie algorytmów dekompozycji liniowej lub rozłącznej dekompozycji funkcjonalnej.

1.2 Teza

Analiza literatury, dotyczącej tematyki poruszanej w ramach prowadzonych badań, pozwoliła na sformułowanie następujących tez pracy:

1. zbiory niezgodności wykorzystane mogą być do poszukiwania dekompozycji liniowej funkcji generowania indeksów,
2. możliwa jest realizacja funkcji generowania indeksów z wykorzystaniem struktur probabilistycznych, charakteryzująca się niewielkim wykorzystaniem pamięci,
3. dekompozycja funkcjonalna pozwala na realizację funkcji generowania indeksów, dla których nie istnieje optymalna dekompozycja liniowa.

1.3 Struktura rozprawy

Rozprawa składa się z sześciu rozdziałów. Rozdział pierwszy stanowi wprowadzenie do rozprawy. Zdefiniowano w nim motywację i charakterystykę prowadzonych badań, a także postawiono tezy pracy. W rozdziale drugim wprowadzono podstawowe pojęcia wykorzystywane w dalszych rozdziałach. Rozdział trzeci zawiera opis algorytmów dekompozycji liniowej. Przedstawiono w nim autorskie modyfikacje algorytmu wykorzystującego zbiory niezgodności. Rozdział czwarty stanowi opis metod realizacji generatorów indeksów. Przedstawiono w nim zaproponowane w literaturze rozwiązania, jak i autorską architekturę wykorzystującą struktury probabilistyczne. W rozdziale piątym przedstawiono algorytmy dekompozycji funkcjonalnej, w tym autorskie rozwiązania wykorzystujące odpowiednio kolorowanie grafów oraz problem SMT. Rozdział szósty stanowi podsumowanie rozprawy.

Rozdział 2

Podstawowe pojęcia i definicje

W ramach niniejszego rozdziału przedstawiono podstawowe pojęcia i definicje niezbędne w opinii autora do analizy dalszych rozdziałów. Zdefiniowano funkcje generowania indeksów oraz wymieniono ich zastosowania i metody syntezy. Następnie przedstawiono zagadnienia związane z rachunkiem podziałów i teorią grafów, które wykorzystane zostaną w rozdziale 5 do znajdowania dekompozycji funkcjonalnej tych funkcji.

2.1 Funkcje generowania indeksów

W ramach niniejszej rozprawy przedstawiono metody efektywnej realizacji funkcji generowania indeksów. Funkcje te zyskały uwagę naukowców dzięki swojej użyteczności w realizacji m.in.:

- skanerów antywirusowych [Nak+09; NSM13],
- tablic adresów IP [Sas11b; Nak+15],
- kontrolerów TAC [Sas11a],
- konwerterów danych cyfrowych [Kok19],
- białych list w zoptymalizowanej postaci [Sas20].

Funkcje generowania indeksów są to silnie nieokreślone funkcje boolowskie następującej postaci:

$$F : D^N \rightarrow \{1, 2, \ldots, K\} \tag{2.1}$$

gdzie $D^N \subseteq \{0,1\}^N$ oraz $|D^N| = K$. Zbiór D^N nazywa się zbiorem wektorów rejestrowanych. Składa się on z K różnych wektorów N bitowych. Zbiór zmiennych wejściowych oznaczany będzie przez $X = \{x_i, i = [1, N]\}$. Dodatkowo przez Q oznaczana będzie wartość $\lceil log_2(K) \rceil$.

W sytuacji gdy zadany wektor wejściowy równy jest jednemu z wektorów ze zbioru wektorów rejestrowanych, funkcja generowania indeksów zwraca odpowiadający mu indeks ze zbioru od 1 do K. Dla wektorów spoza zbioru D^N wartość funkcji równa jest zero, tzn. $\forall \overrightarrow{v} \notin D^N : F(\overrightarrow{v}) = 0$.

Przykład 2.1. Przykład funkcji generowania indeksów przedstawiono w tabeli 2.1. Dla funkcji tej określony jest zbiór wektorów rejestrowanych o liczności $K = 4$ oraz $N = 5$ zmiennych wejściowych.

Tabela 2.1: Przykładowa funkcja generowania indeksów

x_1	x_2	x_3	x_4	x_5	$F(X)$
0	1	1	0	1	1
1	1	1	1	0	2
1	0	0	1	0	3
1	0	0	0	1	4

Należy zwrócić uwagę, że poszczególnym wektorom przypisywane są (jako wartość $F(X)$) kolejne wartości z przedziału $[1, 4]$. Dzięki temu wartość $F(X)$ może posłużyć do identyfikowania konkretnego wektora w tablicy prawdy funkcji. △

Charakterystyczną cechą funkcji generowania indeksów stosowanych w praktyce jest duża liczba zmiennych wejściowych (N) oraz stosunkowo mało liczny zbiór wektorów rejestrowanych ($k \ll 2^N$). Dzięki temu możliwe jest efektywne redukowanie liczby wejść do układu za pomocą algorytmów syntezy logicznej, przy jednoczesnym zachowaniu spójności funkcji, tzn. usunięcie zmiennych nie powoduje pojawienia się sprzeczności w tablicy prawdy funkcji.

W literaturze [Sas20] spotkać się można również z pojęciem nie w pełni zdefiniowanej funkcji generowania indeksów. Charakteryzuje się ona tym, że dla wektorów spoza zbioru D^N jej wartość jest nieokreślona. Oznacza to, że możliwe jest przypisanie wartości z przedziału $[1, K]$ wektorowi, który nie należy do zbioru wektorów rejestrowanych. Dla takiej funkcji minimalna liczba zmiennych do jej reprezentacji wynosi:

$$GD_i = \lceil log_2(K) \rceil \tag{2.2}$$

Dla funkcji zgodnych z przedstawioną wcześniej definicją, generacja wartości 0 dla wektorów spoza zbioru wektorów rejestrowanych pozwala w łatwy sposób odróżnić je od wektorów rejestrowanych. Minimalna liczba zmiennych do reprezentacji funkcji wynosi wtedy:

$$GD = \lceil log_2(K + 1) \rceil \tag{2.3}$$

Mimo że różnica między dwoma ograniczeniami jest niewielka, to może mieć ona kluczowe znaczenie w niektórych realizacjach praktycznych. W dalszej części rozprawy dekompozycja liniowa pozwalająca zrealizować funkcję z wykorzystaniem GD zmiennych nazywana będzie optymalną.

Przykład 2.2. Niech dana będzie funkcja przedstawiona w tabeli 2.2 [MŁ17]. Dla funkcji tej $K = 16$ oraz $N = 15$. Ograniczenia na minimalną liczbę zmiennych wynoszą zatem odpowiednio:

$$GD_i = 4, GD = 5$$

Tabela 2.2: Przykładowa funkcja generowania indeksów ($K = 16, N = 15$)

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	$F(X)$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	3
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	4
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	5
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	6
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	7
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	8
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	9
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	10
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	11
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	12
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	13
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16

Wykorzystując algorytm dekompozycji liniowej, przedstawiony w rozdziale 3, można znaleźć realizację tej funkcji z wykorzystaniem czterech zmiennych wejściowych:

$$F(x_1, x_2, \ldots, x_{15}) = G(y_1, y_2, y_3, y_4)$$

gdzie G - funkcja boolowska o zredukowanej liczbie zmiennych wejściowych oraz

$$y_1 = x_2 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_{10} \oplus x_{11} \oplus x_{13} \oplus x_{14}$$
$$y_2 = x_1 \oplus x_2 \oplus x_4 \oplus x_6 \oplus x_8 \oplus x_9 \oplus x_{13} \oplus x_{14}$$
$$y_3 = x_1 \oplus x_2 \oplus x_4 \oplus x_5 \oplus x_{11} \oplus x_{12} \oplus x_{13} \oplus x_{15}$$
$$y_4 = x_4 \oplus x_5 \oplus x_7 \oplus x_8 \oplus x_{10} \oplus x_{12} \oplus x_{13} \oplus x_{14}$$

Funkcję $G(Y)$ przedstawiono w tabeli 2.3. Jak łatwo zauważyć, spójność funkcji pozostała zachowana. Niech teraz dany będzie wektor $\overrightarrow{v} = (1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$.

Tabela 2.3: Przykładowa funkcja generowania indeksów po zastosowaniu algorytmu dekompozycji liniowej

y_1	y_2	y_3	y_4	$G(Y)$
1	0	0	0	1
0	1	0	1	2
1	1	0	1	3
1	1	0	0	4
1	0	1	1	5
0	1	1	1	6
1	0	0	1	7
0	0	1	1	8
1	0	1	0	9
0	0	0	1	10
1	1	1	1	11
1	1	1	0	12
0	0	1	0	13
0	1	1	0	14
0	1	0	0	15
0	0	0	0	16

Mimo że $\overrightarrow{v} \notin D^N$, to $F(\overrightarrow{v}) = G(1,0,0,0) = 1$. Dla wektora spoza zbioru wektorów rejestrowanych otrzymaliśmy zatem wartość indeksu, która odpowiada wektorowi $(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$. △

W zastosowaniach praktycznych możliwość detekcji czy dany wektor należy do zbioru wektorów rejestrowanych jest często pożądana. W związku z tym najczęściej stosowane jest ograniczenie wynikające ze wzoru 2.3.

W dalszych rozważaniach istotna będzie również górna granica na liczbę zmiennych. Jej zależność od wartości K (przy czym $K \geqslant 3$) jest następująca [Sas20]:

$$GG_S = \lceil 2log_2 K \rceil - 2 \tag{2.4}$$

W literaturze [Ast+16] zaproponowano również ograniczenie, które zależy zarówno od wartości K, jak i N:

$$GG_A = 2\lceil log_2(K+1) \rceil - 1 + \lceil log_2(N-1) \rceil \tag{2.5}$$

W syntezie funkcji generowania indeksów wykorzystuje się następujące techniki:

1. redukcję zmiennych,
2. dekompozycję liniową,
3. dekompozycję funkcjonalną,
4. dedykowaną realizację sprzętową.

Algorytmy redukcji zmiennych, czyli znajdowanie podzbioru zmiennych wejściowych, dla którego odpowiedź układu pozostaje niezmieniona, były intensywnie badane w literaturze [ŁR92; BŁ14; BŁP16; Sas20]. Uzyskano bardzo skuteczne rozwiązania, a niektóre z algorytmów uwzględniają również szczególne własności funkcji generowania indeksów [Bor18; BŁK20]. W związku z tym, w dalszych rozdziałach niniejszej rozprawy skoncentrowano się na zastosowaniu technik 2-4 do efektywnej realizacji funkcji generowania indeksów.

Do oceny efektywności wspomnianych metod, pod względem jakości generowanego rozwiązania, używa się często funkcji *M z N*, zwanych również koderami. Są to funkcje o N zmiennych wejściowych oraz zbiorze wektorów rejestrowanych obejmującym wszystkie wektory N-bitowe o wadze Hamminga M. Liczność tego zbioru równa jest $K = \binom{N}{M}$. Powszechność stosowania tych funkcji wynika z faktu, że redukcja zmiennych gwarantuje redukcję jedynie o jedną zmienną. Na przykład, tabela 2.2 przedstawia funkcję *1 z 16* po operacji redukcji zmiennych. Jak widać, usunięta została z niej jedynie zmienna x_{16}. Z kolei zastosowanie dekompozycji liniowej pozwala na uzyskanie o wiele lepszej minimalizacji. Najczęściej wykorzystywanymi funkcjami są funkcje *M z 16* oraz *M z 20*, gdzie $M \in \{1, 2, 3, 4\}$.

2.2 Rachunek podziałów

Rachunek podziałów [BŁ97; ŁB15] jest zwartym i użytecznym sposobem reprezentowania funkcji boolowskich. Pojęciem niezbędnym do zdefiniowania rachunku podziałów jest relacja równoważności.

Relacją nazywa się dowolny podzbiór iloczynu kartezjańskiego zbiorów. Relacją równoważności na zbiorze A nazywa się relację R, która spełnia następujące własności:

- zwrotność, tzn. $\forall a \in A : aRa$,
- symetryczność, tzn. $\forall a, b \in A : aRb \Rightarrow bRa$,
- przechodniość, tzn. $\forall a, b, c \in A : aRb \wedge bRc \Rightarrow bRc$.

Relację, która spełnia dwie pierwsze własności nazywa się relacją zgodności albo nierozróżnialności. Relacja równoważności dzieli zbiór A na rozłączne podzbiory, co prowadzi do pojęcia podziału. Formalnie, podziałem Π zbioru A nazywa się rodzinę rozłącznych podzbiorów B_i zbioru A takich, że $\bigcup B_i = A$. Podzbiory te nazywane są blokami.

Wykorzystując wprowadzone pojęcie podziału, możliwe jest zdefiniowanie rachunku podziałów. Z punktu widzenia dalszych rozważań kluczowe są dwa zagadnienia: relacja

„$\leqslant$” oraz działanie iloczynu. Relację tę definiuje się następująco:

$$\Pi \leqslant \Upsilon \Leftrightarrow \forall B_i \in \Pi : (\exists B_j \in \Upsilon : B_i \subseteq B_j) \tag{2.6}$$

Iloczynem Π podziałów Υ_1 oraz Υ_2 nazywamy podział o największych blokach taki, że:

$$\Pi = \Upsilon_1 \cdot \Upsilon_2 \Leftrightarrow \Pi \leqslant \Upsilon_1 \wedge \Pi \leqslant \Upsilon_2 \tag{2.7}$$

Podział ten wyznaczyć można poprzez obliczenie iloczynu poszczególnych bloków Υ_1 z blokami Υ_2.

Stosując iloczyn podziałów, wprowadzić można pojęcie ilorazu podziałów. Podział $\Pi|\Pi\Upsilon$ jest podziałem ilorazowym podziałów Π i Υ wtedy i tylko wtedy, gdy jego elementy są blokami iloczynu $\Pi \cdot \Upsilon$, a bloki są blokami Π.

Przykład 2.3. Niech dany jest zbiór $A = \{1, 2, 3, 4, 5\}$. Przykładowymi podziałami są wtedy:

$$\Pi_1 = \{\{1, 2, 3\}, \{4, 5\}\}$$
$$\Pi_2 = \{\{1\}, \{2, 3\}, \{4\}, \{5\}\}$$

W ramach niniejszej pracy stosowany będzie uproszczony zapis, tzn.:

$$\Pi_1 = \{\overline{1, 2, 3}; \overline{4, 5}\}$$
$$\Pi_2 = \{\overline{1}; \overline{2, 3}; \overline{4}; \overline{5}\}$$

Łatwo zauważyć, że $\Pi_2 \leqslant \Pi_1$ oraz $\Pi_1 \cdot \Pi_2 = \Pi_2$. W tym przypadku podział ilorazowy $\Pi_1|\Pi_1\Pi_2 = \{\overline{(1)(2, 3)}; \overline{(4)(5)}\}$. △

Dla funkcji boolowskich istotnym pojęciem jest również pojęcie podziału charakterystycznego (wyjściowego) P_F funkcji F. Spełnia on następującą zależność:

$$(s, t) \in B_i^{P_F} \Leftrightarrow F(\overrightarrow{v}_s) = F(\overrightarrow{v}_t) \tag{2.8}$$

gdzie $s, t \in \mathbb{Z}$ - liczby naturalne odpowiadające numerowi wektorów w tablicy prawdy funkcji, $B_i^{P_F}$ - i-ty blok podziału P_F. Oznacza to, że do bloku podziału charakterystycznego należą wartości reprezentujące wektory, którym przypisywana jest taka sama wartość funkcji F. Co istotne, dla dowolnej funkcji generowania indeksów podział ten ma zawsze postać: $P_F = \{\overline{1}; \overline{2}; \ldots; \overline{K}\}$. Odpowiada on tzw. podziałowi zerowemu Π_0 [ŁB15], czyli najmniejszemu możliwemu podziałowi zbioru $S = \{1, 2, \ldots, K\}$.

Przykład 2.4. Niech dana będzie funkcja przedstawiona w tabeli 2.1. Dla funkcji tej podziały wejściowe, tzn. podziały względem poszczególnych zmiennych wejściowych x_i ($i = [1, 5]$), są następujące:

- $P_1 = \{\overline{1}; \overline{2, 3, 4}\}$,
- $P_2 = \{\overline{1, 2}; \overline{3, 4}\}$,
- $P_3 = \{\overline{1, 2}; \overline{3, 4}\}$,
- $P_4 = \{\overline{1, 4}; \overline{2, 3}\}$,
- $P_5 = \{\overline{2, 3}; \overline{1, 4}\}$.

Poprzez P_i oznaczono podział względem zmiennej wejściowej x_i. Wartości przyjmowane przez zmienną x_i dla wektorów z tablicy prawdy funkcji określają bloki poszczególnych podziałów. Na przykład, $P_4 = \{\overline{1, 4}; \overline{2, 3}\}$. Oznacza to, że dla wektorów numer 1 i 4 wartość x_4 równa jest 0. Z kolei dla wektorów 2 i 3 - $x_4 = 1$. Na podstawie postaci tego podziału wiadomo, że jeżeli wartość x_4 w otrzymanym wektorze wejściowym $\overrightarrow{v}$ ($\overrightarrow{v} \in D^N$) równa jest zero, to wartość funkcji $F(X)$ równa będzie 1 albo 4.

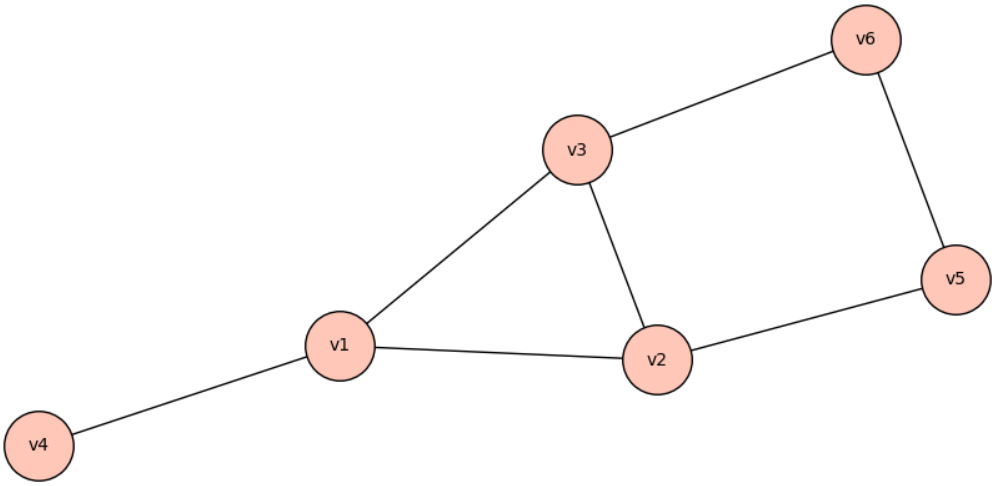
Podział charakterystyczny analizowanej funkcji ma postać $P_F = \{\overline{1}; \overline{2}; \overline{3}; \overline{4}\}$. △

2.3 Elementy teorii grafów

Algorytmy teorii grafów [Cor+09] często znajdują zastosowanie w syntezie logicznej układów cyfrowych [CYP89; Raw+97; Kan12; ŁB15; BŁP16; ŁM19]. W ramach niniejszej pracy wykorzystane zostaną one do znajdowania nierozłącznej dekompozycji funkcjonalnej, jak opisano w rozdziale 5. W związku z tym, w tym podrozdziale przedstawiono podstawowe zagadnienia związane z teorią grafów.

Grafem nazywa się parę $G = (V, E)$, gdzie V - niepusty skończony zbiór wierzchołków, a E - skończony zbiór krawędzi, tzn. par nieuporządkowanych $e = (v_i, v_j)$ takich, że $v_i, v_j \in V$. Liczbę wierzchołków oraz liczbę krawędzi oznaczono odpowiednio $|V|$ oraz $|E|$. Jeżeli graf nie posiada pętli, tzn. $\forall e = (v_i, v_j) \in E, i, j = [1, |V|] : v_i \neq v_j$ oraz nie posiada krawędzi wielokrotnych, tzn. $\forall e_i, e_j \in E, i, j = [1, |E|] : i \neq j \Rightarrow e_i \neq e_j$, to graf nazywa się prostym. W dalszych rozważaniach wykorzystywane będą jedynie grafy tego typu. Przykład grafu, wykorzystywanego w dalszej części rozprawy, przedstawiono na rysunku 2.1.

Dodatkowym pojęciem przydatnym w dalszej części pracy jest stopień grafu. Do jego zdefiniowania niezbędne jest pojęcia stopnia wierzchołka $v \in V$, którym nazywa się liczbę krawędzi incydentnych do tego wierzchołka, tzn. liczbę krawędzi $e = (v_i, v_j) \in E$ takich,



Rysunek 2.1: Przykładowy graf G

że $v_i = v \vee v_j = v$. Stopień grafu równy jest maksymalnej wartości stopnia wierzchołka w rozpatrywanym grafie. Stopień ten oznaczono przez $\Delta(G)$.

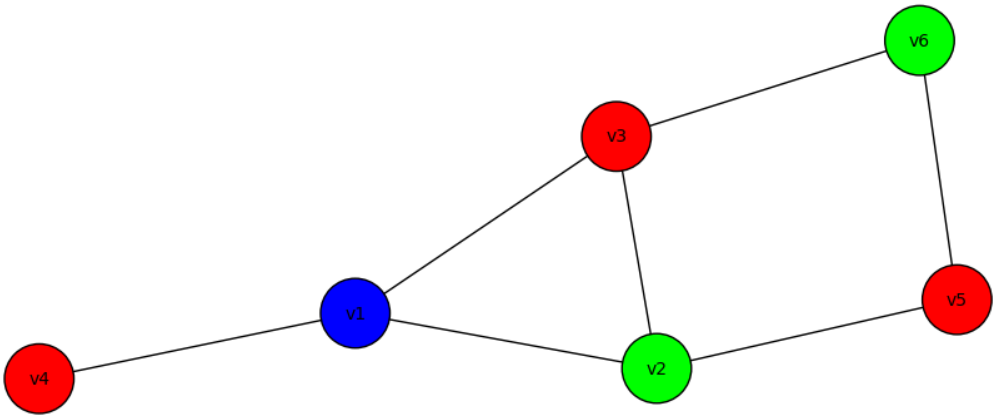
Istotnym problemem w teorii grafów jest problem znalezienia kolorowania grafu. Poszukiwane jest takie przyporządkowanie kolorów do wierzchołków grafu G, aby żadna para wierzchołków przyległych nie miała przypisanego tego samego koloru. Poprzez wierzchołki przyległe rozumie się wierzchołki $v_i, v_j \in V$ takie, że $\exists e \in E : e = (v_i, v_j)$. Kluczowym zagadnieniem jest określenie minimalnej liczby kolorów $\chi(G)$, która zapewnia prawidłowe pokolorowanie. Liczbę tę nazywa się liczbą chromatyczną grafu. W literaturze zaproponowano wiele algorytmów rozwiązujących ten problem [Wer90; Gal+13; AB16]. Do znalezienia pokolorowania wykorzystać można m.in. pojęcie maksymalnych zbiorów niezależnych [ŁB15]. Znalezienie liczby chromatycznej grafu jest jednakże problemem NP-trudnym [GJS74]. Z tego powodu często stosowane są algorytmy heurystyczne, które gwarantują niższą złożoność obliczeniową kosztem potencjalnie gorszej jakości generowanego rozwiązania.

Innym istotnym pojęciem w teorii grafów jest pojęcie kliki. Kilką nazywa się podzbiór wierzchołków $C \subseteq V$ taki, że każde dwa wierzchołki są przyległe. Klikę, która nie jest zawarta w żadnej innej klice i dodatkowo jest najliczniejsza nazywa się największą. Rząd największej kliki (ang. *clique number*) oznaczono w niniejszej rozprawie przez $\omega(G)$. Problem znalezienia maksymalnej kliki w grafie jest problemem NP-zupełnym. W literaturze zostało zaproponowanych wiele algorytmów [WH15] rozwiązujących ten problem, zarówno dokładnych, jak i heurystycznych. Nie istnieje jednak algorytm działający w czasie wielomianowym. Najszybszy algorytm charakteryzuje się złożonością czasową $O(2^{0.249 \cdot |V|})$ [Rob01].

Przykład 2.5. Graf przedstawiony na rysunku 2.1 jest grafem, w którym zbiory wierzchołków i krawędzi mają następującą postać:

- $V = \{v_1, v_2, \ldots, v_6\}$,
- $E = \{(v_1, v_2), (v_2, v_3), (v_1, v_3), (v_2, v_5), (v_3, v_6), (v_1, v_4), (v_5, v_6)\}$.

Największą kliką w tym grafie jest $C = \{v_1, v_2, v_3\}$. Dla tego grafu wyznaczyć można następujące wartości: $\Delta(G) = 3$ oraz $\omega(G) = 3$. Przykładowe pokolorowanie grafu przedstawiono na rysunku 2.2. Jak łatwo można zauważyć, liczba chromatyczna grafu wynosi $\chi(G) = 3$.



Rysunek 2.2: Przykładowe pokolorowanie grafu G

△

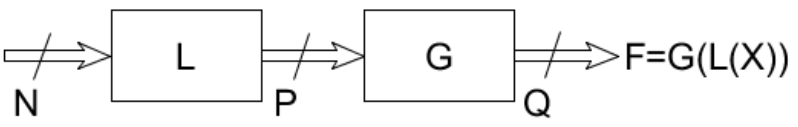
Rozdział 3

Dekompozycja liniowa

W niniejszym rozdziale przedstawiono wyniki badań nad zastosowaniem dekompozycji liniowej do syntezy logicznej funkcji generowania indeksów. W pierwszej kolejności omówiono model dekompozycji liniowej oraz rozwiązania zaprezentowane w literaturze. Następnie przedstawiono wyniki własnych prac nad rozwinięciem iteracyjnej metody wykorzystującej zbiory niezgodności. Zaproponowano dwie metody wyboru dekompozycji i przeanalizowano ich wpływ na efektywność syntezy. Przedstawione wyniki dowodzą ich użyteczności w realizacji funkcji generowania indeksów.

3.1 Model dekompozycji

Metodą umożliwiającą efektywną syntezę funkcji generowania indeksów w strukturach z pamięcią RAM lub ROM jest dekompozycja liniowa. Funkcja wejściowa $F(X)$ przedstawiana jest jako złożenie dwóch funkcji: funkcji liniowej L oraz funkcji ogólnej G, która często jest nieliniowa, tzn. $F(X) = G(L(X))$. Funkcja L zapewnia redukcję liczby zmiennych z N do P ($P < N$). Schemat dekompozycji liniowej przedstawiono na rysunku 3.1. Funkcja L realizowana jest zazwyczaj z wykorzystaniem rejestrów, multiplekserów oraz bramek XOR, podczas gdy funkcja G - z wykorzystaniem pamięci RAM lub ROM [ŁM18; Sas20]. Celem dekompozycji liniowej jest znalezienie takiej funkcji L, że liczba zmiennych wejściowych do funkcji G jest minimalna, tzn. wartość P jest minimalna.



Rysunek 3.1: Schemat dekompozycji liniowej

Rozmiar pamięci do realizacji funkcji G wynosi:

$$MEM_{lin} = 2^P \cdot Q \tag{3.1}$$

Koszt realizacji funkcji liniowej L (wynoszący $N \cdot P$) jest pomijany w dalszych rozważaniach.

Ze względu na to, że dla funkcji generowania indeksów zachodzi $K \ll 2^N$, to zastosowanie dekompozycji liniowej prowadzi do znacznej redukcji zajętości pamięci. W związku z tym, że $p \leqslant GG_S$, rozmiar pamięci można asymptotycznie określić jako $O(MEM_{lin}) = O(K^2 logK)$ [Sas20].

W literaturze zaproponowano szereg metod pozwalających na znalezienie dekompozycji liniowej funkcji generowania indeksów. Większość z nich to metody heurystyczne. Do najważniejszych z nich zaliczyć można metody wykorzystujące:

- tablice dekompozycji [Sas08],
- miarę niejednoznaczności funkcji [Sas12],
- probabilistykę [SZP12; SB18],
- autokorelację [Sas13],
- macierz niezgodności [SUI14],
- mechanizm iteracyjnego poprawiania wyniku [Sas15],
- problem spełnialności [SFI15],
- metody algebraiczne [Ast+16; ASA16; Ast+17; HPC19],
- zbalansowane drzewa decyzyjne [NSB16; NSB17a],
- metodę podziału i ograniczeń [NSB17b],
- binarne diagramy decyzyjne [NSB18].

Przegląd metod dostępny jest w literaturze [Sas17; ŁM18].

W ramach niniejszego rozdziału przedstawiono wyniki prac nad rozwinięciem metody zaproponowanej przez zespół pod kierunkiem prof. Łuby [ŁBJ16; ŁPZ16; Kow17]. Wykorzystuje ona pojęcie zbioru niezgodności do znalezienia dekompozycji liniowej.

3.2 Dekompozycja wykorzystująca zbiór niezgodności

Poprzez $\overrightarrow{v}_i$ oznaczono wektor należący do zbioru wektorów rejestrowanych D^N taki, że $F(\overrightarrow{v}_i) = i$. Zbiorem niezgodności wektorów $\overrightarrow{v}_p$ oraz $\overrightarrow{v}_q$ nazywamy zbiór:

$$C_{p,q} = \{x_s \in X : x_s(\overrightarrow{v}_p) \neq x_s(\overrightarrow{v}_q)\} \tag{3.2}$$

gdzie $x_s(\overrightarrow{v}_p) \in \{0,1\}$ oznacza wartość zmiennej x_s w wektorze $\overrightarrow{v}_p$. Należy zauważyć, że jest to uproszczona definicja tego zbioru [ŁBJ16] wynikająca z faktu, że dla funkcji generowania indeksów zachodzi własność:

$$\forall_{\overrightarrow{v}_p, \overrightarrow{v}_q, p \neq q} : F(\overrightarrow{v}_p) \neq F(\overrightarrow{v}_q) \tag{3.3}$$

Rodzinę zbiorów $C = (C_{1,1}, C_{1,2}, \ldots, C_{K-1,K})$ dla wszystkich wartości $p = [1, K-1], q = [2, K], p < q$ oznacza się poprzez RC_{pq}. W celu ułatwienia dalszej analizy wprowadzono dodatkowe oznaczenie RC^a_{pq} reprezentujące ograniczenie rodziny zbiorów niezgodności do zbiorów o liczności a. Dopełnienie RC^a_{pq} względem rodziny wszystkich podzbiorów o liczności a oznaczono przez $COM(RC^a_{pq})$. Analogicznie, $COM(RC_{pq})$ oznacza dopełnienie względem rodziny wszystkich podzbiorów o liczności od 1 do N.

Algorytm znajdowania dekompozycji liniowej z wykorzystaniem zbiorów niezgodności bazuje na pojęciu funkcji rozdzielającej. Do jego zdefiniowania wykorzystywany jest rachunek podziałów przedstawiony we wcześniejszym rozdziale. Funkcja g jest funkcją rozdzielającą wektory $\overrightarrow{v}_p$ oraz $\overrightarrow{v}_q$ wtedy i tylko wtedy, gdy należą one do różnych bloków podziału P_g. Możliwe jest zatem obliczenie podziału P_g dla dowolnej funkcji o skończonej liczbie zmiennych wejściowych, a następnie usunięcie tych zbiorów $C_{p,q}$, dla których funkcja g jest funkcją rozdzielającą. Pozwala to przedstawić funkcję F jako funkcję o zredukowanej liczbie zmiennych, do których należy też wyjście z funkcji g. Na przykład dla dwuargumentowej funkcji g uzyskujemy następującą reprezentację:

$$F = H(X \backslash \{x_i, x_j\}, g(x_i, x_j)) \tag{3.4}$$

gdzie H to funkcja ze zredukowaną liczbą argumentów. Liczba argumentów jest mniejsza o jeden w stosunku do pierwotnej reprezentacji.

W celu sformułowania metody dekomponowania funkcji wykorzystać można twierdzenie 3.1 [ŁR92]. Prowadzi ono do testu 3.1 [ŁBJ16] weryfikującego istnienie dekompozycji z wykorzystaniem alternatywy wykluczającej (tzn. funkcji XOR).

Twierdzenie 3.1. *Funkcja $g = x_i \oplus x_j$ jest funkcją rozdzielającą parę (p, q) wtedy i tylko wtedy, gdy $\{x_i, x_j\} \notin C_{p,q} \wedge |\{x_i, x_j\} \cap C_{p,q}| = 1$.*

Test 3.1. Funkcja $g = x_i \oplus x_j$ jest dekompozycją funkcji F wtedy i tylko wtedy, gdy $\{x_i, x_j\} \notin RC^2_{pq}$.

Poszukiwanie dekompozycji sprowadzić można zatem do iteracyjnego poszukiwania skończonych zbiorów zmiennych wejściowych takich, że należą one do $COM(RC_{pq})$. Metoda kończy działanie wtedy, gdy $COM(RC_{pq}) = \emptyset$. Przedstawione twierdzenie i test można uogólnić na przypadek dekompozycji wielokrotnej [ŁBJ16]. Na przykład: para funkcji

$g_1 = x_i \oplus x_j$ i $g_2 = x_j \oplus x_k$ jest dekompozycją funkcji F wtedy i tylko wtedy, gdy $\{x_i, x_j, x_k\} \in COM(RC^3_{pq})$. Z kolei para funkcji $g_1 = x_i \oplus x_j$ i $g_3 = x_k \oplus x_l$ jest dekompozycją funkcji F wtedy i tylko wtedy, gdy $\{x_i, x_j, x_k, x_l\} \in COM(RC^4_{pq})$.

Wykorzystując powyższe ustalenia, można zaproponować algorytm dokładny znajdowania dekompozycji liniowej. Wymagałby on jednak iteracyjnego przeglądania wszystkich podzbiorów należących do $COM(RC_{pq})$. W związku z tym, jego efektywność byłaby niewielka. W związku z powyższym przeanalizowano możliwość zastosowania podejścia heurystycznego. Wykorzystuje ono następujące operacje:

1. redukcję zmiennych funkcji wejściowej,
2. wygenerowanie rodziny zbiorów RC_{pq},
3. iteracyjny wybór funkcji rozdzielającej dopóki $COM(RC_{pq}) \neq \emptyset$.

Krok 1. realizowany jest de facto przed wykonaniem algorytmu. Wynika on z założenia, że w dekompozycji liniowej wykorzystywane są jedynie minimalnoargumentowe reprezentacje funkcji F [ŁPZ16; MŁ17]. Do jej znalezienia wykorzystać można jeden z algorytmów dostępnych w literaturze [BŁ14; JBK14]. Krok ten można potraktować jako opcjonalny, ale często istotnie redukuje on czasochłonność kroków numer 2 i 3. Co interesujące, różne reprezentacje funkcji mogą prowadzić do różnej liczby zmiennych P po zastosowaniu algorytmu dekompozycji liniowej [MoŁ19].

Wygenerowanie rodziny zbiorów RC_{pq} również może być potraktowane jako operacja przygotowująca dane do działania algorytmu. Podobne założenia przyjęto w literaturze [SUI14]. Do reprezentacji zbiorów $C_{p,q}$ wykorzystać można macierz rozróżnialności [MŁ19a]. Przykład macierzy, dla danych z tabeli 3.2, przedstawiono w tabeli 3.3. W ogólności macierz ta wykorzystuje $O(N \cdot K^2)$ pamięci. Oznacza to, że metoda wymaga dużo pamięci dla funkcji o stosunkowo licznym zbiorze wektorów rejestrowanych.

W celu zwiększenia efektywności czasowej działania algorytmu dla dużych wartości K, zaproponowano grupowanie uzyskanych zbiorów według ich liczności. W przypadku reprezentacji z wykorzystaniem macierzy, grupowanie odbywa się na podstawie wagi Hamminga wierszy. Pozwala to efektywnie poszukiwać dekompozycji w poszczególnych iteracjach w sytuacji, gdy nie będzie istnieć dekompozycja z relatywnie mało licznym zbiorem $C_{p,q}$. Co więcej, powtarzające się postaci zbiorów, czyli wiersze macierzy, są usuwane. Wynika to z faktu, że im większa wartość K, tym częściej pojawiać się będą zbiory o postaci takiej samej, jak już wcześniej wygenerowane. Na przykład dla funkcji *4 z 20* aż 98.6% wierszy ma tę samą postać, co dodany już wcześniej wiersz [SUI14].

Krok 3. realizowany jest iteracyjnie w dwóch etapach. Najpierw poszukiwana jest funkcja rozdzielająca g, która umożliwia dekompozycję liniową funkcji G. Jeżeli funkcja ta

zostanie znaleziona, to następuje modyfikacja macierzy rozróżnialności. Krok ten powtarzany jest do uzyskania $COM(RC_{pq}) = \emptyset$. Maksymalna liczba iteracji równa jest co najwyżej $N - GD$.

W celu znalezienia funkcji rozdzielającej, realizowany jest algorytm 3.1. Przez B_a oznaczono zbiór wektorów o wadze Hamminga równej a, natomiast B_s oznacza maksymalną wagę Hamminga, czyli liczbę zmiennych analizowanej funkcji. Dla kolejnych wartości a następuje sprawdzenie, czy $|B_a| = \binom{B_s}{a}$. Jeżeli równość ta jest spełniona, to zachodzi $COM(RC^a_{pq}) = \emptyset$. Oznacza to, że nie istnieje dekompozycja z wykorzystaniem a-elementowego zbioru $C_{p,q}$ i konieczne jest zatem poszukiwanie zbioru o liczności o jeden większej. Dzięki pogrupowaniu wierszy macierzy rozróżnialności nie ma zatem konieczności przeglądania $\binom{n}{a}$ wierszy w celu stwierdzenia, że zbiór $COM(RC^a_{pq})$ jest zbiorem pustym. Gdy równość nie jest spełniona, to poszukiwanie funkcji rozdzielającej przy wykorzystaniu reprezentacji z wykorzystaniem macierzy rozróżnialności sprowadza się do znalezienia wektora, o określonej wadze Hamminga, który nie występuje w macierzy.

Należy zauważyć, że w kolejnych iteracjach działania algorytmu wartość B_s będzie coraz mniejsza. Wynika to z iteracyjnego dekomponowania wejściowej funkcji F. W średnim przypadku prowadzi to do sytuacji, że każda kolejna iteracja algorytmu wykonuje się w czasie krótszym od poprzedniej.

Jeżeli $|B_a| \neq \binom{B_s}{a}$, to $COM(RC^a_{pq}) \neq \emptyset$. Istnieje zatem dekompozycja liniowa z funkcją rozdzielającą wyznaczoną na podstawie a–elementowego zbioru $C_{p,q}$. Sposób wyboru funkcji, tzn. sposób działania funkcji *choose_decomp*(), przedstawiony został w rozdziale 3.2.1. W sytuacji, gdy $COM(RC_{pq}) = \emptyset$ algorytm zwraca wartość $None$. Kończy to działanie algorytmu poszukiwania dekompozycji liniowej.

Algorytm 3.1 Poszukiwanie dekompozycji

Input: Macierz rozróżnialności po usunięciu duplikatów
Output: Znaleziona funkcja rozdzielająca

```
1: a ← 2, n ← Bs
2: while a ⩽ n do
3:     if |B_a| = (n a) then
4:         a ← a + 1
5:     else
6:         choose_decomp(B_a)
7:     end if
8: end while
9: return None
```

W sytuacji, gdy znaleziono funkcję rozdzielającą, konieczne jest dokonanie modyfikacji macierzy rozróżnialności. Następuje to na podstawie postaci znalezionej funkcji.

Przykład 3.1. Niech dana jest funkcja przedstawiona w tabeli 3.1. Jest to funkcja generowania indeksów, dla której $K = 4$ oraz $N = 5$.

Tabela 3.1: Funkcja generowania indeksów ($N = 5$, $K = 4$)

x_1	x_2	x_3	x_4	x_5	$F(X)$
1	1	0	1	1	1
1	0	1	1	1	2
1	0	0	0	1	3
1	0	0	1	1	4

W kroku 1. algorytm redukcji argumentów usuwa zmienne x_1 oraz x_5 z tablicy prawdy funkcji. Łatwo zauważyć, że podzbiór zmiennych wejściowych postaci $\{x_2, x_3, x_4\}$ jest wystarczający do reprezentacji tej funkcji. W związku z tym, w dalszych obliczeniach wykorzystano jedynie te trzy zmienne.

W związku z tym, że $K = 4$ w 2. kroku algorytmu należy wyznaczyć $\binom{4}{2} = 6$ zbiorów $C_{p,q}$. Wyznaczone postaci zbiorów przedstawiono w tabeli 3.2, a macierz rozróżnialności w tabeli 3.3.

Tabela 3.2: Postaci zbiorów $C_{p,q}$

p, q	$C_{p,q}$
1,2	$\{x_2, x_3\}$
1,3	$\{x_2, x_4\}$
1,4	$\{x_2\}$
2,3	$\{x_3, x_4\}$
2,4	$\{x_3\}$
3,4	$\{x_4\}$

Tabela 3.3: Macierz rozróżnialności

(a) Przed grupowaniem

x_2	x_3	x_4	(p, q)
1	1	0	$(1, 2)$
1	0	1	$(1, 3)$
1	0	0	$(1, 4)$
0	1	1	$(2, 3)$
0	1	0	$(2, 4)$
0	0	1	$(3, 4)$

(b) Po grupowaniu

$\lvert C_{p,q}\rvert$	x_2	x_3	x_4	(p, q)
1	1	0	0	$(1, 4)$
	0	1	0	$(2, 4)$
	0	0	1	$(3, 4)$
2	1	1	0	$(1, 2)$
	1	0	1	$(1, 3)$
	0	1	1	$(2, 3)$

Co istotne, macierz ta zawiera wszystkie możliwe wektory 3-bitowe o wadze Hamminga 2. W związku z tym, nie istnieje funkcja postaci $g = x_i \oplus x_j$, która byłaby dekompozycją funkcji F. Jednakże, $\{x_2, x_3, x_4\} \in COM(RC^3_{pq})$, czyli para funkcji $g_1 = x_2 \oplus x_3$ oraz $g_2 = x_3 \oplus x_4$ jest dekompozycją funkcji F i wybierana jest w trzecim kroku algorytmu

jako funkcja rozdzielająca. W przykładzie zastosowano ograniczenie 2.2, a otrzymany wynik jest optymalny ($GD_i = 2$), więc algorytm kończy działanie. Ostateczną postać funkcji przedstawiono w tabeli 3.4. Liczba zmiennych została zredukowana z pięciu do zaledwie dwóch.

Tabela 3.4: Funkcja po zastosowaniu algorytmu dekompozycji liniowej

g_1	g_2	$F(X)$
1	1	1
1	0	2
0	0	3
0	1	4

△

3.2.1 Metody wyboru dekompozycji

Metoda wyboru funkcji rozdzielających w poszczególnych iteracjach ma kluczowy wpływ na efektywność algorytmu. Przedstawiony wcześniej algorytm wymusza konieczność przeanalizowania co najwyżej $\binom{Bs}{a}$ możliwych postaci zbioru $C_{p,q}$ w celu znalezienia takiego, który należy do $COM(RC^a_{pq})$. W tym celu analizowane są kolejne kombinacje bez powtórzeń.

W ramach realizowanych prac zastosowano podejście polegające na wyborze najmniejszej zgodnie z porządkiem leksykograficznym kombinacji reprezentującej funkcję rozdzielającą a-argumentową [MŁ17]. Na przykład, kombinacja $\{1, 2\}$ reprezentuje funkcję rozdzielającą, której argumentami jest pierwsza i druga zmienna wejściowa z tablicy prawdy dekomponowanej funkcji. Podejście to nazwano First-Fit. Dla tego podejścia, w najlepszym przypadku pierwszy z analizowanych zbiorów należy do $COM(RC^a_{pq})$, co kończy iterację algorytmu. W najgorszym, konieczne będzie przejrzenie wszystkich kombinacji, gdyż dopiero ostatnia analizowana należeć będzie do tego zbioru.

Niestety, metoda ta nie gwarantuje uzyskania wartości P takiej, jak w przypadku najlepszych algorytmów dekompozycji [ŁM18]. W związku z tym za celowe uznano zaproponowanie innej metody wyboru funkcji w poszczególnych iteracjach algorytmu [MŁ18; MŁ19a]. Metodę tę nazwano MinR.

W celu wyboru funkcji rozdzielającej wyznaczana jest wartość $R(d)$, definiowana jako liczba różnych wektorów, powstałych po usunięciu z wektorów $\overrightarrow{w}_i \in D^{Bs}$ zmiennych ze zbioru d. Proponowana metoda zakłada wybór zbioru $d^* \in COM(RC^a_{pq})$ takiego, że:

$$R(d^*) = \min_{d \in COM(RC^a_{pq})} R(d) \tag{3.5}$$

W przypadku, gdy wartość $R(d^*)$ uzyskana została dla więcej niż jednego zbioru d, do dalszych rozważań wybierany jest pierwszy znaleziony zbiór. Należy zauważyć, że metoda ta zawsze wymaga przeanalizowania wszystkich możliwych zbiorów $C_{p,q}$, należących do $COM(RC^a_{pq})$, w celu znalezienia takiej postaci, dla której uzyskana zostanie wartość $R(d^*)$. Co więcej, dla każdego z tych zbiorów należy wyznaczyć wartość $R(d)$, co wymaga przeanalizowania K wierszy tablicy prawdy dekomponowanej funkcji. Metoda ta ma zatem o wiele gorszą złożoność pesymistyczną oraz średnią w porównaniu do metody First-Fit.

Przykład 3.2. W celu zobrazowania różnic w działaniu obu metod przedstawiono ich działanie na przykładzie funkcji *1 z 7*. W tabeli 3.5a przedstawiono tablicę prawdy tej funkcji po operacji redukcji argumentów. Jak widać, zmienna x_7 została zredukowana. W związku z tym, dla analizowanej funkcji $N = 6$ oraz $K = 7$. Tablica prawdy po wspomnianej operacji zawiera sześć wektorów o wadze Hamminga jeden oraz jeden wektor zerowy. Postać macierzy rozróżnialności, uzyskanej na podstawie tejże tablicy prawdy, przedstawiono w tabeli 3.6.

Tabela 3.5: Funkcja *1 z 7*

(a) Po operacji redukcji argumentów

x_1	x_2	x_3	x_4	x_5	x_6	$F(X)$
1	0	0	0	0	0	1
0	1	0	0	0	0	2
0	0	1	0	0	0	3
0	0	0	1	0	0	4
0	0	0	0	1	0	5
0	0	0	0	0	1	6
0	0	0	0	0	0	7

(b) Po pierwszej iteracji algorytmu dekompozycji

x_4	x_5	x_6	g_1	g_2	$H_1(X')$
0	0	0	1	0	1
0	0	0	1	1	2
0	0	0	0	1	3
1	0	0	0	0	4
0	1	0	0	0	5
0	0	1	0	0	6
0	0	0	0	0	7

Jak łatwo zauważyć, w macierzy znajdują się wszystkie możliwe wektory 6-bitowe o wadze Hamminga równej 2. Oznacza to, że $COM(RC^2_{pq}) = \emptyset$. Nie istnieje zatem funkcja rozdzielająca postaci $g = x_i \oplus x_j$. Jednakże $RC^3_{pq} = \emptyset$. W związku z tym, zbiór $COM(RC^3_{pq})$ zawiera wszystkie możliwe zbiory trójelementowe $\{x_i, x_j, x_k\}$ $(i \neq j \neq k)$. Podejście First-Fit powoduje wybór zbioru $\{x_1, x_2, x_3\}$, ponieważ kombinacja $(1, 2, 3)$ jest najmniejsza zgodnie z porządkiem leksykograficznym. W związku z tym istnieje dekompozycja liniowa z parą funkcji: $g_1 = x_1 \oplus x_2$ oraz $g_2 = x_2 \oplus x_3$. Funkcję F można przedstawić zatem w następującej postaci:

$$F = H_1(X') = H_1(x_4, x_5, x_6, x_1 \oplus x_2, x_2 \oplus x_3)$$

Tabela 3.6: Macierz rozróżnialności dla zredukowanej funkcji *1 z 7*

$\|C_{p,q}\|$	x_1	x_2	x_3	x_4	x_5	x_6	(p,q)
1	1	0	0	0	0	0	(1,7)
	0	1	0	0	0	0	(2,7)
	0	0	1	0	0	0	(3,7)
	0	0	0	1	0	0	(4,7)
	0	0	0	0	1	0	(5,7)
	0	0	0	0	0	1	(6,7)
2	1	1	0	0	0	0	(1,2)
	1	0	1	0	0	0	(1,3)
	1	0	0	1	0	0	(1,4)
	1	0	0	0	1	0	(1,5)
	1	0	0	0	0	1	(1,6)
	0	1	1	0	0	0	(2,3)
	0	1	0	1	0	0	(2,4)
	0	1	0	0	1	0	(2,5)
	0	1	0	0	0	1	(2,6)
	0	0	1	1	0	0	(3,4)
	0	0	1	0	1	0	(3,5)
	0	0	1	0	0	1	(3,6)
	0	0	0	1	1	0	(4,5)
	0	0	0	1	0	1	(4,6)
	0	0	0	0	1	1	(5,6)

Tablicę prawdy dla funkcji po pierwszej iteracji algorytmu przedstawiono w tabeli 3.5b. Należy zwrócić uwagę, że liczba zmiennych zmniejszyła się o jedną (z sześciu do pięciu) w porównaniu do postaci funkcji na wejściu algorytmu.

W przypadku podejścia MinR konieczne jest przeanalizowanie wszystkich możliwych kombinacji wartości i, j oraz k w celu znalezienia zbioru o najmniejszej wartości $R(d)$. Pierwszym analizowanym zbiorem jest $d_1 = \{x_1, x_2, x_3\}$. Jak łatwo zauważyć:

$$R(d_1) = |\{(0,0,0),(1,0,0),(0,1,0),(0,0,1)\}| = 4$$

Wyznaczenie wartości $R(d)$ dla pozostałych podzbiorów należących do $COM(RC^3_{pq})$ skutkuje uzyskaniem takiej samej wartości. W związku z tym, podejście to również skutkuje wyborem zbioru $\{x_1, x_2, x_3\}$ w tej iteracji algorytmu. Funkcja po pierwszej iteracji algorytmu ma zatem taką samą postać jak w przypadku podejścia First-Fit.

Macierz rozróżnialności zmodyfikowana po pierwszej iteracji algorytmu przedstawiona została w tabeli 3.7. Jak widać, ponownie $COM(RC^2_{pq}) = \emptyset$. W związku z tym, poszukiwana jest dekompozycja ze zbiorem trzyelementowym. Przy zastosowaniu podejścia First-Fit następuje sprawdzenie, czy wektor $(1,1,1,0,0)$ występuje w macierzy rozróżnialności. Ze względu na to, że wektor ten nie występuje stwierdzić można przynależność zbioru $\{x_4, x_5, x_6\}$ do zbioru $COM(RC^3_{pq})$. Oznacza to, że istnieje dekompozycja z parą

Tabela 3.7: Macierz rozróżnialności w drugiej iteracji algorytmu

$\|C_{p,q}\|$	x_4	x_5	x_6	g_1	g_2	(p,q)
1	0	0	0	0	1	(1,2), (3,7)
	0	0	0	1	0	(1,7), (2,3)
	1	0	0	0	0	(4,7)
	0	1	0	0	0	(5,7)
	0	0	1	0	0	(6,7)
2	0	0	0	1	1	(1,3), (2,7)
	1	0	0	1	0	(1,4)
	0	1	0	1	0	(1,5)
	0	0	1	1	0	(1,6)
	1	0	0	0	1	(3,4)
	0	1	0	0	1	(3,5)
	0	0	1	0	1	(3,6)
	1	1	0	0	0	(4,5)
	1	0	1	0	0	(4,6)
	0	1	1	0	0	(5,6)
3	1	0	0	1	1	(2,4)
	0	1	0	1	1	(2,5)
	0	0	1	1	1	(2,6)

funkcji: $g_3 = x_4 \oplus x_5$ oraz $g_4 = x_5 \oplus x_6$. Funkcję F można przedstawić zatem w następującej postaci:

$$F = H_2(x_1 \oplus x_2, x_2 \oplus x_3, x_4 \oplus x_5, x_5 \oplus x_6)$$

W kolejnej iteracji uzyskany zbiór $COM(RC_{pq})$ jest zbiorem pustym. Oznacza to, że funkcja nie może zostać dalej zdekomponowana z wykorzystaniem podejścia First-Fit. W wyniku uzyskano zatem reprezentację z wykorzystaniem funkcji czteroargumentowej. Należy zwrócić uwagę, że dla kodera *1 z 7* uzyskany wynik jest najlepszym możliwym, wynikającym z ograniczenia GD.

W przypadku podejścia MinR, dla zbioru wyznaczonego za pomocą podejścia First-Fit uzyskuje się $R(\{x_4, x_5, x_6\} = 4$. Istnieje jednak sześć zbiorów należących do $COM(RC^3_{pq})$, dla których wartość ta wynosi trzy. Najmniejszą kombinacją reprezentującą jeden z takich zbiorów jest $\{1, 2, 4\}$. Oznacza to, że istnieje dekompozycja z parą funkcji postaci: $g_3^{'} = x_4 \oplus x_5$ oraz $g_4^{'} = x_5 \oplus g_1 = x_1 \oplus x_2 \oplus x_5$. W wyniku uzyskano zatem inną dekompozycję, niż w przypadku podejścia First-Fit. Funkcję F można przedstawić jako:

$$F = H_2^{'}(x_6, g_2, g_3^{'}, g_4^{'}) = H_2^{'}(x_6, x_2 \oplus x_3, x_4 \oplus x_5, x_1 \oplus x_2 \oplus x_5)$$

Należy zwrócić uwagę, że obie otrzymane funkcje H_2 oraz $H_2^{'}$ są czteroargumentowe, jednak ich postać się różni. W tym wypadku, po zmodyfikowaniu macierzy rozróżnialności, uzyskuje się niepuste dopełnienie rodziny zbiorów RC_{pq}, a mianowicie $COM(RC^3_{pq}) =$

$\{\{x_6, g_2, g_3^{'}\}\}$. Oznacza to, że podejście MinR pozwala znaleźć reprezentację trzyargumentową funkcji F:

$$F = H_3(x_1 \oplus x_2 \oplus x_5, x_2 \oplus x_3 \oplus x_6, x_2 \oplus x_3 \oplus x_4 \oplus x_5)$$

Jego stosowanie powoduje jednak utratę możliwości rozpoznawania wektorów, które nie należą do zbioru wektorów rejestrowanych. Niemniej, w rezultacie uzyskano wynik o mniejszej liczbie zmiennych niż przy zastosowaniu podejścia First-Fit.

Znalezione reprezentacje prowadzą do uzyskania tablic prawdy funkcji przedstawionych w tabeli 3.8. Dla ułatwienia reprezentacji zmienne wejściowe przy podejściach First-Fit oraz MinR oznaczono odpowiednio $Y = \{y_1, y_2, y_3, y_4\}$ oraz $Z = \{z_1, z_2, z_3\}$. Należy zwrócić uwagę, że obie metody pozwoliły na efektywną redukcję liczby zmiennych z początkowych siedmiu zmiennych.

Tabela 3.8: Reprezentacja funkcji *1 z 7* po zastosowaniu proponowanych podejść

(a) Po zastosowaniu podejścia First-Fit

y_1	y_2	y_3	y_4	$H_2(Y)$
1	0	0	0	1
1	1	0	0	2
0	1	0	0	3
0	0	1	0	4
0	0	1	1	5
0	0	0	1	6
0	0	0	0	7

(b) Po zastosowaniu podejścia MinR

z_1	z_2	z_3	$H_3(Z)$
1	0	0	1
1	1	1	2
0	1	1	3
0	0	1	4
1	0	1	5
0	1	0	6
0	0	0	7

△

3.2.2 Uzyskane wyniki

Ocenę efektywności przedstawionego algorytmu, z wykorzystaniem metod First-Fit (FF) oraz MinR, rozpoczęto od przeanalizowania wyników uzyskanych dla funkcji *1 z N*. Zaprezentowano je w tabeli 3.9. Pod uwagę wzięto uzyskaną liczbę zmiennych po dekompozycji oraz czas obliczeń. Dodatkowo przedstawiono wyniki dostępne w literaturze [SZP12; Sas13]. Poprzez Opt oznaczono liczbę zmiennych uzyskaną z wykorzystaniem metody dokładnej [Sas11a].

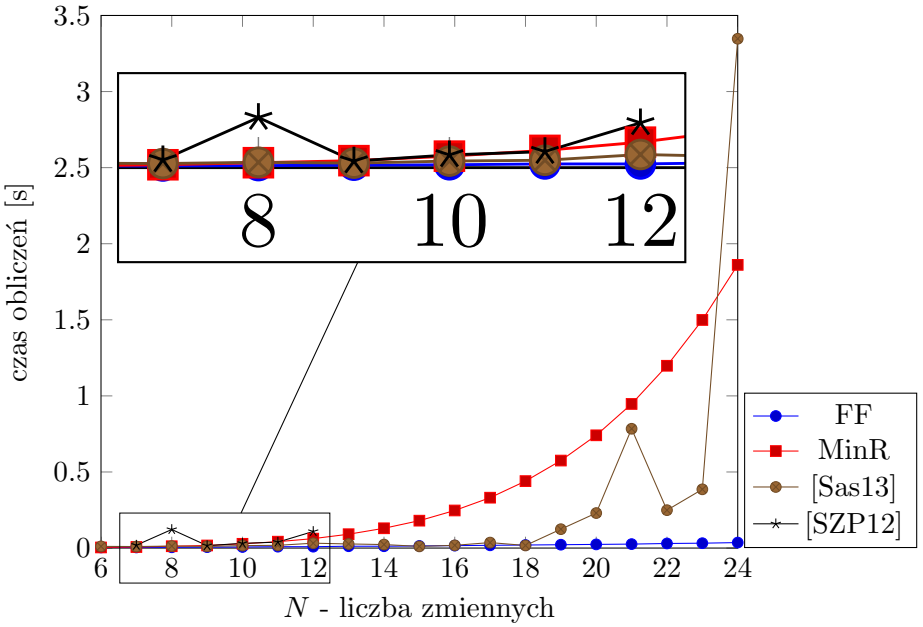
Zastosowanie metody First-Fit pozwoliło znaleźć optymalną reprezentację dla siedmiu spośród analizowanych wartości N. Uzyskane wyniki są jednak niesatysfakcjonujące w porównaniu z metodą probabilistyczną [SZP12]. Z kolei dla dwunastu wartości N

uzyskano wynik niegorszy niż poprzez zastosowanie metody wykorzystującej autokorelację [Sas13]. W przypadku metody MinR uzyskano najlepsze wyniki spośród wszystkich przedstawionych w tabeli algorytmów. Co więcej, jedynie dla $N = 17$ uzyskano nieoptymalną reprezentację funkcji. Niestety, metoda ta cechowała się długim czasem działania. W celu lepszego zobrazowania efektywności czasowej metody, uzyskane czasy obliczeń (w sekundach) przedstawiono dodatkowo na rysunku 3.2.

Tabela 3.9: Wyniki uzyskane dla funkcji *1 z N*

	Liczba zmiennych P					Czas [s]			
N	Opt	FF	MinR	[Sas13]	[SZP12]	FF	MinR	[Sas13]	[SZP12]
6	3	3	3	3	-	0,004	0,004	0,011	-
7	3	4	3	3	3	0,004	0,007	0,010	0,019
8	3	4	3	3	3	0,005	0,012	0,013	0,122
9	4	4	4	4	4	0,006	0,017	0,011	0,016
10	4	4	4	4	4	0,007	0,028	0,016	0,031
11	4	4	4	4	4	0,009	0,042	0,018	0,039
12	4	5	4	4	4	0,009	0,062	0,032	0,109
13	4	5	4	4	-	0,012	0,091	0,027	-
14	4	5	4	4	-	0,012	0,130	0,023	-
15	4	5	4	6	-	0,014	0,180	0,011	-
16	4	4	4	6	-	0,016	0,247	0,018	-
17	4	5	5	5	-	0,018	0,331	0,036	-
18	5	5	5	6	-	0,020	0,440	0,017	-
19	5	5	5	6	-	0,022	0,575	0,124	-
20	5	6	5	5	-	0,024	0,741	0,230	-
21	5	6	5	5	-	0,026	0,947	0,784	-
22	5	6	5	7	-	0,030	1,198	0,249	-
23	5	6	5	7	-	0,032	1,499	0,386	-
24	5	6	5	6	-	0,036	1,861	3,347	-

Jak widać, dla metody MinR czas obliczeń znacząco rośnie wraz ze wzrostem wartości N. Możliwe jest wyznaczenie potęgowej linii trendu $O(N^{4,52})$, przy współczynniku determinacji $R^2 = 0,9966$. Podobną zależność zaobserować można dla metody probabilistycznej. Jej czas działania jest mniejszy dla większości wartości N, jednak dla $N = 21$ różnica jest niewielka, a dla $N = 24$ metoda MinR jest niemal dwa razy szybsza. Dla $8 \leqslant N \leqslant 23$ czas działania zaproponowanej metody jest z kolei dłuższy niż przy zastosowaniu metody wykorzystującej autokorelację. Dla metody First-Fit, czas obliczeń rośnie nieznacznie (linia trendu $O(N^{1,89})$ przy $R^2 = 0,9956$). Co więcej, jest on krótszy niż w przypadku pozostałych algorytmów dla wszystkich wartości N. Należy jednak podkreślić, że w eksperymencie wykorzystano inny sprzęt (Intel Xeon E5-2650v2 2.6Hz, pamięć 64 GB, system operacyjny Windows 7, interpreter języka Python 3.8). Mimo to, przedstawiony wykres dobrze odzwierciedla złożoność metod.



Rysunek 3.2: Średni czas obliczeń dla funkcji *1 z N*

Do oceny efektywności algorytmu dekompozycji liniowej wykorzystano także dostępne w literaturze wyniki dla funkcji *M z 16* oraz *M z 20*. Zestawienie wyników, czyli uzyskanych wartości P z wykorzystaniem poszczególnych metod, przedstawiono w tabelach 3.10 oraz 3.11. Poprzez „-" oznaczono brak wyniku dla określonej wartości M w cytowanej pracy. W obu tabelach zamieszczono również wyznaczoną wartość dolnego ograniczenia na liczbę zmiennych do reprezentacji koderów, czyli wartość GD. Należy podkreślić, że dekompozycja liniowa taka, że $P = GD$ może nie istnieć. Na przykład, dla funkcji *2 z 16* metodą dokładną [NSB18] dowiedziono, że rozwiązanie optymalne nie istnieje. Również dla funkcji *2 z 20* udowodniono, z wykorzystaniem problemu spełnialności (SAT) [SFI15], że najlepszym osiągalnym wynikiem jest $P = 9$ [Sas17]. Na podstawie przedstawionych do tej pory wyników w literaturze można przypuszczać, że dla funkcji *4 z 16* oraz *4 z 20* dekompozycja liniowa z $P = GD$ również nie istnieje.

Mimo że w tabeli przedstawiono również wyniki uzyskane metodami dokładnymi, nie oznacza to, że nie istnieją rozwiązania z mniejszą liczbą argumentów. Wynika to z faktu wprowadzania ograniczenia przez niektórych autorów na maksymalny stopień złożoności[1]. Może zatem istnieć rozwiązanie o mniejszej liczbie zmiennych, ale większym stopniu złożoności.

Podczas analizy otrzymanych wyników należy mieć na uwadze, że zastosowanie algorytmów redukcji zmiennych pozwala zmniejszyć liczbę zmiennych zaledwie o jedną, czyli odpowiednio do 15 i 19 zmiennych.

[1]Pojęcie to zdefiniowano w dalszej części rozdziału

Tabela 3.10: Efektywność algorytmów dekompozycji liniowej dla funkcji *M z 16*

	M			
Praca	1	2	3	4
T. Sasao [Sas12]	6	8	**10**	13
T. Sasao [Sas13]	6	-	-	-
T. Sasao (2-Min) [Sas15]	8	12	13	14
T. Sasao (3-Min) [Sas15]	**5**	11	12	14
T. Sasao, I. Fumishi i Y. Iguchi [SFI15]	**5**	8	**10**	13
S. Nagayama, T. Sasao i J.T. Butler [NSB17b]	**5**	8	**10**	-
S. Nagayama, T. Sasao i J.T. Butler [NSB18]	**5**	8	**10**	-
Metoda First-Fit	**5**	9	12	13
Metoda MinR	**5**	8	**10**	13
Optimum (GD)	**5**	**7**	**10**	**11**

Tabela 3.11: Efektywność algorytmów dekompozycji liniowej dla funkcji *M z 20*

	M				
Praca	1	2	3	4	5
T. Sasao [Sas12]	6	9	**11**	15	17
T. Sasao [Sas13]	**5**	-	-	-	-
T. Sasao, Y. Urano i Y. Iguchi [SUI14]	**5**	10	13	16	-
T. Sasao (2-Min) [Sas15]	10	14	15	17	18
T. Sasao (3-Min) [Sas15]	**5**	14	15	16	18
T. Sasao, I. Fumishi i Y. Iguchi [SFI15]	**5**	9	**11**	15	-
J. Astola et al. [Ast+16]	**5**	9	**11**	15	-
S. Nagayama, T. Sasao i J.T. Butler [NSB16]	6	-	13	-	-
S. Nagayama, T. Sasao i J.T. Butler [NSB17a]	6	-	13	-	-
Metoda First-Fit	6	9	12	16	17
Metoda MinR	**5**	9	**11**	15	17
Optimum (GD)	**5**	**8**	**11**	**13**	**14**

Dla funkcji *1 z 16* obie metody, First-Fit oraz MinR, pozwoliły na uzyskanie wyniku optymalnego. Co więcej, pozwalają one także znaleźć rozwiązanie takie, że $P = GD_i = 4$. Dla funkcji *2 z 16* oraz *3 z 16* reprezentację z mniejszą liczbą zmiennych uzyskano stosując metodę MinR. Co więcej, zgodnie z wcześniejszymi ustaleniami reprezentacje te są najlepszymi, jakie można uzyskać. Potwierdzają to wyniki uzyskane z wykorzystaniem metod dokładnych [NSB17b; NSB18]. Dla $M = 4$ uzyskana liczba zmiennych z wykorzystaniem obu metod jest taka sama, jak najlepszy wynik zaprezentowany w literaturze [Sas12]. Z kolei dla $M = 5$ obie metody pozwoliły znaleźć reprezentację z 14 zmiennymi. Podsumowując, zaproponowane metody pozwalają znaleźć dekompozycje liniowe z mniejszą liczbą zmiennych niż inne metody heurystyczne dla funkcji *M z 16*.

Dla funkcji *M z 20* zaproponowane metody również pozwoliły na efektywną minimalizację liczby zmiennych. Bardzo dobre wyniki otrzymano zwłaszcza dla metody MinR,

gdzie uzyskano takie same wartości jak z wykorzystaniem metody dokładnej [SFI15]. Dla $M \in \{1, 3, 4\}$ metoda First-Fit znajduje rozwiązania z jedną zmienną więcej. Jednak efektywność czasowa tej metody jest o wiele większa. Dla $M = 5$ obie metody pozwoliły uzyskać reprezentację z 17 zmiennymi, przy $GD = 14$. Wynik ten jest identyczny z najlepszym przedstawionym w literaturze.

Należy zauważyć, że w tabeli 3.11 przedstawiono także wyniki uzyskane metodą [SUI14], która podobnie jak algorytm przedstawiony w niniejszej rozprawie wykorzystuje macierz rozróżnialności. Metoda ta znajduje jednak reprezentacje z większą liczbą zmiennych dla analizowanych wartości $M > 1$.

W opracowanym oprogramowaniu eksperymentalnym redukcja argumentów oraz wygenerowanie macierzy rozróżnialności dla metody First-Fit okazują się o wiele bardziej czasochłonne niż znalezienie dekompozycji. Na przykład, dla funkcji *4 z 20* czas działania algorytmu to 136,59 s, przy czym samo znalezienie dekompozycji zajmuje zaledwie 3,62 s. Metoda First-Fit jest zatem bardzo efektywna czasowo. Dla metody MinR zależność ta również ma miejsce dla większych wartości K. Liczbę wektorów rejestrowanych K dla poszczególnych funkcji przedstawiono w tabeli 3.12. Na rysunku 3.3 przedstawiono zależność czasu obliczeń (w sekundach) od wartości K dla funkcji *M z 16* oraz *M z 20*, dla $M = [1, 5]$. Dla obu metod poprzez indeks dolny d oznaczono czas działania procesu poszukiwania dekompozycji. Z kolei indeksem dolnym t oznaczono całkowity czas obliczeń.

Tabela 3.12: Liczność K zbioru wektorów rejestrowanych dla funkcji *M z N*

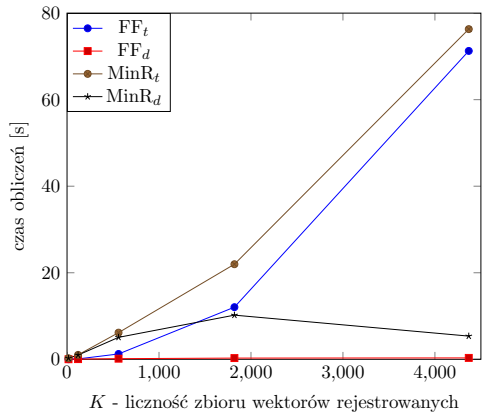
	M				
N	1	2	3	4	5
16	16	120	560	1820	4368
20	20	190	1140	4845	15504

Należy zauważyć, że dla metody MinR czas działania jest silnie zależny od wartości a, dla której znaleziona została dekompozycja. Z tego powodu czas obliczeń dla funkcji *5 z 16* okazał się mniejszy niż dla funkcji *4 z 16*. W literaturze często wprowadza się dodatkowy parametr, który ogranicza złożoność metody, ale wpływa na jakość uzyskiwanego wyniku. Dla zaproponowanego algorytmu takim parametrem może być maksymalna wartość, jaką może przyjąć a.

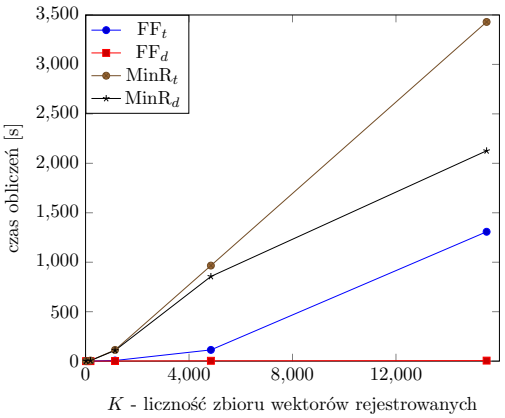
Należy podkreślić, że oprogramowanie eksperymentalne wykonane zostało z wykorzystaniem języka Python, który nie należy do najefektywniejszych czasowo języków programowania. W opinii autora niniejszej rozprawy przedstawione czasy działania można zdecydowanie zredukować poprzez opracowanie zoptymalizowanej implementacji z wykorzystaniem np. języka C. Zastosowanie dedykowanego algorytmu redukcji argumentów [Bor18] również powinno skrócić przedstawione czasy działania.

Rysunek 3.3: Czas obliczeń w zależności od wartości K

(a) Funkcje *M z 16*



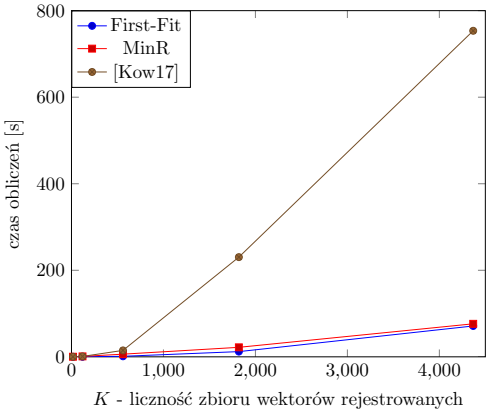
(b) Funkcje *M z 20*



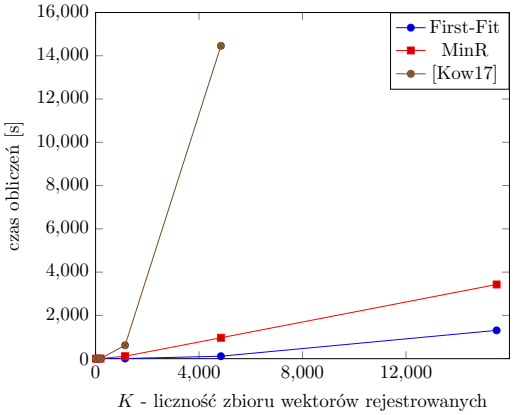
Dla funkcji *M z N* przeanalizowano efekywność autorskich modyfikacji algorytmu. Porównano czas obliczeń z czasem uzyskanym z wykorzystaniem oprogramowania, które zostało wytworzone w ramach wcześniejszych prac nad metodą [Kow17]. Wyniki przedstawiono na rysunku 3.4. Uzyskano znączący wzrost wydajności czasowej dla analizowanych funkcji. Co więcej, dla funkcji *5 z 20* wywołanie drugiego programu skutkuje błędem stosu. Dla pozostałych funkcji uzyskano rozwiązania o takiej samej liczbie zmiennych jak w przypadku metody First-Fit.

Rysunek 3.4: Czas obliczeń w porównaniu z wcześniejszymi pracami

(a) Funkcje *M z 16*



(b) Funkcje *M z 20*



Spośród przedstawionych w tabeli 3.11 metod heurystycznych najlepsze wyniki uzyskuje się metodą wielomianową [Ast+16]. Wyniki te są identyczne jak te otrzymane metodą MinR. Z tego powodu celowe wydaje się dokładniejsze porównanie tych metod. Co więcej, stosowanie funkcji *M z N* do oceny efektywności algorytmów może prowadzić do błędnych wniosków. Wynika to z faktu, że funkcje te charakteryzują się szczególną

strukturą, która może wpływać na uzyskiwane wyniki. Dlatego w dalszej części rozdziału przedstawiono wyniki uzyskane dla losowo wygenerowanych funkcji generowania indeksów.

Porównanie metody algebraicznej z przedstawionymi w niniejszej rozprawie metodami rozpoczęto od analizy stopnia złożoności (ang. Compund degree) otrzymanej reprezentacji [SUI14; MŁ19a]. Stopień ten definiowany jest jako maksymalna liczba zmiennych wejściowych w funktorze wprowadzanym do funkcji ze zredukowaną liczbą zmiennych. Na przykład, dla funkcji $H_3(Z)$ z poprzedniego podrozdziału stopień ten wynosi cztery. Wynika to z faktu, że $z_3 = x_2 \oplus x_3 \oplus x_4 \oplus x_5$, czyli z wykorzystaniem bramek XOR połączone są cztery zmienne wejściowe. Uzyskane wyniki dla funkcji *M z 20* przedstawiono w tabeli 3.13. Jak widać, metody przedstawione w niniejszej rozprawie cechują się generalnie większym stopniem złożoności. W przypadku sprzętowej realizacji funkcji skutkuje to powiększeniem liczby wejść do funkcji. Nie powoduje to jednak pojawienia się dodatkowego poziomu logiki, jak w przypadku dekompozycji funkcjonalnej. Wynika to z własności operacji dodawania modulo dwa [ŁPZ16].

Tabela 3.13: Stopień złożoności reprezentacji dla funkcji *M z 20* [MŁ19a]

	M			
Praca	1	2	3	4
Astola et al. [Ast+16]	10	8	7	4
Metoda First-Fit	9	11	7	3
Metoda MinR	10	10	10	6

W celu dalszego porównania metod wygenerowano po tysiąc losowych funkcji generowania indeksów, dla wybranych par wartości N i K. Funkcje wygenerowano w dwóch wariantach:

1. dla prawdopodobieństwa wystąpienia 0 i 1 w wektorze rejestrowanym wynoszącym po 50% (oznaczony przez $p_0 = p_1 = 0,5$),
2. dla prawdopodobieństwa wystąpienia 0 równego 80% oraz 1 równego 20% (oznaczony przez $p_0 = 0,8, p_1 = 0,2$).

W ramach realizowanych prac przygotowane zostało również oprogramowanie eksperymentalne, z wykorzystaniem środowiska SageMath [Sage] oraz języka Python, które realizowało dekompozycję liniową funkcji generowania indeksów z wykorzystaniem metody wielomianowej. Średnią liczbę zmiennych P, uzyskaną dla zdekomponowanych funkcji, przedstawiono w tabeli 3.14.

Jak widać, metody First-Fit oraz MinR gwarantują uzyskanie reprezentacji o mniejszej liczbie zmiennych niż metoda wielomianowa. Dla wariantu pierwszego uzyskano reprezentacje z ok. $0,7 - 1,1$ zmienną mniej. Dla funkcji rzadkich, tzn. w wariancie drugim,

Tabela 3.14: Wyniki dla metody wielomianowej (M.w.) oraz metod First-Fit (FF) i MinR, dla losowych funkcji generowania indeksów

			$p_0 = p_1 = 0,5$			$p_0 = 0,8, p_1 = 0,2$		
N	K	GD	FF	MinR	M.w.	FF	MinR	M.w.
20	20	5	5,368	5,370	6,124	5,250	5,177	6,265
	40	6	6,987	6,989	7,870	6,915	6,829	7,927
	60	6	7,987	7,985	8,875	7,762	7,490	8,924
	80	7	8,737	8,743	9,477	8,122	8,022	9,586
	100	7	9,009	9,009	9,978	8,953	8,783	9,995
40	20	5	5,162	5,162	6,114	5,321	5,281	6,283
	40	6	6,986	6,986	7,891	6,900	6,836	7,937
	60	6	7,976	7,976	8,868	7,730	7,545	8,932
	80	7	8,664	8,662	9,472	8,087	8,018	9,664
	100	7	9,003	9,003	9,989	8,927	8,783	10,022
60	20	5	5,055	5,055	6,143	5,332	5,302	6,282
	40	6	6,965	6,964	7,884	6,898	6,869	7,940
	60	6	7,955	7,955	8,871	7,724	7,585	8,917
	80	7	8,601	8,601	9,491	8,074	8,021	9,687
	100	7	8,996	8,996	9,986	8,914	8,777	10,022

efektywność przedstawionych w niniejszej rozprawie metod w porównaniu do metody wielomianowej była jeszcze lepsza. Uzyskano reprezentacje z ok. $1,0-1,7$ zmienną mniej.

Analizując przedstawione wyniki, należy zwrócić uwagę, że dla funkcji rzadkich metodą wielomianową uzyskano gorsze wyniki dla funkcji z wariantu pierwszego. W przypadku obu metod przedstawionych w rozprawie, zależność jest odwrotna. Wyjątek stanowią funkcje z $N = 60$ oraz $K = 20$.

Przedstawione wyniki wykorzystać można do porównania efektywności metod First-Fit oraz MinR. Jak widać, w średnim przypadku różnica w jakości uzyskiwanego rozwiązania między tymi metodami jest niewielka. Dla wszystkich analizowanych par wartości N oraz K średnia dla metody MinR była o zaledwie $0,05$ mniejsza, niż w przypadku metody First-Fit. Oznacza to, że obie metody mogą być z powodzeniem stosowane do minimalizacji funkcji generowania indeksów. Różnice dla losowo wygenerowanych funkcji okazały się mniejsze, niż w przypadku funkcji M *z* N analizowanych wcześniej. Z drugiej strony, metoda wielomianowa [Ast+16] okazała się nieefektywna.

W tabeli 3.15 zestawiono przedstawione wcześniej wyniki dla losowo wygenerowanych funkcji z wynikami dostępnymi w literaturze [Sas08; Ast+16; HPC19]. Należy podkreślić, że autorzy tych prac wykorzystali inny zbiór losowo wygenerowanych funkcji generowania indeksów. Mimo to wyniki te pozwalają dokonać ogólnej oceny zaproponowanych metod. Kolumny FF, MinR oraz M.w. reprezentują zaokrąglone wyniki przedstawione w tabeli 3.14 dla wariantu $p_0 = p_1 = 0,5$. Co istotne, wyniki dla metody wielomianowej są

zbliżone do tych uzyskanych dla innego zbioru losowo wygenerowanych funkcji [Ast+16]. Potwierdza to poprawność wykonanego oprogramowania eksperymentalnego.

Tabela 3.15: Wyniki dla losowych funkcji generowania indeksów

N	K	GD	FF	MinR	M.w.	[Ast+16]	[HPC19]	[Sas08]
20	20	5	5,4	5,4	6,1	6,2	5,5	5,5
	40	6	7,0	7,0	7,9	7,9	-	7,3
	60	6	8,0	8,0	8,9	8,9	8,0	8,1
	80	7	8,7	8,7	9,5	9,5	-	9,0
	100	7	9,0	9,0	10,0	10,1	9,0	9,5
40	20	5	5,2	5,2	6,1	6,1	5,5	5,1
	40	6	7,0	7,0	7,9	7,9	-	7,0
	60	6	8,0	8,0	8,9	8,9	8,0	8,0
	80	7	8,7	8,7	9,5	9,5	-	8,9
	100	7	9,0	9,0	10,0	10,0	9,0	9,4
60	20	5	5,1	5,1	6,1	6,2	5,6	5,0
	40	6	7,0	7,0	7,9	7,9	-	7,0
	60	6	8,0	8,0	8,9	8,9	8,0	8,0
	80	7	8,6	8,6	9,5	9,5	-	8,5
	100	7	9,0	9,0	10,0	10,0	9,0	9,0

Jak widać, metody First-Fit oraz MinR gwarantują uzyskanie wyników zbliżonych do algebraicznej metody iteracyjnej [HPC19]. Metoda prof. Sasao [Sas08] okazuje się mniej efektywna w przypadku $N \in \{20, 40\}$ oraz $K \in \{80, 100\}$. Dla pozostałych par wartości uzyskiwane wyniki są zbliżone również do metod zaprezentowanych w niniejszej pracy. Reprezentacje z największą liczbą zmiennych uzyskuje się z kolei metodą wielomianową.

Analizując wyniki przedstawione w tabeli należy zwrócić uwagę, że dla rosnących wartości K różnica między GD a średnim wynikiem uzyskiwanym dowolną z metod rośnie. Oznacza to najprawdopodobniej nieistnienie dekompozycji liniowej takiej, że $P = GD$ dla funkcji o większej wartości K. Celowe jest zatem zaproponowanie innych metod umożliwiających dalszą minimalizację funkcji generowania indeksów.

3.2.3 Wnioski z przeprowadzonych badań

W niniejszym rozdziale przedstawiono wyniki prac nad metodami znajdowania dekompozycji liniowej funkcji generowania indeksów. W tym celu zaproponowano autorskie modyfikacje algorytmu wykorzystującego zbiory niezgodności.

Przedstawione wyniki potwierdzają efektywność zaproponowanej metody. Dla funkcji *M z N*, przy zastosowaniu podejścia MinR uzyskano takie same wyniki, jak w przypadku zastosowania metod dokładnych.

Wyniki dekompozycji tych funkcji nie mogą być uznane za obiektywny wyznacznik efektywności algorytmów. Świadczą o tym m.in. wyniki osiągane z wykorzystaniem metody wielomianowej [Ast+16]. Z związku z tym, przeanalizowano również efektywność proponowanego algorytmu dla losowo wygenerowanych funkcji generowania indeksów. Oba podejścia do wyboru dekompozycji w poszczególnych iteracjach, First-Fit oraz MinR, pozwoliły na uzyskanie bardzo dobrych wyników.

Wyniki otrzymane dla metody First-Fit potwierdzają, że może być ona wykorzystana do efektywnego znalezienia dekompozycji liniowej funkcji generowania indeksów. Jest to o tyle istotne, że cechuje się ona mniejszą złożonością obliczeniową niż drugie zaproponowane podejście.

Uzyskane wyniki sugerują, że dla wielu funkcji nie istnieje dekompozycja liniowa, która pozwala na reprezentację funkcji z GD zmiennymi. Dla niektórych funkcji M *z* N brak istnienia takiej dekompozycji został udowodniony z wykorzystaniem metod dokładnych [Sas17; NSB18]. W związku z tym celowe jest zaproponowanie innych metod. W rozdziale 5 przeanalizowano możliwość zastosowania algorytmów dekompozycji funkcjonalnej do syntezy logicznej takich funkcji.

Rozdział 4

Realizacja generatorów indeksów

W niniejszym rozdziale przedstawiono wyniki badań nad realizacją generatorów indeksów. W pierwszej kolejności wyjaśniono czym są generatory indeksów oraz przedstawiono rozwiązania dostępne w literaturze. Skupiono się przede wszystkim na rozwiązaniu zaproponowanym przez prof. Sasao zwanym IGU [Sas06; Sas11b; Sas20]. Wynika to z faktu, że stanowi ono również podstawę innych konstrukcji. Następnie określono wady tego rozwiązania i zaproponowano alternatywną architekturę, która wykorzystuje struktury probabilistyczne. Dokonano analizy możliwości zastosowania trzech wybranych struktur oraz wykazano ich użyteczność w porównaniu do IGU.

4.1 Generatory indeksów

Funkcjonalność funkcji generowania indeksów realizowana jest zwykle przez zastosowanie pamięci typu CAM. Jednakże, pamięci te cechuje relatywnie wysoki koszt i rozproszenie mocy. Inną możliwością jest wykorzystanie pojedynczej pamięci RAM o N wejściach oraz $Q = GD$ wyjściach. Całkowita zajętość pamięci wynosi wtedy $Q \cdot 2^N$. Realizacja ta jest jednak niecelowa, ze względu na silną nieokreśloność funkcji generowania indeksów. W związku z tym, w ostatnich latach istotne stały się badania nad sprzętową realizacją generatorów indeksów, czyli cyfrowych układów sprzętowych, które realizują funkcję generowania indeksów.

Jedno z pierwszych podejść do problemu realizacji generatorów indeksów [Sas06; Sas11b] zakłada wykorzystanie kaskady elementów pamiętających o p_k wejściach i q wyjściach. Wejście do pierwszego elementu stanowi pierwsze p_k zmiennych realizowanej funkcji. Dla pozostałych elementów jako wejście traktowane są wyjścia z poprzedniego elementu oraz kolejne $p_k - q$ zmiennych wejściowych. Liczba elementów pamiętających wymaganych

do realizacji tego typu struktury wynosi co najwyżej:

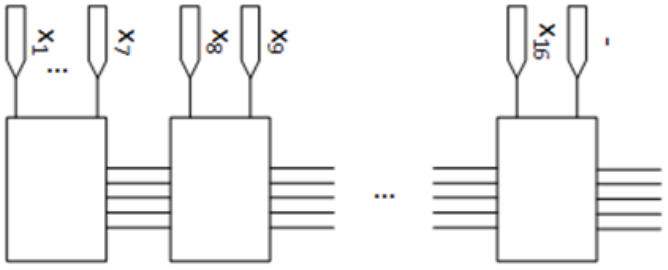
$$\left\lceil \frac{N-q}{p_k-q} \right\rceil \tag{4.1}$$

Całkowite wykorzystanie pamięci w takiej strukturze jest najmniejsze dla $p_k - q = 1$ albo $p_k - 2$. Wynosi ono (w bitach):

$$\left\lceil \frac{N-q}{p_k-q} \right\rceil \cdot q \cdot 2^{p_k} \tag{4.2}$$

Jak łatwo zauważyć, liczbę wykorzystanych elementów pamiętających można zmniejszyć, poprzez zwiększenie liczby wejść do pojedynczego elementu. Wiążę się to jednak ze zwiększeniem całkowitej zajętości pamięci.

Przykład 4.1. Niech dana będzie funkcja generowania indeksów o $N = 16$ zmiennych wejściowych. Dodatkowo, niech $p_k = 7$ oraz $q = 5$. Funkcję tę możemy zrealizować wtedy za pomocą architektury przedstawionej na rysunku 4.1. Wykorzystuje ona $6 \cdot 5 \cdot 2^7 = 3840$ bitów pamięci. Jest to istotna redukcja w porównaniu do realizacji z pojedynczą kością pamięci.

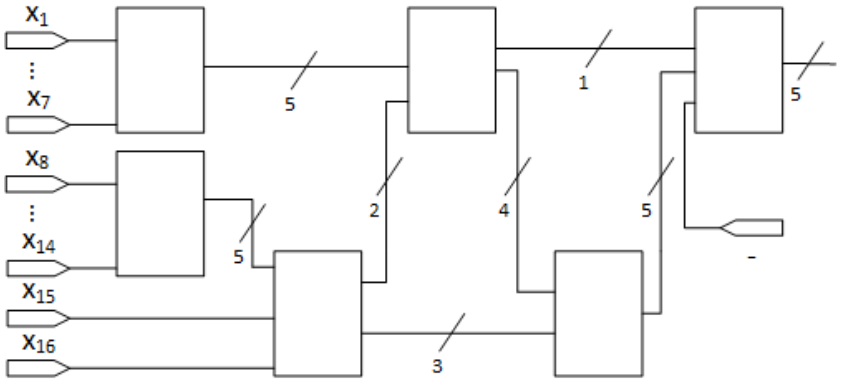


Rysunek 4.1: Realizacja funkcji generowania indeksów z wykorzystaniem kaskady elementów pamiętających

△

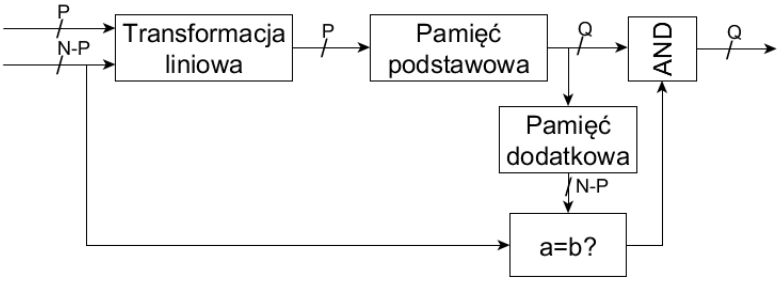
Kaskada elementów pamiętających może zostać zrealizowana również jako wielopoziomowa sieć. Nie wpływa to na liczbę wykorzystanych elementów oraz rozmiar pojedynczego elementu, a co za tym idzie całkowitą zajętość pamięci. Zmniejsza się jednak liczba poziomów kaskady, a co za tym idzie czas przetwarzania pojedynczego wektora jest krótszy ze względu na krótszą ścieżkę krytyczną. Na rysunku 4.2 przedstawiono przykład wielopoziomowej kaskady pamięci, realizujący tę samą funkcjonalność co struktura z przedstawionego powyżej przykładu. Liczba poziomów kaskady zmniejszona została z 6 do 5.

Opisana struktura jest stosowana zwłaszcza w przypadku układów FPGA starszych generacji. Odzwierciedla ona implementację funkcji z wykorzystaniem bloków logicznych,



Rysunek 4.2: Realizacja funkcji generowania indeksów z wykorzystaniem wielopoziomowej kaskady elementów pamiętających

z których zbudowane są tego typu struktury. Podstawowym elementem dla takich układów jest blok tablicowy LUT, który umożliwia realizację dowolnej funkcji boolowskiej. Charakteryzuje się on niewielką liczbą wejść (np. 4 lub 5) i wyjść (1). Nowsze układy mają bloki adaptowalne, dzięki czemu jest możliwe zastosowanie bloków o różnej liczbie wejść i wyjść w ramach jednej struktury. Minimalizacja funkcji boolowskiej umożliwia zmniejszenie liczby wykorzystywanych bloków, a co za tym idzie – implementację w układach FPGA wykorzystującą mniej zasobów logicznych, a często gwarantujących również większą wydajność przez skrócenie ścieżki krytycznej.



Rysunek 4.3: Architektura IGU (na podstawie [Sas20])

Jak podkreśla twórca, kaskada elementów pamiętających gwarantuje efektywną implementację w sytuacji, gdy $\lceil log_2(K+1) \rceil < p_k$. Dla pozostałych funkcji architektura ta nie może być wprost zastosowana [Sas11b]. W związku z tym konieczne jest zaproponowanie bardziej uniwersalnych metod. Przykładem alternatywnej realizacji generatorów indeksów jest wykorzystanie architektury IGU zaproponowanej przez prof. Sasao [Sas11b; Sas20]. Przedstawiono ją na rysunku 4.3. Generator indeksów zrealizowany jest z wykorzystaniem dwóch pamięci: podstawowej oraz pomocniczej, a także komparatora oraz

bramki AND. Pamięć podstawowa przechowuje informację o wartości indeksu przypisywanego analizowanemu wektorowi wejściowemu, podczas gdy pozostałe elementy wykorzystywane są do detekcji czy wektor ten jest wektorem rejestrowanym. Dodatkowo, wejścia do pamięci podstawowej modyfikowane są za pomocą modułu realizującego funkcję liniową. Zastosowanie modułu zmniejszającego liczbę zmiennych z N do P skutkuje istotną redukcją rozmiaru stosowanych w generatorze pamięci. W związku z tym znalezienie efektywnej transformacji liniowej jest czynnikiem kluczowym, warunkującym efektywność pamięciową realizowanego generatora.

Pamięć podstawowa generuje prawidłowe wyniki dla wektorów rejestrowanych, jednak może generować niewłaściwe wyniki dla pozostałych wektorów wejściowych. Z tego powodu pamięć pomocnicza i komparator wykorzystywane są do sprawdzenia czy wygenerowana wartość jest właściwa. Całkowita zajętość pamięci wynosi (w bitach) wynosi

$$MEM_{IGU} = MEM_{main} + MEM_{aux} = Q \cdot 2^P + (N - P) \cdot 2^Q \tag{4.3}$$

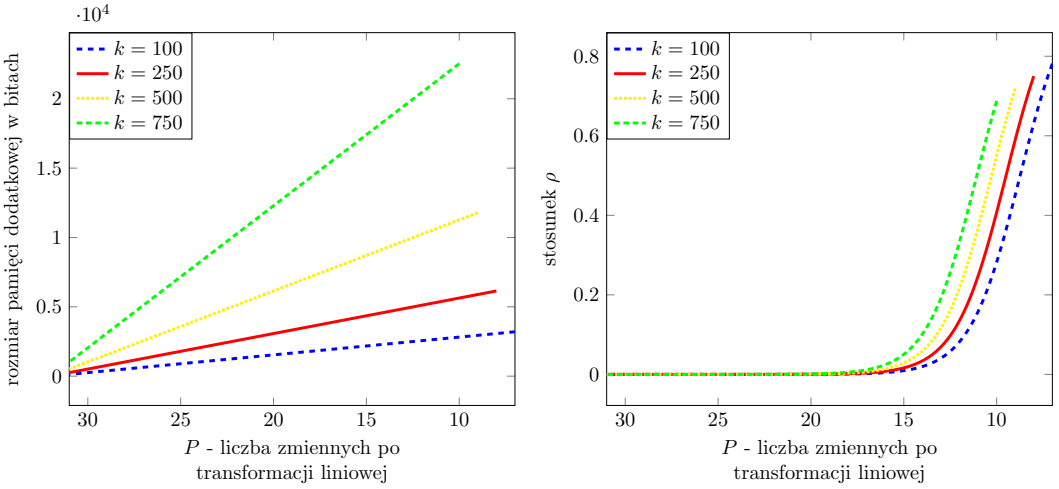
W celu porównania efektywności pamięciowej IGU oraz kaskady elementów pamiętających, rozważmy następujący przykład.

Przykład 4.2. Niech dana będzie przykładowa funkcja generowania indeksów o następujących parametrach: $N = 32$ oraz $K = 1000$. Na podstawie równania 2.3 otrzymujemy $GD = 10$. Niech $P = GD$, $q = GD$ oraz $p_k = q+1$. Wykorzystanie pamięci, przy realizacji z użyciem kaskady pamięci, na podstawie równania 4.2 wynosi wtedy 450560 bitów. Z kolei dla architektury IGU, na podstawie równania 4.3, otrzymujemy 32768 bitów, czyli niemal czternastokrotnie mniejsze wykorzystanie pamięci. △

Architektura IGU stanowi również podstawę architektur wykorzystujących dodatkowo programowalną macierz logiczną, tzw. metody hybrydowe albo wykorzystujących wiele struktur IGU połączonych bramką OR [SMN10; Sas11b]. Ze względu na to, że głównym elementem tych metod jest architektura IGU, dalsze badania skupiać będą się na zaproponowaniu architektury realizującej tę samą funkcjonalność, a wiążącej się z mniejszym wykorzystaniem pamięci. Potencjalnie architektura ta zastąpić może IGU we wspominanych metodach.

W IGU pamięć dodatkowa wykorzystywana jest do wykrycia, czy wektor wejściowy należy do zbioru wektorów rejestrowanych. Istotną wadą architektury tej architektury jest wzrost wykorzystania pamięci dodatkowej dla malejącej wartości P. Oznacza to, że im lepsze (pod względem redukcji liczby zmiennych) przekształcenie liniowe zostanie wyznaczone, tym większy będzie rozmiar tej pamięci. Co więcej, dla niektórych wartości parametrów N, K oraz P rozmiar ten może być większy od rozmiaru pamięci głównej. Na

rysunku 4.4a przedstawiono rozmiar pamięci dodatkowej w IGU dla funkcji generowania indeksów z $N = 32$ dla różnych wartości K. Na rysunku 4.4b przedstawiono z kolei rozmiar ten w stosunku do całkowitego rozmiaru pamięci, tzn. $\rho = \frac{MEM_{aux}}{MEM_{IGU}} \cdot 100\%$. Na obu rysunkach przedstawiono zależność w stosunku do malejącej wartości P. Jak widać, dla $P = GD$ rozmiar pamięci dodatkowej jest około trzykrotnie większy od rozmiaru pamięci podstawowej dla wszystkich analizowanych wartości K, zgodnie ze wzorem 4.3.



(A) Rozmiar pamięci dodatkowej

(B) Rozmiar MEM_{aux} w stosunku do MEM_{IGU}

RYSUNEK 4.4: Rozmiar pamięci dodatkowej w IGU ($N = 32$) [Maz19a]

Uzyskane wyniki przeczą idei pamięciowo-efektywnej realizacji funkcji generowania indeksów. W związku z tym w ramach prowadzonych badań zaproponowano wykorzystanie innej metody sprawdzenia czy wektor wejściowy należy do zbioru wektorów rejestrowanych.

4.2 Generatory wykorzystujące struktury probabilistyczne

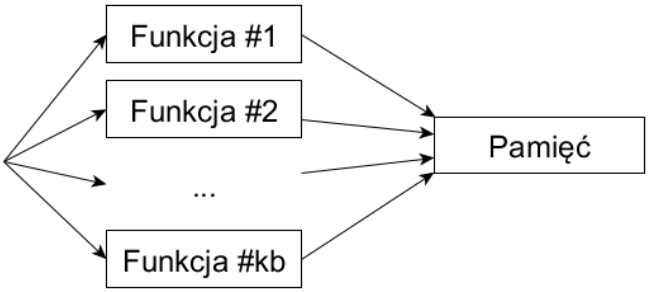
Jak już wspomniano, rozmiar pamięci dodatkowej w IGU silnie zależy od wartości liczby zmiennych po transformacji liniowej. Za celowe uznano zaproponowanie innej architektury, która nie będzie posiadała takiej zależności. W związku z tym, zaproponowano [MBŁ18] wykorzystanie struktur probabilistycznych do określania, czy dany wektor należy do zbioru wektorów rejestrowanych. Pozwoliło to uzyskać większą efektywność pamięciową kosztem niewielkiego prawdopodobieństwa ϵ uzyskania wyniku fałszywie pozytywnego. Oznacza to, że możliwe jest wygenerowanie wartości różnej od zera dla wektora, który nie należy do zbioru wektorów rejestrowanych. Ponadto, zastosowanie struktur probabilistycznych wprowadza konieczność realizacji dodatkowych obliczeń. W dalszej części przeanalizowano trzy struktury probabilistyczne:

1. filtr Blooma,
2. filtr Blooma z pojedynczą funkcją skrótu,
3. filtr Cuckoo.

4.2.1 Filtr Blooma

Filtr Blooma [Blo70] jest strukturą probabilistyczną zaproponowaną w latach 70. Dzięki rozwojowi szeroko rozumianej informatyki zyskuje ona coraz większą popularność, m.in. w zastosowaniach sieciowych oraz systemach rozproszonych. Jej działanie sprowadza się do odpowiedzi na pytanie, czy zadany wektor wejściowy jest elementem ustalonego zbioru S czy nie. Filtr Blooma reprezentowany jest za pomocą m_b-bitowej tablicy, której elementy oznaczono jako $M[i]$. Dodatkowo, do działania wymaga on k_b niezależnych funkcji skrótu generujących wartości z przedziału $[0, m_b - 1]$. Wartość k_b zależna jest od akceptowanego prawdopodobieństwa wyniku fałszywie pozytywnego.

Początkowo tablica wypełniona jest zerami. Następnie, dla każdego wektora z ustalonego zbioru $\overrightarrow{v} \in S$ obliczane jest k_b wartości skrótu $h_i(\overrightarrow{v})$. Wartości te określają, którym bitom w tablicy należy przypisać wartość jeden. Po dodaniu wszystkich wektorów ze zbioru S do filtru, możliwe jest sprawdzanie czy dowolny wektor $\overrightarrow{w}$ należał do zbioru S czy nie. W tym celu należy sprawdzić, czy $\forall_{i=[1,k_b]} : M[h_i(\overrightarrow{w})] = 1$. Jeżeli tak, to $\overrightarrow{w} \in S$ z prawdopodobieństwem $1 - \epsilon$. W przeciwnym wypadku, wektor wejściowy nie należał do zbioru S. Ogólną ideę działania filtru Blooma przedstawiono na rysunku 4.5. W przypadku funkcji generowania indeksów zbiorem S jest zbiór wektorów rejestrowanych.



Rysunek 4.5: Idea działania filtru Blooma [Maz19b]

Przykład 4.3. Niech $S = \{\overrightarrow{v}, \overrightarrow{w}\}$, $m_b = 8$ oraz $k_b = 2$. Początkowo tablica jest wypełniona zerami - (00000000). Niech teraz $h_1(\overrightarrow{v}) = 0$ oraz $h_2(\overrightarrow{v}) = 7$. Dodanie wektora $\overrightarrow{v} \in S$ do filtru powoduje ustawienie wartości bitów numer jeden oraz osiem na jeden - (10000001). Podobnie przyjmijmy, że $h_1(\overrightarrow{w}) = 1$ oraz $h_2(\overrightarrow{w}) = 7$. W wyniku dodania wektora $\overrightarrow{w} \in S$ tablica ma postać (11000001).

Po dodaniu wszystkich wektorów ze zbioru S do filtru, możliwe jest sprawdzenie, czy dowolny wektor $\overrightarrow{u}$ należał do zbioru czy nie. Załóżmy, że $h_1(\overrightarrow{u}) = 0$ oraz $h_2(\overrightarrow{u}) = 7$. Wtedy w wyniku otrzymujemy informację, że $\overrightarrow{u} \in S$ z prawdopodobieństwem $1-\epsilon$, gdyż zarówno pierwszy, jak i ósmy bit mają wartość jeden. Zauważmy, że jeżeli wartość skrótu $h_2(\overrightarrow{u})$ miałaby wartość 1, to otrzymujemy wynik fałszywie pozytywny. Wartość bitów pierwszego oraz drugiego to jeden, jednak żaden z wektorów ze zbioru S nie generował takiej pary skrótów. Z kolei w sytuacji, gdy $h_2(\overrightarrow{u})$ ma wartość np. siedem, to wiemy, że $\overrightarrow{u} \notin S$, gdyż siódmy bit ma wartość zero. $\triangle$

W celu określenia prawdopodobieństwa wyniku fałszywie pozytywnego załóżmy, że wykorzystane funkcje skrótu generują każdą z wartości $[1, m_b]$ z takim samym prawdopodobieństwem[1], które równe jest $\frac{1}{m_b}$. Przyjmijmy, że $|S| = n_b$. W takiej sytuacji prawdopodobieństwo, że wartość określonego bitu w tablicy równa jest zero po dodaniu n_b wektorów ze zbioru S wynosi:

$$\delta = \left(1 - \frac{1}{m_b}\right)^{k_b \cdot n_b} \tag{4.4}$$

Prawdopodobieństwo zdarzenia przeciwnego, tzn. że wartość równa jest zero, wynosi zatem $1 - \delta$. W związku z tym, prawdopodobieństwo wyniku fałszywie pozytywnego równe jest:

$$\epsilon = (1 - \delta)^{k_b} \approx \left(1 - e^{\frac{-k_b \cdot n_b}{m_b}}\right)^{k_b} \tag{4.5}$$

Wartość ϵ jest najmniejsza w sytuacji, gdy:

$$k_b = \frac{m_b}{n_b} \cdot ln(2) \approx \frac{9 \cdot m_b}{13 \cdot n_b} \Rightarrow \epsilon \approx (0,6185)^{\frac{n_b}{m_b}} \tag{4.6}$$

Jak łatwo zauważyć, im większa wartość m_b, tym mniejsze prawdopodobieństwo wyniku fałszywie pozytywnego. Wykorzystując równanie 4.5 możliwe jest określenie optymalnej liczby funkcji skrótu do zastosowania w filtrze:

$$k_b^* = -\log_2(\epsilon) \tag{4.7}$$

Ze względu na to, że liczba funkcji skrótu powinna być liczbą całkowitą, otrzymana wartość w dalszych rozważaniach będzie zaokrąglana w górę. Na podstawie powyższych rozważań możliwe jest również wyznaczenie optymalnej liczby bitów na element, która

[1]Możliwe jest przeprowadzenie alternatywnej analizy [MU05]. Prowadzi ona do takich samych wyników, jednak nie wymaga przedstawionego założenia.

wynosi:

$$\psi^* = \frac{m_b}{n_b} \approx -1,44 \log_2(\epsilon) \tag{4.8}$$

Przykład 4.4. Niech $\epsilon = 1\%$. Otrzymujemy wtedy $\psi^* \approx -1,44 \cdot \log_2(0,01) = 9,57$ oraz $k_b^* = -\log_2(0,01) = 6,64$. Wartości dla wybranych wartości ϵ przedstawiono w tabeli 4.1.

Tabela 4.1: Parametry dla filtru Blooma

ϵ	0,01%	0,10%	0,25%	0,50%	1,00%	2,00%	5,00%
ψ^*	19,13	14,35	12,45	11,01	9,57	8,13	6,22
k_b^*	13,29	9,97	8,64	7,64	6,64	5,64	4,32

△

Rozmiar pamięci wykorzystywanej przez filtr może być określony na podstawie zakładanego prawdopodobieństwa wyniku fałszywie pozytywnego, tzn.:

$$m_b = \psi^* \cdot K = -1,44 \cdot K \cdot \log_2(\epsilon) \tag{4.9}$$

Oczywiście, po przekształceniu powyższego wzoru lub z wykorzystaniem wzoru 4.5, możliwe jest określenie prawdopodobieństwa wyniku fałszywie pozytywnego na podstawie dostępnej wielkości pamięci.

Na podstawie danych z tabeli 4.1 można zauważyć, że realizacja filtru Blooma z $\epsilon = 1\%$ wprowadza konieczność wykorzystania aż siedmiu niezależnych funkcji skrótu. Co więcej, "klasyczny" pozwala na wykonanie jedynie dwóch operacji: dodania wektora do filtru oraz sprawdzenia, czy wektor wejściowy należy do zbioru S. Nie pozwala z kolei na np. usunięcie już dodanego wektora. Należy zauważyć, że w literaturze zaproponowano przeszło 20 wariantów filtru [TRL12], które dodają nowe funkcjonalności. Zazwyczaj jednak kosztem gorszej efektywności pamięciowej.

Istotnym zagadnieniem jest wybór funkcji skrótu wykorzystywanych w filtrze. Właściwy dobór funkcji jest zadaniem skomplikowanym [Lu+18]. Na przykład korelacja pomiędzy funkcjami powoduje znaczny wzrost prawdopodobieństwa wyniku fałszywie pozytywnego. Funkcje wykorzystywane w realizacjach filtru można podzielić na trzy grupy:

1. funkcje kryptograficzne,
2. funkcje niekryptograficzne,
3. funkcje uniwersalne [CW79].

Funkcje z pierwszej grupy, np. MD5 czy SHA-1, gwarantują dobre właściwości statystyczne. Jednakże, zazwyczaj ich implementacja cechuje się relatywnie dużą złożonością

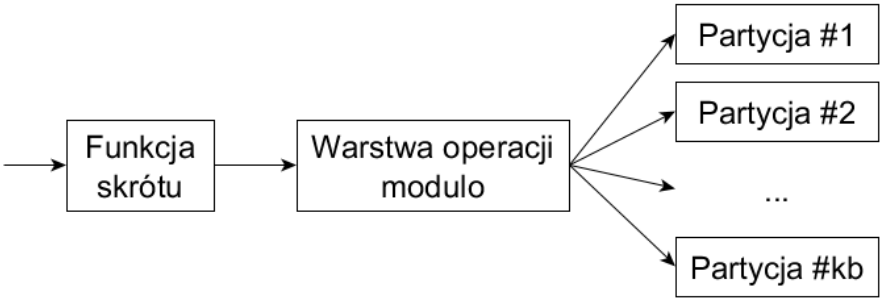
i wykorzystaniem zasobów logicznych. Z tego powodu często stosuje się funkcje z drugiej grupy. Nie zapewniają one tak dobrych właściwości statystycznych, co wiąże się z wyższym prawdopodobieństwem wyniku fałszywie pozytywnego. Z drugiej strony, mogą być one efektywnie implementowane w układach programowalnych. Pytaniem otwartym [Maz19a] pozostaje ocena możliwości zastosowania funkcji kryptograficznych zaprojektowanych z myślą o użyciu w urządzeniach o ograniczonym dostępie do zasobów logicznych, tzw. lekkich (ang. lightweight). Trzeci typ funkcji zakłada losowy wybór funkcji skrótu spośród rodziny funkcji o określonych właściwościach statystycznych. W praktyce do implementacji tych funkcji wykorzystuje się funkcje z obu przedstawionych wcześniej kategorii. Cechują się one największą złożonością i wykorzystaniem zasobów logicznych.

4.2.2 Filtr Blooma z pojedynczą funkcją skrótu

W celu wyeliminowania problemu ze skomplikowanym zadaniem doboru funkcji skrótu do filtru Blooma, można rozważyć zastosowanie jego wariantu z pojedynczą funkcją skrótu [Lu+18] (ang. One-Hashing Bloom Filter). Jak nazwa wskazuje, filtr ten wykorzystuje zaledwie jedną funkcję skrótu. Dodatkowo, realizowane jest k_b operacji modulo w celu zapewnienia tej samej funkcjonalności, co filtr Blooma przedstawiony we wcześniejszym podrozdziale.

Idea działania filtru z pojedynczą funkcją skrótu przedstawiona została na rysunku 4.6. Działanie filtru podzielone jest na dwie fazy:

1. faza wyliczania skrótu,
2. faza operacji modulo.



Rysunek 4.6: Idea działania filtru Blooma z pojedynczą funkcją skrótu [Maz19b]

Pierwsza odpowiada za wyznaczenie skrótu z wektora wejściowego. Skrót ten zazwyczaj ma długość słowa maszynowego, np. 32 lub 64 bity. W drugiej fazie, na podstawie wartości skrótu i wartości m_i uzyskiwana jest informacja o bitach, których wartość powinna zostać ustawiona na jeden. Wykorzystywana pamięć podzielona jest na k_b partycji. Dla

każdej z partycji wykorzystywana jest inna wartość modulnika m_i. Rozmiar poszczególnych partycji równy jest wartości modulnika.

Przykład 4.5. Niech $m_i \in \{5, 7, 9, 11\}$. Rozmiar filtru wynosi wtedy 32 bity $(5 + 7 + 9 + 11)$. Początkowo, wszystkie partycje P_i $(i = [1, 4])$ wypełnione są zerami. Podczas pierwszej fazy obliczana jest wartość skrótu dla wektora wejściowego. Niech na przykład $h(\overrightarrow{v}) = 0xFA0$. Wtedy w drugiej fazie otrzymujemy następujące wyniki:

$$h(\overrightarrow{v}) \equiv 0 \pmod 5$$
$$h(\overrightarrow{v}) \equiv 3 \pmod 7$$
$$h(\overrightarrow{v}) \equiv 4 \pmod 9$$
$$h(\overrightarrow{v}) \equiv 7 \pmod{11}$$

Uzyskane wartości wskazują, którym bitom powinna zostać przypisana wartość jeden: $P_1 = (10000)$, $P_2 = (0001000)$, $P_3 = (000010000)$ oraz $P_4 = (00000001000)$. Dalsze działanie jest analogiczne do filtru Blooma. △

Metoda doboru wartości modulników przedstawiona została przez twórców filtru[2]. Wartości m_i są kolejnymi liczbami pierwszymi. Dzięki temu generowane wartości skrótów są parami niezależne. Należy zauważyć, że suma długości poszczególnych partycji zazwyczaj będzie różna od zakładanej wartości m_b. Suma ta oznaczana będzie poprzez:

$$m_o = \sum_{i=1}^{k_b} m_i \tag{4.10}$$

Prawdopodobieństwa wyniku fałszywie pozytywnego w filtrze oszacować można na podstawie nierówności [Lu+18]:

$$\epsilon_o \leqslant \left(1 - \sqrt[k_b]{\prod_{i=1}^{k_b} e^{\frac{-n_b}{m_i}}}\right)^{k_b} \tag{4.11}$$

Istotną wadą filtru jest jego bardziej skomplikowana generacja niż w przypadku "klasycznego" filtru. Wymaga ona zastosowania tablicy liczb pierwszych lub metod wyznaczania kolejnych takich liczb. Z drugiej strony, filtr z pojedynczą funkcją skrótu zapewnia lepsze właściwości statystyczne [AA19] i cechuje się mniejszą liczbą wyników fałszywie pozytywnych. Co więcej, filtr z pojedynczą funkcją skrótu może być wykorzystany we wspomnianych przy filtrze Blooma różnych jego wariantach.

[2]Należy zauważyć, że jeżeli rozmiar filtru jest bardzo mały, to metoda ta może zakończyć się niepowodzeniem.

4.2.3 Filtr Cuckoo

Inną strukturą probabilistyczną, która może być wykorzystana do realizacji generatorów indeksów jest filtr Cuckoo [Fan+14]. Główną zaletą tego filtru w porównaniu do filtru Blooma jest mniejsze wykorzystanie pamięci przy małym prawdopodobieństwie wyniku fałszywie pozytywnego oraz możliwość dynamicznego dodawania i usuwania elementów (wektorów) z filtru.

Filtr Cuckoo jest kompaktową wersją tablic Cuckoo z haszowaniem [PR04]. Tablice te składają się z listy kubełków (ang. Bucket). Dla każdego elementu wejściowego obliczane są dwie wartości skrótów ($k_c = 2$). Wartości te oznaczone będą odpowiednio $h_1(\overrightarrow{v})$ oraz $h_2(\overrightarrow{v})$. W procedurze dodawania elementu do filtru wykorzystywane są one do określenia dwóch potencjalnych kubełków, gdzie dany element może zostać umieszczony. Jeżeli któryś z kubełków jest pusty, to element dodawany jest do tego kubełka. W przeciwnym wypadku, tzn. gdy oba kubełki są pełne, realizowana jest operacja tzw. relokacji. Polega ona na umieszczeniu elementu w jednym z kubełków oraz przeniesieniu poprzednio przechowywanej wartości w inne miejsce w filtrze. W tym celu ponownie wyliczane są dwie wartości skrótu i określane są potencjalne kubełki do umieszczenia wartości. Oczywiście, oba kubełki mogą ponownie być pełne. Procedura wykonywana jest iteracyjnie do momentu, gdy zostanie znaleziony pusty kubełek albo osiągnięta zostanie maksymalna liczba przeniesień. W drugim wypadku filtr określa się wtedy jako zbyt pełny do dodania elementu. Procedura dodania elementu kończy się wtedy niepowodzeniem. Konieczność realizacji operacji relokacji na etapie jego konstrukcji jest istotną wadą filtru Cuckoo, jeżeli zbiór wektorów rejestrowanych będzie modyfikowany.

W momencie dodania wszystkich wektorów ze zbioru S do filtru, możliwa jest realizacja procedury sprawdzenia czy dowolny wektor wejściowy należał do tego zbioru. Sprowadza się ona do sprawdzenia czy któryś z wyznaczonych kubełków zawiera poszukiwany wektor.

Liczba wykorzystywanych funkcji skrótu k_c oraz rozmiar kubełka b_c, tzn. liczba możliwych elementów do przechowywania w kubełku, mogą być modyfikowane w celu uzyskania lepszej efektywności tablicy.

W filtrze Cuckoo zamiast przechowywać pary klucz-wartość, przechowuje się jedynie tzw. odcisk (ang. fingerprint) wektora, tzn. wartość wyznaczoną z wykorzystaniem funkcji skrótu. Wymaga to innej procedury dodawania elementu niż opisywana wcześniej. Wyznaczane są następujące trzy wartości:

1. $f_{\overrightarrow{v}} = \text{fingerprint}(\overrightarrow{v})$,
2. $h_1(\overrightarrow{v}) = \text{hash}(\overrightarrow{v})$,

3. $h_2(\overrightarrow{v}) = h_1(\overrightarrow{v}) \oplus \text{hash}(f_{\overrightarrow{v}})$.

Dla wektora wejściowego $\overrightarrow{v}$ w filtrze przechowywana jest jedynie wartość $f_{\overrightarrow{v}}$. Wartości $h_1(\overrightarrow{v})$ oraz $h_2(\overrightarrow{v})$ służą do wskazania potencjalnych kubełków. W przypadku, gdy któryś z kubełków jest pusty, wartość $f_{\overrightarrow{v}}$ zapisywana jest w tym kubełku. W przeciwnym wypadku, następuje relokacja, a kolejny potencjalny kubełek wyznaczany jest poprzez wyznaczenie wartości $j = i \oplus \text{hash}(f_{old})$, gdzie i oznacza numer obecnego kubełka, a f_{old} - wartość przechowywaną do tej pory w kubełku.

Przykład 4.6. Niech $S = \{\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w}, \overrightarrow{x}\}$, $b_c = 1$ oraz $k_c = 2$. Początkowo filtr jest pusty. Dla każdego wektora z S obliczana jest wartość jego odcisku. Niech teraz $h_1(\overrightarrow{u}) = 2$ oraz $h_2(\overrightarrow{u}) = 7$. Ze względu na to, że oba kubełki są puste, wartość $f_{\overrightarrow{u}}$ wyznaczona dla wektora $\overrightarrow{u}$ zapisywana jest do jednego z nich. Przyjmijmy, że wykorzystano wartość $h_1(\overrightarrow{u})$. Niech teraz $h_1(\overrightarrow{v}) = 2$ oraz $h_2(\overrightarrow{v}) = 5$. Ze względu na to, że wartość $h_1(\overrightarrow{v})$ wskazuje na pełny kubełek, wartość $f_{\overrightarrow{v}}$ umieszczana jest w kubełku wyznaczonym z wykorzystaniem wartości $h_2(\overrightarrow{v})$. Niech teraz $h_1(\overrightarrow{x}) = 5$ oraz $h_2(\overrightarrow{x}) = 7$. Dodatkowo załóżmy, że wartość $f_{\overrightarrow{w}}$ umieszczona została wcześniej w kubełku numer siedem. Ze względu na to, że oba kubełki są pełne, konieczna jest realizacja procedury relokacji. Wartość $f_{\overrightarrow{x}}$ zapisana zostaje w kubełku wyznaczonym z wykorzystaniem wartości $h_1(\overrightarrow{x})$. Przechowywana tam wcześniej wartość $f_{\overrightarrow{v}}$ musi zostać przeniesiona do innego kubełka. Obliczana jest zatem wartość $j = 5 \oplus hash(f_{\overrightarrow{v}})$, która wskazuje kubełek, do którego przeniesiona zostanie wartość $f_{\overrightarrow{v}}$. Jeżeli kubełek jest pusty, procedura relokacji kończy się. W przeciwnym wypadku, wartość która była tam zapisana przenoszona jest analogicznie do wartości $f_{\overrightarrow{v}}$. △

Procedura sprawdzenia, czy dowolny wektor wejściowy należał do zbioru wektorów wejściowy, polega na wyznaczeniu wartości $h_1(\overrightarrow{v})$ oraz $h_2(\overrightarrow{v})$ i sprawdzenia czy zawierają one odcisk wektora $\overrightarrow{v}$. W przypadku, gdy operacja zwróciła wynik pozytywny, to możliwe jest usunięcie wektora z filtru poprzez usunięcie jego odcisku z kubełka.

Optymalny rozmiar filtru analizowany był przez jego twórców. Udowodnili oni, że zajętość filtru poprawia się dla większych rozmiarów kubełków. Większe kubełki wymagają dłuższych odcisków w celu uzyskania tego samego prawdopodobieństwa wyniku fałszywie pozytywnego. Optymalne wartości parametrów określone zostały w następujący sposób: $k_c = 2$ oraz $b_c = 4$. Wartości te wykorzystane zostały w kolejnym podrozdziale do oceny filtru.

Niech l_c oznacza długość odcisku f. Prawdopodobieństwo, że podczas operacji sprawdzenia czy wektor v należał do zbioru S natrafimy na wartość f taką, że $f = f_{\overrightarrow{v}}$, a będzie to wynik fałszywie pozytywny wynosi $\frac{1}{2^{l_c}}$. Liczba operacji porównania dla wszystkich wpisów w filtrze wynosi z kolei $2 \cdot b_c$. Prawdopodobieństwo wyniku fałszywie pozytywnego

dla filtru można oszacować z wykorzystaniem następującej nierówności:

$$\epsilon \geqslant \frac{2 \cdot b_c}{2^{l_c}} \tag{4.12}$$

Minimalna długość odcisku f w filtrze wynosi z kolei:

$$l_c^* \approx \left\lceil log_2 \left(\frac{2 \cdot b_c}{\epsilon} \right) \right\rceil = \lceil -log_2(\epsilon) + log_2(b_c) + 1 \rceil \tag{4.13}$$

Ze względu na to, że przyjęto $b_c = 4$ otrzymujemy $l_c^* \approx \lceil -log_2(\epsilon) + 3 \rceil$. Należy zwrócić uwagę, że filtr Cuckoo wymaga większej ilości pamięci, niż wynika jedynie z długości odcisków. Związane jest to z koniecznością posiadania pustych kubełków, aby procedura relokacji nie kończyła się niepowodzeniem. W związku z tym, wprowadza się dodatkowy parametr α. Jego wartość określa maksymalne zapełnienie filtru i zależna jest od rozmiaru kubełków. Dla $k_c = 2$ oraz $b_c = 4$ wartość tego parametru wynosi $\alpha = 95,5\%$. Optymalna liczba bitów na element określona może być na podstawie następującej zależności:

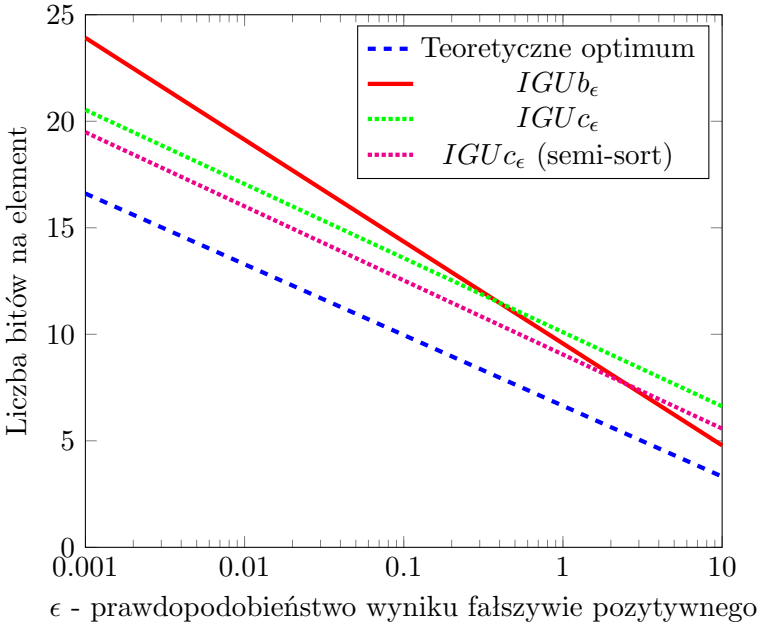
$$\psi_c^* \approx \frac{l_c^*}{\alpha} \tag{4.14}$$

W przypadku, gdy $b_c = 4$ możliwe jest uzyskanie jeszcze lepszej efektywności pamięciowej, dzięki zastosowaniu techniki na wpół-sortowania (ang. semi-sorting). Polega ona na sortowaniu elementów w kubełkach i kodowaniu sekwencji posortowanych odcisków. Wymaga ona dodatkowych tablic kodowania i dodatkowych operacji w procedurze sprawdzenia czy wektor jest w filtrze. Jednakże zmniejsza ona minimalną długość odcisku o jeden bit.

W tabeli 4.2 przedstawiono optymalną liczbę bitów na element dla filtru Blooma oraz filtru Cuckoo w zależności od wartości prawdopodobieństwa wyniku fałszywie pozytywnego na podstawie równań 4.8 oraz 4.14. W celu lepszego zobrazowania zależności, na rysunku 4.7 przedstawiono zależność wykres optymalnej liczby bitów dla $\epsilon < 10\%$. Dodatkowo, zamieszczono prostą odpowiadającą za teoretyczne optimum wynikające z teorii informacji. Należy zauważyć, że $\psi_b^* = \psi_c^* \Leftrightarrow \epsilon \approx 0,39\%$. Oznacza to, że dla mniejszych wartości prawdopodobieństwa filtr Cuckoo wykorzystuje mniej pamięci od filtru Blooma. Zastosowanie techniki na wpół-sortowania powoduje zwiększenie wartości granicznej prawdopodobieństwa, dla której wykorzystanie pamięci w filtrze Cuckoo jest mniejsze do $\epsilon \approx 2,49\%$. Dla większych wartości ϵ celowe okazuje się wykorzystanie filtru Blooma.

Tabela 4.2: Optymalna liczba bitów na element w stukturze

Struktura	Liczba bitów na element
$IGUb_\epsilon$	$-1.44 \cdot log_2(\epsilon)$
$IGUc_\epsilon$	$\frac{-log_2(\epsilon)+3}{\alpha}$
$IGUc_\epsilon$ (z techniką na wpół-sortowania)	$\frac{-log_2(\epsilon)+2}{\alpha}$



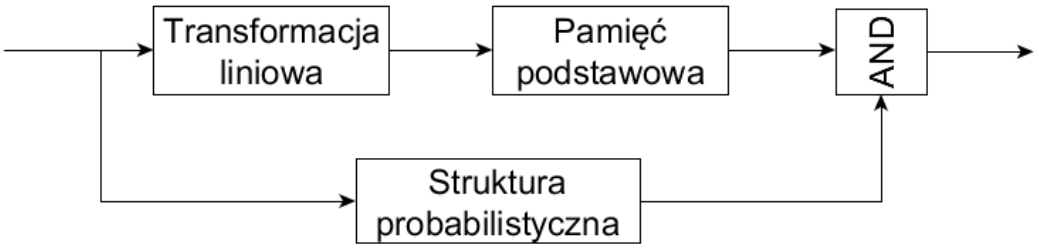
Rysunek 4.7: Liczba bitów na element [Maz19a]

4.2.4 Uzyskane wyniki

Badania nad wykorzystaniem struktur probabilistycznych do realizacji generatorów indeksów podzielone były na dwa etapy. W pierwszym z nich dokonano analizy możliwości zastosowania filtrów Blooma i Cuckoo [MBŁ18]. W drugim etapie dokonano analizy możliwości zastąpienia filtru Blooma filtrem z pojedynczą funkcją skrótu [Maz19a]. Przeanalizowano również możliwość realizacji tego filtru w strukturach programowalnych w celu oceny kosztu (pod względem wykorzystania zasobów pamięciowych) dodatkowych operacji.

Jak już wspomniano, struktury probabilistyczne mogą być zastosowane w celu zastąpienia pamięci dodatkowej w IGU. Proponowana architektura rozwiązania z ich wykorzystaniem przedstawiona została na rysunku 4.8. Transformacja liniowa oraz pamięć podstawowa realizują tę samą funkcjonalność co w IGU. Struktura probabilistyczna jest z kolei wykorzystywana do określenia czy wektor wejściowy należy do zbioru wektorów rejestrowanych. Zarówno filtr Blooma, jak i filtr Cuckoo powodują konieczność wykonania dodatkowych obliczeń:

- filtr Blooma - konieczne jest obliczenie k_b skrótów z wektora wejściowego,
- filtr Cuckoo - konieczne jest obliczenie k_c skrótów z wektora wejściowego oraz ewentualne zrealizowanie operacji relokacji.



Rysunek 4.8: Generator indeksów wykorzystujący strukturę probabilistyczną

Zaproponowana architektura nie korzysta jednak z pamięci dodatkowej oraz komparatora. W zależności od wybranej struktury probabilistycznej stosowane będą następujące oznaczenia (przy ustalonym prawdopodobieństwie wyniku fałszywie pozytywnego ϵ):

1. $IGUb_\epsilon$ - generator indeksów wykorzystujący filtr Blooma,
2. $IGUo_\epsilon$ - generator indeksów wykorzystujący filtr Blooma z pojedynczą funkcją skrótu,
3. $IGUc_\epsilon$ - generator indeksów wykorzystujący filtr Cuckoo,

Należy zwrócić uwagę, że zaproponowana architektura cechuje się dużą uniwersalnością. Dowolna inna struktura, która rozwiązuje problem przynależności do zbioru może być wykorzystana w proponowanej architekturze. Należy podkreślić, że w ostatnim czasie w literaturze pojawia się wiele propozycji nowych struktur probabilistycznych [GL20]. W związku z tym, istnieje szansa na dalszą redukcję wykorzystywanej pamięci albo na redukcję prawdopodobieństwa wyniku fałszywie pozytywnego, przy zastosowaniu proponowanej architektury (rysunek 4.8).

Na podstawie równań 4.8 oraz 4.14 możliwe jest określenie całkowitego wykorzystania pamięci w zaproponowanej architekturze. Dla filtru Blooma wynosi ono:

$$MEM_{IGUb_\epsilon} = 2^P \cdot Q + K \cdot \psi_b^* \approx 2^P \cdot Q - 1,44 \cdot K \cdot log_2(\epsilon) \tag{4.15}$$

Z kolei dla filtru Cuckoo:

$$MEM_{IGUc_\epsilon} = 2^P \cdot Q + K \cdot \psi_c^* \approx 2^P \cdot Q + 1,05 \cdot K \cdot (-log_2(\epsilon) + 3) \tag{4.16}$$

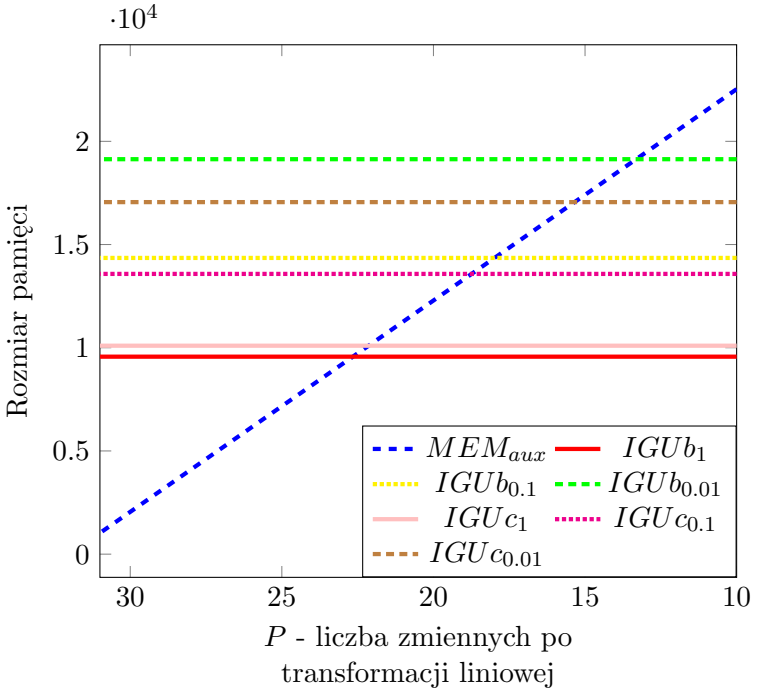
Łatwo zauważyć, że całkowity rozmiar pamięci różni się jedynie drugim czynnikiem w sumie. Na tej podstawie możliwe jest określenie granicznego prawdopodobieństwa

wyniku fałszywie pozytywnego, dla którego struktura probabilistyczna wykorzystuje tyle samo pamięci co pamięć dodatkowa w IGU. Otrzymujemy odpowiednio:

$$\widehat{\epsilon_b} \approx 2^{\frac{2^Q(N-P)}{-1.44*K}} \tag{4.17}$$

$$\widehat{\epsilon_c} \approx 2^{-\left(\frac{\alpha}{K}2^Q(N-P)\right)+3} \tag{4.18}$$

Na rysunku 4.9 przedstawiono porównanie rozmiaru pamięci dodatkowej w IGU oraz wykorzystania pamięci przez filtr Blooma oraz Cuckoo. Przedstawiono wyniki dla trzech wartości prawdopodobieństwa wyniku fałszywie pozytywnego: $\epsilon = 0,01\%$, $\epsilon = 0,1\%$ oraz $\epsilon = 1\%$. Przyjęto, że $N = 32$ oraz $K = 1000$. Jak łatwo zauważyć, istotną różnicą w porównaniu do IGU jest brak zależności rozmiaru pamięci wykorzystywanej przez struktury probabilistyczne od wartości P. W przypadku IGU, im lepsza transformacja liniowa zostanie znaleziona, tym rozmiar pamięci dodatkowej jest większy. W przypadku gdy wartość P zbliża się do wartości optymalnej, rozmiar wykorzystywanej pamięci przez struktury probabilistyczne jest mniejszy nawet dla $\epsilon = 0,01\%$. Dowodzi to użyteczności zaproponowanej architektury.



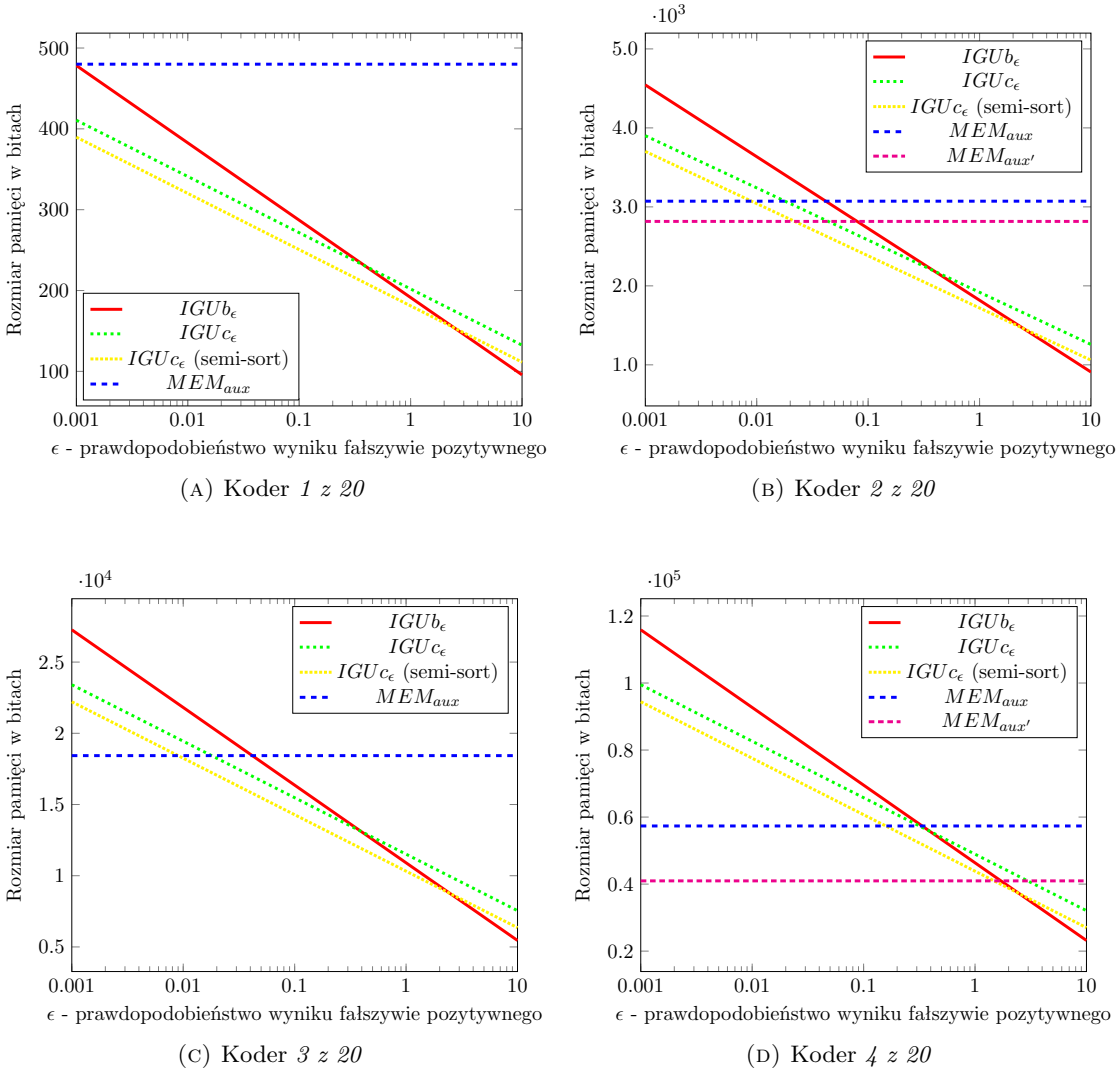
Rysunek 4.9: Wykorzystanie pamięci w zależności od wartości P

Ze względu na to, że kodery M z N powszechnie wykorzystywane są do oceny algorytmów syntezy logicznej funkcji generowania indeksów, celowe wydaje się przeanalizowanie zaproponowanej architektury pod względem realizacji tychże funkcji. W tabeli 4.3 przedstawiono wartości prawdopodobieństwa granicznego, wyznaczone zgodnie z równaniami

4.17 oraz 4.18. Założono, że $P = GD$, tzn. że została znaleziona optymalna transformacja liniowa. Należy jednak zauważyć, że np. dla kodera *2 z 20* nie istnieje taka transformacja [Sas17]. Przyjmując dla tego kodera wartość $P = GD + 1$ [MŁ19a] (zgodnie z wynikami z rozdziału 3) uzyskano odpowiednio $\widehat{\epsilon_b} = 0,0797\%$ oraz $\widehat{\epsilon_c} = 0,0439\%$. Co więcej, dla kodera *4 z 20* również nie znaleziono do tej pory optymalnej transformacji liniowej, a najlepsza znana reprezentacja ma $P = GD + 2$ zmienne wejściowe.

Tabela 4.3: Wartości $\widehat{\epsilon_b}$ oraz $\widehat{\epsilon_c}$ dla funkcji *M z 20*

M	1	2	3	4
$\widehat{\epsilon_b}$	0,0010%	0,0417%	0,0417%	0,3356%
$\widehat{\epsilon_c}$	0,0001%	0,0180%	0,0180%	0,3166%



(A) Koder *1 z 20*

(B) Koder *2 z 20*

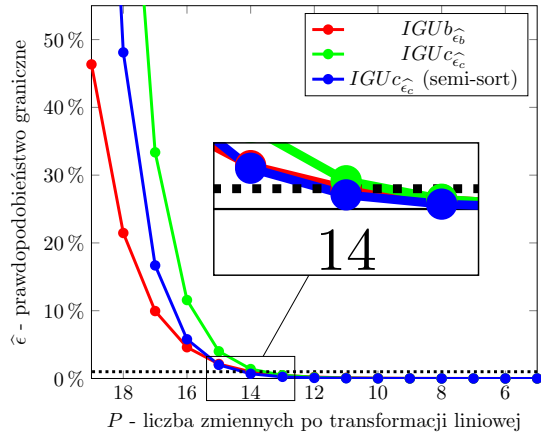
(C) Koder *3 z 20*

(D) Koder *4 z 20*

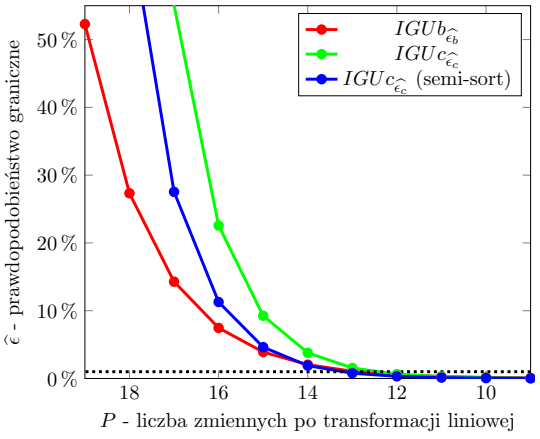
Rysunek 4.10: Rozmiar pamięci dodatkowej w IGU w porównaniu do rozmiaru struktur probabilistycznych, w zależności od wartości ϵ dla koderów *M z 20*

Rysunek 4.10 przedstawia porównanie rozmiaru pamięci dodatkowej w IGU z rozmiarem struktur probabilistycznych w zależności od wartości ϵ dla koderów *M z 20*. Niebieską linią, oznaczoną jako MEM_{aux}, przedstawiono wykorzystanie pamięci w sytuacji, gdy $P = GD$, tzn. gdy znaleziono optymalną transformację. Dla koderów *2 z 20* oraz *4 z 20* przedstawiono dodatkowo fioletową linię oznaczoną jako $MEM_{aux'}$, gdzie wartość P przyjęta została na podstawie wyników przedstawionych w literaturze [MŁ19a].

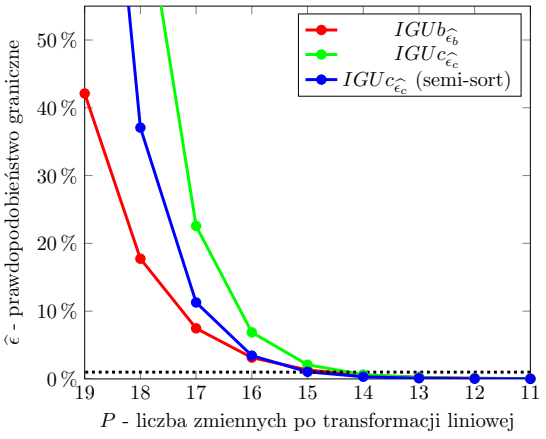
W sytuacji, gdy istnieje optymalna transformacja, to większa rzadkość funkcji implikuje mniejsze prawdopodobieństwo graniczne wyniku fałszywie pozytywnego. Jak już wspomniano, cechą charakterystyczną funkcji generowania indeksów jest spełnianie zależności $K \ll 2^N$. Dlatego proponowana architektura jest szczególnie efektywna dla tego typu funkcji boolowskich.



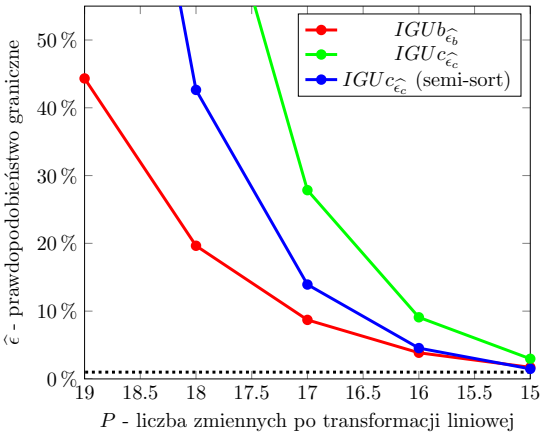
(A) Koder *1 z 20*



(B) Koder *2 z 20*



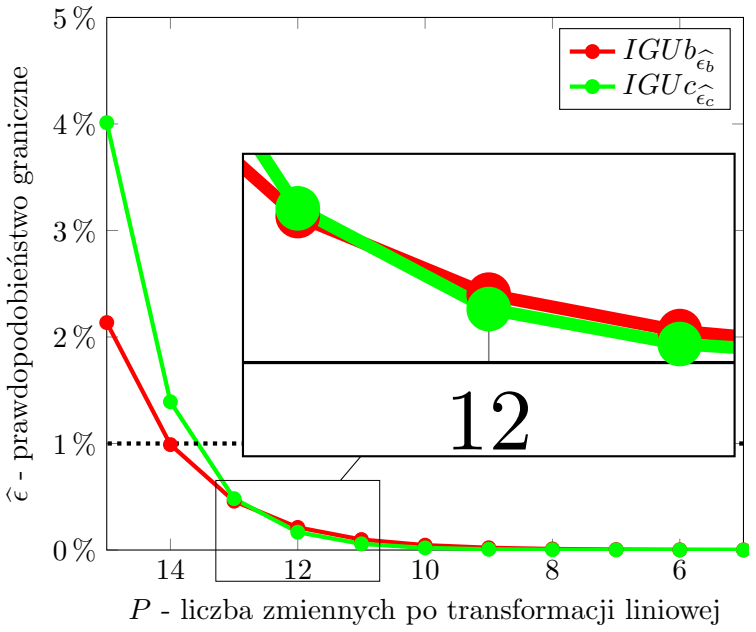
(C) Koder *3 z 20*



(D) Koder *4 z 20*

Rysunek 4.11: Prawdopodobieństwa graniczne dla koderów *M z 20* w zależności od wartości P

Na rysunku 4.11 przedstawiono wartości granicznego prawdopodobieństwa wyniku fałszywie pozytywnego w zależności od uzyskanej wartości P. Zarówno filtr Blooma, jak i filtr Cuckoo osiągają prawdopodobieństwo poniżej 1% dla wartości P znacząco odbiegającej od optymalnej - odpowiednio dla $P = 14$ oraz $P = 13$. Przy zastosowaniu techniki na wpół-sortowania w filtrze Cuckoo wartość P rośnie do 14. Dla większych wartości M prawdopodobieństwo graniczne rośnie. Analizując przedstawiony rysunek można zauważyć, że w sytuacji gdy $N \approx P$, to prawdopodobieństwo wyniku fałszywie pozytywnego jest bardzo wysokie. Celowe jest zatem wykorzystanie IGU w takich sytuacjach. W przypadku, gdy $P \ll N$, to proponowana architektura wykorzystująca struktury probabilistyczne zapewnia mniejsze wykorzystanie pamięci oraz relatywnie niskie prawdopodobieństwo wyniku fałszywie pozytywnego.



Rysunek 4.12: Prawdopodobieństwa graniczne dla kodera *1 z 20* w zależności od wartości P

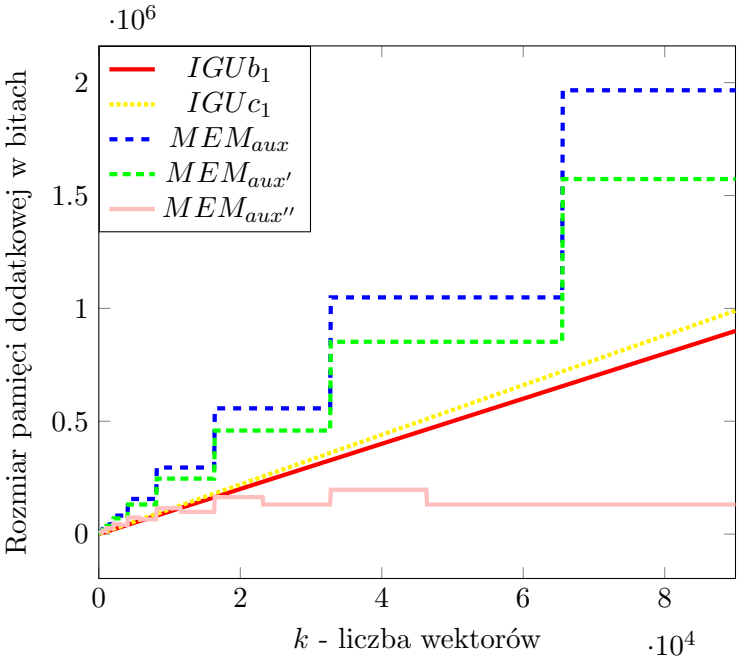
Należy zauważyć, że w sytuacji gdy prawdopodobieństwo to przekracza ok. $0,39\%$, to $\widehat{\epsilon_b} < \widehat{\epsilon_c}$. W celu lepszego zobrazowania tej sytuacji, na rysunku 4.12 przedstawiono wycinek rysunku 4.11a. Dla $P > 12$ prawdopodobieństwo dla filtru Blooma jest mniejsze niż w przypadku filtru Cuckoo. Dla $P = 12$ otrzymujemy natomiast odpowiednio $\widehat{\epsilon_b} = 0,211\%$ oraz $\widehat{\epsilon_c} = 0,167\%$. Dla mniejszych wartości P zachodzi $\widehat{\epsilon_b} > \widehat{\epsilon_c}$. Potwierdza to wcześniejszą obserwację, że filtr Cuckoo należy stosować w sytuacji, gdy $\epsilon < 0,39\%$. W przypadku zastosowania techniki na wpół-sortowania wartość graniczna wynosi ok. $2,49\%$. Oznacza to, że dla $P \leqslant 15$ gwarantuje ona najmniejsze wykorzystanie pamięci oraz zapewnia najmniejszą wartość prawdopodobieństwa wyniku fałszywie pozytywnego.

Aby nie zmniejszać przejrzystości rysunku, nie przedstawiono na nim wykresu funkcji odpowiadającego tej strukturze.

Ostatnim analizowanym zagadnieniem jest wpływ liczby wektorów K na zajętość pamięci. W tym celu porównano rozmiar filtru Blooma i filtru Cuckoo przy ustalonym prawdopodobieństwie wyniku fałszywie pozytywnego $\epsilon = 1\%$ z rozmiarem pamięci dodatkowej w IGU. Dla funkcji z $N = 32$ przeanalizowano trzy scenariusze:

1. kiedy $P = GD$,
2. kiedy $P = GD + 3$,
3. kiedy $P = GG$.

Otrzymane wyniki przedstawiono na rysunku 4.13. Wspomniane scenariusze oznaczono odpowiednio MEM_{aux}, $MEM_{aux'}$ oraz $MEM_{aux''}$. Jak widać, w sytuacji gdy wartość P jest zbliżona do GD, wykorzystanie pamięci przez struktury probabilistyczne jest mniejsze niż rozmiar pamięci dodatkowej w IGU. Z kolei, gdy wartość ta zbliżona jest do GG, to struktury probabilistyczne okazują się nieefektywne. Należy zauważyć, że dla $K > 18000$ liczba wektorów nie wpływa na zaobserwowane zależności. Dla wybranych mniejszych wartości wartość $MEM_{aux''}$ jest większa niż rozmiar $IGUb_1$ oraz $IGUc_1$.



Rysunek 4.13: Rozmiar pamięci dodatkowej w zależności od liczby wektorów (na podstawie [MBŁ18])

Jak wykazano, zastosowanie filtrów Blooma i Cuckoo gwarantuje lepszą efektywność pamięciową generatorów indeksów. Jednakże wymaga ono zastosowania wielu funkcji skrótu. Przekłada się na dodatkowe wykorzystanie zasobów logicznych w przypadku

realizacji sprzętowych. Na przykład pojedynczy moduł wyznaczający wartość skrótu algorytmem MD5 wykorzystuje ponad 730 ALM [Int19]. Zastosowanie algorytmów niekryptograficznych, ze względu na ich gorsze właściwości statystyczne, może z kolei prowadzić do zwiększenia prawdopodobieństwa wyniku fałszywie pozytywnego. Dlatego celowe wydaje się zastosowanie innej struktury probabilistycznej, która ogranicza liczbę skomplikowanych operacji z punktu widzenia implementacji sprzętowych. W dalszej części oceniono możliwość zastosowania filtru Blooma z pojedynczą funkcją skrótu w zaproponowanej architekturze. Ograniczenie liczby funkcji skrótu z kilku do zaledwie jednej pozwala w łatwy sposób zmniejszyć zajętość warstwy obliczeń skrótów. Wprowadza jednak konieczność zastosowania warstwy operacji modulo. Jak wykazano, operacja ta może być efektywnie zrealizowana w strukturach programowalnych dzięki zastosowaniu podejścia iteracyjnego [BS11].

W tabeli 4.4 przedstawiono przykładowe wartości modulników. Uzyskane zostały one za pomocą metody zaproponowanej przez twórców filtru, zaimplementowanej z wykorzystaniem środowiska SageMath [Sage]. Przyjęto następujące wartości parametrów: $\epsilon = 1\%$, $N = 20$, $m_b = 10 \cdot K$ oraz $k_b = 7$. Jak łatwo zauważyć, dla wszystkich przypadków uzyskano $m_b \neq m_o$. Ostatnia kolumna przedstawia oszacowanie prawdopodobieństwa wyniku fałszywie pozytywnego w filtrze, wyznaczone na podstawie nierówności 4.11. Uważny czytelnik zauważy, że wartość K odpowiada liczbie wektorów rejestrowanych w koderach *M z 20*, dla $M = [1, 4]$.

Tabela 4.4: Wartości poszczególnych m_i dla $\epsilon = 1\%$ [Maz19b]

K	$\{m_i : i = [1, k_b]\}$	m_b	m_o	ϵ_o
20	17, 19, 23, 29, 31, 37, 41	200	197	1,37%
190	257, 263, 269, 271, 277, 281, 283	1900	1901	0,82%
1140	1609, 1613, 1619, 1621, 1627, 1637, 1657	11400	11383	0,83%
4845	6883, 6899, 6907, 6911, 6917, 6947, 6949	48450	48413	0,82%

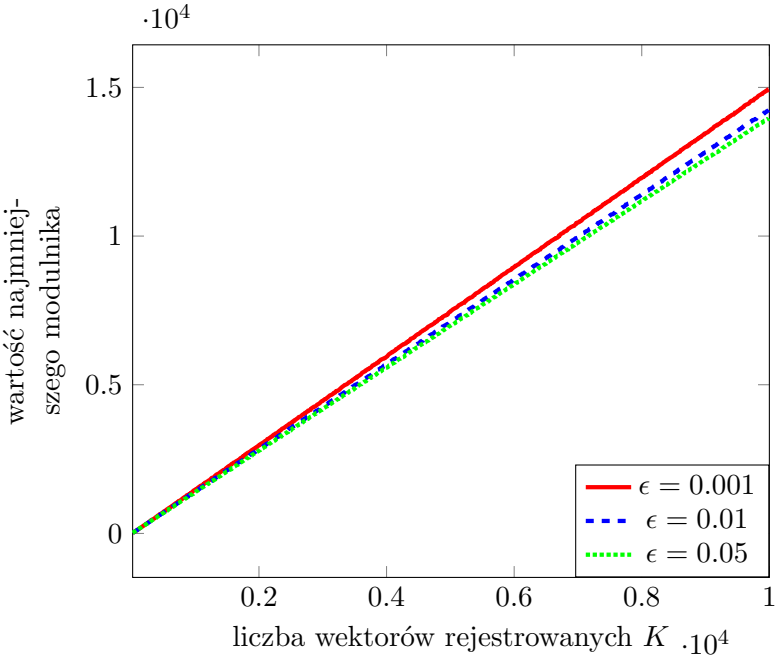
Tabela 4.5: Wartości poszczególnych m_i dla $\epsilon = 0,5\%$ [Maz19a]

K	$\{m_i : i = [1, k_b]\}$	m_b	m_o	ϵ_o
20	17, 19, 23, 29, 31, 37, 41, 43	240	240	0,56%
190	269, 271, 277, 281, 283, 293, 307, 311	2280	2292	0,31%
1140	1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733	13680	13644	0,32%
4845	7237, 7243, 7247, 7253, 7283, 7297, 7307, 7309	58140	58176	0,31%

Należy zauważyć, że dla małych wartości m_b, w tym wypadku $m_b = 200$, górna granica wynikająca z oszacowania wartości ϵ_o jest większa od oczekiwanego prawdopodobieństwa wyniku fałszywie pozytywnego, tzn. $\epsilon_o > \epsilon$. Dla pozostałych wartości uzyskano $\epsilon_o < \epsilon$. Podobne zależności uzyskano dla $\epsilon = 0,5\%$. Wyniki przedstawiono w tabeli 4.5. Przyjęto, że $m_b = 12 \cdot K$. Warto zauważyć, że dla $K = 20$ uzyskano równość $m_b = m_o$.

Co istotne, mniejsze wartość parametru ϵ implikuje wykorzystanie modulników o większych wartościach. Na przykład, dla $K = 4845$ konieczne jest zastosowanie liczby 7309. Dla $\epsilon = 1\%$ największą liczbą pierwszą dla tej wartości K było 6949. Konieczność zastosowania większej liczby modulników wynika z równania 4.7. W celu zobrazowania tej zależności, na rysunku 4.14 przedstawiono jak zmienia się wartość najmniejszego spośród modulników dla rosnącej wartości K. Łatwo zauważyć, że im większa liczba wektorów rejestrowanych, tym różnica w wartościach m_1 dla poszczególnych wartości prawdopodobieństwa wyniku fałszywie pozytywnego ϵ rośnie.

Należy zauważyć, że rozmiar filtru z pojedynczą funkcją skrótu wyznaczany jest w taki sposób, aby $m_b \approx m_o$. Oznacza to, że wykorzystanie pamięci przez tę strukturę probabilistyczną w proponowanej architekturze będzie porównywalne z filtrem Blooma. Dlatego we wcześniejszych analizach porównawczych filtrów w miejsce filtru Blooma można wstawić filtr z pojedynczą funkcją skrótu, bez utraty poprawności analizy.

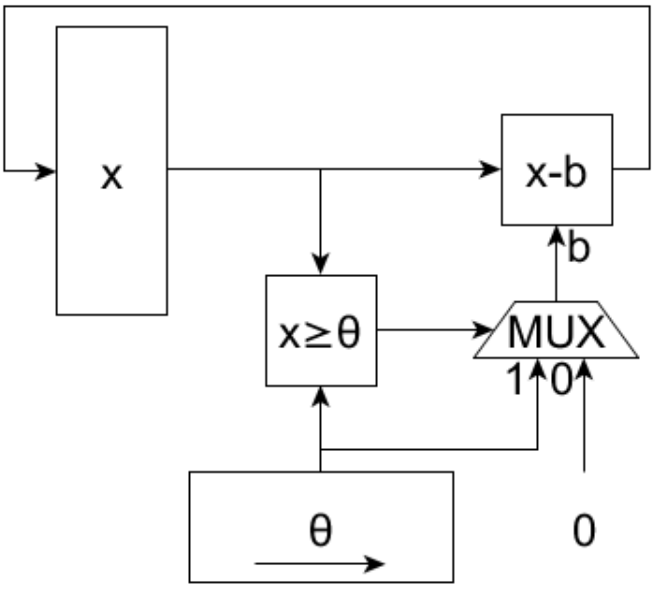


Rysunek 4.14: Wartość najmniejszego modulnika w zależności od liczby wektorów rejestrowanych [Maz19a]

Implementacja filtru Blooma z pojedynczą funkcją skrótu wymaga implementacji dwóch modułów: funkcji skrótu oraz warstwy operacji modulo. Ze względu na to, że w filtrze wykorzystywana jest tylko jedna funkcja skrótu, możliwe jest zastosowanie niekryptograficznej funkcji skrótu. Przykładem takiej funkcji jest algorytm FNV [FNV91]. Do wyznaczania wartości skrótu wykorzystuje on jedynie mnożenie przez liczbę pierwszą oraz operację XOR, dzięki czemu jest on łatwo implementowalny w sprzęcie [Aug18]. Opracowana implementacja [Maz19a] dla układu Cyclone V firmy Intel wykorzystuje zaledwie

51 ALM, 128 rejestrów oraz dwa bloki DSP. Bloki te realizują operację mnożenia. Innym rozwiązaniem jest zastosowanie algorytmu CRC-32, który zdaje się zapewniać najlepsze właściwości statystyczne spośród niekryptograficznych funkcji skrótu [Lu+18]. Implementacja tego algorytmu [Maz19a], przy założeniu 32-bitowej szyny wejściowej, zajmuje jedynie 72 ALM i wymaga zaledwie jednego taktu zegara.

Moduł realizujący warstwę operacji modulo wykorzystywać może architekturę przedstawioną na rysunku 4.15. Wynik otrzymywany jest poprzez iteracyjne wykonywanie operacji odejmowania. Dzięki temu moduł ten wykorzystuje mniej zasobów niż w pracy [BS11], w której zaproponowany został jako schemat szybkiego sprzętowego obliczania wartości $Z \equiv X \pmod{Y}$. Maksymalna częstotliwość zegara jest wysoka, przy jednoczesnym niewielkim wykorzystaniu zasobów logicznych.



Rysunek 4.15: Układ do realizacji pojedynczej operacji modulo [Maz19a] (na podstawie [BS11])

Niech wartość wejściowa X reprezentowana będzie jako η-bitowa liczba zapisana binarnie. Możemy ją wtedy przedstawić z wykorzystaniem następującej równości

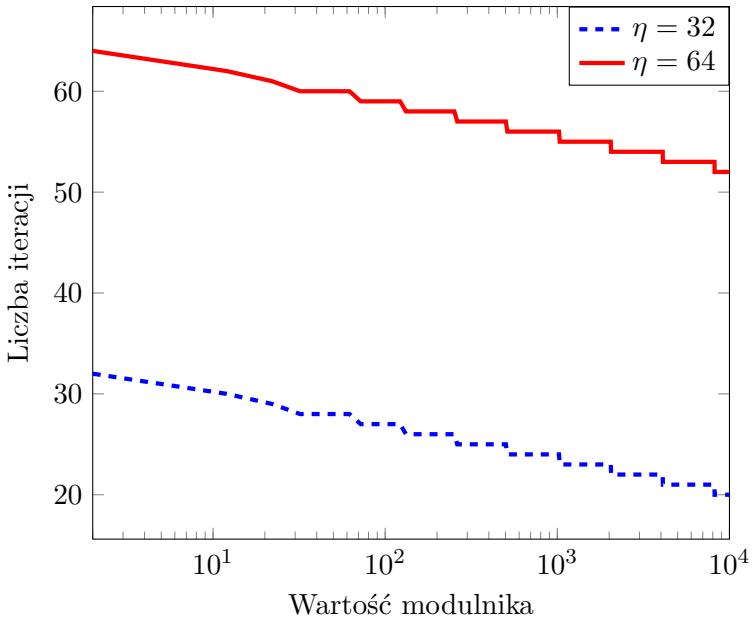
$$X = Z_i + \sum_{j=0}^{\tau} 2^j \cdot q_j \cdot m_i \tag{4.19}$$

Poprzez q_j oznaczono j-ty bit z binarnej reprezentacji X. Wartość $\tau \in \mathbb{Z}$ jest najmniejszą liczbą taką, że $2^\tau \cdot m_i > 2^\eta$. Zaproponowana architektura wymaga $(\tau + 1)$ iteracji, aby wyznaczyć poszukiwaną wartość m_i $(i = [1, k_b])$.

Rejestr x inicjowany jest wartością wejściową X. Rejestr θ inicjowany jest z kolei wartością $2^\tau \cdot m_i$. W implementacjach sprzętowych wartość ta wyznaczana jest poprzez konkatenację wartości m_i z τ zerami, tzn. $\{Y \parallel 0^\tau\}$. Każda iteracja działania modułu wymaga dzielenia wartości przechowywanej w tym rejestrze przez dwa. Realizowane jest to poprzez przesunięcie wartości w prawo o jeden.

W celu realizacji całej warstwy operacji modulo konieczne jest zastosowanie k_b modułów. Każdy z nich realizuje operację $Z_i \equiv X \pmod{m_i}$. Działania te wykonywane są równolegle. W przypadku konieczności dalszej redukcji wykorzystywanych zasobów, możliwe jest zrealizowanie wszystkich obliczeń z wykorzystaniem pojedynczego modułu. Skutkować będzie to jednak dłuższym czasem działania oraz koniecznością przechowywania wyników poszczególnych obliczeń. Z drugiej strony, dla małych wartości m_i układ kombinacyjny może generować wynik w jeden cykl zegara. Zależność pomiędzy wartością modulnika, a wartością τ przedstawiono na rysunku 4.16. Przyjęto $\eta = 32$ oraz $\eta = 64$, gdyż są to długości słowa maszynowego wykorzystywanego przez filtr Blooma z pojedynczą funkcją skrótu. Jak widać, im większa wartość modulnika m_i, tym mniejsza liczba iteracji jest potrzebna do wyliczenia poszukiwanej wartości Z_i. Na podstawie tej obserwacji oraz zależności z rysunku 4.14 sformułować można wniosek, że filtr jest najbardziej efektywny pamięciowo, pod względem liczby cykli zegara potrzebnych do realizacji obliczeń, dla dużych wartości K oraz małych wartości ϵ.

Przy takiej realizacji warstwy modulo zakłada się, że wartości modulników są znane na etapie implementacji filtru.



Rysunek 4.16: Liczba iteracji w zależności od wartości modulnika [Maz19a]

Przykład 4.7. Niech $X = 99$ oraz $m_i = 17$. Wtedy $x = 99$, $\tau = 3$ oraz $\theta = 2^3 \cdot 17 = 136$. Ze względu na to, że $x < \theta$ multiplekser wysterowany jest wartością zero, a do rejestru x zapisywany jest wynik działania $x - 0$, tzn. jego wartość nie jest modyfikowana. Rejestr θ przesuwany jest o jeden w prawo, więc jego wartość po pierwszej iteracji równa jest $\theta = 68$. W drugiej iteracji $x = 99 > 68 = \theta$. Wartość x modyfikujemy zatem przez odjęcie od niej wartości θ, tzn. $x = 99 - 68 = 31$. Rejestr θ ponownie przesuwany jest w prawo. W trzeciej iteracji $x > \theta$, więc wartość x pozostaje bez zmian. Czwarta iteracja wyznacza końcową wartość działania, tzn. $x = 31 > 17 = \theta \Rightarrow x = 31 - 17 = 14$. W wyniku otrzymujemy $99 \equiv 14 \pmod{17}$. △

W tabeli 4.6 przedstawiono wyniki sprzętowej implementacji warstwy operacji modulo w układach rodziny Cyclone V firmy Intel. Uwzględniono wykorzystanie zasobów logicznych ALM oraz rejestrów, liczbę taktów zegara wymaganych do uzyskania wyniku T_{clk} oraz maksymalną częstotliwość zegara F_{max} [MHz]. Przyjęto dwie różne wartości prawdopodobieństwa wyniku fałszywie pozytywnego: $\epsilon = 0,1\%$ oraz $\epsilon = 1\%$.

Dla obu wartości ϵ uzyskano wysokie wartości maksymalnej częstotliwości zegara. Prowadzi to do niskich czasów potrzebnych do uzyskania wyniku końcowego, na poziomie $127 - 195$ ns. Wykorzystanie zasobów logicznych jest niewielkie, zwłaszcza przy $\epsilon = 1\%$ i nie przekracza 1% zasobów dostępnych w wykorzystywanym układzie. Na podstawie przedstawionych wyników możliwe jest określenie wykorzystania zasobów logicznych niezbędnych do realizacji generatora indeksów z wykorzystaniem filtru z pojedynczą funkcją skrótu. Na przykład, przy $\epsilon = 0,1\%$ i $K = 1140$ implementacja ta wymagała by ok. 640 ALM, 720 rejestrów i dwóch bloków DSP przy zastosowaniu algorytmu FNV. Jest to koszt dodatkowych operacji w filtrze, poza wykorzystaniem m_o bitów pamięci, które przeanalizowane zostało wcześniej. Liczba taktów zegara maleje dla rosnących wartości K, co związane jest z jej zależnością od wartości najmniejszego modulnika.

Tabela 4.6: Wyniki sprzętowej implementacji warstwy operacji modulo [Maz19a]

(a) $\epsilon = 0,1\%$

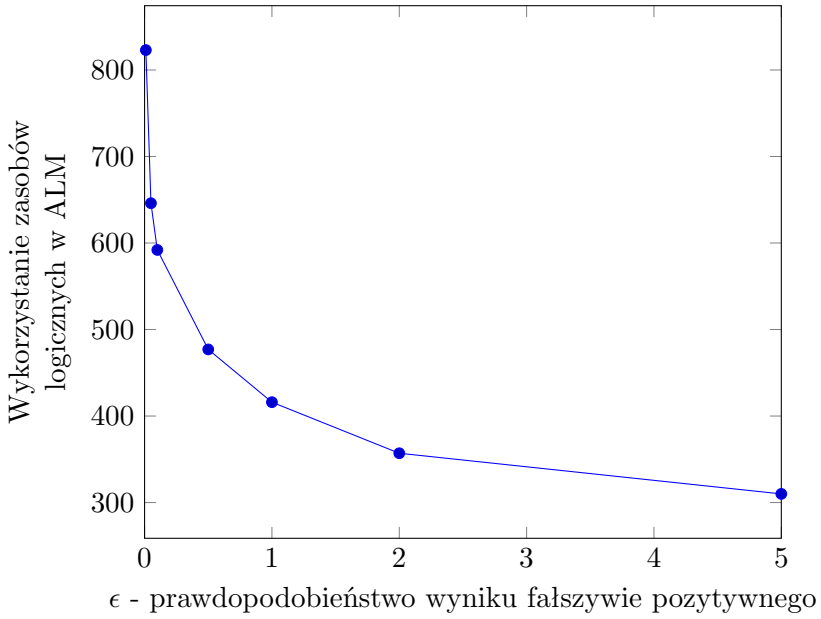
K	ALM	Rejestry	T_{clk}	F_{max}
20	606	641	29	148,63
190	585	601	24	148,52
1140	592	588	22	147,58
4845	617	574	20	141,18

(b) $\epsilon = 1\%$

K	ALM	Rejestry	T_{clk}	F_{max}
20	413	450	28	149,08
190	399	421	24	152,31
1140	416	415	22	157,51
4845	426	407	20	157,31

Na podstawie danych z tabeli 4.6 widać, że dla większych wartości ϵ wykorzystanie zasobów logicznych maleje. W celu lepszego zobrazowania tej zależności, przeanalizowano wykorzystanie zasobów dla różnych wartości ϵ przy ustalonym $K = 1140$. Wyniki przedstawiono na rysunku 4.17. Potwierdzają one spostrzeżenia wynikające z wyników z tabeli.

Wykorzystanie zasobów maleje logarytmicznie dla rosnących wartości ϵ. Związane jest to z mniejszą liczbą obliczeń w warstwie operacji modulo.



Rysunek 4.17: Wykorzystanie zasobów logicznych w zależności od wartości ϵ [Maz19a]

4.2.5 Wnioski z przeprowadzonych badań

W niniejszym rozdziale przedstawiono wyniki prac nad alternatywną w stosunku do IGU realizacją generatorów indeksów. W tym celu zaproponowano wykorzystanie struktur probabilistycznych i przeanalizowano możliwość zastosowania trzech z nich: filtr Blooma, filtr Blooma z pojedynczą funkcją skrótu oraz filtr Cuckoo.

Do najważniejszych zalet wspomnianych struktur należy brak zależności od wartości P. Dzięki temu nie otrzymuje się zależności jak w IGU, że im lepsza transformacja liniowa zostanie znaleziona, tym większe wykorzystanie pamięci potrzebnej do realizacji sprawdzenia przynależności wektora v do zbioru wektorów rejestrowanych. Wykazano, że w przypadku gdy $P \approx GD$, to zaproponowana architektura gwarantuje o wiele lepszą efektywność pamięciową. Co istotne, wraz z pojawianiem się w literaturze nowych struktur probabilistycznych o funkcjonalności takiej, jak filtr Blooma czy filtr Cuckoo, ale o mniejszej zajętości pamięci, możliwe powinno być ich wykorzystanie w zaproponowanej architekturze.

Wadą zaproponowanej architektury jest konieczność realizacji dodatkowych operacji, co przekłada się na dodatkowe wykorzystanie zasobów logicznych w przypadku implementacji w układach FPGA. Celowe wydaje się zatem stosowanie filtru Blooma z pojedynczą

funkcją skrótu. Struktura ta ogranicza liczbę wykorzystanych funkcji skrótu do jednej. Z kolei operacje modulo mogą być efektywnie zaimplementowane sprzętowo.

Oczywiście, proponowana architektura może być zastosowana tylko tam, gdzie akceptowane jest niezerowe prawdopodobieństwo wyniku fałszywie pozytywnego ϵ. Jednak jak pokazano (por. tabela 4.3), jeżeli $P \approx GD$ oraz $K \ll 2^N$ to prawdopodobieństwo to jest niewielkie. Co więcej, w niektórych zastosowaniach [Nak+09; NSM13] wynik działania generatora indeksów przetwarzany jest przez inny układ, np. MPU (ang. Microprocessing Unit), który może weryfikować czy nie otrzymano wyniku fałszywie pozytywnego.

Rozdział 5

Dekompozycja funkcjonalna

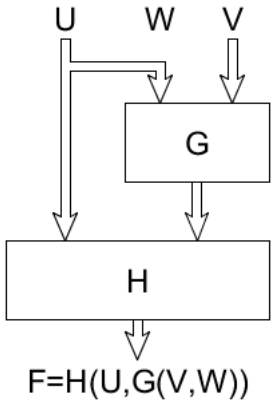
W niniejszym rozdziale przedstawiono wyniki badań nad zastosowaniem dekompozycji funkcjonalnej do minimalizacji funkcji generowania indeksów. Omówiono model dekompozycji funkcjonalnej oraz rozwiązania zaprezentowane w literaturze. Następnie zaproponowano heurystyczną metodę znajdowania nierozłącznej dekompozycji funkcjonalnej wykorzystującą elementy teorii grafów. Przedstawiono również metodę dokładną wykorzystującą uogólnienie problemu SAT. Dokonano analizy możliwości zastosowania zaproponowanych metod oraz wykazano ich użyteczność w realizacji funkcji generowania indeksów.

5.1 Model dekompozycji

Analizowane w ramach niniejszej pracy funkcje mogą być efektywnie realizowane sprzętowo z wykorzystaniem dekompozycji liniowej. Jednakże, nie dla wszystkich funkcji możliwe jest znalezienie dekompozycji takiej, że $P = GD$. W związku z tym konieczne jest zaproponowanie innej metody dalszej syntezy takich funkcji. Przykładem takiej metody jest dekompozycja funkcjonalna. Od wielu lat wykorzystywana jest ona w syntezie funkcji boolowskich [Ash57; Cur62; ŁS95; RJŁ01], jednak jej zastosowanie dla funkcji generowania indeksów nie zostało jeszcze dogłębnie przeanalizowane. Świadczy o tym niewielka liczba publikacji w literaturze krajowej [ŁM18; ŁM19] i zagranicznej [SMI16; SMI17].

Dekompozycja funkcjonalna polega na przedstawieniu funkcji $F(X)$ jako złożenia dwóch funkcji, oznaczanych jako G i H, tzn. $F(X) = H(U, G(V))$, gdzie $U, V \subset X$ oraz $U \cup V = X$. Funkcje te charakteryzują się mniejszą liczbą zmiennych wejściowych. W metodzie klasycznej zakłada się, że zbiory U i V są rozłączne, tzn. $U \cap V = \emptyset$. Schemat ten nazywa

się dekompozycją rozłączną. Bardziej prawdopodobne jest jednak istnienie dekompozycji nierozłącznej, tzn. $F(X) = H(U, G(V \cup W))$, gdzie $W \subseteq U$. Schemat takiej realizacji funkcji przedstawiono na rysunku 5.1.



Rysunek 5.1: Schemat (nierozłącznej) dekompozycji funkcjonalnej

Podstawowym problemem przy realizacji funkcji z wykorzystaniem przedstawionego schematu jest optymalny dobór zmiennych do zbiorów U, V oraz W. Należy zauważyć, że w przypadku realizacji obu funkcji z wykorzystaniem pamięci, całkowity rozmiar pamięci wynosi:

$$MEM_{fun} = 2^{|V|+|W|} \cdot \mu + 2^{\mu+|U|} \cdot Q \tag{5.1}$$

gdzie μ - liczba wyjść z funkcji G. Wybór zmiennych ma więc kluczowy wpływ na całkowity rozmiar pamięci. Istotnymi problemami jest również określenie liczby wejść do funkcji H oraz znalezienie odpowiedniej realizacji funkcji G.

Prace nad dekompozycją funkcjonalną funkcji generowania indeksów realizowane były przez zespół prof. Sasao [SMI16; SMI17]. W pierwszej z prac zaproponowano heurystyczną metodę wyznaczania postaci zbioru V. Metoda ta wykorzystuje miarę niejednoznaczności funkcji względem zbioru zmiennych, która wykorzystywana była również w algorytmie dekompozycji liniowej [Sas12]. W drugiej z prac zaproponowano algorytm dokładny znajdowania dekompozycji rozłącznej. Polega ona na znajdowaniu postaci zbioru V o określonej liczności s takiego, dla którego uzyskuje się najmniejszą wartość μ. Następnie, spośród znalezionych dekompozycji dla $s = [2, N-2]$ wybierana jest ta, która gwarantuje uzyskanie najmniejszego wykorzystania pamięci. Dodatkowe prace [SB18] realizowane były nad analizą możliwości wykorzystania metody Monte Carlo do zwiększenia wydajności algorytmu.

Podstawową wadą wspomnianych prac było poszukiwanie jedynie dekompozycji rozłącznej funkcji generowania indeksów. W dalszej części niniejszego rozdziału przedstawione

zostaną wyniki potwierdzające, że metoda ta jest niewystarczająca. Dekompozycja nierozłączna pozwala na uzyskanie oszczędniejszego wykorzystania pamięci.

5.2 Dekompozycja wykorzystująca rachunek podziałów

Do znalezienia reprezentacji funkcji boolowskiej z wykorzystaniem dekompozycji funkcjonalnej wykorzystany może zostać rachunek podziałów, który przedstawiony został w rozdziale 2.2. Warunek istnienia dekompozycji wynika z twierdzenia 5.1 [ŁB15].

Twierdzenie 5.1. *Funkcja* $F : D^N \rightarrow D^m$ *ma dekompozycję funkcjonalną postaci* $F = H(U, G(V, W))$ *wtedy i tylko wtedy, gdy* $\exists \Pi_G \geqslant P_{V \cup W} : P_U \cdot \Pi_G \leqslant P_F$.

W najlepszym przypadku zbiór W jest zbiorem pustym, a funkcja ma dekompozycję rozłączną. W najgorszym możliwe jest znalezienie dekompozycji takiej, że $W = U$. Jednakże, dekompozycja taka może prowadzić do realizacji funkcji wykorzystującej więcej pamięci.

W celu realizacji funkcji zgodnie z przedstawionym schematem konieczne jest rozwiązanie następujących problemów:

1. W jaki sposób przydzielić zmienne do zbiorów U i V?
2. Jak określić liczbę zmiennych wejściowych do funkcji H?
3. W jaki sposób przydzielić zmienne do zbioru W?
4. W jaki sposób znaleźć postać funkcji G?

Do rozwiązania dwóch pierwszych problemów wykorzystane może zostać pojęcie r-przydatności [ŁL94; Łu95; BŁT09]. Jego definicja wynika bezpośrednio z przedstawionego wcześniej rachunku podziałów. Zbiór $\{P_1 P_2 \dots P_k\}$ jest r-przydatny względem P_F wtedy i tylko wtedy, gdy:

$$r = k + \lceil log_2(\gamma(P_1 P_2 \dots P_k | P_1 P_2 \dots P_k P_F)) \rceil \tag{5.2}$$

gdzie $\gamma(P_1 P_2 \dots P_k | P_1 P_2 \dots P_k P_F)$ - liczba elementów w najliczniejszym bloku podziału ilorazowego. Poprzez P_i $(i = [1, N])$ rozumie się podział względem zmiennej wejściowej x_i.

Uzyskanie określonej r-przydatności warunkuje możliwość istnienia dekompozycji jak na rys. 5.1, gdzie funkcja H ma r wejść, tzn. $\mu + |U| = r$. Nie gwarantuje jednak istnienia dekompozycji takiej, że $W = \emptyset$. Funkcja G ma wtedy $r - |U|$ wyjść, a zbiór U tworzą zmienne wejściowe, które wykorzystane zostały do wyznaczenia podziału ilorazowego.

Wybierany jest podzbiór zmiennych wejściowych, który gwarantuje uzyskanie najmniejszej r-przydatności. Zbiór V tworzą z kolei pozostałe zmienne wejściowe, tzn. $V = X \backslash U$. Warunek konieczny r-przydatności zbioru wynika z lematu 5.2 [BŁ97].

Lemat 5.2. *Zbiór P jest r-przydatny wtedy i tylko wtedy, gdy wszystkie jego podzbiory są s-przydatne, gdzie $s \leqslant r$.*

Znalezienie r-przydatnych zbiorów mogących zostać wykorzystanych do realizacji funkcji wymaga iteracyjnego wyznaczania tej miary dla zbiorów podziałów o coraz większej liczności. Przy dużej liczbie zmiennych wejściowych N obliczenia stają się czasochłonne ze względu na dużą liczbę możliwych podzbiorów. Czas obliczeń można zmniejszyć wykorzystując lemat 5.3 [ŁB15].

Lemat 5.3. *Jeżeli $\{P_a\}$ oraz $\{P_b\}$ są r-przydatne, to zbiór $\{P_a, P_b\}$ jest co najwyżej $(r+1)$-przydatny. Jeżeli z kolei $\{P_b\}$ jest $(r+1)$-przydatny, to zbiór $\{P_a, P_b\}$ jest $(r+1)$-przydatny.*

Tabela 5.1: Przykładowa funkcja generowania indeksów

x_1	x_2	x_3	x_4	$F(X)$
0	1	1	0	1
1	1	0	1	2
0	0	0	0	3
1	1	1	0	4
0	0	0	1	5
0	1	0	0	6

Przykład 5.1. Niech dana jest funkcja przedstawiona w tablicy 5.1 [Maz20a]. Jest to funkcja generowania indeksów, dla której $K = 6$ oraz $N = 4$. Dla podziałów względem zmiennych wejściowych $x_i, i = [1, 4]$ otrzymano następujące wartości r-przydatności:

- $P_1 = \{\overline{1,3,5,6}; \overline{2,4}\} \Rightarrow r = 1 + \lceil log_2(4) \rceil = 3,$
- $P_2 = \{\overline{1,2,4,6}; \overline{3,5}\} \Rightarrow r = 1 + \lceil log_2(4) \rceil = 3,$
- $P_3 = \{\overline{1,4}; \overline{2,3,5,6}\} \Rightarrow r = 1 + \lceil log_2(4) \rceil = 3,$
- $P_4 = \{\overline{2,5}; \overline{1,3,4,6}\} \Rightarrow r = 1 + \lceil log_2(4) \rceil = 3.$

Ze względu na to, że wszystkie $\{P_i\}$ są 3-przydatne należy przypuszczać, że istnieje dwuelementowy zbiór, który również będzie 3-przydatny. W tym celu wykonuje się następujące obliczenia:

- $P_1P_2 = \{\overline{1,6}; \overline{3,5}; \overline{2,4}\} \Rightarrow r = 2 + \lceil log_2(2) \rceil = 3,$
- $P_1P_3 = \{\overline{1}; \overline{3,5,6}; \overline{4}; \overline{2}\} \Rightarrow r = 2 + \lceil log_2(3) \rceil = 4,$
- $P_1P_4 = \{\overline{5}; \overline{1,3,6}; \overline{2}; \overline{4}\} \Rightarrow r = 2 + \lceil log_2(3) \rceil = 4.$
- $P_2P_3 = \{\overline{1,4}; \overline{2,6}; \overline{3,5}\} \Rightarrow r = 2 + \lceil log_2(2) \rceil = 3,$

- $P_2P_4 = \{\overline{2}; \overline{5}; \overline{3}; \overline{1,4,6}\} \Rightarrow r = 2 + \lceil log_2(3) \rceil = 4$,
- $P_3P_4 = \{\overline{1,4}; \overline{2,5}; \overline{3,6}\} \Rightarrow r = 2 + \lceil log_2(2) \rceil = 3$.

Uzyskano trzy zbiory 3-przydatne oraz trzy 4-przydatne. Korzystając z lematu 5.2, wiadomo, że nie istnieje zbiór o liczności 3, który jest 3-przydatny. W związku z tym zbiór U może mieć jedną z następujących postaci: $\{x_1, x_2\}$, $\{x_2, x_3\}$ albo $\{x_3, x_4\}$. Ze względu na to, że $|U| = 2$ to liczba wyjść z funkcji G wynosi $r - |U| = 3 - 2 = 1$. Funkcja H ma z kolei $r = 3$ wejścia. Określenie czy funkcja ta posiada dekompozycję rozłączną, tzn. czy $W = \emptyset$, omówione zostanie w dalszej części rozprawy. △

Postać funkcji G może zostać określona z wykorzystaniem problemu kolorowania grafu [Raw+97; BŁP16]. W tym celu konieczne jest jednak wcześniejsze określenie postaci zbioru W. Kluczowym problemem jest znalezienie zbioru o jak najmniejszej liczności. Wybór zmiennych wykonany może być poprzez analizę podziału P_V, czyli iloczynu podziałów względem zmiennych wejściowych tworzących zbiór V [BŁT12]. Polega ona na poszukiwaniu podziału $P_G \geqslant P_{V \cup W}, W \subset U$, który spełnia twierdzenie 5.1. Sprowadza się to do konstruowania podziału P_G poprzez łączenie bloków $P_{V \cup W}$ w taki sposób, aby spełniona była relacja $P_U \cdot P_G \leqslant P_F$. W ramach niniejszej rozprawy zaproponowano dwie systematyczne metody znajdowania podziału P_G i zbioru W o minimalnej liczności. Metoda heurystyczna wykorzystuje elementy teorii grafów, z kolei metoda dokładna uogólnienie problemu SAT. Obie metody omówione zostały w dalszej części rozprawy.

5.2.1 Metoda heurystyczna

Do znalezienia postaci funkcji G oraz zbioru W zastosować można pojęcia i algorytmy teorii grafów [Raw+97]. W tym celu wykorzystane zostaną podziały $P_U|P_UP_F$ oraz P_V wyznaczone z wykorzystaniem przedstawionego wcześniej pojęcia r-przydatności.

Niech podział $P_{i,j}$ - podział powstały poprzez połączenie bloków B_i, B_j podziału P_V w jeden blok. Bloki te są zgodne [ŁB15], jeżeli podział $P_{i,j}$ spełnia warunek z twierdzenia 5.1, tzn. $P_U \cdot P_{i,j} \leqslant P_F$. W przeciwnym razie bloki te są niezgodne. Na podstawie znajomości podziałów $P_U|P_UP_F$ oraz P_V możliwe jest skonstruowanie grafu $\Gamma = (V_\Gamma, E_\Gamma)$ oraz wykorzystanie problemu kolorowania grafu do znalezienia postaci funkcji G. Wierzchołki $v \in V_\Gamma$ reprezentują bloki podziału P_V, podczas gdy krawędzie $e = (B_i, B_j) \in E_\Gamma$ reprezentują niezgodne bloki podziału ilorazowego $P_U|P_UP_F$. Należy zwrócić uwagę, że graf ten ma co najwyżej K wierzchołków.

Istotnym problemem pozostaje efektywne znalezienie postaci zbioru W. W tym celu zaproponowano heurystyczne podejście iteracyjne [MŁ19b], przedstawione jako Algorytm 5.1. Bazuje ono na obserwacji, że liczba chromatyczna grafu Γ nie może być większa niż

liczba możliwych różnych wartości na wyjściu z funkcji G. W przedstawionym schemacie dekompozycji oznacza to, że dla zadanych zbiorów U, V oraz wartości r funkcję można będzie zrealizować wtedy i tylko wtedy, gdy:

$$\chi(\Gamma) \leqslant \beta = 2^{r-|U|}. \tag{5.3}$$

W tym celu do zbioru W dodawane są iteracyjnie zmienne ze zbioru U, dopóki nie jest spełniony wspomniany warunek. Po dodaniu zmiennej x_i obliczana jest nowa postać podziału $P_{V\cup W}$ poprzez wyznaczenie iloczynu podziałów $P_{V\cup W}P_i$. Powoduje to konieczność zmodyfikowania grafu Γ. Nowy podział, indukowany argumentami $V \cup W$, powoduje zwiększenie liczby wierzchołków grafu. Jednocześnie liczba krawędzi pozostaje niezmieniona. Algorytm kończy działanie w momencie, gdy znaleziono pokolorowanie grafu takie, że $\chi(\Gamma) \leqslant \beta$ albo gdy nie można już dodać żadnej zmiennej do zbioru W, tzn. $W = U$. Druga sytuacja oznacza, że nie znaleziono postaci zbioru W umożliwiającej realizację funkcji.

Algorytm 5.1 Heurystyczne znajdowanie dekompozycji

Wejście: Tablica prawdy funkcji F, r-przydatność oraz postaci zbiorów U i V
Wyjście: Postać zbioru W oraz funkcji G

1: $W \leftarrow \emptyset, P_{V\cup W} = P_V$
2: Stwórz graf $\Gamma = (N_\Gamma, E_\Gamma)$ na podstawie podziałów $P_{V\cup W}$ oraz $P_U|P_UP_F$
3: $\beta \leftarrow 2^{r-|U|}$
4: Zastosuj algorytm kolorowania grafu w celu wyznaczenia liczby chromatycznej $\chi(\Gamma)$
5: Dopóki $\chi(\Gamma) > \beta$:
6: Znajdź zmienną x_i, która powinna być dodana do zbioru W
7: $W \leftarrow W \cup \{x_i\}$
8: $P_{V\cup W} \leftarrow P_{V\cup W}P_i$
9: Zmodyfikuj graf Γ uwzględniając nową postać podziału $P_{V\cup W}$
10: Zastosuj algorytm kolorowania grafu
11: Jeżeli $W = U$, to wyjdź z pętli
12: Zwróć zbiór W oraz pokolorowanie grafu Γ

Przykład 5.2. Dla funkcji przedstawionej w tablicy 5.1 w przykładzie 5.1 wyznaczono możliwe postaci zbioru U, które są 3-przydatne. Przyjmijmy, że do dalszych obliczeń wybrano następujący zbiór $U = \{x_1, x_2\}$. Wtedy $V = X\backslash U = \{x_3, x_4\}$.

Wykorzystując rachunek podziałów wyznaczyć można następujące podziały:

- $P_U = P_1P_2 = \{\overline{1,6}; \overline{3,5}; \overline{2,4}\}$,
- $P_U|P_UP_F = \{\overline{(1)(6)}; \overline{(3)(5)}; \overline{(2)(4)}\}$,
- $P_V = \{\overline{1,4}; \overline{2,5}; \overline{3,6}\} = \{B_1, B_2, B_3\}$.

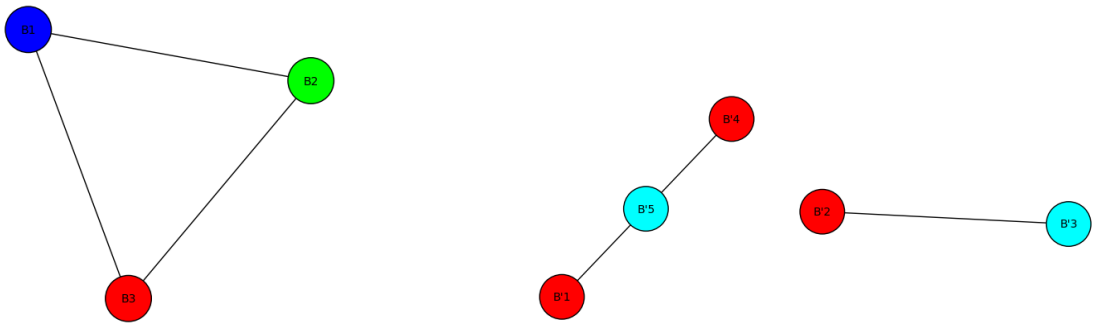
Analizując postaci podziałów P_V oraz $P_U|P_UP_F$ można zauważyć, że na ich podstawie tworzony jest graf o trzech wierzchołkach B_i i trzech krawędziach.

Krawędzie tworzą następujące pary wierzchołków:

- $1 \in B_1 \wedge 6 \in B_3 \Rightarrow (B_1, B_3) \in E_\Gamma$,
- $3 \in B_3 \wedge 5 \in B_2 \Rightarrow (B_2, B_3) \in E_\Gamma$,
- $2 \in B_2 \wedge 4 \in B_1 \Rightarrow (B_1, B_2) \in E_\Gamma$.

Otrzymany graf przedstawiono na rysunku 5.2a. Graf ten jest 3-kolorowalny. Jednakże, $\chi(\Gamma) = 3 > 2^{r-|U|} = 2$. Oznacza to, że nie istnieje dekompozycja rozłączna analizowanej funkcji i konieczne jest określenie postaci zbioru W. W tym celu należy znaleźć zmienną x_i, która rozdzieli jeden z bloków podziału P_V. Na przykład, dodanie zmiennej x_1 do zbioru W rozdziela wartości należące do bloku B_1. W związku z tym wyznaczany jest podział $P_{V \cup W}$, który jest iloczynem P_V oraz P_1:

$$P_{V \cup W} = P_V P_1 = \{\overline{1}; \overline{4}; \overline{2}; \overline{5}; \overline{3,6}\} = \{B_1^{'}, B_2^{'}, B_3^{'}, B_4^{'}, B_5^{'}\}$$



(A) Graf Γ (B) Pokolorowany graf Γ po jego modyfikacji

RYSUNEK 5.2: Postać grafu Γ w dwóch iteracjach algorytmu [Maz20a]

Graf powstały na podstawie tego podziału, przedstawiony na rysunku 5.2b, ma pięć wierzchołków. Jak widać, jest on 2-kolorowalny. Uzyskano zatem dekompozycje nierozłączną z następującymi postaciami zbiorów: $U = \{x_1, x_2\}$, $V = \{x_3, x_4\}$ oraz $W = \{x_1\}$. Wierzchołkom pokolorowanym z wykorzystaniem tego samego koloru przypisywana jest ta sama wartość wyjściowa funkcji G i otrzymano:

$$P_G = \{B_1^{'} \cup B_2^{'} \cup B_4^{'}; B_3^{'} \cup B_5^{'}\} = \{\overline{1,4,5}; \overline{2,3,6}\}$$

Uzyskane postaci funkcji G i H przedstawiono w tabeli 5.2. Należy podkreślić, że zastosowanie dostępnych wcześniej metod, wykorzystujących rozłączną dekompozycję funkcjonalną [Iwa16], prowadzi do uzyskania reprezentacji z $U = \{x_1\}$ oraz $V = \{x_2, x_3, x_4\}$. Wykorzystanie pamięci przy takiej reprzentacji jest aż o 25% wyższe niż przy zastosowaniu proponowanego algorytmu.

△

Tabela 5.2: Postać funkcji po dekompozycji

(a) Funkcja G

x_1	x_3	x_4	G
0	1	0	0
1	0	1	1
0	0	0	1
1	1	0	0
0	0	1	0
0	0	0	1

(b) Funkcja H

x_1	x_2	G	$F(X)$
0	1	0	1
1	1	1	2
0	0	1	3
1	1	0	4
0	0	0	5
0	1	1	6

Kluczowe znaczenie z punktu widzenia jakości otrzymywanego rozwiązania oraz złożoności obliczeniowej mają:

1. wybór algorytmu kolorowania grafu, wykorzystywanego w liniach 4. i 10. algorytmu 5.1,
2. metoda wyboru zmiennej x_i dodawanej do zbioru W w poszczególnych iteracjach algorytmu (linia 6.).

Jak już wspomniano w rozdziale 2.3, istnieje wiele algorytmów rozwiązujących problem kolorowania grafu. Algorytmy dokładne gwarantują uzyskanie wyniku optymalnego, jednak ich złożoność obliczeniowa jest bardzo duża. W ramach przedstawionej metody znajdowania dekompozycji funkcjonalnej konieczne może być kilkukrotne znalezienie pokolorowania grafu. W związku z tym należy spodziewać się małej efektywności czasowej w przypadku zastosowania algorytmu dokładnego, szczególnie dla funkcji o bardzo dużej liczbie zmiennych wejściowych N. W związku z tym celowe jest przeanalizowanie jakości uzyskiwanego wyniku w przypadku wyboru heurystycznego algorytmu kolorowania grafu [Maz20a].

W ramach przeprowadzonych eksperymentów przeanalizowano dwa algorytmy kolorowania grafu:

K1. dokładny - wykorzystujący mieszane całkowitoliczbowe programowanie liniowe (MILP, ang. *Mixed-Integer Linear Programming*),
K2. heurystyczny - Welsha-Powella, zwany również algorytmem LF (ang. *Largest First*) [WP67].

Pierwszy z algorytmów wybrany został dlatego, że oprogramowanie eksperymentalne przygotowane zostało z wykorzystaniem środowiska SageMath [Sage]. W środowisku tym wykorzystać można funkcję *sage.graphs.graph_coloring.vertex_coloring()* do znalezienia pokolorowania grafu. W tym celu wykorzystuje ona narzędzie rozwiązujące wspominany program liniowy. Algorytm Welsha-Powella wybrany został z kolei ze względu na

jego dużą efektywność obliczeniową [AB16]. Złożoność czasowa tego algorytmu wynosi $O(|N_\Gamma|^2)$. Nie zawsze jednak gwarantuje on uzyskanie wyniku zbliżonego do optymalnego.

Wybór zmiennej wejściowej $x_i \in U$ sprowadza się do określenia, dla której zmiennej liczba chromatyczna grafu $\chi(\Gamma)$ zostanie najbardziej zredukowana. Problem ten można rozwiązać poprzez znalezienie w grafie Γ wierzchołka, który warunkuje uzyskanie określonej wartości $\chi(\Gamma)$. W ten sposób znalezienie zmiennej x_i, która rozdziela blok podziału $P_{V\cup W}$ w dwa bloki podziału $P_{V\cup W}P_i$ może doprowadzić do zmniejszenia wartości $\chi(\Gamma)$. Zgodnie z najlepszą wiedzą autora niniejszej rozprawy nie istnieje efektywna metoda znajdowania takiego wierzchołka w grafie. Możliwa byłaby próba adaptacji algorytmów o złożoności wykładniczej [HH02]. Jednakże należy zwrócić uwagę, że wybór zmiennej x_i powoduje modyfikację grafu na podstawie podziału $P_{V\cup W}P_i$. W związku z tym zmienia się liczba wierzchołków, a także krawędzie. Zastosowanie tych algorytmów byłoby zatem nieefektywne. W ramach zrealizowanych prac przeanalizowano następujące metody wyboru zmiennej do zbioru W [MŁ19b; Maz20a]:

W1. poprzez wybór zmiennej rozdzielającej blok reprezentowany przez wierzchołek o największym stopniu,
W2. poprzez wybór zmiennej rozdzielającej blok reprezentowany przez wierzchołek należący do maksymalnej kliki w grafie,
W3. poprzez losowy wybór zmiennej ze zbioru U, która nie została jeszcze dodana do zbioru W.

Pierwsza z metod wynika z obserwacji, że dla grafu prostego zachodzi $\chi(\Gamma) \leqslant \Delta(\Gamma) + 1$. W związku z tym, poprzez rozdzielenie bloku reprezentowanego przez wierzchołek o maksymalnym stopniu możliwe jest zmniejszenie górnego ograniczenia na wartość $\chi(\Gamma)$. Z drugiej strony zachodzi $\chi(\Gamma) \geqslant \omega(\Gamma)$. W związku z tym rozdzielenie bloku reprezentowanego przez wierzchołek z kliki maksymalnej prowadzi do zmniejszenia dolnego ograniczenia na wartość $\chi(\Gamma)$. Należy jednak przypomnieć, że problem znalezienia kliki maksymalnej jest problemem NP-zupełnym. W ramach zrealizowanych wykorzystano funkcję *clique_maximum()* dostępną w środowisku SageMath do rozwiązania tego problemu. Wykorzystuje ona metodę podziału i ograniczeń [NO03]. Dodatkowo przeanalizowano losowy wybór zmiennej wejściowej x_i do zbioru W, ze względu na jego małą złożoność obliczeniową. Losowy charakter metody powoduje jednak, że każde jej wywołanie może prowadzić do różnych wyników, tzn. różnej liczności zbioru W.

5.2.2 Metoda dokładna

Zaproponowana metoda heurystyczna pozwala na efektywną redukcję wykorzystania pamięci. Nie gwarantuje jednak uzyskania wyniku optymalnego. W związku z tym należy zaproponować metodę dokładną znajdowania realizacji funkcji zgodnie ze schematem przedstawionym na rysunku 5.1. W tym celu wykorzystać można problem spełnialności modulo teorie (SMT, ang. *Satisfiability Modulo Theories*) [Maz20b].

Problem spełnialności (SAT, ang. *Satisfiability*) ze względu na bycie pierwszym znanym problemem NP-zupełnym, jest jednym z najważniejszych zagadnień w informatyce. Wiele innych problemów NP-zupełnych może być efektywnie rozwiązanych poprzez ich redukcję do problemu SAT. Jak już wspomniano wcześniej, metoda ta została wykorzystana m.in. do znajdowania dekompozycji liniowej funkcji generowania indeksów [SFI15]. Głównym celem w problemie SAT jest określenie, czy formuła boolowska jest spełnialna, tzn. czy zmiennym wejściowym do problemu można przypisać wartości logiczne *Prawda* albo *Fałsz* w taki sposób, aby cała formuła zwracała wartość *Prawda*. Jeżeli nie uda się znaleźć takiego przypisania, to formuła nazywana jest niespełnialną.

Problem SMT jest uogólnieniem problemu SAT. Istotę różnicy pomiędzy problemami bardzo dobrze przedstawia następujący cytat: [Yur20]

> SAT/SMT solvers can be viewed as solvers of huge systems of equations. The difference is that SMT solvers takes systems in arbitrary format, while SAT solvers are limited to boolean equations in CNF form.

Co w wolnym tłumaczeniu oznacza:

> Narzędzia rozwiązujące problem SAT/SMT można traktować jako narzędzia rozwiązujące wielkie układy równań. Różnica polega na tym, że dla SMT narzędzia przyjmują dowolne równania, a dla SAT ograniczone są one do równań boolowskich w koniunkcyjnej postaci normalnej.

Formalnie, problem SMT polega na określeniu czy dana formuła jest spełnialna z uwzględnieniem teorii wyrażonych za pomocą rachunku predykatów pierwszego rzędu. W informatyce często stosuje się m.in. teorie dotyczące liczb całkowitych oraz liczb rzeczywistych. Działanie narzędzia rozwiązującego problem SMT sprowadza się często do konwersji problemu do SAT, przy wykorzystaniu metod rozumowania dostosowanych do zastosowanych teorii [Bar+09].

Dostępnych jest wiele narzędzi rozwiązujących problem SMT. W ramach eksperymentów opisanych w niniejszej rozprawie wykorzystano narzędzie Z3 [MB08], udostępnione

za darmo przez Microsoft Research. Narzędzie to osiąga bardzo dobre wyniki w corocznym SMT-Competition [Web+19], choć dla niektórych problemów cechuje się gorszą efektywnością [Nie+18]. Plikiem wejściowym dla Z3 jest sekwencja komend zgodnych ze standardem SMT-LIB 2.0 [BST12]. Twórcy udostępnili również szereg API pozwalających na obsługę narzędzia z wykorzystaniem języka wysokiego poziomu, np. Pythona czy .NET.

Przykład 5.3. Przykładowy plik wejściowy w formacie SMT-LIB przedstawiono jako listing 5.1. Opisuje on model zadania, w którym należy określić, czy istnieje $a, b \in \mathbb{Z}$ takie, że $a < 4, b < 2$ oraz $a^2 + b^2 > 10$.

Dwie pierwsze linie służą do zdefiniowania cech istotnych. Następnie określone są związki między wybranymi cechami. Linie 4. i 5. wprowadzają ograniczenie na wartość liczb a i b. Linia 6. wprowadza ograniczenie $a^2+b^2 > 10$. Komenda *check-sat* powoduje sprawdzenie spełnialności, natomiast komenda *get-value* zwraca uzyskaną wartość a i b, jeżeli problem był spełnialny. Dla tego zadania narzędzie Z3 zwraca przykładowy wynik $a = 3$ i $b = -2$.

```
1 (declare-const a Int)
2 (declare-const b Int)
3
4 (assert (< a 4))
5 (assert (< b 2))
6 (assert (> (+ (^ a 2) (^ b 2)) 10))
7
8 (check-sat)
9 (get-value (a b))
```

Listing 5.1: Przykładowy plik w formacie SMT-LIB

△

Zagadnienie znalezienia dekompozycji funkcjonalnej funkcji generowania indeksów może zostać zamodelowane jako problem SMT [Maz20b]. Korzystając z pojęcia r-przydatności, znana jest postać zbiorów U i V. Co więcej, wiadomo że funkcja G ma $r - |U|$ wyjść. Konieczne jest określenie, które zmienne ze zbioru U powinny zostać wykorzystane do utworzenia zbioru W w taki sposób, aby jego liczność była jak najmniejsza. Można zaproponować dwie strategie postępowania:

1. sformułować zadanie optymalizacyjne, w którym funkcją celu jest minimalizacja liczności zbioru W,
2. znaleźć rozwiązanie z wykorzystaniem przedstawionej wcześniej metody heurystycznej, a następnie zweryfikować jego minimalność z wykorzystaniem narzędzia Z3.

W ramach niniejszej rozprawy przeprowadzono badania z wykorzystaniem strategii numer dwa.

W celu zamodelowania problemu definiowane jest N zmiennych (*const*) typu Bool oznaczonych poprzez x_i. Wartość *True* uzyskana dla i-tej zmiennej oznaczać będzie, że zmienna ta powinna być wykorzystana w zbiorze W. Ze względu na to, że $W \subset U$, wszystkim zmiennym należącym do zbioru V przypisywana jest w modelu wartość *False*, tzn. $\forall x_i \in V, i = [1, N] : x_i = False$. Wykorzystywana jest do tego komenda *assert*. Liczność zbioru W może być zatem zdefiniowana jako liczba zmiennych, którym przypisano wartość *True* w uzyskanym rozwiązaniu.

Niech $\overrightarrow{v}_i$ - wektor wejściowy do funkcji G, $\overrightarrow{w}_i$ - wektor wejściowy do funkcji H oraz $i = [1, K]$ - numer wektora w tablicy prawdy funkcji. Wektory $\overrightarrow{v}_i$ tworzone są poprzez konkatenację zmiennych ze zbiorów V oraz W. Wektory $\overrightarrow{w}_i$ tworzone są z kolei poprzez konkatenację zmiennych ze zbioru U oraz wartości $G(\overrightarrow{v}_i)$, czyli wartości funkcji G dla wektora $\overrightarrow{v}_i$, która zakodowana jest na $r - |U|$ bitach. Łatwo zauważyć, że $G(\overrightarrow{v}_i) = [0, 2^{r-|U|} - 1]$. W modelu wektory te traktowane są jako binarna reprezentacja liczby całkowitej.

Modelując związki pomiędzy cechami należy zauważyć, że w schemacie dekompozycji przedstawionym na rysunku 5.1 funkcja H musi być funkcją generowania indeksów. W związku z tym zachodzić musi następująca zależność:

$$\forall_{i,j=[1,K],i\neq j} : \overrightarrow{w}_i \neq \overrightarrow{w}_j. \tag{5.4}$$

W SMT-LIB zależność ta może być zamodelowana z wykorzystaniem pojedynczej komendy *assert distinct*. W przypadku funkcji G konieczne jest z kolei zagwarantowanie, że jeżeli wartość dwóch wektorów $\overrightarrow{v}_i$, $\overrightarrow{v}_j, i \neq j$ jest taka sama, to wartość przypisywana im przez funkcję G też będzie taka sama:

$$\forall_{i,j=[1,K],i\neq j} : \overrightarrow{v}_i = \overrightarrow{v}_j \Rightarrow P_G(\overrightarrow{v}_i) = P_G(\overrightarrow{v}_j), \tag{5.5}$$

Zależność ta wprowadza $\frac{K(K-1)}{2}$ związków do modelu. Dodatkowe $4K$ zależności wynika z potrzeby ograniczenia wartości $G(\overrightarrow{v}_i)$, a także konieczności wyznaczenia postaci $\overrightarrow{v}_i$ oraz $\overrightarrow{w}_i$. Ze względu na to, że łączna liczba zależności w modelu jest $O(K^2)$, proponowana metoda będzie najbardziej efektywna dla funkcji o niewielkiej liczbie wektorów K w tablicy prawdy. Należy jednak przypomnieć, że dla funkcji generowania indeksów $K \ll 2^N$.

Niech ξ - liczność zbioru W uzyskana z wykorzystaniem algorytmu heurystycznego. Jeżeli spełnialny jest model, w którym liczność zbioru W jest mniejsza niż ξ, to uzyskane

rozwiązanie heurystyczne nie było optymalne. Możliwe jest wtedy iteracyjne sprawdzanie spełnialności modelu dla malejącej wartości ξ. W przypadku stwierdzenia, że model jest niespełnialny dla wartości ξ, to rozwiązaniem optymalnym jest liczność $\xi + 1$. Dodatkowo w wyniku działania narzędzia Z3 uzyskuje się przypisanie wartości funkcji G dla poszczególnych wektorów wejściowych.

5.2.3 Uzyskane wyniki

W rozdziale 3. przedstawiono wyniki pokazujące, że dekompozycja liniowa może być wykorzystana do efektywnej minimalizacji funkcji generowania indeksów. Jednakże, część funkcji nie posiada optymalnej reprezentacji uzyskanej z wykorzystaniem algorytmów dekompozycji liniowej. W związku z tym, analizując efektywność dekompozycji funkcjonalnej, przyjęto następującą metodykę:

1. analizowana funkcja generowana indeksów jest minimalizowana z wykorzystaniem algorytmu dekompozycji liniowej, który przedstawiony został w rozdziale 3,
2. jeżeli nie znaleziono optymalnej dekompozycji liniowej, tzn. $P \neq GD$, funkcja jest minimalizowana z wykorzystaniem algorytmu dekompozycji funkcjonalnej.

O ile nie określono inaczej, eksperymenty prowadzone były z wykorzystaniem metody MinR.

Przeprowadzone badania można podzielić na trzy części. W pierwszej z nich dokonano analizy zaproponowanej metody heurystycznego znajdowania dekompozycji [MŁ19b; Maz20a]. Należy zauważyć, że w sumie należało przeanalizować sześć podejść. Wynika to z możliwości zastosowania różnych algorytmów kolorowania grafu (K1, K2), jak i różnych metod wyboru zmiennej do zbioru W (W1, W2, W3). W drugiej części dokonano porównania efektywności metody heurystycznej oraz metody dokładnej [Maz20b]. W ostatniej części przedstawiono wyniki uzyskane dla nieredukowalnych funkcji generowania indeksów [Sas20].

Do oceny podejść zaproponowanych w ramach metody heurystycznej zastosowano funkcje generowania indeksów o liczbie zmiennych wejściowych $N = 32$ oraz liczbie wektorów z następującego zbioru $K \in \{20, 30, 40, 50\}$. Dla każdej wartości K wygenerowano tysiąc funkcji. Zgodnie z przedstawioną metodyką, funkcje te zostały zminimalizowane z wykorzystaniem algorytmu dekompozycji liniowej. Uzyskane wyniki przedstawiono w tabeli 5.3. Przez P_{avg} oznaczono średnią liczbę zmiennych po dekompozycji, natomiast przez P_{opt} - liczbę funkcji, dla których $GD = P$. Dla funkcji, które nie posiadały optymalnej

Tabela 5.3: Wyniki uzyskane po zastosowaniu algorytmu dekompozycji liniowej

K	GD	P_{avg}	P_{opt}
20	5	5,21	787
30	5	6,16	0
40	6	6,98	20
50	6	7,48	0

dekompozycji liniowej, poszukiwano dekompozycji funkcjonalnej. W tym celu przygotowano eksperymentalne oprogramowanie z wykorzystaniem środowiska SageMath [Sage]. Uzyskane wyniki przedstawiono w tabeli 5.4. Przyjęto następujące oznaczenia:

- $|W| = 0, 1, \ldots, 4$ - liczba funkcji o określonej liczności zbioru W,
- $|W|_{avg}$ - średnia liczność zbioru W,
- $|W|_{rel}$ - stosunek średniej liczności zbioru W do najmniejszej średniej liczności spośród wszystkich podejść,
- T_{rel} - stosunek średniego czasu obliczeń do najmniejszego średniego czasu obliczeń spośród wszystkich podejść.

Pierwszym wnioskiem wynikającym z przedstawionych wyników jest to, że wybór algorytmu kolorowania grafu ma o wiele większy wpływ na uzyskiwaną jakość wyniku od metody wyboru zmiennej dodawanej do zbioru W. Dla heurystycznego algorytmu kolorowania grafu uzyskano wyniki średnio o 27% gorsze niż w przypadku algorytmu dokładnego. Dla $K = 50$ uzyskane wyniki były aż o ok. 70% większe. Należy również zwrócić uwagę, że liczba funkcji dla których $|W| = 0$ różni się w zależności od zastosowanego algorytmu. Dla niektórych funkcji nie udało się potwierdzić istnienia dekompozycji rozłącznej z wykorzystaniem algorytmu heurystycznego.

Wybór innej metody dodawania zmiennej do zbioru W skutkuje różnicą w uzyskiwanej liczności zbioru na poziomie zaledwie kilku procent. Biorąc pod uwagę średnią liczność zbioru W dla wszystkich wartości K, metoda wykorzystująca problem znajdowania maksymalnej kliki w grafie gwarantuje uzyskanie najmniejszej wartości. Jednakże, wybór zmiennej rozdzielającej blok reprezentowany przez wierzchołek o największym stopniu skutkuje licznością zaledwie o 0.7% większą.

Zastosowanie algorytmu heurystycznego kolorowania grafu gwarantuje z kolei uzyskanie dużej wydajności czasowej. Średni czas obliczeń dla poszczególnych wartości K przedstawiono na rysunku 5.3. Czas obliczeń z wykorzystaniem algorytmu Welsha-Powella nie wzrasta gwałtownie dla rosnących wartości K. Zgodnie z przypuszczeniami, dla algorytmu dokładnego kolorowania czas obliczeń jest silnie zależny od wartości K. Dla $K = 50$ czas obliczeń z wykorzystaniem algorytmu dokładnego jest ponad 20 razy większy niż w przypadku algorytmu heurystycznego.

Tabela 5.4: Wyniki uzyskane dla różnych wartości K [Maz20a]

(a) Wyniki dla $K = 20$.

	W1,K1	W1,K2	W2,K1	W2,K2	W3,K1	W3,K2
$\|W\| = 0$	16	15	16	15	16	15
$\|W\| = 1$	103	67	103	78	99	83
$\|W\| = 2$	80	102	80	96	75	77
$\|W\| = 3$	14	29	14	23	23	38
$\|W\| = 4$	0	0	0	0	0	0
$\|W\|_{avg}$	1,43	1,68	1,43	1,60	1,49	1,65
$\|W\|_{rel}$	1,00	1,17	1,00	1,12	1,04	1,15
T_{rel}	2,39	1,67	2,49	1,59	1,64	1,00

(b) Wyniki dla $K = 30$.

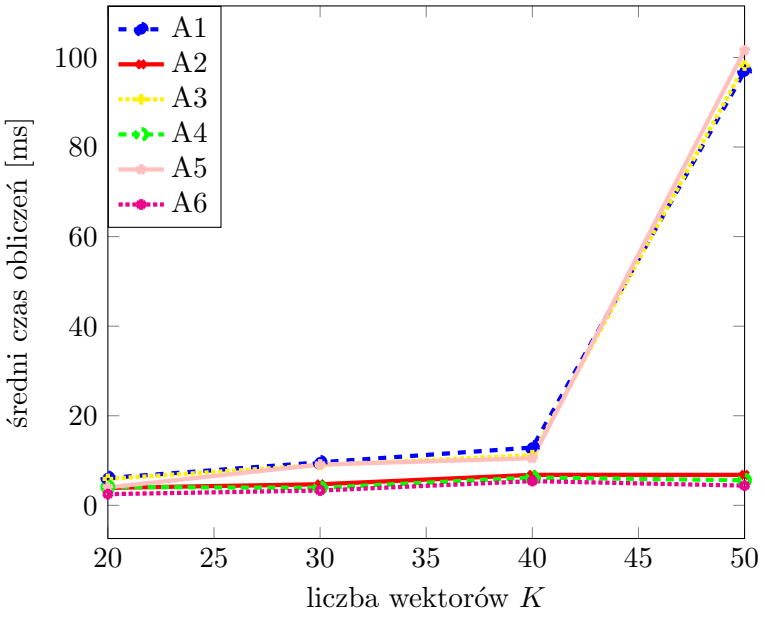
	W1,K1	W1,K2	W2,K1	W2,K2	W3,K1	W3,K2
$\|W\| = 0$	590	406	590	406	590	406
$\|W\| = 1$	247	373	259	379	261	387
$\|W\| = 2$	145	201	133	196	131	189
$\|W\| = 3$	3	3	3	3	3	3
$\|W\| = 4$	0	0	0	0	0	0
$\|W\|_{avg}$	0,55	0,80	0,54	0,79	0,54	0,79
$\|W\|_{rel}$	1,03	1,48	1,00	1,47	1,00	1,45
T_{rel}	2,77	1,19	2,96	1,45	2,78	1,00

(c) Wyniki dla $K = 40$.

	W1,K1	W1,K2	W2,K1	W2,K2	W3,K1	W3,K2
$\|W\| = 0$	81	51	81	52	81	51
$\|W\| = 1$	216	165	210	151	216	156
$\|W\| = 2$	297	305	326	338	278	325
$\|W\| = 3$	302	337	294	319	313	301
$\|W\| = 4$	81	124	69	109	92	147
$\|W\|_{avg}$	2,09	2,34	2,06	2,29	2,12	2,34
$\|W\|_{rel}$	1,01	1,13	1,00	1,11	1,03	1,14
T_{rel}	2,06	1,15	2,37	1,25	1,94	1,00

(d) Wyniki dla $K = 50$.

	W1,K1	W1,K2	W2,K1	W2,K2	W3,K1	W3,K2
$\|W\| = 0$	540	388	540	388	540	388
$\|W\| = 1$	275	208	270	201	263	207
$\|W\| = 2$	124	246	141	260	149	275
$\|W\| = 3$	48	142	40	138	42	125
$\|W\| = 4$	1	1	4	4	2	1
$\|W\|_{avg}$	0,68	1,15	0,69	1,16	0,70	1,14
$\|W\|_{rel}$	1,00	1,69	1,02	1,71	1,03	1,68
T_{rel}	22,10	1,25	21,82	1,53	22,87	1,00



Rysunek 5.3: Średni czas obliczeń dla różnych wartości K

W ramach przeprowadzonych eksperymentów poszukiwano dekompozycji funkcjonalnej gwarantującej, że $|W| < |U|$. W związku z tym, nie dla wszystkich funkcji znaleziono realizację funkcji zgodnie z omówionym wcześniej schematem. Liczbę funkcji, dla których znaleziono dekompozycję funkcjonalną przedstawiono w tabeli 5.5. Należy zauważyć, że dla $K = 20$ większość funkcji miała optymalną dekompozycję liniową (por. tabela 5.3). Powoduje to, że liczba funkcji, dla których znaleziono dekompozycję funkcjonalną, jest relatywnie niewielka w porównaniu do pozostałych analizowanych wartości K.

Do oceny zaproponowanych podejść wykorzystano również kodery *M z 20*. W tabeli 5.6 przedstawiono uzyskaną liczność zbioru W oraz czas obliczeń. Poprzez „-" oznaczono sytuację, kiedy nie uzyskano wyniku w czasie 60 minut. Miało to miejsce przy zastosowaniu dokładnego algorytmu kolorowania grafu dla $M = 4$. Ze względu na to, że losowy wybór zmiennej dodawanej do zbioru W może prowadzić do różnych rozwiązań, przedstawiono największą i najmniejszą liczność zbioru W uzyskaną w dziesięciu wywołaniach metody. Przedstawiony czas dotyczy sytuacji, gdy znaleziono rozwiązanie najmniejsze.

Tabela 5.5: Liczba funkcji zminimalizowana z wykorzystaniem dekompozycji funkcjonalnej [Maz20a]

K	W1,K1	W1,K2	W2,K1	W2,K2	W3,K1	W3,K2
20	213	213	213	212	213	213
30	985	983	985	984	985	985
40	977	973	980	969	980	980
50	988	985	995	991	996	996

Tabela 5.6: Wyniki dla koderów *M z 20* [Maz20a]

Algorytmy	$M = 2, K = 190$		$M = 4, K = 4845$	
	czas [s]	$\|W\|$	czas [s]	$\|W\|$
W1,K1	0,25	2	-	-
W1,K2	0,04	2	17,92	3
W2,K1	0,17	1	-	-
W2,K2	0,05	2	19,33	3
W3,K1	0,17	1-3	-	-
W3,K2	0,02	2-3	7,98	2-4

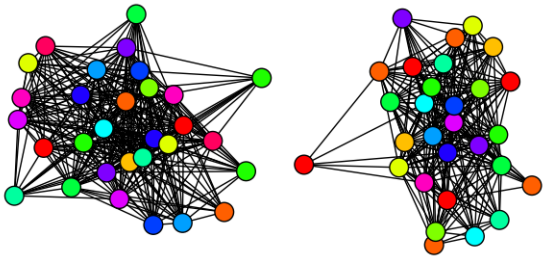
Dla $M = 2$ uzyskano $|W| = 1$ przy wykorzystaniu podejść W2 oraz W3, przy jednoczesnym zastosowaniu dokładnego algorytmu kolorowania grafu. Dla $M = 4$ najlepszy wynik również uzyskano poprzez losowy wybór zmiennej do zbioru W. Należy podkreślić, że jakość otrzymywanych rozwiązań za pomocą tej metody okazała się różna. Pozwoliła ona uzyskać zarówno najlepsze, jak i najgorsze rozwiązanie spośród wszystkich metod, w zależności od wylosowanych zmiennych w poszczególnych iteracjach. Przedstawione wyniki dowodzą także, że zastosowanie algorytmu dokładnego kolorowania grafu nie gwarantuje uzyskania optymalnego rozwiązania.

Jak już wspomniano, dokładny algorytm kolorowania grafu doprowadził do przekroczenia czasu eksperymentu dla $M = 4$. Dowodzi to, że do minimalizacji funkcji o wielu wektorach rejestrowanych celowe wydaje się stosowanie algorytmu heurystycznego kolorowania grafu. Dla $M = 2$ czas działania algorytmu dokładnego był kilkukrotnie większy, niż czas działania algorytmu Welsha-Powella. Należy również podkreślić, że czas działania metody wykorzystującej losowe dodawanie zmiennych do zbioru W był najmniejszy.

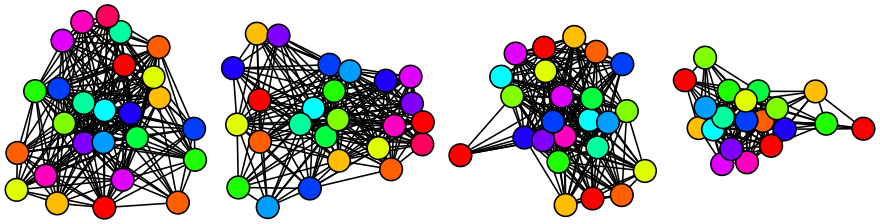
Co ciekawe, dla $M = 4$ oraz metody First-Fit, możliwe jest znalezienie dekompozycji rozłącznej. Otrzymuje się wtedy $|U| = 3$ oraz $|V| = 13$. Całkowita zajętość pamięci jest jednak większa, niż w przypadku wyników przedstawionych w tabeli 5.6, uzyskanych z wykorzystaniem metody MinR.

Na rysunku 5.4 przedstawiono pokolorowany graf Γ uzyskany dla zbioru W o różnych licznościach dla funkcji *2 z 20*. Jak widać, większa liczność zbioru W prowadzi do uzyskania grafu o większej liczby wierzchołków. Do pokolorowania obu grafów wykorzystano 16 kolorów. Dowodzi to, że istnieje dekompozycja nierozłączna, gdzie funkcja G ma cztery wyjścia. Biorąc pod uwagę wartości przedstawione dalej w tabeli 5.8a, funkcję tę można zrealizować w sposób przedstawiony na rysunku 5.5.

W celu porównania efektywności metody heurystycznej oraz dokładnej znajdowania dekompozycji funkcjonalnej funkcji generowania indeksów, wygenerowano po tysiąc funkcji dla różnych wartości N oraz K. W ramach badań w pierwszej kolejności dokonano minimalizacji funkcji z wykorzystaniem metody heurystycznej. Na podstawie uzyskanych

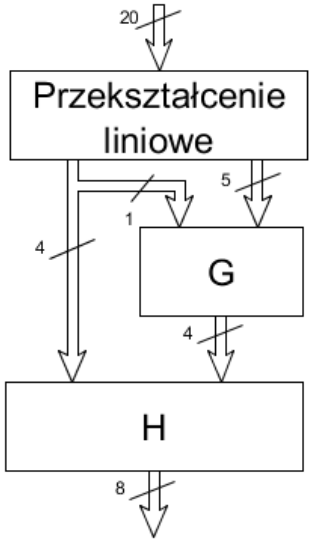


(A) Graf Γ dla $W = 1$



(B) Graf Γ dla $W = 2$ [MŁ19b]

RYSUNEK 5.4: Uzyskany graf Γ w zależności od postaci zbioru W



RYSUNEK 5.5: Realizacja funkcji *2 z 20*

wyników, dla funkcji spełniających $W \neq 0$ wywołano metodę dokładną. Wyniki przedstawiono w tabeli 5.7. W przypadku metody heurystycznej zastosowano dokładny algorytm kolorowania grafu oraz wybór zmiennej rozdzielającej blok reprezentowany przez wierzchołek o największym stopniu. Podobnie jak wcześniej, wygenerowane funkcje były najpierw dekomponowane za pomocą algorytmu dekompozycji liniowej. Funkcje, dla których nie znaleziono optymalnej dekompozycji liniowej, były dalej minimalizowane z wykorzystaniem algorytmu dekompozycji funkcjonalnej. Przez l oznaczono liczbę takich funkcji dla poszczególnych wartości N i K.

Tabela 5.7: Porównanie efektywności metody heurystycznej i dokładnej [Maz20b]

(a) Wyniki uzyskane metodą heurystyczną (W1,K1)

N	K	l	$\|W\\|_{avg}$	$\|W\| = 0$	$\|W\| = 1$	$\|W\| = 2$	$\|W\| = 3$	$\|W\| = 4$
20	20	370	1,43	42	162	132	34	0
20	40	794	1,37	283	161	172	129	49
40	20	162	1,48	10	77	62	13	0
40	40	952	2,22	38	189	336	307	82
60	20	55	1,49	5	23	22	5	0
60	40	960	2,36	11	151	368	337	93

(b) Wyniki uzyskane metodą dokładną

N	K	l	$\|W\\|_{avg}$	$\|W\| = 0$	$\|W\| = 1$	$\|W\| = 2$	$\|W\| = 3$	$\|W\| = 4$
20	20	370	0,99	42	288	40	0	0
20	40	794	0,93	283	298	198	15	0
40	20	162	0,99	10	144	8	0	0
40	40	952	1,52	38	418	457	39	0
60	20	55	0,93	5	49	1	0	0
60	40	960	1,58	11	406	516	27	0

Przedstawione wyniki potwierdzają efektywność zaproponowanej metody dokładnej. Średnia liczność zbioru W w przypadku zastosowania jedynie metody heurystycznej jest wyższa o ok. 44-60% niż w przypadku zastosowania metody dokładnej. Należy także zwrócić uwagę, że metoda dokładna pozwoliła znaleźć dekompozycje z licznością zbioru $|W| < 4$ dla wszystkich analizowanych funkcji.

Dla ponad 60% spośród analizowanych funkcji nie znaleziono dekompozycji liniowej takiej, że $P = GD$. Potwierdza to zasadność zastosowania dekompozycji funkcjonalnej do dalszej minimalizacji tych funkcji. Co więcej, większość z tych funkcji ma nierozłączną dekompozycję funkcjonalną. Uzasadnia to potrzebę prowadzenia prac nad opracowaniem efektywnego algorytmu jej znajdowania.

Do oceny efektywności zaproponowanych w ramach niniejszej rozprawy metod wykorzystano również funkcje *M z N*. Pod uwagę wzięto te funkcje, dla których nie istnieje optymalna dekompozycja liniowa (por. rozdział 3). Uzyskane wyniki przedstawiono w tabeli 5.8. Zastosowano następujące oznaczenia:

- W_G - liczność zbioru W wyznaczona z wykorzystaniem metody heurystycznej,
- W_{SMT} - liczność zbioru W wyznaczona z wykorzystaniem metody dokładnej,
- MEM_{lin} - rozmiar pamięci po zastosowaniu algorytmu dekompozycji liniowej, zgodnie ze wzorem 3.1,

- MR - współczynnik minimalizacji, wyznaczony z następującej zależności:

$$MR = (1 - \frac{MEM_{func}}{MEM_{lin}}) \cdot 100\% \tag{5.6}$$

Jako W_G przedstawiono najlepszy wynik uzyskany z wykorzystaniem metody heurystycznej, na podstawie wyników przedstawionych w tabeli 5.6. Dla trzech spośród analizowanych funkcji znaleziono dekompozycję rozłączną. Z kolei dla funkcji *M z 20* znaleziono dekompozycję nierozłączną. Przedstawiony wcześniej wynik uzyskany dla funkcji *2 z 20* z wykorzystaniem metody heurystycznej okazał się być optymalnym. Zostało to potwierdzone z wykorzystaniem metody dokładnej. Dla funkcji *4 z 20* metoda dokładna nie zwróciła jednak wyniku w założonym czasie eksperymentu.

Tabela 5.8: Otrzymane wyniki dla funkcji *M z N* [Maz20b; Maz20a]

(a) Otrzymane wartości $|U|, |V|, |W|$

Funkcja	N	K	GD	P	r	$\|U\|$	$\|V\|$	$\|W\|_G$	$\|W\|_{SMT}$
2 z 16	16	120	7	8	7	2	6	0	0
4 z 16	16	1820	11	13	11	1	12	0	0
2 z 20	20	190	8	9	8	4	5	1	1
4 z 20	20	4845	13	15	13	6	9	2	-
2 z 32	32	496	9	11	9	1	10	0	0

(b) Wykorzystanie pamięci

Funkcja	N	K	MEM_{fun}	MEM_{lin}	MR
2 z 16	16	120	1216	1792	32,1%
4 z 16	16	1820	63488	90112	29,5%
2 z 20	20	190	2304	4096	43,7%
4 z 20	20	4845	120832	425984	71,6%
2 z 32	32	496	12800	18432	30,6%

Zastosowanie dekompozycji funkcjonalnej pozwoliło na uzyskanie znacznej redukcji wykorzystywanej pamięci. Na podstawie wyników przedstawionych w tabeli 5.8b wynosi ono od ok. 30% do nawet ok. 70%. Warto zauważyć, że największą redukcję pamięci uzyskano dla funkcji posiadających nierozłączną dekompozycję funkcjonalną.

W celu oceny efektywności dekompozycji funkcjonalnej przeanalizowano możliwość minimalizacji nieredukowalnych funkcji generowania indeksów [Sas20]. Funkcje te definiowane są następująco: funkcja generowania indeksów z N zmiennymi wejściowymi jest redukowalna, jeżeli może być przedstawiona z wykorzystaniem mniej niż N zmiennych za pomocą dekompozycji liniowej. W przeciwnym wypadku, funkcja F jest nieredukowalna. Możliwość redukcji funkcji można określić poprzez sprawdzenie czy rodzina zbiorów niezgodności zawiera wszystkie możliwe wektory niezerowe.

W literaturze przedstawiono postaci funkcji nieredukowalnych dla $4 \leqslant N \leqslant 8$ o zbiorze wektorów rejestrowanych z minimalną licznością. Uzyskane dla nich wyniki z wykorzystaniem algorytmu dekompozycji funkcjonalnej przedstawiono w tabeli 5.9. Dla wszystkich funkcji znaleziono dekompozycję rozłączną, tzn. $W = \emptyset$. Jak łatwo zauważyć, zastosowanie dekompozycji funkcjonalnej gwarantuje istotną minimalizację wykorzystywanej pamięci dla wszystkich funkcji. Potwierdza to tezę, że dekompozycja funkcjonalna pozwala na efektywną redukcję funkcji, dla których nie istnieje dekompozycja liniowa spełniająca zależność $P = GD$.

Tabela 5.9: Wyniki dla funkcji nieredukowalnych

N	K	Q	MEM_{lin}	$\|U\|$	$\|V\|$	r	MEM_{func}	MR
4	6	3	48	1	3	3	40	16,7%
5	10	4	128	2	3	4	80	37,5%
6	14	4	256	2	4	5	176	31,3%
7	20	5	640	2	5	5	256	60,0%
8	27	5	1280	2	6	5	352	72,5%

5.2.4 Wnioski z przeprowadzonych badań

W niniejszym rozdziale przedstawiono wyniki prac nad metodami znajdowania dekompozycji funkcjonalnej funkcji generowania indeksów. W tym celu zaproponowano dwie metody: heurystyczną, wykorzystującą elementy teorii grafów oraz dokładną, wykorzystującą uogólnienie problemu SAT. Obie metody pozwalają znaleźć dekompozycję nierozłączną.

Przedstawione wyniki potwierdzają efektywność zaproponowanych metod. Metoda heurystyczna pozwala na znalezienie dekompozycji w relatywnie krótkim czasie. Jednakże, jakość uzyskiwanego rozwiązania jest gorsza niż w przypadku metody dokładnej. Należy również zauważyć, że zastosowanie dokładnego algorytmu kolorowania grafu nie gwarantuje uzyskania wyniku optymalnego. Z drugiej strony, metoda dokładna cechuje się słabą skalowalnością. Dla funkcji o dużej liczbie wektorów rejestrowanych K zastosowanie tej metody nie doprowadziło do uzyskania wyniku w określonym czasie. Należy przy tym pamiętać, że funkcje generowania indeksów cechują się tym, że są rzadkie, tzn. $K \ll 2^N$.

Prace innych autorów [SMI16; SMI17] skupiają się na dekompozycji rozłącznej. Przedstawione wyniki potwierdzają jednak zasadność poszukiwania dekompozycji nierozłącznej. Spośród przeanalizowanych funkcji, wygenerowanych w sposób losowy, większość posiadała dekompozycję nierozłączną, która pozwoliła na znaczną redukcję wykorzystywanej pamięci. Zaproponowane metody pozwoliły także na dalszą minimalizację funkcji M *z* N, które nie posiadają optymalnej dekompozycji liniowej. Co istotne, jakość uzyskanego rozwiązania z wykorzystaniem metody heurystycznej była bardzo dobra.

Rozdział 6

Podsumowanie pracy i dalsze kierunki badań

Minimalizacja funkcji boolowskich szczególnej postaci, zwanych funkcjami generowania indeksów, przyciągnęła w ostatnich latach uwagę naukowców z całego świata. W ramach niniejszej rozprawy przeanalizowano następujące metody, które pozwalają na redukcję wykorzystania pamięci dla realizacji funkcji generowania indeksów:

- dekompozycja liniowa z wykorzystaniem zbiorów niezgodności,
- realizacja sprzętowa z wykorzystaniem dedykowanej architektury,
- dekompozycja funkcjonalna z wykorzystaniem algorytmów teorii grafów oraz problemu SMT.

W rozdziale 3. przedstawiono autorskie modyfikacje algorytmu dekompozycji liniowej z wykorzystaniem zbioru niezgodności. Zaproponowano iteracyjny algorytm oraz dwie metody wyboru funkcji rozdzielającej: First-Fit oraz MinR. Przedstawione wyniki dowodzą efektywności zaproponowanego rozwiązania w minimalizacji liczby zmiennych do realizacji funkcji M *z* N oraz losowo wygenerowanych funkcji generowania indeksów. Wadą algorytmu jest konieczność wyznaczenia macierzy rozróżnialności, co wiążę się z silną zależnością złożoności pamięciowej od liczności zbioru wektorów rejestrowanych.

Dalsze badania, przedstawione w rozdziale 4., dotyczyły wykorzystania struktur probabilistycznych do realizacji generatorów indeksów. Wykazano, że wykorzystanie pamięci w proponowanej architekturze nie zależy od liczby zmiennych po transformacji liniowej. Dzięki temu, gdy liczba ta zbliżona jest do optymalnej, to uzyskuje się mniejsze wykorzystanie pamięci niż w przypadku architektury przedstawionej w literaturze. Wadą

zaproponowanej architektury jest konieczność realizacji dodatkowych operacji oraz wprowadzanie niewielkiego prawdopodobieństwa wyniku fałszywie pozytywnego, co ograniczania możliwości jej zastosowania.

Ze względu na to, że niektóre funkcje nie posiadają optymalnej dekompozycji liniowej, w rozdziale 5. przeanalizowano możliwość zastosowania dekompozycji funkcjonalnej do minimalizacji takich funkcji. Zaproponowano dwie metody poszukiwania nierozłącznej dekompozycji funkcjonalnej funkcji generowania indeksów: heurystyczną oraz dokładną. Przedstawione wyniki potwierdzają efektywność zaproponowanych metod. Metoda heurystyczna pozwala znaleźć dekompozycję o relatywnie niewielkiej liczbie zmiennych w krótkim czasie. Metoda dokładna gwarantuje z kolei uzyskanie wyniku optymalnego, kosztem dużej złożoności obliczeniowej.

Biorąc pod uwagę powyższe można stwierdzić, że wszystkie tezy rozprawy przedstawione w rozdziale 1.2. zostały udowodnione.

Przedstawione w ramach niniejszej rozprawy badania oraz ich wyniki mają duży potencjał dalszego rozwoju. W zakresie zagadnień przedstawionych w rozdziale 3 należy zauważyć, że ograniczenie problemu dekompozycji liniowej jedynie do funkcji takich, że $|D^N| = K$ wydaje się niepotrzebne [ŁPZ16; MŁ19a]. Dlatego w literaturze zaproponowano [Sas18] generalizację pojęcia funkcji generowania indeksów. Przedstawiony algorytm dekompozycji liniowej może być w prosty sposób zaadaptowany do syntezy funkcji tej postaci, o binarnych wektorach rejestrowanych [MŁ19a]. Co więcej, najnowsze prace z zakresu funkcji generowania indeksów wskazują, że algorytmy dekompozycji liniowej mogą być z powodzeniem stosowane do minimalizacji funkcji symetrycznych [NSB20; Maz20c] oraz klasyfikujących [Sas20b]. Analiza możliwości zastosowania w tym celu zaproponowanego algorytmu wydaje się celowa.

Innym interesującym kierunkiem jest analiza możliwości sprzętowej implementacji algorytmu dekompozycji liniowej. Zgodnie z wiedzą autora niniejszej rozprawy, realizowane do tej pory badania skupiają się na poszukiwaniu dekompozycji z wykorzystaniem implementacji programowych. Implementacja z wykorzystaniem układów programowalnych FPGA może zapewnić większą efektywną czasową poszukiwania dekompozycji liniowej. Do realizacji operacji poszukiwania funkcji rozdzielającej w poszczególnych iteracjach wykorzystać można algorytm Bincombgen [Kok97], który może być efektywnie zrealizowany sprzętowo [Maz17].

Rachunek podziałów wykorzystany może być również do poszukiwania dekompozycji nieliniowej funkcji generowania indeksów [ŁBJ16], tzn. wykorzystującej bramki AND

zamiast bramek XOR. W literaturze zagadnienie to nie zostało jednak dokładnie przeanalizowane, o czym świadczy niewielka liczba prac naukowych [ASA16]. W ramach realizowanych badań wykazano potencjalną użyteczność tego typu algorytmów [MŁ19a]. Badania te wymagają jednak dalszej dogłębnej analizy.

W zakresie zagadnień przedstawionych w rozdziale 4 można wyróżnić kilka potencjalnych kierunków dalszych badań. Należy podkreślić, że w ostatnich latach można zaobserwować gwałtowny wzrost liczby nowych struktur, bazujących na filtrze Blooma. Z tego powodu, pierwszym z kierunków jest analiza możliwości zastosowania innych struktur probabilistycznych niż te opisane w ramach niniejszej pracy. Przykładem takiej struktury jest np. filtr Xor [GL20]. Filtr ten jest w stanie zapewnić mniejsze wykorzystanie pamięci niż filtry Blooma oraz Cuckoo. Możliwa powinna być zatem dalsza redukcja wykorzystania pamięci w stosunku do struktury IGU. Drugim potencjalnym kierunkiem dalszych badań jest analiza możliwości zastosowania struktury [Kis+18], w której prawdopodobieństwo wyniku fałszywie pozytywnego jest zerowe dla określonej liczby wektorów wejściowych. Interesująca może okazać się też analiza możliwości zastosowana lekkich funkcji kryptograficznych do realizacji filtrów.

W rozdziale 5 wykazano, że problem SMT może być wykorzystany do znalezienia dokładnej nierozłącznej dekompozycji funkcjonalnej funkcji generowania indeksów. W niektórych przypadkach nie uzyskano jednak wyniku ze względu na przekroczenie założonego maksymalnego czasu obliczeń. W związku z tym celowa jest analiza możliwości zastosowania innego narzędzia niż Z3 do rozwiązania problemu SMT [Nie+18; Web+19]. Należy również zauważyć, że podobnie jak w przypadku algorytmu dekompozycji liniowej, przedstawione algorytmy dekompozycji funkcjonalnej można zaadaptować do wspomnianego uogólnienia funkcji generowania indeksów. Wynika to m.in. z uniwersalności wykorzystywanych twierdzeń, bazujących na rachunku podziałów [BŁ97; Raw+97].

Podsumowując, przeprowadzone w ramach niniejszej rozprawy badania nad każdym z przedstawionych zagadnień (dekompozycja liniowa, dedykowana realizacja sprzętowa, dekompozycja funkcjonalna) stanowią obiecujący punkt wyjścia do realizacji dalszych eksperymentów.

Bibliografia

[Ash57] R. Ashenhurst. „International Symposium on the Theory of Switching". W: *The decomposition of switching functions*. 1957, 74—116.

[AB16] M. Aslan i N. Baykan. „A Performance Comparison of Graph Coloring Algorithms". W: *International Conference on Advanced Technology and Sciences (ICAT)*. 2016, s. 266–273.

[ASA16] H. Astola, R. S. Stankovic i J. Astola. „Index Generation Functions Based on Linear and Polynomial Transformations". W: *2016 IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL)* (2016), s. 102–106.

[Ast+16] J. T. Astola i in. „An Algebraic Approach to Reducing the Number of Variables of Incompletely Defined Discrete Functions". W: *2016 IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL)*. 2016, s. 107–112.

[Ast+17] J. T. Astola i in. „Algebraic and Combinatorial Methods for Reducing the Number of Variables of Partially Defined Discrete Functions". W: *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*. 2017, s. 167–172.

[Aug18] P. Augustynowicz. „Dobór funkcji skrótu spełniających wymagania sprzętowej implementacji filtru Blooma". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 8-9 (2018), s. 605–608.

[AA19] P. Augustynowicz i A. Augustynowicz. „Sprzętowa implementacja filtru Blooma bazująca na pojedynczej funkcji". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 7 (2019).

[BST12] C. Barrett, A. Stump i C. Tinelli. *The SMT-LIB Standard - Version 2.0*. 2012.

[Bar+09] C. Barrett i in. „Satisfiability modulo theories". W: *Handbook of Satisfiability*. 1 wyd. Frontiers in Artificial Intelligence and Applications 1. IOS Press, 2009, s. 825–885.

[Blo70] B. H. Bloom. „Space/Time Trade-offs in Hash Coding with Allowable Errors". W: *Commun. ACM* 13.7 (1970), s. 422–426.

[Bor18] G. Borowik. „Optimization on the complementation procedure towards efficient implementation of the index generation function". W: *International Journal of Applied Mathematics and Computer Science* Vol. 28, no. 4 (2018), s. 803–815.

[BŁ14] G. Borowik i T. Łuba. „Fast Algorithm of Attribute Reduction Based on the Complementation of Boolean Function". W: *Advanced Methods and Applications in Computational Intelligence*. Heidelberg: Springer International Publishing, 2014, s. 25–41.

[BŁK20] G. Borowik, T. Łuba i R. Klempous. „Comparison of algorithms for dimensionality reduction and their application to index generation functions". W: *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*. 2020, s. 283–288.

[BŁP16] G. Borowik, T. Łuba i K. Poźniak. „New trends in logic synthesis for both digital designing and data processing". W: *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2016*. Red. R. S. Romaniuk. T. 10031. International Society for Optics & Photonics. SPIE, 2016, s. 1371 –1381.

[BŁT09] G. Borowik, T. Łuba i P. Tomaszewicz. „A notion of r-admissibility and its application in logic synthesis". W: *IFAC Proceedings Volumes* 42.21 (2009). 4th IFAC Workshop on Discrete-Event System Design, s. 172 –177.

[BŁT12] G. Borowik, T. Łuba i P. Tomaszewicz. „On Memory Capacity to Implement Logic Functions". W: *Computer Aided Systems Theory – EUROCAST 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 343–350.

[BŁ97] J. Brzozowski i T. Łuba. „Decomposition of boolean functions specified by cubes". W: *Journal of Multi-Valued Logic & Soft Computing* 9 (1997), s. 377–417.

[BS11] J. T. Butler i T. Sasao. „Fast Hardware Computation of x Mod z". W: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, s. 294–297.

[CW79] J. Carter i M. N. Wegman. „Universal classes of hash functions". W: *Journal of Computer and System Sciences* 18.2 (1979), s. 143 –154.

[CYP89] M. J. Ciesielski, S. Yang i M. A. Perkowski. „Multiple-valued Boolean minimization based on graph coloring". W: *Proceedings 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 1989, s. 262–265.

[Cor+09] T. Cormen i in. *Introduction to Algorithms, Third Edition*. London, England: The MIT Press, 2009.

[Cur62] H. Curtis. *A new approach to the design of switching circuits*. 1962.

[Fan+14] B. Fan i in. „Cuckoo Filter: Practically Better Than Bloom". W: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia, 2014, s. 75–88. ISBN: 978-1-4503-3279-8.

[FNV91] G. Fowler, L. Noll i P. Vo. *Fowler/Noll/Vo (FNV) Hash*. 1991.

[Gal+13] P. Galinier i in. „Recent Advances in Graph Vertex Coloring". W: *Handbook of Optimization: From Classical to Modern Approach*. Red. I. Zelinka, V. Snášel i A. Abraham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, s. 505–528.

[GJS74] M. Garey, D. Johnson i L. Stockmeyer. „Some simplified NP-complete problems". W: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*. 1974, s. 47–63.

[GL20] T. Graf i D. Lemire. „Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters". W: *J. Exp. Algorithmics* 25 (2020).

[HH02] F. Herrmann i A. Hertz. „Finding the Chromatic Number by Means of Critical Graphs". W: *J. Exp. Algorithmics* 7 (2002).

[HPC19] S. Hodžić, E. Pasalic i A. Chattopadhyay. „An iterative method for linear decomposition of index generating functions". W: *Cryptography and communications* 11.5 (2019), s. 1079–1102.

[Int19] Intel-FPGA. *MD5: MD5 Processor Core*. 2019. URL: `https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/cast-inc-/ip/md5--md5-processor-core.html#features`.

[Iwa16] T. Iwaszko. „Dekompozycja funkcji boolowskich z zastosowaniem do syntezy generatorów adresów". Promotor: prof. Tadeusz Łuba. Prac. inż. Wydział Elektroniki i Technik Informacyjnych, Politechnika Warszawska, 2016.

[JBK14] C. Jankowski, G. Borowik i K. Kowalski. „Dyskretyzacja danych numerycznych metodami przekształceń boolowskich". W: *Przegląd Telekomunikacyjny+ Wiadomości Telekomunikacyjne* 10 (2014), s. 1334–1342.

[Kan12] D. Kania. *Układy logiki programowalnej. Podstawy syntezy i sposoby odwzorowania technologicznego*. Warszawa: Wydawnictwo Naukowe PWN, 2012. ISBN: 9788301170523.

[Kis+18] S. Kiss i in. „Bloom Filter with a False Positive Free Zone". W: *IEEE Conference on Computer Communications (INFOCOM)* (2018), s. 1412–1420.

[Kok97] Z. Kokosiński. „On Parallel Generation of Combinations in Associative Processor Architectures". W: *Euro-PDS*. 1997.

[Kok19] Z. Kokosiński. „Digital Data Convertion Using Content Addressable Memory". W: *2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. T. 2. 2019, s. 680–684.

[Kow17] K. Kowalski. „Synteza funkcji generowania indeksów metodą redukcji i kompresji argumentów". Promotor: prof. Tadeusz Łuba. Prac. mag. Wydział Elektroniki i Technik Informacyjnych, Politechnika Warszawska, 2017.

[Lu+18] J. Lu i in. „Low Computational Cost Bloom Filters". W: *IEEE/ACM Transactions on Networking* 26.5 (2018), s. 2254–2267.

[Łu95] T. Łuba. „Decomposition of Multiple-Valued Functions". W: *1995 IEEE 25th International Symposium on Multiple-Valued Logic (ISMVL)*. 1995, s. 256–261.

[ŁB15] T. Łuba i G. Borowik. *Synteza logiczna*. Warszawa: Oficyna Wydawnicza Politechniki Warszawskiej, 2015. ISBN: 9788378143123.

[ŁBJ16] T. Łuba, G. Borowik i C. Jankowski. „Gate-based decomposition of index generation functions". W: *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2016*. T. 10031. International Society for Optics & Photonics. 2016, 100314A–1.

[ŁL94] T. Łuba i R. Lasocki. „Decomposition of Multiple-valued Boolean Functions". W: *Applied Mathematics and Computer Science* 4.1 (1994), s. 125–138.

[ŁM18] T. Łuba i T. Mazurkiewicz. „Synteza generatorów indeksów metodami dekompozycji liniowej i funkcjonalnej". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 5 (2018), s. 122–128.

[ŁM19] T. Łuba i T. Mazurkiewicz. „Dekompozycja funkcjonalna w syntezie funkcji generowania indeksów". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 1 (2019), s. 19–25.

[ŁPZ16] T. Łuba, K. Poźniak i B. Zbierzchowski. „Redukcja i kompresja zmiennych w syntezie funkcji generowania indeksów". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 10 (2016), s. 1230–1236.

[ŁR92] T. Łuba i J. Rybnik. „Algorithmic Approach to Discernibility Function with Respect to Attributes and Object Reduction". W: 1992.

[ŁS95] T. Łuba i H. Selvaraj. „A General Approach to Boolean Function Decomposition and its Application in FPGA-Based Synthesis". W: *Vlsi Design* 3 (1995), s. 289–300.

[Maz17] T. Mazurkiewicz. „An efficient hardware implementation of a combinations generator". W: *Technical Sciences / University of Warmia and Mazury in Olsztyn* 4.20 (2017), 405–413.

[Maz19a] T. Mazurkiewicz. „Computationally efficient index generation unit using a Bloom filter". W: *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019*. T. 11176. International Society for Optics i Photonics. 2019, s. 111761L.

[Maz19b] T. Mazurkiewicz. „Realizacja sprzętowego generatora indeksów z wykorzystaniem filtru Blooma". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 3 (2019), s. 61–66.

[Maz20a] T. Mazurkiewicz. „Application of graph theory algorithms in the non-disjoint functional decomposition of specific Boolean functions". W: *Journal of Telecommunications and Information Technology* 3 (2020), s. 67–74.

[Maz20b] T. Mazurkiewicz. „Non-disjoint functional decomposition of index generation functions". W: *2020 IEEE 50th International Symposium on Multiple-Valued Logic*. List. 2020, s. 137–141.

[Maz20c] T. Mazurkiewicz. „On Heuristic Linear Decomposition of Symmetric Index Generation Functions". W: *36th International Business Information Management Association (IBIMA) Conference*. List. 2020, s. 11011–11018.

[MBŁ18] T. Mazurkiewicz, G. Borowik i T. Łuba. „Construction of index generation unit using probabilistic data structures". W: *2018 26th International Conference on Systems Engineering (ICSEng)*. 2018, s. 1–7.

[MŁ17] T. Mazurkiewicz i T. Łuba. „Redukcja liczby zmiennych do reprezentacji funkcji generowania indeksów". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 8-9 (2017), s. 795–798.

[MŁ18] T. Mazurkiewicz i T. Łuba. „Metody wyboru dekompozycji dla algorytmu z wykorzystaniem zbiorów niezgodności i ich wpływ na minimalizację generatorów indeksów". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 8-9 (2018), s. 722–725.

[MŁ19a] T. Mazurkiewicz i T. Łuba. „Linear and Non-linear Decomposition of Index Generation Functions". W: *2019 MIXDES - 26th International Conference Mixed Design of Integrated Circuits and Systems*. 2019, s. 246–251.

[MŁ19b] T. Mazurkiewicz i T. Łuba. „Non-disjoint Decomposition Using r-admissibility and Graph Coloring and Its Application in Index Generation Functions Minimization". W: *2019 MIXDES - 26th International Conference Mixed Design of Integrated Circuits and Systems*. 2019, s. 252–256.

[MU05] M. Mitzenmacher i E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. New York, NY, USA: Cambridge University Press, 2005. ISBN: 0521835402.

[MoŁ19] Ł. Morawski i T. Łuba. „Synteza generatorów indeksów metodą dekompozycji liniowej". W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne* nr 10 (2019), s. 761–771.

[MB08] L. de Moura i N. Bjørner. „Z3: An Efficient SMT Solver". W: *14th Conference on Tools and algorithms for the construction and analysis of systems*. 2008, s. 336–340.

[NSB16] S. Nagayama, T. Sasao i J. T. Butler. „An Efficient Heuristic for Linear Decomposition of Index Generation Functions". W: *2016 IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL)*. 2016, s. 96–101.

[NSB17a] S. Nagayama, T. Sasao i J. T. Butler. „A Balanced Decision Tree Based Heuristic for Linear Decomposition of Index Generation Functions". W: *IEICE Transactions on Information and Systems* E100.D.8 (2017), s. 1583–1591.

[NSB17b] S. Nagayama, T. Sasao i J. T. Butler. „An Exact Optimization Algorithm for Linear Decomposition of Index Generation Functions". W: *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*. 2017, s. 161–166.

[NSB18] S. Nagayama, T. Sasao i J. T. Butler. „An Exact Optimization Method Using ZDDs for Linear Decomposition of Index Generation Functions". W: *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*. 2018, s. 144–149.

[NSB20] S. Nagayama, T. Sasao i J. T. Butler. „On Optimum Linear Decomposition of Symmetric Index Generation Functions". W: *2020 IEEE 50th International Symposium on Multiple-Valued Logic*. Maj 2020, s. 130–136.

[NSM13] H. Nakahara, T. Sasao i M. Matsuura. „A Virus Scanning Engine Using an MPU and an IGU Based on Row-Shift Decomposition". W: *IEICE Trans. Inf. Syst.* 96-D (2013), s. 1667–1675.

[Nak+09] H. Nakahara i in. „The Parallel Sieve Method for a Virus Scanning Engine". W: *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. 2009, s. 809–816.

[Nak+15] H. Nakahara i in. „A Memory-Based IPv6 Lookup Architecture Using Parallel Index Generation Units". W: *IEICE Transactions on Information and Systems* E98.D.2 (2015), s. 262–271.

[Nie+18] A. Niewiadomski i in. „SMT-Solvers in Action: Encoding and Solving Selected Problems in NP and EXPTIME". W: *Scientific Annals of Computer Science* 28.2 (2018), 269–288.

[NO03] S. Niskanen i P. Ostergard. *Cliquer User's Guide*. Spraw. tech. Communcations Laboratory at Helsinki University of Technology, 2003.

[PR04] R. Pagh i F. F. Rodler. „Cuckoo hashing". W: *Journal of Algorithms* 51.2 (2004), s. 122 –144.

[RJŁ01] M. Rawski, L. Jóźwiak i T. Łuba. „Functional decomposition with an efficient input support selection for sub-functions based on information relationship measures". W: *Journal of Systems Architecture* 47.2 (2001), s. 137–155.

[Raw+97] M. Rawski i in. „Non-Disjoint Decomposition of Boolean Functions and Its Application in FPGA-oriented Technology Mapping". W: *Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology*. 1997, s. 24–30.

[Rob01] J. Robson. *Finding a maximum independent set in time* $O(2^{n/4})$. 2001.

[Sas06] T. Sasao. „Design Methods for Multiple-Valued Input Address Generators". W: *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*. 2006.

[Sas08] T. Sasao. „On the numbers of variables to represent sparse logic functions". W: *ICCAD 2008*. 2008.

[Sas11a] T. Sasao. „Index generation functions: Recent developments". W: *2011 IEEE 41st International Symposium on Multiple-Valued Logic*. 2011, s. 1–9.

[Sas11b] T. Sasao. *Memory-Based Logic Synthesis*. 1st. Springer Publishing Company, Incorporated, 2011.

[Sas12] T. Sasao. „Linear decomposition of index generation functions". W: *17th Asia and SouthPacific Design Automation Conference (DAC)*. 2012, 781–788.

[Sas13] T. Sasao. „An Application of Autocorrelation Functions to Find Linear Decompositions for Incompletely Specified Index Generation Functions". W: *2013 IEEE 43rd International Symposium on Multiple-Valued Logic*. 2013, s. 96–102.

[Sas15] T. Sasao. „A Reduction Method for the Number of Variables to Represent Index Generation Functions: s-Min Method". W: *2015 IEEE International Symposium on Multiple-Valued Logic*. 2015, s. 164–169.

[Sas17] T. Sasao. „Index Generation Functions: Minimization Methods". W: *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*. 2017, s. 197–206.

[Sas18] T. Sasao. „On a Memory-Based Realization of Sparse Multiple-Valued Functions". W: *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*. 2018, s. 50–55.

[Sas20] T. Sasao. *Index generation functions*. Synthesis lectures on digital circuits and systems. San Rafael, CA: Morgan & Claypool Publishers, 2020.

[Sas20b] T. Sasao. „On the Minimization of Variables to Represent Partially Defined Classification Functions". W: *2020 IEEE 50th International Symposium on Multiple-Valued Logic*. Maj 2020, s. 117–123.

[SB18] T. Sasao i J. T. Butler. „Decomposition of Index Generation Functions Using a Monte Carlo Method". W: *Advanced Logic Synthesis*. Springer International Publishing, 2018, s. 209–225.

[SFI15] T. Sasao, I. Fumishi i Y. Iguchi. „A method to minimize variables for incompletely specified index generation functions using a SAT solver". W: *International Workshop on Logic and Synthesis, (IWLS-2015)*. 2015, s. 161–167.

[SMI16] T. Sasao, K. Matsuura i Y. Iguchi. „A heuristic decomposition of index generation functions with many variables". W: *The 20th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI-2016)*. 2016, s. 23–28.

[SMI17] T. Sasao, K. Matsuura i Y. Iguchi. „An algorithm to find optimum support-reducing decompositions for index generation functions". W: *Design, Automation and Test in Europe (DATE-2017)*. 2017.

[SMN10] T. Sasao, M. Matsuura i H. Nakahara. „A Realization of Index Generation Functions Using Modules of Uniform Sizes". W: *International Workshop on Logic and Synthesis, (IWLS-2010)*. 2010, s. 201–224.

[SUI14] T. Sasao, Y. Urano i Y. Iguchi. „A Method to Find Linear Decompositions for Incompletely Specified Index Generation Functions Using Difference Matrix". W: *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* 97-A (2014), s. 2427–2433.

[SZP12] D. A. Simovici, M. Zimand i D. Pletea. „Several Remarks on Index Generation Functions". W: *2012 IEEE 42nd International Symposium on Multiple-Valued Logic*. 2012, s. 179–184.

[TRL12] S. Tarkoma, C. E. Rothenberg i E. Lagerspetz. „Theory and Practice of Bloom Filters for Distributed Systems". W: *IEEE Communications Surveys Tutorials* 14.1 (2012), s. 131–155.

[Sage] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.3)*. 2018. URL: `https://www.sagemath.org`.

[Web+19] T. Weber i in. „The SMT Competition 2015–2018". W: *Journal on Satisfiability, Boolean Modeling and Computation* 11.1 (2019), s. 221–259.

[WP67] D. J. A. Welsh i M. B. Powell. „An upper bound for the chromatic number of a graph and its application to timetabling problems". W: *The Computer Journal* 10.1 (sty. 1967), s. 85–86.

[Wer90] D. de Werra. „Heuristics for Graph Coloring". W: *Computational Graph Theory*. Red. G. Tinhofer i in. Vienna: Springer Vienna, 1990, s. 191–208.

[WH15] Q. Wu i J. Hao. *A review on algorithms for maximum clique problems*. European Journal of Operational Research. 2015.

[Yur20] D. Yurichev. *SAT/SMT by Example*. Maj 2020.