POLITECHNIKA POZNAŃSKA

Laboratoria Wprowadzenie do Sztucznej Inteligencji

Projekt Zaliczeniowy

Wiktor Kowalewski 155080
Andrzej Kowalski 155070

Poznań 2024

1 Opis Problemu

1.1 Ogólny zarys problemu

Opracowanie pochyla się nad problemem minimalizacji funkcji logicznych stosowanych w projektowaniu układów cyfrowych. Efektem końcowym ma być zoptymalizowana forma funkcji która minimalizuje liczbę użytych bramek logicznych. Minimalizacja jest kluczowym elementem projektowania układów ponieważ każdy element układu wiąże się z kosztem jego produkcji, zajmuje miejsce na chipie oraz zużywa energię. Minimalizując ich ilość – maksymalizujemy wydajność powierzchni układu.

1.2 Zastosowany Algorytm

Tradycyjne metody opierające się na tablicach Karnaugh, stają się trudne do zastosowania w przypadkach kiedy na funkcję składa się większa liczba zmiennych. Dlatego zastosowany został Algorytm Quine-McCluskey, który jest bardziej systematycznym podejściem, pozwala na skuteczną optymalizację nawet dla funkcji o większej złożoności, jednocześnie zapewniając jednoznaczne rozwiązanie.

1.3 Dane wejsciowe i wyjściowe

Danymi wejściowym jest układ cyfrowy typu $MISO$(Many Inputs Single Output), reprezentowany przez funkcje $f(x_1, x_2, \ldots, x_n)$ gdzie $f : B^n \to B$ oraz $x_1, \ldots, x_n \in B$ gdzie $B = \{0, 1\}$ dla którego zdefiniowane jest zbiór operacji $\circ = \{and, not, xor, or\}$, układ ten potem jest konwertowany na formę bardziej przyjazną operacji minimalizacji przy użyciu wybranego przez nas algorytmu, następnie zminimalizowana funkcja jest z powrotem konwertowana do postaci początkowej.

1.4 Powiązanie z dziedziną Sztucznej Inteligencji

Temat wiąże się z obszarem sztucznej inteligencji, jakim jest optymalizacja i teoria decyzji. Optymalizacja funkcji logicznych jest przykładem rozwiązywania problemu kombinatorycznego, gdzie celem jest znalezienie najlepszej możliwej konfiguracji spełniającej zadane kryteria. Wykorzystanie algorytmu Quine-McCluskey wpisuje się również w kontekst algorytmów deterministycznych – algorytmów w pełni zależnych od danych wejściowych.

2 State of the art - opis znanych koncepcji minimalizacji funkcji logicznych

2.1 Algorytmy genetyczne

Algorytmy genetyczne opierają się na mechanizmach inspirowanych procesami ewolucji biologicznej, takimi jak selekcja, krzyżowanie i mutacja. Rozwiązania (funkcje logiczne) są kodowane jako chromosomy w populacji, następnie porównywane za pomocą funkcji przystosowania. Kolejnym krokiem są procesy ewolucji opisane wcześniej. Proces ewolucji prowadzi do stopniowej poprawy funkcji logicznej w kierunku jej minimalnej postaci poprzez stosowanie wyników poprzedniej generacji do inicjalizacji następnego pokolenia.

2.2 Sieci neuronowe z wnioskowaniem logicznym

Sieci neuronowe wykorzystywane do minimalizacji funkcji logicznych opierają się na uczeniu maszynowym, gdzie model uczy się reguł uproszczania na podstawie zestawu treningowego składającego się z par: funkcja wejściowa – zminimalizowana postać. Proces minimalizacji jest realizowany poprzez wnioskowanie logiczne w sieci.

2.3 Algorytm Quine-McCluskey

Algorytm Quine-McCluskey jest deterministycznym i systematycznym podejściem do minimalizacji funkcji logicznych. Działa na podstawie dwóch głównych kroków: redukcji implikantów-wyrażeń logiczych i wyboru pokrycia minimalnego-najmniejszej liczby implikantów opisujących daną funkcję. Używa tablic i reguł upraszczania, co pozwala na dokładną analizę funkcji logicznych.

2.4 Porównanie

Każde z podejść do przedstawionego posiada swoje wady oraz zalety.

- Algorytmy genetyczne dobrze sprawdzają się w przypadku większych i bardziej skomplikowanych funkcji logicznych. Jednak wymagają dokładnego tuningu parametrów ewolucji oraz nie gwarantują najbardziej optymalnego rozwiązania.
- Sieci neuronowe to najszybsze dostępne rozwiązanie. Świetnie radzą sobie ze złożonymi funkcjami, uczą się wraz z każdym przedstawionym problemem. Jednak nie obędzie się bez minusów: wymagany czas i zasoby do początkowego wytrenowania i wyuczenie reguł są bardzo duże, ogromne ilości danych treningowych. Na dodatek interpretacja wewnętrznego działania staje się praktycznie niemożliwa ze względu na wymaganą liczbę warstw neuronów a wraz z tym wag i przesunięć-mamy do czynienia z tzw. black-boxem. A na koniec i tak nie mamy 100% pewności że rozwiązanie jest optymalne
- Algorytm Quine-McCluskey w tym zestawieniu jest jedynym rozwiązaniem deterministycznym. Zawsze mamy pewność że wynik został zoptymalizowany, do tego cały proces jest przejrzysty i łatwy do zinterpretowania. Niestety ze względu na swoją naturę, w przypadku dużej liczby zmiennych wejściowych wymagany czas i pamięć obliczeniowa rośnie wykładniczo.

2.5 Nasz wybór

Wybór padł na algorytm Quine-McCluskey. Zestawiając go z resztą możliwości, w tym przypadku jest najlepszym z wyborów - nie wymaga zasobów danych i czasu do wyuczenia. Nie potrzeba również niezliczonych godzin wymaganych do tuningu i testu parametrów jak ma to miejsce w przypadku rozwiązania genetycznego. Wyniki są jednoznaczne, kroki postępowania przejrzyste i łatwe do prześledzenia a implementacja osiągalna. Rosnąca wykładniczo wymagana moc obliczeniowa nie jest problemem ponieważ układ testowany będzie dla relatywnie małej liczby zmiennych wejściowych.

3 Opis rozwiązania

3.1 Zasada działania algorytmu Quine-McCluskey

3.1.1 Reprezentacja funkcji w postaci kanonicznej

Funkcja logiczna musi być przedstawiona jako suma iloczynów - mintermów - dla których przyjmuje wartość 1. Początkowo każdy minterm jest zapisany jako reprezentacja binarna danej wartości.

3.1.2 Grupowanie mintermów

Następnie każdy z mintermów zostaje przydzielony do swojej grupy na podstawie liczby 1 w jego zapisie binarnym:

- Grupa 0: mintermy bez jedynek.
- Grupa 1: mintermy z 1 jedynką.
- Grupa 2: mintermy z 2 jedynkami.

I tak dalej aż do grupy z maksymalną liczbą jedynek.

3.1.3 Porównywanie i upraszczanie mintermów

Każdy z mintermów w danej grupie jest porównywany z mintermami z następnej grupy - posiadającej jedną 1 więcej. Jeśli różnią się one tylko jednym bitem to zostają złączone w nowy minterm którego wspomniany bit zostaje zastąpiony symbolem `"-"` oznaczając że może on przyjąć dowolną wartość jeśli pozostałe bity są w danej konfiguracji.
Przykładowo 10101 i 11101 różnią się jednym bitem więc wynikiem połączenia będzie 11-01.

3.1.4 Rekursywne porównywanie oraz zapis pierwotnych implicantów

Stare i nowo powstałe mintermy są porównywane aż do momentu utworzenia pierwotnych implicantów. Pierwotne implicanty są mintermami których nie da się już uprościć.

3.1.5 Tabela pokrycia

Tworzona jest tabela pokrycia która sprawdza porównuje implicanty z podstawowymi mintermami z kroku 1 i określie które implicanty pokrywają które mintermy.

3.1.6 Wybór istotnych implicantów

Z tabeli wybierane są istotne implicanty. One jako jedyne pokrywają konkretne mintermy. Oznacza to że muszą one znaleźć się w zminimalizowanej funkcji logicznej

3.1.7 Pokrywanie reszty mintermów

Jeśli po wyborze istotnych implicantów jakieś mintermy zostaną niepokryte, wybiera się dodatkowe implicanty żeby zapewnić 100% pokrycia.

3.1.8 Wynikowa funkcja logiczna

Po uzyskaniu pełnego pokrycia, wynikowa funkcja logiczna jest zapisana jako suma wybranych implicantów.

3.2 Opis działania programu oraz wymaganych danych

Danymi wejściowymi realizacji algorytmu będzie ciągu znaków opisujący funkcję logiczną przy pomocy podstawowych elementów logicznych oraz bitów. Jako przykład możemy dać funkcję booleanowska

$$f(x) = x_3 + x_1 \oplus (\overline{(x_0 * \overline{x_2})} * (x_3 + (x_1 \oplus x_3 + \overline{x_5}) \oplus x_4))$$

gdzie $\oplus$ oznacza operacja logiczna XOR. Na wyjściu dostaniemy zminimalizowaną funkcję booleanowska opisaną w konwencji POS(produkt sum logicznych). Program można podzielić na 3 części:

- Część 1: Parser układu MISO na listę mintermów.
- Część 2: Algorytm minimalizacji.
- Część 3: Parser zminimalizowanej funkcji logicznej na układ MISO

Ze względu na deterministyczny charakter wewnętrznego algorytmu odpowiedzialnego za minimalizację, nie są wymagane żadne dane do uczenia sieci neuronowych. Jednymi potrzebnymi zasobami jest komputer z zainstalowanym językiem Python.
Niestety problemem tego rozwiązania jest jego złożoność czasowa tj. $O(N2^N)$ gdzie: **N** to liczba bitów wymagana do zapisania mintermów. Jak widać zależność jest wykładnicza co skutecznie ogranicza zakres sensowności działania tego rozwiązania. Sprawdzanie poprawności opieramy o porównanie funkcji wynikowej z początkową oraz dwa publicznie dostępne kalkulatory wykorzystujące algorytm Quine-McCluskey.
Znaleźć je można pod tymi linkami: Kalkulator nr.1 Kalkulator nr.2

4 Proof of Concept

W celu demonstracji działania opracowanego algorytmu, przygotowano testowy przykład, który przechodzi przez wszystkie etapy minimalizacji funkcji logicznej, bazując na wprowadzonym algorytmie Quine-McCluskey.
Program korzysta z wersji Pythona 3.12 oraz poniższych bibliotek:

```
1 import re
  import itertools
```

Proces ten obejmuje:

1. Wprowadzenie funkcji logicznej w formie nieskomplikowanej
2. Konwersja funkcji do zestawu mintermów
3. Minimzalizacja funkcji przy użyciu algorytmu Quine-McCluskey
4. Prezentacja wyniku w formie zminimalizowanej funkcji logicznej

4.1 Funkcja logiczna

Wprowadzana funkcja jest zapisana przy użyciu zmiennych oraz operatorów logicznych:

- $\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}$ - zmienne logiczne
- $\sim$ - negacja zmiennej
- & - Logiczny AND
- | - Logiczny OR
- ^ - Logiczny XOR

$$f(x) = x_3|x_1\hat{}(\sim(x_0\&(\sim x_2))\&(x_3|(x_1\hat{}x_3|(\sim x_5))\hat{}x_4))$$

4.2 Konwersja

Zaprojektowany został prosty parser pozwalający przełożyć zapis funkcji f(x) na mintermy:

```
def parser(str):
    if re.fullmatch("([ \\^\\&\\|\\~\\(\\)]|(x\\d+))+", str) is None:
        raise ValueError('Incorect expression.')
    a = sorted(list(set(re.findall("x\\d+", str))))
    lst = list(itertools.product([0, -1], repeat=len(a)))
    l = eval("lambda " + ",".join(a) + ":" + str)
    return ([bit2num(x) for x in lst if l(*x) == -1], a)
```

Zwraca on kolejno listę mintermów oraz nazwy użytych zmiennych. Następnie lista mintermów jest przekazywana do algorytmu optymalizującego.

4.3 Minimalizacja

Algorytm korzysta z szeregu funkcji pomocniczych które kolejno wykonują kroki algorytmu:

```
def count_ones(binary_str):
    """
    Zlicza liczbę jedynek w reprezentacji binarnej.
    """
    return binary_str.count('1')

def is_adjacent(term1, term2):
    """
    Sprawdza, czy dwa terminy różnią się dokładnie jedną pozycją.
    """
    diff_count = sum(1 for a, b in zip(term1, term2) if a != b)
    return diff_count == 1
```

```python
def merge_terms(term1, term2):
    """
    Łączy dwa terminy, które różnią się dokładnie jedną pozycją.
    """
    return ''.join(a if a == b else '-' for a, b in zip(term1, term2))

def is_covered(minterm, implicant):
    """
    Sprawdza, czy minterm jest pokrywany przez implicant.
    """
    return all(m == i or i == '-' for m, i in zip(minterm, implicant))

def remove_redundant_implicants(essential_prime_implicants, coverage, minterms_bin):
    """
    Usuwa nadmiarowe implicanty, pozostawiając minimalny zestaw pokrywający
    wszystkie mintermy.
    """
    minimized_implicants = set(essential_prime_implicants)
    for implicant in list(minimized_implicants):
        # Sprawdź, czy implicant można usunąć
        if all(any(is_covered(m, other) for other in minimized_implicants - {implicant})
               for m in minterms_bin if is_covered(m, implicant)):
            minimized_implicants.remove(implicant)
    return list(minimized_implicants)
```

Całość zamknięta jest w funkcji głównej, której ze względu na rozmiar nie będziemy umieszczać w opracowaniu gdyż byłaby kompletnie nieczytelna. Istotnym jest rodzaj zwracanych przez nią danych tj. lista zminimalizowanych implicantów reprezentowanych w poniższy sposób:

- **0** - ten bit musi mieć wartość 0
- **1** - ten bit musi mieć wartość 1
- **"-"** - Wartość tego bitu nie ma znaczenia (obydwie spełnią funkcję)

W tym przykładzie będzie to lista o postaci: ['00–00', '—1–', '–1-11', '110—', '0—11', '-01-00', '-1–1-']

4.4 Wynik końcowy

Aby wynik końcowy był bardziej czytelny, stosujemy ponowne parsowanie na postać zapisaną z użyciem podanych zmiennych logicznych.

```
def minterm_print(minterms, a):
    ret = ""
    for term in minterms:
        ret += "("
        for i, t in enumerate(term):
            if t == '-':
                continue
            if i > 0 and ret[-1] != "(":
                ret += " & "
            if t == '1':
                ret += a[i]
            elif t == '0':
                ret += '~' + a[i]
        ret += ") | "
    return ret[:-3]
```

Stosując ją dla naszego wyniku minimalizacji algorytmem otrzymujemy zapis pod postacią:

$$f(x) = (\sim x_0 \& \sim x_1 \& \sim x_4 \& \sim x_5) | (x_3) | (x_2 \& x_4 \& x_5) | (x_0 \& x_1 \& \sim x_2) | (\sim x_0 \& x_4 \& x_5) | (\sim x_1 \& x_2 \& \sim x_4 \& \sim x_5) | (x_1 \& x_4)$$

Na pierwszy rzut oka funkcja wydaje się być bardziej skomplikowana niż wejściowa, jednak mając na uwadze efektywność energetyczną oraz fakt obniżenia kosztów produkcji przez zmniejszenie potrzebnych tranzystorów do zaprojektowania układu. Postać przez nas uzyskana jest postacią zminimalizowaną/zoptymalizowaną. Dodatkowo ewentualna naprawa czy wymiana wadliwej bramki staje się zdecydowanie prostsza.

Finalnie przeprowadzamy test czy nowo powstała funkcja jest taką samą jak ta podana na początku przy użyciu funkcji parsowania i porównania uzyskanych mintermów.

```
def test():
    l, a = parser("x3|x1^(~(x0&(~x2))&(x3|(x1^x3|(~x5))^x4))")
    terms = quine_mccluskey(l)
    str = minterm_print(terms, a)
    l2, a2 = parser(str)
    print(l == l2)
```

Wyświetloną wartością jest wartość True.

References

[1] A.-T. N. T. Hoang-Gia Vu Ngoc-Dai Bui, "Performance evaluation of quine-mccluskey method on multi-core cpu," *2021 8th NAFOSTED Conference on Information and Computer Science (NICS)*, 2021.