

UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”
FEIS - FACULDADE DE ENGENHARIA - CÂMPUS DE ILHA SOLTEIRA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

TRABALHO DE GRADUAÇÃO

**ESTUDO DA FASE DE GERAÇÃO DE IMPLICANTES PRIMOS DO
MÉTODO DE QUINE-MCCLUSKEY BUSCANDO UMA HEURÍSTICA**

Manuel José Vasconcelos dos Santos



Ilha Solteira
Fevereiro de 2025

Manuel José Vasconcelos dos Santos

**ESTUDO DA FASE DE GERAÇÃO DE IMPLICANTES PRIMOS DO
MÉTODO DE QUINE-MCCLUSKEY BUSCANDO UMA HEURÍSTICA**

Trabalho de Graduação apresentado à
Faculdade de Engenharia de Ilha
Solteira - UNESP como parte dos
requisitos para obtenção do título de
bacharelado em Engenharia Elétrica.

Orientador:
Prof. Dr. Carlos Antônio Alves

Ilha Solteira
Fevereiro de 2025

FICHA CATALOGRÁFICA

Desenvolvido pelo Serviço Técnico de Biblioteca e Documentação

S237e Santos, Manuel José Vasconcelos dos.
Estudo da fase de geração de implicantes primos do método de Quine-McCluskey buscando uma heurística / Manuel José Vasconcelos dos Santos. -- Ilha Solteira: [s.n.], 2025
92 f. : il.

Trabalho de conclusão de curso (Graduação em Engenharia Elétrica) - Universidade Estadual Paulista (UNESP), Faculdade de Engenharia, Ilha Solteira, 2025

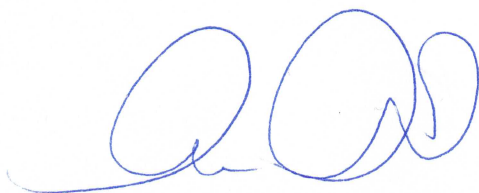
Orientador: Carlos Antônio Alves

Inclui bibliografia

1. Método de Quine-Mccluskey. 2. Geração de implicantes primos. 3. Programa espresso. 4. Linguagem de programação Python.

ATA DE DEFESA DE TRABALHO DE GRADUAÇÃO

Aos seis dias do mês de fevereiro do ano de dois mil e vinte e cinco, o discente **Manuel José Vasconcelos dos Santos**, matriculado sob o nº 152052941, tendo como banca examinadora o seu orientador, o Prof. Dr. Carlos Antonio Alves, o Prof. Dr. Jean Marcos de Souza Ribeiro e o Prof. Dr. Jonatas Boas Leite, apresentou o Trabalho de Graduação intitulado "ESTUDO DA FASE DE GERAÇÃO DE IMPLICANTES PRIMOS DO MÉTODO DE QUINE-MCCLUSKEY BUSCANDO UMA HEURÍSTICA", obtendo a nota 9.0 (NOVE) e conceito APROVADO.



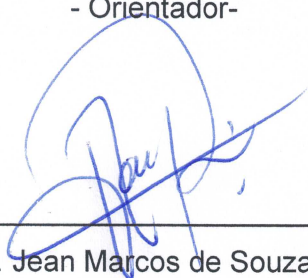
Prof. Dr. Carlos Antonio Alves

- Orientador -



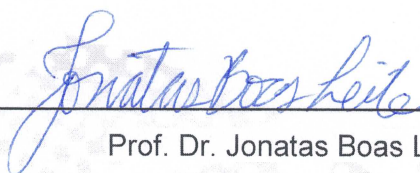
Manuel José Vasconcelos dos Santos

- Discente -



Prof. Dr. Jean Marcos de Souza Ribeiro

- Membro da Banca -



Prof. Dr. Jonatas Boas Leite

- Membro da Banca -

Aos meus pais Marta e Raimundo,
dedico este trabalho.

AGRADECIMENTOS

Aos meus pais, Marta e Raimundo, gratidão, pelo suporte que me permitiu concretizar esta conquista.

À minha parceira Camylla por se fazer presente ao meu lado em todos os momentos.

Ao professor Carlos pela oportunidade de orientação, além do conhecimento compartilhado na graduação.

Ao professor Alexandre, pelo conhecimento compartilhado na graduação, e o auxílio na confecção deste trabalho.

Aos professores Jonatas e Jean pelas contribuições e sugestões para enriquecimento deste trabalho.

À UNESP de Ilha Solteira, seu corpo docente, o conselho de curso da engenharia elétrica, ao professor Kitano, e demais funcionários.

RESUMO

O problema de minimização de funções Booleanas busca otimizar os circuitos lógicos mantendo sua funcionalidade e reduzindo custo de implementação. Há diversas técnicas de minimização de funções booleanas descritas na literatura, como o método de Quine-McCluskey, que gera todos os implicantes primos para então fazer a cobertura dos mintermos da função, empregando os implicantes primos gerados. É notável o baixo desempenho computacional do Método de Quine-McCluskey para funções com muitas variáveis. Algoritmos heurísticos têm atingido resultados satisfatórios quanto a essas limitações computacionais, como o programa Espresso. Na maioria das aplicações não se faz necessária a solução exata ou global da função, uma solução local ou mínima será suficiente. Neste trabalho implementou-se em linguagem de programação Python um algoritmo que modifica a fase de geração de implicante primos do método de Quine-McCluskey, de modo a gerar implicantes primos num período relativamente curto de tempo, quando comparado com o método de Quine-McCluskey tradicional. Para avaliar a eficiência, o algoritmo proposto foi comparado com o método de Quine-McCluskey tradicional e com o programa Espresso, considerando a quantidade de memória utilizada, o tempo de execução, os termos gerados e os custos das soluções geradas. A partir dos resultados obtidos pode-se concluir que, o método Quine-McCluskey modificado tem menor tempo de execução que o método de Quine-McCluskey tradicional, e menor tempo de execução que o programa Espresso para menos de 8 variáveis de entrada. Para funções com poucas variáveis o método de Quine-McCluskey tradicional faz menor uso de memória que o método de Quine-McCluskey modificado. Entretanto, do ponto de vista da minimização o método de Quine-McCluskey modificado não apresenta um resultado satisfatório, pois não gera uma solução de custo mínimo para a maioria dos casos ensaiados, como geram o método de Quine-McCluskey tradicional e o programa Espresso.

Palavras-chave: Método de Quine-McCluskey. Geração de implicantes primos. Programa Espresso. Linguagem de programação Python.

ABSTRACT

The minimization problem of boolean functions seeks to optimize logic circuits keeping its functionality and reducing implementation cost. There are many minimization techniques in literature, such as the Quine-McCluskey method, that generates all prime implicants to then do the function minterms coverage using the generated prime implicants. The low computing performance of the Quine-McCluskey method for functions with many product terms is noticeable. Heuristic algorithms have achieved satisfactory results regarding these computational limitations, such as the Espresso program. In most applications an exact or global solution of the function is not necessary, a local or minimal solution will suffice. In this work an algorithm was implemented in the Python programming language that modifies the prime implicant generation phase of the Quine-McCluskey method, in order to generate prime implicants in a relatively short period of time, when compared to the traditional Quine-McCluskey method. To evaluate the method's efficiency, the proposed method was compared to the traditional Quine-McCluskey method and the Espresso program, the amount of memory used, execution time, generated terms and the costs of generated solutions were considered. From the results obtained, it can be concluded that the modified Quine-McCluskey method generates fewer implicants than the other methods, therefore it requires a shorter execution time than the traditional Quine-McCluskey method, and a shorter execution time than the program Espresso for less than 8 input variables. For functions with few variations, the traditional Quine-McCluskey method uses less memory than the modified Quine-McCluskey method. However, from a minimization standpoint the modified Quine-McCluskey method does not present a satisfactory result, since it does not generate a minimum cost solution for many of the cases tested, as does the traditional Quine-McCluskey method and the ESPRESSO program.

Keywords: Prime Implicants Generation. Quine-McCluskey method. Espresso program. Python programming language.

LISTA DE FIGURAS

Figura 1 - Simbologia da porta lógica NOT.....	15
Figura 2 - Simbologia da porta lógica AND.....	15
Figura 3 - Porta lógica OR.....	16
Figura 4 - Mapa de Karnaugh 2 variáveis.....	22
Figura 5 - Mapa de Karnaugh: (a) 3 variáveis e (b) 4 variáveis.....	22
Figura 6 - Mapa de Karnaugh 5 variáveis.....	22
Figura 7 - Mapa de Karnaugh $F(A,B,C,D) = \sum m(0,1,2,5,6,7,8,9,10,13,14,15)$ - Solução não mínima.....	23
Figura 8 - Mapa de Karnaugh $F(A,B,C,D) = \sum m(0,1,2,5,6,7,8,9,10,13,14,15)$ - Solução mínima.....	24
Figura 9 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2.....	25
Figura 10 - Cobertura dos Implicantes Primos.....	26
Figura 11 - Função $F(A,B,C,D) = \sum m(0, 1, 2, 4, 5, 6, 9, 10, 11, 13, 14, 15)$	27
Figura 12 - Operação Expand.....	28
Figura 13 - Operação IrredundantCover.....	28
Figura 14 - Operação Reduce.....	29
Figura 15 - Operação Iterate: (a) Expand 2 e (b) IrredundantCover 2.....	29
Figura 16 - Solução Reshaping.....	30
Figura 17 - Fluxograma do método de Quine-McCluskey modificado.....	34
Figura 18 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 0	35
Figura 19 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 1.....	35
Figura 20 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 2.....	36
Figura 21 - Cobertura dos Implicantes Primos.....	36
Figura 23 - Plotagem de Implicantes gerados na fase modificada (Tudo um).....	42
Figura 24 - Plotagem do uso de memória (Tudo um).....	43
Figura 25 - Geração de Implicantes Primos: (a) Grupo 0 e (b) Grupo 1 - (FI1).....	44
Figura 26 - Cobertura dos Implicantes Primos - (FI1).....	44

LISTA DE FIGURAS

Figura 27 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FI2).....	45
Figura 28 - Cobertura dos Implicantes Primos - (FI2).....	46
Figura 29 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FI3).....	46
Figura 30 - Cobertura dos Implicantes Primos - (FI3).....	47
Figura 31 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FI4).....	47
Figura 32 - Cobertura dos Implicantes Primos - (FI4).....	48
Figura 33 - Geração de Implicantes Primos: (a) Grupo 0 e (b) Grupo 1 - (FI5).....	49
Figura 34 - Cobertura dos Implicantes Primos - (FI5).....	49
Figura 35 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP1).....	50
Figura 36 - Cobertura dos Implicantes Primos - (FP1).....	50
Figura 37 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP2).....	51
Figura 38 - Cobertura dos Implicantes Primos - (FP2).....	52
Figura 39 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP3).....	52
Figura 40 - Cobertura dos Implicantes Primos - (FP3).....	53
Figura 41 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP4).....	54
Figura 42 - Cobertura dos Implicantes Primos - (FP4).....	54
Figura 43 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP5).....	55
Figura 44 - Cobertura dos Implicantes Primos - (FP5).....	56
Figura 46 - Plotagem de Implicantes gerados na fase modificada (Outros casos).....	58
Figura 47 - Plotagem do uso de memória (Outros casos).....	59
Figura 48 - Plotagem dos tempos de execução dos métodos (Outros casos).....	60

LISTA DE FIGURAS

Figura 49 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 0 (Comparativo).....	61
Figura 50 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 1: (a) Quine-McCluskey tradicional; (b) Quine-McCluskey modificado.....	62
Figura 51 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 2: (a) Quine-McCluskey tradicional; (b) Quine-McCluskey modificado.....	62
Figura 52 - Cobertura dos Implicantes Primos - Quine-McCluskey tradicional.....	63
Figura 53 - Cobertura dos Implicantes Primos - Quine-McCluskey modificado.....	63
Figura 54 - Soluções obtidas: (a) Quine-McCluskey tradicional; (b) Quine-McCluskey modificado.....	64

LISTA DE TABELAS

Tabela 1 - Tabela Verdade da porta lógica NOT.....	15
Tabela 2 - Tabela Verdade AND.....	16
Tabela 3 - Tabela Verdade OR.....	16
Tabela 4 - Propriedades básicas da Álgebra Booleana.....	17
Tabela 5 - Tabela Verdade $F(A,B,C) = (A \cdot B) + C$	19
Tabela 6 - Tabela Verdade $F(A,B,C) = \sum (m(1, 3, 5, 6, 7) + d(2, 4))$	20
Tabela 7 - Gasto de tempo do programa Espresso por etapa.....	31
Tabela 8 - Tempo de execução dos métodos implementados (Tudo um).....	39
Tabela 9 - Implicantes gerados na fase modificada dos métodos de Quine-McCluskey e método de Quine McCluskey modificado (Tudo um).....	41
Tabela 10 - Uso de memória em Megabytes na fase modificada dos métodos de Quine-McCluskey e método de Quine McCluskey modificado (Tudo um).....	42
Tabela 11 - Custo de implementação das soluções (Outros casos).....	57
Tabela 12 - Implicantes gerados na fase modificada (Outros casos).....	58
Tabela 13 - Uso de memória em Megabytes na fase modificada (Outros casos).....	59
Tabela 14 - Tempo de execução dos métodos implementados (Outros casos).....	60

SUMÁRIO

1 INTRODUÇÃO.....	10
1.1 OBJETIVOS.....	11
1.2 ORGANIZAÇÃO DO TEXTO.....	11
2 REVISÃO DA LITERATURA.....	12
3 FUNDAMENTAÇÃO TEÓRICA.....	14
3.1 ÁLGEBRA DE BOOLE.....	14
3.1.1 OPERADORES DA ÁLGEBRA BOOLEANA.....	14
3.1.2 PROPRIEDADES DA ÁLGEBRA BOOLEANA.....	17
3.2 FUNÇÕES BOOLEANAS.....	19
3.3 CRITÉRIO DE CUSTO.....	20
3.4 MAPA DE KARNAUGH.....	21
3.5 MÉTODO DE QUINE-MCCLUSKEY.....	24
3.6 PROGRAMA ESPRESSO.....	27
4 DESENVOLVIMENTO.....	32
4.1 MÉTODO DE QUINE-MCCLUSKEY MODIFICADO.....	32
4.1.1 DESCRIÇÃO DO ALGORITMO.....	33
4.1.2 FLUXOGRAMA DO ALGORITMO.....	34
4.2 EXEMPLO EM MAPA DE KARNAUGH.....	35
5 RESULTADOS E DISCUSSÕES.....	37
5.1 A LINGUAGEM PYTHON.....	37
5.1.1 O IDE PYCHARM.....	37
5.1.2 O IDE JUPYTER.....	37
5.1.3 BIBLIOTECAS UTILIZADAS.....	38
5.2 TESTES “TUDO UM”.....	38
5.2.1 TEMPO DE EXECUÇÃO.....	39
5.2.2 IMPLICANTES GERADOS NA FASE MODIFICADA.....	40
5.2.3 USO DE MEMÓRIA.....	42

SUMÁRIO

5.3 OUTROS CASOS TESTADOS.....	44
5.3.1 TESTES EM QUE OBTEVE-SE O CUSTO MÍNIMO.....	44
5.3.1.1 $F(A, B, C, D) = \sum m(0, 2, 5, 6, 10, 11)$	44
5.3.1.2 $F(A, B, C, D) = \sum m(1, 2, 5, 6, 9, 10, 13, 14)$	45
5.3.1.3 $F(A, B, C, D) = \sum m(1, 2, 3, 6, 9, 10, 11, 12, 14)$	46
5.3.1.4 $F(A, B, C, D) = \sum m(0, 1, 2, 4, 5, 6, 9, 10, 11, 13, 14, 15)$	47
5.3.1.5 $F(A, B, C, D) = \sum m(0, 5, 8, 9, 10, 12, 15, 16, 18, 19, 21, 26, 28, 31)$	48
5.3.2 TESTES EM QUE NÃO SE OBTEVE O CUSTO MÍNIMO.....	50
5.3.2.1 $F(A, B, C, D) = \sum m(2, 4, 6, 10, 12, 14)$	50
5.3.2.2 $F(A, B, C, D) = \sum m(3, 8, 9, 10, 11, 12, 14)$	51
5.3.2.3 $F(A, B, C, D) = \sum m(0, 1, 2, 3, 5, 6, 7, 10, 11, 13, 15)$	52
5.3.2.4 $F(A, B, C, D) = \sum m(3, 6, 11, 14, 16, 18, 19, 24, 26, 27, 30)$	53
5.3.2.5 $F(A, B, C, D) = \sum m(0, 1, 2, 7, 8, 10, 12, 14, 15)$	55
5.4 COMPARAÇÃO ENTRE MÉTODO DE QUINE-MCCLUSKEY TRADICIONAL E MÉTODO DE QUINE-MCCLUSKEY MODIFICADO PARA OS OUTROS CASOS TESTADOS.....	56
5.4.1 CUSTO DE IMPLEMENTAÇÃO.....	56
5.4.2 IMPLICANTES GERADOS NA FASE MODIFICADA.....	57
5.4.3 USO DE MEMÓRIA.....	59
5.4.4 TEMPO DE EXECUÇÃO.....	60
5.5 COMPARATIVO EM MAPA DE KARNAUGH.....	61
6 CONCLUSÃO E SUGESTÃO PARA TRABALHOS FUTUROS.....	65
6.1 CONCLUSÃO.....	65
6.2 SUGESTÃO PARA TRABALHOS FUTUROS.....	66
REFERÊNCIAS.....	67
APÊNDICE A - CÓDIGO COMENTADO.....	69
APÊNDICE B - CÓDIGO INDENTADO.....	83

1 INTRODUÇÃO

A pesquisa e desenvolvimento na área de semicondutores possibilitaram a democratização dos sistemas digitais, tornando-os acessíveis ao grande público. Esse progresso impulsiona ainda mais o avanço do setor, ocasionando em um incessante aumento da complexidade de *design* em sistemas digitais.

As técnicas de projetos para a eletrônica digital são então impulsionadas de modo a viabilizar a implementação de circuitos integrados cada vez mais complexos, buscando minimizar grandezas como espaço físico ocupado e consumo de energia, sem perder sua funcionalidade e desempenho.

Os sistemas de eletrônica digital em sua essência, operam com sinais elétricos discretos através de circuitos de chaveamento, que baseados na álgebra booleana (BOOLE, 1854), interpretam esses sinais como valores binários, 0 (zero) e 1 (um), na tecnologia TTL(Transistor-Transistor Logic) o 0 lógico corresponde à faixa de tensão entre 0 V e 0,8 V, para um sinal de entrada, e tensão na faixa entre 0 V e 0,4 V, para um sinal de saída. O 1 lógico corresponde a faixa de tensão de 2,0 V a V_{cc} para um sinal de entrada e a faixa de tensão de 2,4 V a V_{cc} para um sinal de saída (TOCCI; WIDMER; MOSS, 2011).

As saídas dos elementos dos circuitos de chaveamento, como as portas lógicas e os transistores, podem ser modeladas e analisadas através de equações booleanas, com os sinais elétricos alimentando as entradas do circuito. Estas equações podem ser simplificadas com base nos teoremas e axiomas da álgebra Booleana, proporcionando a otimização de circuitos lógicos.

São diversas as técnicas para simplificação de funções booleanas desenvolvidas ao longo dos anos, cada uma com suas vantagens e limitações de abordagem. Dentre as técnicas mais populares de minimização lógica de dois níveis, pode-se citar o método de Quine-McCluskey (MCCLUSKEY, 1956), (MCCLUSKEY, 1961), baseado nos procedimentos de Quine (QUINE, 1952), que consiste na geração de todos os implicantes primos da função (1ª fase do método) e na cobertura dos mintermos (2ª fase do método) para obter uma função mínima.

1.1 OBJETIVOS

Este trabalho de graduação teve como objetivo desenvolver e implementar em programa de computador, um algoritmo que modifica a etapa de geração de implicants primos do algoritmo tradicional de Quine-McCluskey, de modo a evitar redundâncias. Dessa forma, a minimização da função Booleana demanda menos recurso computacional, atacando diretamente uma das fraquezas do algoritmo de Quine-McCluskey, que perde muito em desempenho para funções com muitas variáveis.

Desenvolve-se o algoritmo com o emprego da linguagem de programação Python e as várias bibliotecas associadas. Visando avaliar o desempenho do algoritmo desenvolvido, foram realizados testes, comparando os resultados obtidos pelo programa proposto com os resultados gerados pelo Método de Quine-McCluskey tradicional (MCCLUSKEY, 1956) e o programa Espresso-II (BRAYTON, 1984), também chamado de programa Espresso.

1.2 ORGANIZAÇÃO DO TEXTO

Este trabalho está subdividido em 6 capítulos. No Capítulo 2 é feita uma revisão da literatura com respeito aos métodos tabulares e heurísticos de simplificação de funções booleanas. No Capítulo 3 são apresentados os conceitos básicos que fundamentam a teoria utilizada neste trabalho, a álgebra booleana, suas leis, teoremas e operadores lógicos, os métodos tabulares de simplificação de funções booleanas como o Mapa de Karnaugh e o Método de Quine-McCluskey, a heurística do programa Espresso. No Capítulo 4 é descrita a metodologia adotada neste trabalho e o método modificado proposto. O capítulo 5 corresponde aos testes realizados, e os resultados obtidos. No Capítulo 6 apresenta-se a conclusão do trabalho e a sugestão para trabalhos futuros. No Apêndice apresenta-se o código comentado (**APÊNDICE A**) e o código indentado (**APÊNDICE B**).

2 REVISÃO DA LITERATURA

O matemático e filósofo George Boole, buscando sistematizar a lógica de proposições, adaptou conceitos lógicos para um formato algébrico no propósito de representar proposições e suas correlações. Publicou em 1854 um livro intitulado “*An investigation into the Laws of Thought*”, em Português “Uma investigação sobre a as Leis do Pensamento” (BOOLE, 1854), onde foi descrito um modo de tomar decisões lógicas com base em circunstâncias verdadeiras ou falsas.

Posteriormente em 1938, Claude Elwood Shannon, aplicou este conceito no trabalho denominado “*A Symbolic Analysis of Relay and Switching Circuits*”, em Português “Uma análise simbólica de circuitos de relé e comutação” (SHANNON, 1938), para mostrar que as propriedades de circuitos elétricos de chaveamento (*switching*) podem ser representadas por uma álgebra Booleana com dois valores 0 e 1, o que deu início ao campo da eletrônica digital. Já em 1952, Quine teorizou um método para encontrar formas normais simples disjuntivas de uma função verdade (QUINE, 1952). Em outro trabalho, apresentou a utilização da operação de consenso entre dois termos adjacentes para gerar todos os implicantes primos de uma função. O procedimento não considera estados irrelevantes (*don't care*) e era utilizado para funções simples.

Em 1956 McCluskey, baseado nos procedimentos de Quine, propôs um método que consistia na geração de implicantes primos e cobertura dos mintermos para obter uma função mínima (MCCLUSKEY, 1956), e em 1961 McCluskey desenvolveu um método para minimizar funções booleanas com múltiplas saídas (MCCLUSKEY, 1961). Nos anos seguintes, a área de simplificação de funções booleanas perdeu o interesse econômico, que somente voltou a ser impulsionado na década de 70, com o advento dos PLA (*Programmable Logic Array*), empregados na implementação de funções booleanas. Este tipo de dispositivo eletrônico permitiu o desenvolvimento de métodos para a minimização de funções com múltiplas saídas, cujos implicantes primos compartilhados são gerados uma única vez, reduzindo o número de portas lógicas do circuito e o custo das funções resultantes. As primeiras abordagens heurísticas bem sucedidas foram o programa MINI (HONG, 1974), desenvolvido na IBM, e o programa PRESTO (BROWN, 1981).

Em 1982, Brayton criou o programa ESPRESSO-I (BRAYTON, 1982), que comparou e apresentou melhorias em diversas estratégias empregadas pelos programas MINI e PRESTO. Com o intuito de chegar numa solução mais próxima de um ótimo global com os recursos computacionais da época, é publicado em 1984 um método heurístico muito importante para a minimização de funções booleanas que se tornou referência para a comparação para os métodos que o sucederam, o programa denominado ESPRESSO-II (BRAYTON, 1984). Em 2001 foi proposto o método BOOM. Em sua estrutura, a heurística combina a fase de geração dos implicantes primos com a fase do problema de cobertura, o que reduz os implicantes primos gerados (HLAVICKA; FISER, 2001).

Em 2012, Guha (GUHA, 2012), propõe em seu trabalho uma heurística que tenta obter custo benefício melhor ou igual ao Espresso, ao cobrir os minterms na etapa inicial buscando o mínimo de redundâncias nos implicantes gerados. O algoritmo de Guha remove a etapa *Reduce* utilizada no Espresso, uma vez que *Expand* e *Irredundant* levam a uma solução próxima da ótima.

Utilizou-se uma estratégia de poda para identificar e priorizar implicantes primos essenciais, selecionando apenas os termos que cobrem todas as combinações verdadeiras necessárias. Assim há uma redução do espaço de busca acelerando o processamento e reduzindo a quantidade de candidatos iniciais a serem simplificados. Também é imposto um limite máximo de iterações de modo a evitar que o algoritmo entre em ciclos desnecessários.

É introduzida uma abordagem *greedy-heuristic*, heurística gananciosa que consiste em aplicar repetidamente a etapa *Expand* e *Irredundant*, até chegar num implicante irredundante e de custo mínimo.

O algoritmo de Guha superou o Espresso em termos de eficiência para a maioria dos casos testados, reduzindo significativamente o tempo de execução em casos de alta complexidade.

Neste trabalho de graduação, inspirando-se no trabalho de Guha, é empregada uma modificação que busca obter uma forma irredundante dos minterms da função no início do algoritmo, reduzindo a complexidade dos casos trabalhados. Busca-se uma alternativa de custo benefício melhor ou igual ao método de Quine-McCluskey tradicional, eliminando etapas redundantes do algoritmo.

3 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo apresenta-se os principais conceitos teóricos e ferramentas que foram utilizadas neste trabalho, os fundamentos da álgebra de Boole que possibilita a representação de circuitos de sistemas digitais como funções matemáticas computáveis, para que venham a ser minimizadas através de técnicas de simplificação como o Mapa de Karnaugh, o método de Quine-McCluskey e o algoritmo Espresso.

3.1 ÁLGEBRA DE BOOLE

Em busca de sistematizar a lógica de proposições, George Boole adaptou conceitos lógicos para um formato algébrico no propósito de representar proposições e suas correlações (BOOLE, 1854). Tal conceituação matemática no contexto histórico reflete o desenvolvimento do pensamento lógico e matemático, e possui forte conexão com a revolução tecnológica. A Álgebra de Boole, Álgebra Booleana, ou Álgebra Booleana Binária, é um conjunto de leis e teoremas onde suas operações lógicas podem ser expressas matematicamente. O engenheiro Claude Shannon publicou uma obra (SHANNON, 1938), na qual representou circuitos de chaveamento através da álgebra booleana, fazendo com que a partir de então fosse amplamente utilizada para a construção de projetos de circuitos e a análise e simplificação de sistemas digitais (SANCHES, 2017).

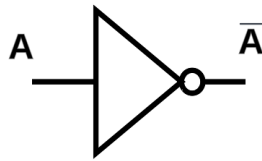
3.1.1 OPERADORES DA ÁLGEBRA BOOLEANA

A estruturação de circuitos lógicos podem ser representadas a partir da utilização das três operações fundamentais da álgebra booleana, sendo elas o operador unário *NOT*, e os operadores binários *OR* e *AND* (TOCCI; WIDMER; MOSS, 2011).

A operação *NOT* opera a negação da variável. Se a variável é representada por *A*, então sua negação, ou complementar, assume a forma $\bar{A}$ (lê-se “A barrado” ou “not A”). As operações lógicas podem ser representadas por circuitos

denominados gates, ou portas lógicas, o símbolo da porta lógica NOT é mostrado na Figura 1.

Figura 1 - Simbologia da porta lógica NOT



Fonte: Elaborado pelo autor

Na Tabela 1 apresenta-se a tabela verdade da porta lógica NOT.

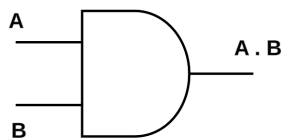
Tabela 1 - Tabela Verdade da porta lógica NOT.

INPUT	OUTPUT
A	$\bar{A}$
1	0
0	1

Fonte: Elaborado pelo autor

A operação AND ou “E”, opera a multiplicação da variável, que resulta em 1 se cada uma das entradas do sistema forem iguais a 1 e resulta em 0 se uma ou mais entradas forem iguais a 0. Sua representação algébrica é dada por “ $\cdot$ ”, da seguinte forma, $A \cdot B$. O símbolo da porta lógica AND é mostrado na Figura 2.

Figura 2 - Simbologia da porta lógica AND.



Fonte: Elaborado pelo autor

Na Tabela 2 apresenta-se a tabela verdade da porta lógica AND.

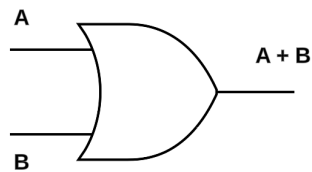
Tabela 2 - Tabela Verdade *AND*

<i>INPUT</i>		<i>OUTPUT</i>
A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

Fonte: Elaborado pelo Autor

A operação OR ou “OU” opera a adição, que resulta em 1 se uma ou mais das entradas corresponder a 1 e igual a 0 apenas se as entradas corresponderem a 0. Sua representação é dada por “ + “, da seguinte forma, $A + B$. O símbolo da porta lógica OR é mostrado na Figura 3.

Figura 3 - Porta lógica OR



Fonte: Elaborado pelo autor

Na Tabela 3 apresenta-se a tabela verdade da porta lógica OR.

Tabela 3 - Tabela Verdade OR

<i>INPUT</i>		<i>OUTPUT</i>
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

Fonte: Elaborado pelo autor

3.1.2 PROPRIEDADES DA ÁLGEBRA BOOLEANA

São diversas as propriedades da álgebra booleana que permitem a simplificação de expressões, para que circuitos tenham implementação mais eficiente. Para uma variável booleana A, valem as seguintes propriedades apresentadas na Tabela 4.

Tabela 4 - Propriedades básicas da Álgebra Booleana

Propriedades	Exemplo
1 - Identidade	$A + 0 = A$ $A \cdot 1 = A$
2 - Dominação	$A + 1 = 1$ $A \cdot 0 = 0$
3 - Idempotência	$A + A = A$ $A \cdot A = A$
4 - Complemento	$A + \bar{A} = 1$ $A \cdot \bar{A} = 0$

Fonte: Elaborado pelo autor

A álgebra booleana é comutativa, associativa e distributiva em relação às duas operações binárias AND e OR. A Lei Comutativa postula que a ordem das variáveis não altera o resultado, seja para a operação de adição ou multiplicação:

$$A + B = B + A \quad (1)$$

$$A \cdot B = B \cdot A \quad (2)$$

A Lei Associativa dita que as operações podem ser agrupadas sem ter nenhuma alteração no resultado final:

$$(A + B) + C = A + (B + C) \quad (3)$$

$$(A \cdot B) \cdot C = A \cdot (B \cdot C) \quad (4)$$

A Lei Distributiva refere-se que resultado não é alterado mesmo que se expanda uma operação sobre a outra. A soma é distributiva sobre o produto e o produto é distributivo sobre a soma:

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C) \quad (5)$$

$$A + (B \cdot C) = (A + B) \cdot (A + C) \quad (6)$$

Essas propriedades podem ser provadas por indução perfeita, exaustivamente.

É notável que estas propriedades se apresentam aos pares, apresentando a mesma estrutura com as operações OR e AND e os valores 0 e 1 invertidos entre si.

Isto é conhecido como o Princípio da Dualidade da álgebra booleana.

A propriedade de Absorção simplifica redundâncias, permitindo converter uma expressão em outra equivalente de menor custo:

$$A + A \cdot B = A \quad (7)$$

$$A \cdot (A + B) = A \quad (8)$$

$$A + \bar{A} \cdot B = A + B \quad (9)$$

$$A \cdot (\bar{A} + B) = A \cdot B \quad (10)$$

No Teorema de De Morgan o complemento de uma expressão qualquer pode ser obtido trocando-se simultaneamente cada variável pelo seu complemento e as operações OR por AND e AND por OR, ou seja:

$$\overline{(A + B)} = \bar{A} \cdot \bar{B} \quad (11)$$

$$\overline{(A \cdot B)} = \bar{A} + \bar{B} \quad (12)$$

3.2 FUNÇÕES BOOLEANAS

Funções booleanas são expressões lógicas cujas entradas são variáveis que assumem valores restritos a 0s e 1s, ou indeterminadas Xs, os chamados *don't care states*. Existem 2^n combinações possíveis para serem consideradas para determinar o valor assumido pela função, onde n representa a quantidade de variáveis. São diversas as formas de representar funções booleanas em programa de computador.

A maneira mais usual é a única a ser utilizada neste trabalho é a maneira tabular. A tabela verdade é um exemplo de representação tabular.

Para $n = 3$, existem 8 combinações possíveis para a saída da função $F(A, B, C) = (A \cdot B) + C$, como é ilustrado na Tabela 5.

Tabela 5 - Tabela Verdade $F(A, B, C) = (A \cdot B) + C$

A	B	C	$(A \cdot B) + C$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Fonte: Elaborado pelo autor

Há diferentes formas de descrever uma função booleana, a função $F(A, B, C) = (A \cdot B) + C$ está escrita em soma de produto, e pode ser reescrita em soma de mintermos, onde são sinalizados os mintermos da função, ou seja, os termos produtos para os quais o resultado da função é igual a 1. Um mintermo é um produto que contém todas as n variáveis, complementadas ou não, de uma função como fatores e assume o valor 1 para somente uma combinação de valores das variáveis.

Um maxtermo funciona da mesma maneira, mas assume o valor 0 para somente uma combinação de valores das variáveis. A representação em soma de mintermos é o formato utilizado para este trabalho.

Reescrevendo a função $F(A, B, C) = (A \cdot B) + C$ como soma de mintermos $F(A, B, C) = \sum m(1, 3, 5, 6, 7)$, ou seja, representando cada mintermo empregando as variáveis e variáveis negadas. Cada representação associada a soma. As variáveis com valor indeterminado, os *don't care states* também podem ser representadas com essa simbologia. Para uma função escrita em soma de mintermos $F(A, B, C) = \sum (m(1, 3, 5, 6, 7) + d(2, 4))$, a tabela verdade correspondente está na Tabela 6.

Tabela 6 - Tabela Verdade $F(A, B, C) = \sum (m(1, 3, 5, 6, 7) + d(2, 4))$

A	B	C	$(F(A, B, C))$
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	1
1	0	0	X
1	0	1	1
1	1	0	1
1	1	1	1

Fonte: Elaborado pelo autor

3.3 CRITÉRIO DE CUSTO

A simplificação de funções booleanas busca minimizar o custo de implementação do circuito digital. O critério da função de custo adotado neste trabalho é a contagem dos terminais de entrada das portas lógicas envolvidas na realização da função, sem contar os inversores na expressão escrita em soma de produtos (SILVA, 1989).

3.4 MAPA DE KARNAUGH

O Mapa de Karnaugh, desenvolvido por Maurice Karnaugh em meados de 1953 (KARNAUGH, 1953), foi criado com o objetivo de facilitar as simplificações de funções booleanas, reduzindo a complexidade e permitindo encontrar de forma mais simples uma função lógica, minimizando a quantidade de portas lógicas e variáveis utilizadas para a implementação de um circuito digital.

Trata-se de um método de muita popularidade, principalmente devido a sua didática para fins educacionais. É válido ressaltar que o mapa de Karnaugh depende da experiência e habilidade do usuário para escolher agrupamentos ótimos.

É um método tradicional que se baseia na identificação visual, e não algébrica, de grupos de mintermos, grupos de maxtermos, ou *don't care states*, que podem ser representados em um arranjo de células de uma tabela verdade, já que contém a representação de todos os termos da função em questão.

As células são posicionadas de forma que somente uma variável é alterada entre células adjacentes do Mapa, permitindo a aplicação da propriedade que deriva da lei distributiva $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$, e do teorema de complemento, $A + \bar{A} = 1$, resultando em $AB + A\bar{B} = A(B + \bar{B}) = A$.

A função simplificada a partir deste método pode ser representada como uma *SoP* (*sum of products*) ou *PoS* (*product of sums*). Na identificação de grupos, se escrito como uma soma de produtos deverão ser considerados os mintermos (1s) e *don't cares* e se escrito como um produto de somas deverão ser considerados os mintermos (0s) e *don't cares*, resultando assim nas combinações simplificadas das variáveis.

A formação de um mapa de Karnaugh trata-se de uma matriz com 2^n células, sendo n o número de variáveis de entradas do circuito. Cada uma dessas células está associada a uma das três opções: mintermos, maxtermos ou *don't care*.

Nas figuras 4, 5 e 6 é ilustrado como é feita a estruturação de mapas de Karnaugh considerando *inputs* de 2, 3, 4 e 5 variáveis, respectivamente.

Figura 4 - Mapa de Karnaugh 2 variáveis.

		B	
		0	1
A	0	0	1
	1	2	3

Fonte: Elaborado pelo autor.

Figura 5 - Mapa de Karnaugh: (a) 3 variáveis e (b) 4 variáveis.

AB		C	
		0	1
AB	00	0	1
	01	2	3
	11	6	7
	10	4	5

(a)

AB		CD			
		00	01	11	10
AB	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

(b)

Fonte: Elaborado pelo autor.

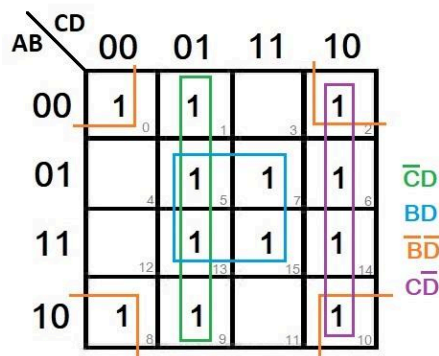
Figura 6 - Mapa de Karnaugh 5 variáveis.

E = 0						E = 1					
AB		CD				AB		CD			
		00	01	11	10			00	01	11	10
AB	00	0	1	3	2	AB		0	1	3	2
	01	4	5	7	6			4	5	7	6
	11	12	13	15	14			12	13	15	14
	10	8	9	11	10			8	9	11	10

Fonte: Elaborado pelo autor.

Todas as células correspondem a um mintermo, as colunas adjacentes se diferenciam por só um literal deixando mais clara a variável que pode ser simplificada. As células da coluna na extremidade esquerda são adjacentes às da coluna na extrema direita, da mesma forma que as células da linha inferior são adjacentes das células da linha superior. Está exemplificado um processo com $n = 4$ para a função $F(A, B, C, D) = \sum m(0, 1, 2, 5, 6, 7, 8, 9, 10, 13, 14, 15)$ na Figura 7.

Figura 7 - Mapa de Karnaugh $F(A, B, C, D) = \sum m(0, 1, 2, 5, 6, 7, 8, 9, 10, 13, 14, 15)$ - Solução não mínima.



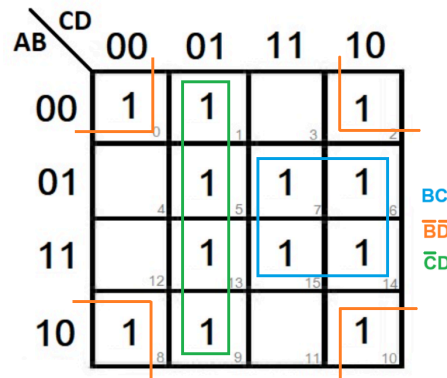
Fonte: Elaborado pelo autor

Nos termos produtos que forma um agrupamento, as variáveis que assumem valor o 1 em um termo e o valor 0 no outro termo, podem ser eliminadas. Por outro lado, as variáveis que permanecem constantes em todas as células do agrupamento serão mantidas na expressão geral. Sendo assim, a função resultante minimizada por mintermos é $F(A, B, C, D) = \overline{C} \cdot D + B \cdot D + \overline{B} \cdot \overline{D} + C \cdot \overline{D}$ e de custo igual a 12, sendo 4 AND de 2 entradas e uma OR de 4 entradas.

Como a minimização via mapa de Karnaugh depende da experiência e habilidade do usuário para escolher agrupamentos ótimos, foi escolhido de maneira não ótima o agrupamento da Figura 7 para ilustrar um exemplo que pode passar despercebido mesmo sendo não ótimo.

No mapa de Karnaugh ilustrado na Figura 8 foi feito um agrupamento ótimo o qual retorna a solução mínima.

Figura 8 - Mapa de Karnaugh $F(A, B, C, D) = \sum m(0, 1, 2, 5, 6, 7, 8, 9, 10, 13, 14, 15)$ - Solução mínima.



Fonte: Elaborado pelo autor

A função minimizada por mintermos é $F(A, B, C, D) = \overline{C} \cdot D + B \cdot C + \overline{B} \cdot \overline{D}$ e de custo igual a 9, sendo 3 AND de duas entradas e uma OR de 3 entradas.

O método do Mapa de Karnaugh tem aplicação recomendada para casos de até 6 variáveis, acima desse número a abordagem acaba se tornando inviável. Assim, outras técnicas são indicadas para a minimização das funções que possuem um número maior de variáveis.

3.5 MÉTODO DE QUINE-MCCLUSKEY

O método de Quine-McCluskey, um método em formato tabular, que sempre garante a obtenção da solução de custo mínimo, consiste em aplicar repetidamente uma propriedade que deriva da lei distributiva $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$, e do teorema de complemento, $A + \overline{A} = 1$, resultando em $A \cdot B + A \cdot \overline{B} = A \cdot (B + \overline{B}) = A$ (KOHAVI, 1970).

O método é dividido em duas etapas, a etapa de geração de todos os implicants primos e a etapa de cobertura dos mintermos da função empregando os implicants primos gerados na etapa de geração. É aplicado como exemplo o método de Quine-McCluskey na função $F(A, B, C, D) = \sum m(0, 1, 2, 5, 6, 7, 8, 9, 10, 13, 14, 15)$.

A etapa geração dos implicants primos consiste em organizar os mintermos e *don't care states* em caixas de índices correspondentes à quantidade de 1s presentes na representação binária do mintermo. Esse conjunto de caixas compõe um grupo, o

primeiro grupo é chamado de Grupo 0. Após a organização em caixas é feita a comparação de cada mintermo de uma caixa com todos os mintermos de uma caixa adjacente.

Um mintermo na caixa de índice i é pareável com um mintermo na caixa adjacente $i + 1$ quando diferem em um e apenas um dígito na sua representação binária, ou seja, tem sua distância de Hamming igual a 1. O bit de pareamento é reescrito com um 'X', e assim gerando um implicante. Os termos combinados devem ser marcados com “*” e reorganizados em um novo grupo, no caso inicial Grupo 1, da mesma maneira que o grupo anterior, onde os termos são dispostos em caixas de índices correspondentes à quantidade de 1s presentes na representação binária dos termos, assim como ilustrado na Figura 9.

Figura 9 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2

Grupo 0			Grupo 1		
Caixa	Mintermos	Mintermos em Binário	Caixa	Mintermo	Mintermo em Binário
=====			=====		
0:			0:		
	0	0000 *		0,1	000X *
-----				0,2	00X0 *
1:				0,8	X000 *
	1	0001 *	-----		
	2	0010 *	1:		
	8	1000 *		1,5	0X01 *
-----				1,9	X001 *
2:				2,6	0X10 *
	5	0101 *		2,10	X010 *
	6	0110 *		8,9	100X *
	9	1001 *		8,10	10X0 *
	10	1010 *	-----		
-----			2:		
3:				5,7	01X1 *
	7	0111 *		5,13	X101 *
	13	1101 *		6,7	011X *
	14	1110 *		6,14	X110 *
-----				9,13	1X01 *
4:				10,14	1X10 *
	15	1111 *	-----		
-----			3:		
				7,15	X111 *
				13,15	11X1 *
				14,15	111X *
-----			-----		

(a)

(b)

Grupo 2		
Caixa	Mintermo	Mintermo em Binário
=====		
0:		
	0,1,8,9	X00X
	0,2,8,10	X0X0

1:		
	1,5,9,13	XX01
	2,6,10,14	XX10

2:		
	5,7,13,15	X1X1
	6,7,14,15	X11X

Implicantes Primos da tabela: XX01, X1X1, XX10, X00X, X0X0, X11X		

(c)

Fonte: Elaborado pelo autor.

Termos que não forem pareados, ou seja, os termos não marcados, são considerados implicantes primos, pois, são os únicos termos capazes de possibilitar a cobertura de algum ou de alguns mintermos presentes. É repetida a etapa de combinação até que não seja possível realizar mais combinações dos implicantes.

A segunda etapa faz a cobertura dos mintermos da função empregando os implicantes primos gerados na primeira etapa. É formada uma tabela com os mintermos originais dispostos em colunas ignorando os *don't cares* e os implicantes primos dispostos em linhas como feito na Figura 10, Todos os mintermos que cada implicante primo cobre deve ser marcado. Caso um mintermo seja coberto por apenas um implicante primo, esse implicante primo é denominado um implicante primo essencial e deve constar, obrigatoriamente, na solução mínima.

Figura 10 - Cobertura dos Implicantes Primos

Mintermos		0	1	2	5	6	7	8	9	10	13	14	15
=====													
0,1,8,9		x	x						x	x			

5,7,13,15					x		x				x		x

6,7,14,15						x	x					x	x

0,2,8,10		x		x					x		x		

1,5,9,13			x		x					x		x	

2,6,10,14				x		x					x		x

Fonte: Elaborado pelo autor.

Assim, a função de custo mínimo é $F(A, B, C, D) = \overline{C} \cdot D + B \cdot C + \overline{B} \cdot \overline{D}$, sendo os respectivos implicantes, $\overline{B} \cdot \overline{D} = X0X0 (0, 2, 8, 10)$, $\overline{C} \cdot D = XX01(1, 5, 9, 13)$ e $B \cdot C = X11X(6, 7, 14, 15)$. O custo da função minimizada é igual a 9, sendo 3 AND de duas entradas e uma OR de 3 entradas.

3.6 PROGRAMA ESPRESSO

O programa ESPRESSO-II (BRAYTON, 1984), um aprimoramento do programa ESPRESSO-I (BRAYTON, 1982), referenciado neste trabalho apenas como programa ESPRESSO, consiste num programa de abordagem heurística que minimiza funções booleanas a dois níveis e tenta encontrar a solução de menor custo de implementação em um circuito. Pode minimizar diversas funções ao mesmo tempo e aceita *don't cares* como entradas. A solução mínima não é garantida por se tratar de uma heurística.

É inicialmente aplicado nas entradas um procedimento auxiliar, o *UNWRAP*, que converte expressões possivelmente em lógica de multinível em expressões de lógica a dois níveis, *SoP* ou *PoS* (FERNANDES, 1991). O programa ESPRESSO executa 8 operações, compondo uma coleção de heurísticas. A primeira operação do programa é chamada *Complement*, onde se computam os mintermos de entrada, em formato de *OFF-set*. Tomando como exemplo a função $F(A, B, C, D) = \sum m(0, 1, 2, 4, 5, 6, 9, 10, 11, 13, 14, 15)$, em soma de mintermos, ilustrada na Figura 11, sua representação em formato *OFF-set* é $OFF(F) = \{(0,0,1,1), (0,1,1,1), (1,0,0,0), (1,1,0,0)\}$, onde os mintermos $OFF(F) = \{3,7,8,12\}$ de $F(A, B, C, D)$ são iguais a 0.

Figura 11 - Função $F(A, B, C, D) = \sum m(0, 1, 2, 4, 5, 6, 9, 10, 11, 13, 14, 15)$.

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01	1 ₄	1 ₅		1 ₆
11		1 ₁₂	1 ₁₃	1 ₁₄
10		1 ₈	1 ₉	1 ₁₀

Fonte: Elaborado pelo autor

A segunda operação do programa é chamada de *Expand* na qual os implicantes são expandidos o máximo possível de modo a se tornarem primos, ou seja, não existe combinação que simplifica mais esse implicante, como ilustrado na Figura 12.

Figura 12 - Operação *Expand*.

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01	1 ₄	1 ₅		1 ₆
11		1 ₁₂	1 ₁₃	1 ₁₄
10		1 ₈	1 ₉	1 ₁₀

Fonte: Elaborado pelo autor

A terceira operação é chamada *Essential Primes* na qual os implicantes primos essenciais para a cobertura são extraídos nessa fase inicial, deixando o problema menor e colocados num set de cobertura de *don't cares* de modo a evitar que esses implicantes acabem sofrendo alterações durante a execução. A quarta operação ilustrada na Figura 13 é chamada *IrredundantCover*, onde são removidos implicantes primos que tem todos seus termos cobertos por outros implicantes primos, mantendo a cobertura mínima mesmo que local, uma vez que nenhum outro implicante pode ser removido sem quebrar a cobertura.

Figura 13 - Operação *IrredundantCover*.

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01	1 ₄	1 ₅		1 ₆
11		1 ₁₂	1 ₁₃	1 ₁₄
10		1 ₈	1 ₉	1 ₁₀

REDUNDANT

Fonte: Elaborado pelo autor

A quinta operação, chamada *Reduce*, reduz cada implicante para implicante essencial mínimo, sem permitir que dois implicantes primos estejam sobrepostos mas deixar que a cobertura seja quebrada, como demonstrado na Figura 14.

Figura 14 - Operação *Reduce*.

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01	1 ₄	1 ₅		1 ₆
11		1 ₁₃	1 ₁₅	1 ₁₄
10		1 ₉	1 ₁₁	1 ₁₀

Fonte: Elaborado pelo autor

Esta solução é visivelmente pior, ou seja, de maior custo de implementação, entretanto ao reduzir os implicantes estes ficam disponíveis para expandir para um maior número de direções e possivelmente uma nova e melhor solução. Na sexta operação *Iterate* é feita iteração das operações *Expand IrredundantCover* e *Reduce* até não haver mais melhora, como mostra a Figura 15.

Figura 15 - Operação *Iterate*: (a) *Expand 2* e (b) *IrredundantCover 2*.

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01	1 ₄	1 ₅		1 ₆
11		1 ₁₃	1 ₁₅	1 ₁₄
10		1 ₉	1 ₁₁	1 ₁₀

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01	1 ₄	1 ₅		1 ₆
11		1 ₁₃	1 ₁₅	1 ₁₄
10		1 ₉	1 ₁₁	1 ₁₀

Fonte: Elaborado pelo autor

Após a etapa *Reduce 2* da operação *Iterate* remover o implicante redundante da

solução, todos seus termos passam a ser cobertos com outro formato, um *Reshaping* como na Figura 16, por outros implicantes primos e de forma irredundante, mantendo a cobertura mínima mesmo que local, uma vez que nenhum outro implicante pode ser removido para minimizar sem quebrar a cobertura.

Figura 16 - Solução *Reshaping*.

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01	1 ₄	1 ₅		1 ₆
11		1 ₁₂	1 ₁₃	1 ₁₄
10		1 ₈	1 ₉	1 ₁₀

Fonte: Elaborado pelo autor

Neste caso a solução de menor custo foi obtida, custo igual a 9, sendo 3 AND de duas entradas e uma OR de 3 entradas, $F(A, B, C, D) = \overline{A} \cdot \overline{C} + A \cdot D + C \cdot \overline{D}$, porém a produção da melhor solução não é sempre garantida. É sempre garantida uma solução mínima, mesmo que local, irredundante e de implicantes primos.

A Sétima operação, *Lastgasp*, uma heurística que procura reduções na função, tenta executar *Reduce Expand* e *IrredundantCover* mais uma vez com uma união lógica dos implicantes primos antigos com os implicantes primos novos. Caso dê certo, a iteração deve continuar, a redução dos implicantes é realizada individualmente, não mais em sequência e independente da ordenação, com isso aumenta o tempo de execução, especialmente para funções de muitas variáveis.

A oitava operação, *Makesparse*, busca minimizar expressão booleana deixando-a mais escassa possível heurísticamente, ou seja, reduzir a sobreposição entre implicantes. Este programa funciona de maneira muito rápida e robusta mesmo com diversas etapas heurísticas complexas, minimizando funções booleanas de centenas de variáveis de entrada. Na Tabela 7 apresenta-se o percentual de tempo que o programa Espresso gasta por operação (RUTENBAR, 2013).

Tabela 7 - Gasto de tempo do programa Espresso por etapa.

Operação	% do Tempo de Execução
<i>Complement</i>	14%
<i>Expand</i>	29%
<i>Essential primes</i>	13%
<i>IrredundantCover</i>	12%
<i>Reduce</i>	8%
<i>Special case optimizations (Iterate, LastGasp e Makesparse)</i>	24%

Fonte: (RUTENBAR, 2013).

4 DESENVOLVIMENTO

Neste trabalho é desenvolvido um método heurístico alternativo para a simplificação de funções booleanas baseado no método de Quine-McCluskey tradicional, que reduz a complexidade computacional média do algoritmo aplicado.

4.1 MÉTODO DE QUINE-MCCLUSKEY MODIFICADO

A modificação consiste numa diferente abordagem da primeira etapa do método de Quine-McCluskey, a geração de implicantes primos. A etapa geração dos implicantes primos consiste em organizar os mintermos e *don't care states* em caixas de índices correspondentes à quantidade de 1s presentes na representação binária do termo. Esse conjunto de caixas compõe um grupo, o primeiro grupo é chamado de Grupo 0. Após a organização em caixas é feita a comparação de cada mintermo de uma caixa com os mintermos de caixas adjacentes.

A comparação entre as caixas do grupo inicial, o grupo 0, é feita de maneira a cobrir todos os mintermos apresentando o mínimo de redundâncias nos implicantes gerados. É feita a comparação de um mintermo numa caixa de índice i até que este encontre um par na caixa adjacente $i + 1$ não marcado, reescrevendo o bit de pareamento como um "X", e assim gerando um implicante.

Ambos os mintermos pareados são marcados com " * " para não serem comparados novamente com outros mintermos nessa etapa. Caso um mintermo não tenha sido pareado é executada uma operação denominada de RESGATE. A operação de RESGATE ocorre apenas no grupo 0, com um mintermo não marcado de uma caixa i que após ter sido comparado com os mintermos não marcados da caixa adjacente $i + 1$ não tenha sido pareado. A partir disso é feita uma comparação forçada com os termos marcados da caixa $i - 1$.

Caso o mintermo a ser resgatado ainda não tenha sido pareado é feita uma última comparação forçada com os termos marcados da caixa $i + 1$. Caso o termo ainda não tenha sido pareado é então considerado como um implicante primo da função. Os implicantes gerados são reorganizados em um novo grupo da mesma maneira que o grupo inicial, onde os termos são dispostos em caixas de índices

correspondentes à quantidade de 1s presentes em sua representação binária.

Com isso o Grupo 1 é formado com o mínimo de redundância possível, em casos que não necessitam passar pela etapa de RESGATE, a forma obtida ao final do Grupo 0 é irredundante. Após a formação do grupo 1, o método segue os passos do método de Quine-McCluskey tradicional até o final, a cobertura dos mintermos, de modo a buscar uma solução além de irredundante, prima e mínima, não marcando mais os termos combinados e não caindo na operação de Resgate.

4.1.1 DESCRIÇÃO DO ALGORITMO

PASSO 1. Entrada dos Mintermos/ Don't Cares.

PASSO 2. São declaradas e atribuídas zero às variáveis j e i . Cria-se um grupo de índice j e neste classificam-se os mintermos em caixas de índice i .

PASSO 3. Comparam-se os mintermos não marcados da caixa[i] com os mintermos não marcados da caixa[$i+1$], mintermos não pareados neste passo prosseguem para o PASSO 4(RESGATE), após pareamento executa-se o PASSO 5.

PASSO 4. RESGATE - Compara-se o mintermo não marcado da caixa[i] com os termos marcados da caixa[$i-1$](se $i>0$), caso o mintermo da caixa[i] continue sem parar é comparado com os termos marcados da caixa[$i+1$].

PASSO 5. Marcam-se termos pareados como combinados.

PASSO 6. Marcam-se termos não pareados como implicantes primos.

PASSO 7. Se a caixa[i] é a penúltima caixa, executa-se o PASSO 8, senão executa-se o PASSO 9.

PASSO 8. Soma-se 1(um) à j e atribui-se 0(zero) à i . Cria-se grupo de índice j e neste classificam-se os implicantes em caixas índice i . Executa-se o PASSO 12.

PASSO 9. Soma-se 1(um) à i .

PASSO 10. Se j for igual a 0(zero) executa-se o PASSO 3, senão executa-se o PASSO 11.

PASSO 11. Comparam-se todos os implicantes da caixa[i] com todos os implicantes da caixa[$i+1$] e retorna-se o PASSO 5.

PASSO 12. Se houver combinações possíveis no novo grupo executa-se o PASSO 10, senão executa-se o PASSO 13.

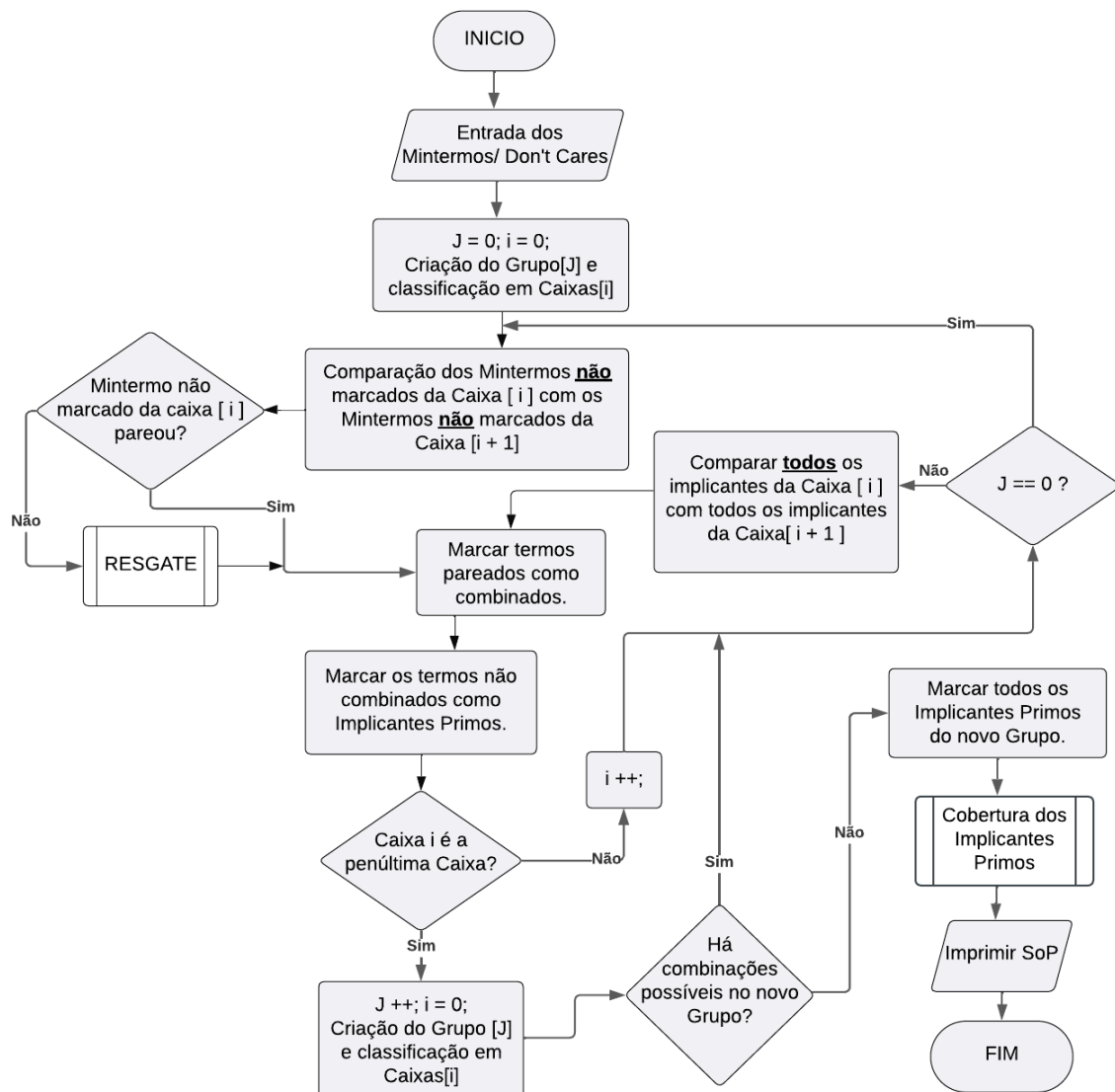
PASSO 13. Cobertura dos Implicantes Primos.

PASSO 14. Imprimir SoP.

4.1.2 FLUXOGRAMA DO ALGORITMO

Na Figura 17 apresenta-se o fluxograma do algoritmo:

Figura 17 - Fluxograma do método de Quine-McCluskey modificado.



Fonte: Elaborado pelo autor.

4.2 EXEMPLO EM MAPA DE KARNAUGH

Aplicando o método proposto na função $F(A, B, C, D) = \sum m(0, 1, 2, 5, 6, 7, 8, 9, 10, 13, 14, 15)$, tem-se o grupo 0 e seu Mapa de Karnaugh correspondente apresentados na Figura 18.

Figura 18 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 0 .

AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01		1 ₄	1 ₅	1 ₆
11		1 ₁₂	1 ₁₃	1 ₁₄
10	1 ₈	1 ₉		1 ₁₀

(a)

Fonte: Elaborado pelo autor

É obtida nesse caso, ao final do grupo 0 uma forma irredundante de implicantes que não passa pela etapa de Resgate. Os implicantes gerados formam o grupo 1 e seu Mapa de Karnaugh apresenta-se na Figura 19.

Figura 19 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 1.

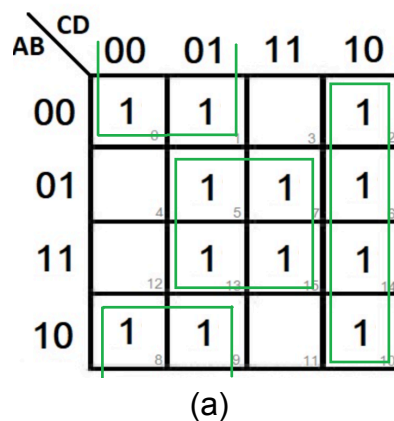
AB \ CD	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01		1 ₄	1 ₅	1 ₆
11		1 ₁₂	1 ₁₃	1 ₁₄
10	1 ₈	1 ₉		1 ₁₀

(a)

Fonte: Elaborado pelo autor

A partir dos implicantes gerados formam o grupo 2 e seu Mapa de Karnaugh apresenta-se na Figura 20.

Figura 20 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 2.



Fonte: Elaborado pelo autor.

A partir dos implicantes primos gerados é feita a etapa de cobertura dos mintermos da função na Figura 21.

Figura 21 - Cobertura dos Implicantes Primos.

Mintermos	0	1	2	5	6	7	8	9	10	13	14	15
2,6,10,14			x		x				x		x	
5,7,13,15				x		x				x		x
0,1,8,9	x	x					x	x				

Fonte: Elaborado pelo autor.

Assim, a função minimizada é $F(A, B, C, D) = C \cdot \overline{D} + B \cdot D + \overline{B} \cdot \overline{C}$, sendo os respectivos implicantes, $\overline{B} \cdot \overline{C} = X00X$ (0, 1, 8, 9), $C \cdot \overline{D} = XX10$ (2, 6, 10, 14) e $BC = X1X1$ (5, 7, 13, 15). O custo da função minimizada é igual a 9, sendo 3 AND de duas entradas e uma OR de 3 entradas.

5 RESULTADOS E DISCUSSÕES

Foi utilizado um computador (*Notebook*) para implementar o algoritmo que em busca de uma heurística, modifica fazendo uma redução de passo do método de minimização de funções booleanas de Quine-McCluskey (QUINE, 1956), em linguagem Python.

5.1 A LINGUAGEM PYTHON

A linguagem de programação Python foi criada em 1991 por Guido van Rossum é reconhecida por sua simplicidade, flexibilidade e aplicabilidade, em campos como engenharia, ciência de dados e automação (ROSSUM; DRAKE, 2001). Trata-se de uma linguagem interpretada, o que facilita o processo de *debugging*, é de alto nível, ou seja, sintaxe intuitiva e multiparadigma, que combina elementos de programação estruturada, funcional e orientada a objetos. Possibilita a implementação de soluções computacionais de forma eficiente, com menor complexidade em comparação a linguagens tradicionais, como a linguagem C, além de ter compatibilidade com os principais sistemas operacionais.

5.1.1 O IDE PYCHARM

O algoritmo foi implementado no IDE (*Integrated Development Environment*) *Pycharm Community Edition* 2023. O ambiente Pycharm é um IDE gratuito e de código aberto desenvolvido pela JetBrains, apresenta desde uma assistência de codificação a debugger e estruturas de teste, todas as ferramentas são integradas com o intuito de agilizar os pequenos detalhes de desenvolver programas.

A heurística consiste numa uma redução de passo do método de minimização de funções booleanas de Quine-McCluskey (QUINE, 1956).

5.1.2 O IDE JUPYTER

Foi utilizado o IDE, ambiente de desenvolvimento Jupyter Notebook,

pertencente ao pacote Anaconda, para as simulações do programa Espresso, via biblioteca PyEDA, em um ambiente interativo para experimentação (JUPYTER, 2024). O Jupyter Notebook é uma aplicação web interativa que permite a execução célula a célula de código em diversas linguagens de programação, sendo uma delas a linguagem Python. Cada célula pode conter código, equações matemáticas ou gráficos, e permite documentar todo o processo computacional em um único ambiente.

5.1.3 BIBLIOTECAS UTILIZADAS

Para as simulações foram importadas 4 poderosas bibliotecas que simplificaram a confecção e geração de testes do algoritmo, as bibliotecas **time**, **psutil**, **os** e **random**.

A biblioteca **time** fornece funções relacionadas ao tempo, foi utilizada para monitorar o tempo de execução do programa, métrica importante para avaliação do desempenho do algoritmo.

A biblioteca **psutil** possui funções que trazem informações sobre a utilização do sistemas, como o uso de memória. A biblioteca **os** fornece funções que possibilitam interação com funcionalidades que dependem do sistema operacional, assim as bibliotecas **psutil** e **os** foram utilizadas para medir o uso de memória de trechos de código, no caso a geração de implicantes primos dos algoritmos. O uso de memória foi utilizado como métrica de desempenho do algoritmo.

A biblioteca **random** foi utilizada para gerar entradas aleatórias de maneira controlada como, por exemplo, quantas mintermos devem ser gerados e qual a faixa de valores esses mintermos devem pertencer, a depender da quantidade de variáveis de entrada da função booleana problema dada como input (DRAKE, 2012).

5.2 TESTES “TUDO UM”

Com o objetivo de validar o programa do método de Quine-McCluskey modificado foi realizada uma série de testes. Executaram-se os programas para diversos casos, onde foi possível explorar as vantagens e desvantagens do método

proposto. Executou-se o programa para o pior caso, o caso “Tudo um”, onde para n variáveis de entrada há 2^n mintermos.

5.2.1 TEMPO DE EXECUÇÃO

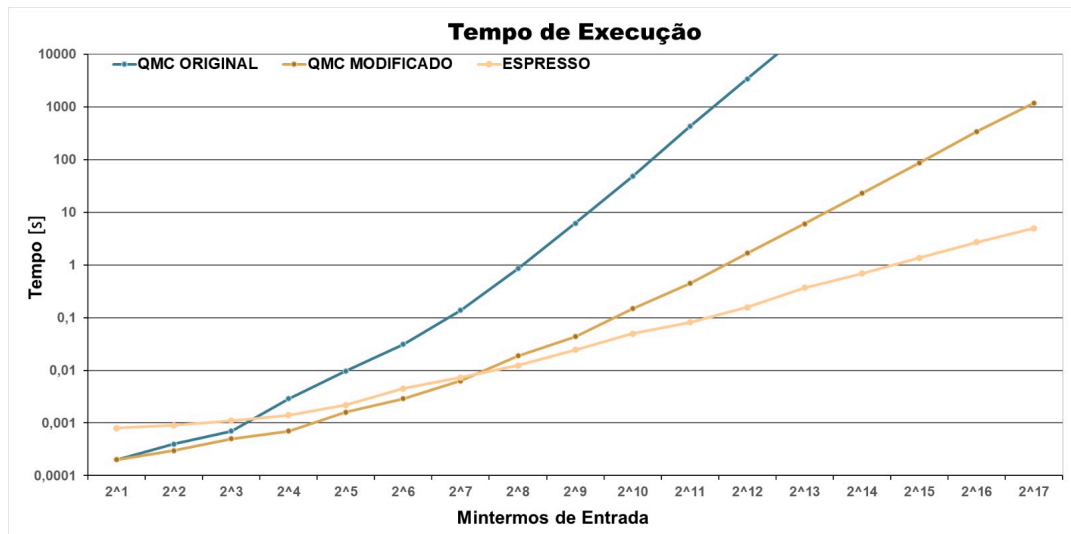
Foi monitorado o tempo de execução do método de Quine-McCluskey tradicional, do programa Espresso e do método de Quine-McCluskey modificado, os valores obtidos estão na Tabela 8. Apresenta-se a plotagem dos dados da Tabela 8 na Figura 22.

Tabela 8 - Tempo de execução dos métodos implementados (Tudo um).

Variáveis de Entrada	Método de Quine-McCluskey [s]	Método de Quine-McCluskey modificado [s]	Programa ESPRESSO [s]
1	0,0002	0,0002	0,0008
2	0,0004	0,0003	0,0009
3	0,0007	0,0005	0,0011
4	0,0029	0,0007	0,0014
5	0,0097	0,0016	0,0022
6	0,0312	0,0029	0,0045
7	0,1377	0,0063	0,0073
8	0,8533	0,0187	0,0123
9	6,1652	0,0437	0,0246
10	48,2836	0,1479	0,0495
11	428,3315	0,4456	0,0811
12	3428,0781	1,6814	0,1566
13	-	6,0229	0,3693
14	-	22,8597	0,6885
15	-	87,0848	1,3715
16	-	342,3945	2,7215
17	-	1191,8167	4,9726

Fonte: Elaborado pelo autor

Figura 22 - Plotagem dos tempos de execução dos métodos (Tudo um).



Fonte: Elaborado pelo autor

A partir de 12 variáveis tornou-se inviável utilizar o Quine-McCluskey tradicional, que apresentou tempo de execução superior a 1 hora.

Nota-se que o método de Quine-McCluskey modificado foi o mais rápido para solucionar problemas com até 7 variáveis.

A partir de 7 variáveis o programa Espresso foi o mais rápido para o caso particular de funções contendo todos os mintermos assumindo valor lógico 1, caso denominado “Tudo um”.

Essa redução no tempo de execução é devida a modificação no método de Quine-McCluskey, na fase de geração de implicantes com o grupo 0, onde acabam sendo gerados muito menos implicantes.

5.2.2 IMPLICANTES GERADOS NA FASE MODIFICADA

A modificação proposta na fase de geração de implicantes primos, ocasiona em menos implicantes sendo gerados, como demonstra-se na Tabela 9.

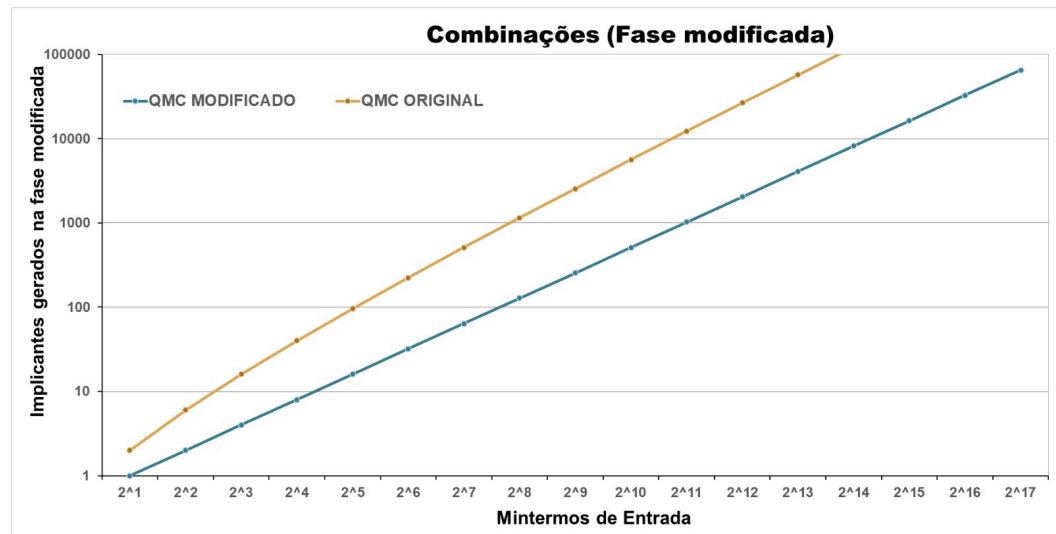
Tabela 9 - Implicantes gerados na fase modificada dos métodos de Quine-McCluskey e método de Quine McCluskey modificado (Tudo um).

Variáveis de Entrada	Método de Quine-McCluskey [s]	Método de Quine-McCluskey modificado [s]
1	1	1
2	4	2
3	12	4
4	32	8
5	80	16
6	192	32
7	448	64
8	1024	128
9	2304	256
10	5120	512
11	11264	1024
12	24576	2048
13	53248	4096
14	114688	8192
15	245760	16384
16	-	32768

Fonte: Elaborado pelo autor

A plotagem dos dados da Tabela 9 encontra-se na Figura 23.

Figura 23 - Plotagem de Implicantes gerados na fase modificada (Tudo um).



Fonte: Elaborado pelo autor

5.2.3 USO DE MEMÓRIA

Foram observados valores diferentes referentes ao uso de memória na execução dos programas do método de Quine-McCluskey tradicional e do método de Quine-McCluskey modificado, como apresenta-se na Tabela 10 e na plotagem dos dados da Tabela 10, que apresenta-se na Figura 24.

Tabela 10 - Uso de memória em Megabytes na fase modificada dos métodos de Quine-McCluskey e método de Quine McCluskey modificado (Tudo um).

(continua)

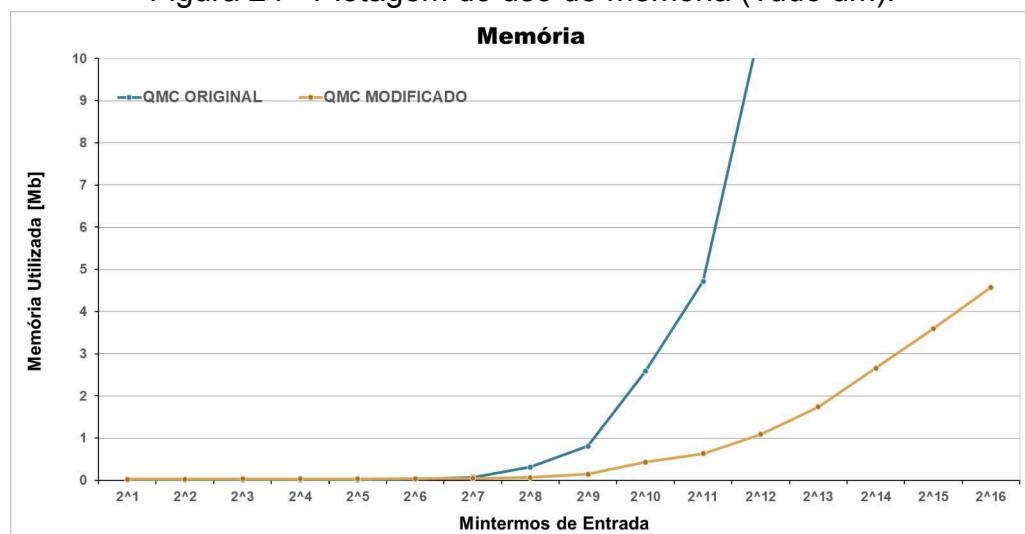
Variáveis de Entrada	Método de Quine-McCluskey [MB]	Método de Quine-McCluskey modificado [MB]
1	0,004	0,027
2	0,004	0,028
3	0,004	0,031
4	0,008	0,033
5	0,012	0,035
6	0,027	0,039

(conclusão)

Variáveis de Entrada	Método de Quine-McCluskey [MB]	Método de Quine-McCluskey modificado [MB]
7	0,070	0,047
8	0,312	0,066
9	0,811	0,141
10	2,591	0,43
11	4,719	0,633
12	10,754	1,088
13	24,506	1,742
14	55,846	2,656
15	127,265	3,599
16	290,018	4,575

Fonte: Elaborado pelo autor

Figura 24 - Plotagem do uso de memória (Tudo um).



Fonte: Elaborado pelo autor

Notou-se inviável a execução do Quine-McCluskey tradicional para muitas variáveis. Salienta-se que o caso denominado Tudo um representa o melhor caso para o algoritmo desenvolvido e o pior caso para o método de Quine-McCluskey tradicional.

5.3 OUTROS CASOS TESTADOS

De forma a medir o custo das soluções geradas, foram realizados testes com diversas funções booleanas.

5.3.1 TESTES EM QUE OBTEVE-SE O CUSTO MÍNIMO

5.3.1.1 $F(A, B, C, D) = \sum m(0, 2, 5, 6, 10, 11)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(0, 2, 5, 6, 10, 11)$, geram-se os grupos apresentados na Figura 25.

Figura 25 - Geração de Implicantes Primos: (a) Grupo 0 e (b) Grupo 1 - (F11).

Caixa.	Mintermos	Mintermos em Bin	Caixa	Mintermos	Mintermos em Binário
0:			0:		
	0	0000		0,2	00X0
1:			1:		
	2	0010		2,6	0X10
2:			2:		
	5	0101		10,11	101X
	6	0110			
	10	1010			
3:					
	11	1011			
Implicantes Primos da tabela: 0101			Implicantes Primos da tabela: 00X0, 101X, 0X10		

(a)

(b)

Fonte: Elaborado pelo autor

Apresenta-se a cobertura dos implicantes primos na Figura 26.

Figura 26 - Cobertura dos Implicantes Primos - (F11).

Mintermos		0	2	5	6	10	11
10,11						x	x
5				x			
0,2		x	x				
2,6			x		x		

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução $S = \overline{A} \cdot \overline{B} \cdot \overline{D} + A \cdot \overline{B} \cdot C + \overline{A} \cdot C \cdot \overline{D} + \overline{A} \cdot B \cdot \overline{C} \cdot D$, correspondendo a um custo igual a 17, sendo 3 AND de três entradas, 1 AND de quatro entradas e 1 OR de quatro entradas, que é o custo mínimo para a função.

5.3.1.2 $F(A, B, C, D) = \sum m(1, 2, 5, 6, 9, 10, 13, 14)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(1, 2, 5, 6, 9, 10, 13, 14)$, geram-se os grupos apresentados na Figura 27.

Figura 27 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (F12).

Caixa.	Mintermos	Minterms em Bin	
=====			
1:			
	1	0001	*
	2	0010	*

2:			
	5	0101	*
	6	0110	*
	9	1001	*
	10	1010	*

3:			
	13	1101	*
	14	1110	*

(a)

Caixa	Mintermos	Minterms em Binário	
=====			
0:			
	1,5	0X01	*
	2,6	0X10	*

1:			
	9,13	1X01	*
	10,14	1X10	*

(b)

Caixa	Mintermos	Minterms em Binário
=====		
0:		
	1,5,9,13	XX01
	2,6,10,14	XX10

Implicantes Primos da tabela: XX01, XX10		

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 28.

Figura 28 - Cobertura dos Implicantes Primos - (FI2).

Mintermos		1	2	5	6	9	10	13	14
=====									
1,5,9,13		x		x		x		x	

2,6,10,14			x		x		x		x

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução $S = C \cdot \overline{D} + \overline{C} \cdot D$, correspondendo a um custo igual a 6, sendo 2 AND de duas entradas e 1 OR de duas entradas, que é o custo mínimo para a função.

5.3.1.3 $F(A, B, C, D) = \sum m(1, 2, 3, 6, 9, 10, 11, 12, 14)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(1, 2, 3, 6, 9, 10, 11, 12, 14)$, geram-se os grupos apresentados na Figura 29.

Figura 29 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FI3).

Caixa.	Mintermos	Minterms em Bin	Caixa	Mintermos	Mintermos em Binário
=====					
1:			0:		
	1	0001 *		1,3	00X1 *
	2	0010 *		2,6	0X10 *

2:			1:		
	3	0011 *		9,11	10X1 *
	6	0110 *		10,14	1X10 *
	9	1001 *		12,14	11X0
	10	1010 *			
	12	1100 *			

3:					
	11	1011 *			
	14	1110 *			

(a)

=====					
0:					
	1,3			00X1	*
	2,6			0X10	*

1:					
	9,11			10X1	*
	10,14			1X10	*
	12,14			11X0	

Implicantes Primos da tabela: 11X0

(b)

(a)

(b)

Caixa	Mintermos	Mintermos em Binário
=====		
0:		
	1,3,9,11	X0X1
	2,6,10,14	XX10

Implicantes Primos da tabela: XX10, X0X1		

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 30.

Figura 30 - Cobertura dos Implicantes Primos - (FI3).

Mintermos		1	2	3	6	9	10	11	12	14
=====										
2,6,10,14			x			x		x		x

12,14									x	x

1,3,9,11		x			x		x		x	

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução $S = A \cdot B \cdot \overline{D} + C \cdot \overline{D} + \overline{B} \cdot D$, correspondendo a um custo igual a 10, sendo 1 AND de três entradas, 2 AND de duas entradas e 1 OR de três entradas, que é o custo mínimo para a função.

5.3.1.4 $F(A, B, C, D) = \sum m(0, 1, 2, 4, 5, 6, 9, 10, 11, 13, 14, 15)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(0, 1, 2, 4, 5, 6, 9, 10, 11, 13, 14, 15)$, geram-se os grupos apresentados na Figura 31.

Figura 31 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FI4).

Caixa.

Mintermos

Minterms em Bin

0:

0

0000 *

1:

1

0001 *

2

0010 *

4

0100 *

2:

5

0101 *

6

0110 *

9

1001 *

10

1010 *

3:

11

1011 *

13

1101 *

14

1110 *

4:

15

1111 *

Caixa

Mintermos

Minterms em Binário

0:

0,1

000X *

1:

2,6

0X10 *

4,5

010X *

2:

9,11

10X1 *

10,14

1X10 *

3:

13,15

11X1 *

(a)

(b)

(a)

(b)

Caixa	Mintermos	Mintermos em Binário
0:	0,1,4,5	0X0X
1:	2,6,10,14	XX10
2:	9,11,13,15	1XX1
Implicantes Primos da tabela: 1XX1, XX10, 0X0X		

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 32.

Figura 32 - Cobertura dos Implicantes Primos - (FI4).

Mintermos	0	1	2	4	5	6	9	10	11	13	14	15
9,11,13,15							x		x	x		x
2,6,10,14			x			x		x			x	
0,1,4,5	x	x		x	x							

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução $S = A \cdot D + C \cdot \overline{D} + \overline{A} \cdot \overline{C}$, correspondendo a um custo igual a 9, sendo 3 AND de duas entradas, e 1 OR de três entradas, que é o custo mínimo para a função.

5.3.1.5 $F(A, B, C, D) = \sum m(0, 5, 8, 9, 10, 12, 15, 16, 18, 19, 21, 26, 28, 31)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(0, 5, 8, 9, 10, 12, 15, 16, 18, 19, 21, 26, 28, 31)$, geram-se os grupos apresentados na Figura 33.

Figura 33 - Geração de Implicantes Primos: (a) Grupo 0 e (b) Grupo 1 - (FI5).

Caixa.	Mintermos	Mintermos em Bin	Caixa	Mintermos	Mintermos em Binário
0:			0:		
	0	00000 *		0,8	0X000
1:					
	8	01000 *	1:		
	16	10000 *		16,18	100X0
2:				8,9	0100X
	5	00101 *			
	9	01001 *	2:		
	10	01010 *		5,21	X0101
	12	01100 *		10,26	X1010
	18	10010 *		12,28	X1100
3:				18,19	1001X
	19	10011 *			
	21	10101 *	4:		
	26	11010 *		15,31	X1111
	28	11100 *			
4:					
	15	01111 *			
5:					
	31	11111 *			

(a)

(b)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 34.

Figura 34 - Cobertura dos Implicantes Primos - (FI5).

Mintermos	0	5	8	9	10	12	15	16	18	19	21	26	28	31
15,31								x						x
8,9				x	x									
18,19										x	x			
10,26						x							x	
5,21			x									x		
16,18									x	x				
0,8		x		x										
12,28						x							x	

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução

$$S = B \cdot C \cdot D \cdot E + \bar{A} \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D + B \cdot \bar{C} \cdot D \cdot \bar{E} + \bar{B} \cdot C \cdot \bar{D} \cdot E + A \cdot \bar{B} \cdot \bar{C} \cdot \bar{E} + \bar{A} \cdot \bar{C} \cdot \bar{D} \cdot \bar{E} + B \cdot C \cdot \bar{D} \cdot \bar{E}$$

, correspondendo a um custo igual a 40, sendo 8 AND de quatro entradas, e 1 OR de oito entradas, que é o custo mínimo para a função.

5.3.2 TESTES EM QUE NÃO SE OBTVE O CUSTO MÍNIMO

5.3.2.1 $F(A, B, C, D) = \sum m(2, 4, 6, 10, 12, 14)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(2, 4, 6, 10, 12, 14)$, geram-se os grupos apresentados na Figura 35.

Figura 35 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP1).

Caixa.	Mintermos	Minterms em Bin		Caixa	Mintermos	Mintermos em Binário
1:				0:		
	2	0010 *			2,6	0X10 *
	4	0100 *			4,12	X100
2:				1:		
	6	0110 *			10,14	1X10 *
	10	1010 *				
	12	1100 *				
3:						
	14	1110 *				

Implicantes Primos da tabela: X100

(a) (b)

Caixa	Mintermos	Mintermos em Binário
0:		
	2,6,10,14	XX10

Implicantes Primos da tabela: XX10

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 36.

Figura 36 - Cobertura dos Implicantes Primos - (FP1).

Mintermos		2	4	6	10	12	14
2,6,10,14		x		x	x		x
4,12			x			x	

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução $S = B \cdot \overline{C} \cdot \overline{D} + C \cdot \overline{D}$, correspondendo a um custo igual a 7, sendo 1 AND de três entradas, 1 AND de duas entradas e 1 OR de duas entradas, que não é o custo mínimo para a função. A solução de custo mínimo, via Quine-McCluskey tradicional ou programa Espresso é $S = B \cdot \overline{D} + C \cdot \overline{D}$, correspondendo a um custo igual a 6, sendo 2 AND de duas entradas e 1 OR de duas entradas.

5.3.2.2 $F(A, B, C, D) = \sum m(3, 8, 9, 10, 11, 12, 14)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(3, 8, 9, 10, 11, 12, 14)$, geram-se os grupos apresentados na Figura 37.

Figura 37 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP2).

Caixa.	Mintermos	Minterms em Bin	
1:	8	1000	*
2:	3	0011	*
	9	1001	*
	10	1010	*
	12	1100	*
3:	11	1011	*
	14	1110	*

(a)

Caixa	Mintermos	Mintermos em Binário	
0:	8,9	100X	
	8,12	1X00	*
1:	3,11	X011	
	10,14	1X10	*

Implicantes Primos da tabela: X011, 100X

(b)

Caixa	Mintermos	Mintermos em Binário
0:	8,10,12,14	1XX0

Implicantes Primos da tabela: 1XX0

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos apresenta-se a cobertura dos mintermos na Figura 38.

Figura 38 - Cobertura dos Implicantes Primos - (FP2).

Mintermos		3	8	9	10	11	12	14
=====								
8,10,12,14			x		x		x	x

3,11		x				x		

8,9			x	x				

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução $S = A \cdot \overline{B} \cdot \overline{C} + \overline{B} \cdot C \cdot D + A \cdot \overline{D}$, correspondendo a um custo igual a 11, sendo 2 AND de três entradas, 1 AND de duas entradas e 1 OR de três entradas, que não é o custo mínimo para a função. A solução de custo mínimo, via Quine-McCluskey tradicional ou programa Espresso é $S = A \cdot \overline{B} + \overline{B} \cdot C \cdot D + A \cdot \overline{D}$, correspondendo a um custo igual a 10, sendo 1 AND de três entradas, 2 AND de duas entradas e 1 OR de três entradas.

5.3.2.3 $F(A, B, C, D) = \sum m(0, 1, 2, 3, 5, 6, 7, 10, 11, 13, 15)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(0, 1, 2, 3, 5, 6, 7, 10, 11, 13, 15)$, geram-se os grupos apresentados na Figura 39.

Figura 39 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP3).

Caixa.	Mintermos	Minterms em Bin		Caixa	Mintermos	Minterms em Binário
=====						
0:				0:		
	0	0000	*		0,1	000X *

1:				1:		
	1	0001	*		2,3	001X *
	2	0010	*		2,6	0X10

2:				2:		
	3	0011	*		5,7	01X1 *
	5	0101	*		10,11	101X *
	6	0110	*	-----		
	10	1010	*	3:		
-----					13,15	11X1 *
3:				-----		
	7	0111	*	Implicantes Primos da tabela: 0X10		
	11	1011	*			
	13	1101	*			

4:						
	15	1111	*			

(a)

(b)

Caixa	Mintermos	Mintermos em Binário
0:	0,1,2,3	00XX
1:	2,3,10,11	X01X
2:	5,7,13,15	X1X1

Implicantes Primos da tabela: X1X1, X01X, 00XX

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 40.

Figura 40 - Cobertura dos Implicantes Primos - (FP3).

Mintermos		0	1	2	3	5	6	7	10	11	13	15
0,1,2,3		x	x	x	x							
5,7,13,15						x		x			x	x
2,3,10,11				x	x				x	x		
2,6				x			x					

Fonte: Elaborado pelo autor

A cobertura dos implicantes primos resulta na solução $S = A\bar{B}\bar{C} + \bar{B}C\bar{D} + A\bar{D}$, correspondendo a um custo igual a 11, sendo 2 AND de três entradas, 1 AND de duas entradas e 1 OR de três entradas, que não é o custo mínimo para a função. A solução de custo mínimo, via Quine-McCluskey tradicional ou programa Espresso é $S = A\bar{B} + \bar{B}C\bar{D} + A\bar{D}$, correspondendo a um custo igual a 10, sendo 1 AND de três entradas, 2 AND de duas entradas e 1 OR de três entradas.

5.3.2.4 $F(A, B, C, D) = \sum m(3, 6, 11, 14, 16, 18, 19, 24, 26, 27, 30)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(3, 6, 11, 14, 16, 18, 19, 24, 26, 27, 30)$, geram-se os grupos apresentados na Figura 41.

Figura 41 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP4).

Grupo 0		
Caixa.	Mintermos	Minterms em Bin
=====		
1:		
	16	10000 *

2:		
	3	00011 *
	6	00110 *
	18	10010 *
	24	11000 *

3:		
	11	01011 *
	14	01110 *
	19	10011 *
	26	11010 *

4:		
	27	11011 *
	30	11110

Implicantes Primos da tabela: 11110		

Grupo 1		
Caixa	Mintermos	Minterms em Binário
=====		
0:		
	16,18	100X0 *

1:		
	3,11	0X011 *
	6,14	0X110
	24,26	110X0 *

2:		
	19,27	1X011 *

Implicantes Primos da tabela: 0X110		

(a)

(b)

Grupo 2		
Caixa	Mintermos	Mintermos em Binário
=====		
0:		
	16,18,24,26	1X0X0

1:		
	3,11,19,27	XX011

Implicantes Primos da tabela: 1X0X0, XX011		

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 42.

Figura 42 - Cobertura dos Implicantes Primos - (FP4).

Mintermos		3	6	11	14	16	18	19	24	26	27	30
=====												
16,18,24,26						x	x		x	x		

3,11,19,27		x		x				x			x	

30												x

6,14			x		x							

Fonte: Elaborado pelo autor

Apresenta solução igual a $S = \overline{A} \cdot C \cdot D \cdot \overline{E} + A \cdot \overline{C} \cdot \overline{E} + \overline{C} \cdot D \cdot E + A \cdot B \cdot C \cdot D \cdot \overline{E}$, correspondendo a um custo igual a 19, sendo 1 AND de cinco entradas, 1 AND de quatro entradas, 2 AND de três entradas e 1 OR de quatro entradas, que não é o custo mínimo para a função. A solução de custo mínimo, via Quine-McCluskey tradicional ou programa Espresso é $S = \overline{A} \cdot C \cdot D \cdot \overline{E} + A \cdot \overline{C} \cdot \overline{E} + \overline{C} \cdot D \cdot E + B \cdot C \cdot D \cdot \overline{E}$, correspondendo a custo igual a 18, sendo 2 AND de quatro entradas, 2 AND de três entradas e 1 OR de quatro entradas.

5.3.2.5 $F(A, B, C, D) = \sum m(0, 1, 2, 7, 8, 10, 12, 14, 15)$

Aplicando-se o método proposto na função $F(A, B, C, D) = \sum m(0, 1, 2, 7, 8, 10, 12, 14, 15)$, geram-se os grupos apresentados na Figura 43.

Figura 43 - Geração de Implicantes Primos: (a) Grupo 0; (b) Grupo 1 e (c) Grupo 2 - (FP5).

Grupo 0		
Caixa.	Mintermos	Minterms em Bin
=====		
0:		
	0	0000 *

1:		
	1	0001 *
	2	0010 *
	8	1000 *

2:		
	10	1010 *
	12	1100 *

3:		
	7	0111 *
	14	1110 *

4:		
	15	1111 *

(a)

Caixa	Mintermos	Minterms em Binário
=====		
0:		
	0,1	000X

1:		
	2,10	X010
	8,12	1X00 *

2:		
	10,14	1X10 *

3:		
	7,15	X111

Implicantes Primos da tabela: X111, 000X, X010

(b)

(a)

(b)

Caixa	Mintermos	Mintermos em Binário
=====		
1:	8,10,12,14	1XX0

Implicantes Primos da tabela: 1XX0		

(c)

Fonte: Elaborado pelo autor

A partir dos Implicantes primos é feita a cobertura dos mintermos, apresentada na Figura 44.

Figura 44 - Cobertura dos Implicantes Primos - (FP5).

Mintermos		0	1	2	7	8	10	12	14	15
7,15						x				x
0,1		x	x							
8,10,12,14						x	x	x	x	
2,10				x			x			

Fonte: Elaborado pelo autor

Apresenta solução igual a $S = B \cdot C \cdot D + \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{D} + \bar{B} \cdot C \cdot \bar{D}$, correspondendo a um custo igual a 15, sendo 3 AND de três entradas, 1 AND de duas entradas e 1 OR de quatro entradas, que não é o custo mínimo para a função. A solução de custo mínimo, via Quine-McCluskey tradicional ou programa Espresso é $S = B \cdot C \cdot D + \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{D} + \bar{B} \cdot \bar{D}$, correspondendo a custo igual a 14, sendo 2 AND de três entradas, 2 AND de duas entradas e 1 OR de quatro entradas.

5.4 COMPARAÇÃO ENTRE MÉTODO DE QUINE-MCCLUSKEY TRADICIONAL E MÉTODO DE QUINE-MCCLUSKEY MODIFICADO PARA OS OUTROS CASOS TESTADOS

Para comparar o método modificado com o método Quine-McCluskey tradicional utilizaram-se com parâmetros, o custo de implementação das soluções, os implicantes gerados na fase modificada, o uso de memória e o tempo de execução do programa de computador.

5.4.1 CUSTO DE IMPLEMENTAÇÃO

O custo de implementação obtido para as funções testadas no Quine-McCluskey tradicional e Quine-McCluskey modificado encontram-se na Tabela 11.

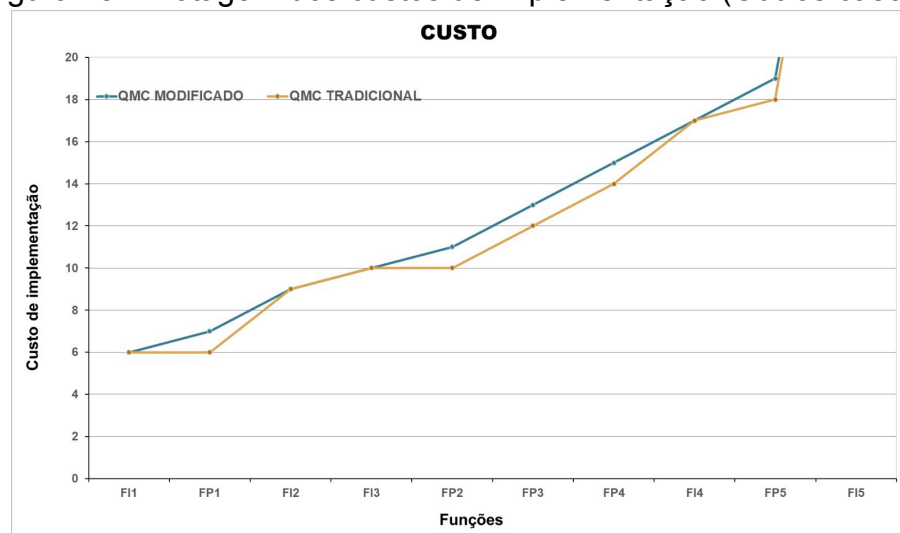
Apresenta-se a plotagem dos dados da Tabela 11 na Figura 45.

Tabela 11 - Custo de implementação das soluções (Outros casos).

Função	Variáveis de entrada	Método de Quine-McCluskey	Método de Quine-McCluskey modificado
5.2.1.1 (FI1)	4	17	17
5.2.1.2 (FI2)	4	6	6
5.2.1.3 (FI3)	4	10	10
5.2.1.4 (FI4)	4	9	9
5.2.1.5 (FI5)	5	40	40
5.2.2.1 (FP1)	4	6	7
5.2.2.2 (FP2)	4	10	11
5.2.2.3 (FP3)	4	12	13
5.2.2.4 (FP4)	5	18	19
5.2.2.5 (FP5)	4	14	15

Fonte: Elaborado pelo autor

Figura 45 - Plotagem dos custos de implementação (Outros casos).



Fonte: Elaborado pelo autor

Nota-se que o método de Quine-McCluskey modificado não garante o custo mínimo devido a modificação realizada.

5.4.2 IMPLICANTES GERADOS NA FASE MODIFICADA

A modificação proposta na fase de geração de implicantes primos, ocasiona em

menos implicantes sendo gerados, como demonstra-se na Tabela 12.

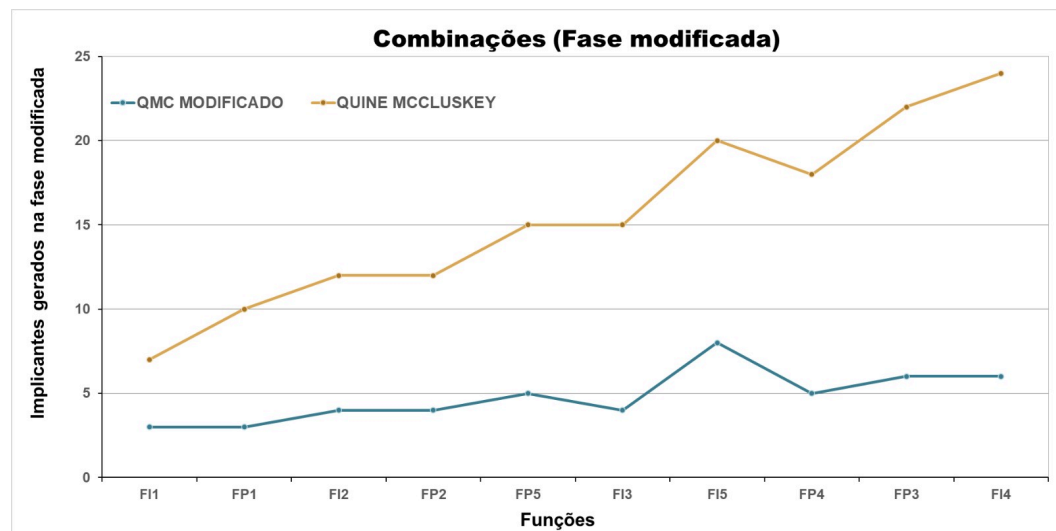
Tabela 12 - Implicantes gerados na fase modificada (Outros casos).

Função	Variáveis de entrada	Mintermos iguais a 1	Método de Quine-McCluskey	Método de Quine-McCluskey modificado
5.2.1.1 (FI1)	4	6	4	3
5.2.1.2 (FI2)	4	8	8	4
5.2.1.3 (FI3)	4	9	11	4
5.2.1.4 (FI4)	4	12	18	6
5.2.1.5 (FI5)	5	14	12	8
5.2.2.1 (FP1)	4	6	7	3
5.2.2.2 (FP2)	4	7	8	4
5.2.2.3 (FP3)	4	11	16	6
5.2.2.4 (FP4)	5	11	13	5
5.2.2.5 (FP5)	4	9	10	5

Fonte: Elaborado pelo autor

A plotagem dos dados da Tabela 12 encontra-se na Figura 46.

Figura 46 - Plotagem de Implicantes gerados na fase modificada (Outros casos).



Fonte: Elaborado pelo autor

5.4.3 USO DE MEMÓRIA

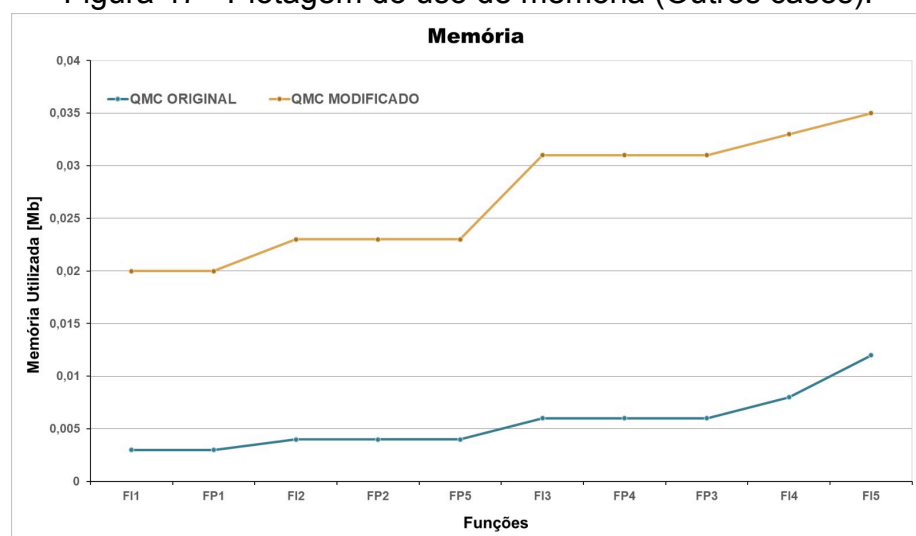
Apresenta-se os valores referentes ao uso de memória na execução dos métodos de Quine-McCluskey tradicional e do método de Quine-McCluskey modificado na Tabela 13. A plotagem dos dados da Tabela 13, apresenta-se na Figura 47.

Tabela 13 - Uso de memória em Megabytes na fase modificada (Outros casos).

Função	Variáveis de entrada	Mintermos iguais a 1	Método de Quine-McCluskey [MB]	Método de Quine-McCluskey modificado [MB]
5.2.1.1 (FI1)	4	6	0,003	0,020
5.2.1.2 (FI2)	4	8	0,004	0,023
5.2.1.3 (FI3)	4	9	0,006	0,031
5.2.1.4 (FI4)	4	12	0,008	0,033
5.2.1.5 (FI5)	5	14	0,012	0,035
5.2.2.1 (FP1)	4	6	0,003	0,020
5.2.2.2 (FP2)	4	7	0,004	0,023
5.2.2.3 (FP3)	4	11	0,006	0,031
5.2.2.4 (FP4)	5	11	0,006	0,031
5.2.2.5 (FP5)	4	9	0,004	0,023

Fonte: Elaborado pelo autor

Figura 47 - Plotagem do uso de memória (Outros casos).



Fonte: Elaborado pelo autor

5.4.4 TEMPO DE EXECUÇÃO

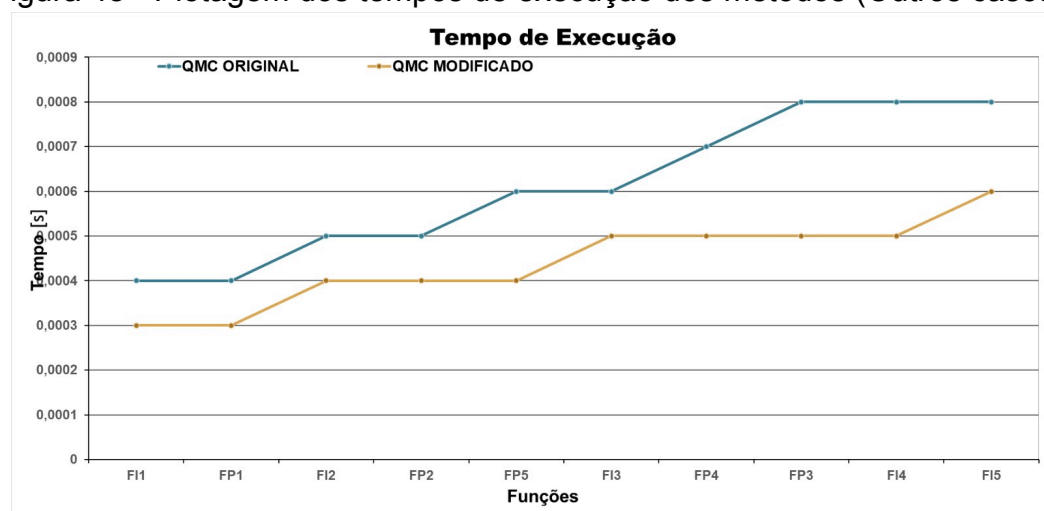
Apresenta-se os valores referentes ao tempo de execução do método de Quine-McCluskey tradicional e do método de Quine-McCluskey modificado, na Tabela 14. Apresenta-se a plotagem dos dados da Tabela 14 na Figura 48.

Tabela 14 - Tempo de execução dos métodos implementados (Outros casos).

Função	Variáveis de entrada	Mintermos iguais a 1	Método de Quine-McCluskey [MB]	Método de Quine-McCluskey modificado [MB]
5.2.1.1 (FI1)	4	6	0,0004	0,0003
5.2.1.2 (FI2)	4	8	0,0005	0,0004
5.2.1.3 (FI3)	4	9	0,0006	0,0005
5.2.1.4 (FI4)	4	12	0,0008	0,0005
5.2.1.5 (FI5)	5	14	0,0008	0,0006
5.2.2.1 (FP1)	4	6	0,0004	0,0003
5.2.2.2 (FP2)	4	7	0,0005	0,0004
5.2.2.3 (FP3)	4	11	0,0008	0,0005
5.2.2.4 (FP4)	5	11	0,0007	0,0005
5.2.2.5 (FP5)	4	9	0,0006	0,0004

Fonte: Elaborado pelo autor

Figura 48 - Plotagem dos tempos de execução dos métodos (Outros casos).



Fonte: Elaborado pelo autor

Evidencia-se que o método de Quine-McCluskey modificado é vantajoso quando a função é composta por uma grande quantidade de mintermos que assumem valor lógico 1, pois nestes casos o Quine-McCluskey tradicional têm um tempo de execução maior. Nota-se que há uma deficiência para a obtenção do custo mínimo no método proposto. Para explicar esse comportamento propõe-se analisar o que acontece em cada passo da geração dos implicantes primos do método proposto, em comparação ao que acontece no método de Quine-McCluskey tradicional, através do mapa de Karnaugh.

5.5 COMPARATIVO EM MAPA DE KARNAUGH

Para comparar a etapa de geração dos implicantes primos dos métodos de Quine-McCluskey e do método de Quine-McCluskey modificado utilizou-se o Mapa de Karnaugh, pois fica mais didático observar o que ocorre em cada passo dos métodos.

Considera-se a função $F(A, B, C, D) = \sum m (0, 1, 2, 7, 8, 10, 12, 14, 15)$, que apresenta-se na Figura 49 para o comparativo.

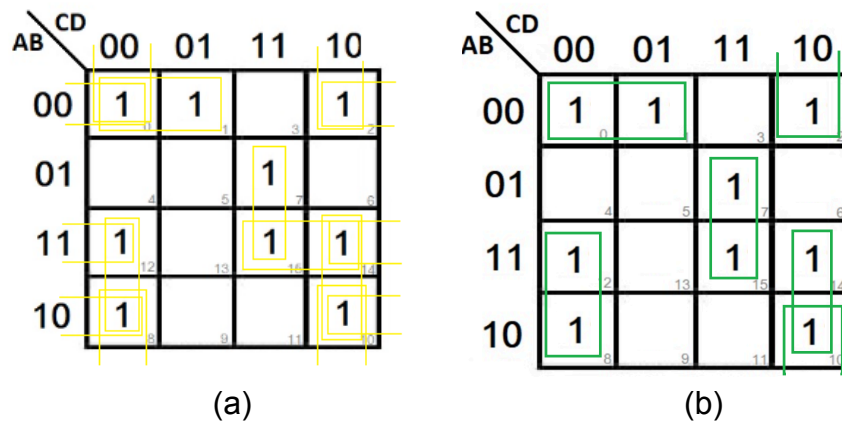
Figura 49 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 0 (Comparativo)

AB \ CD	CD			
	00	01	11	10
00	1 ₀	1 ₁		1 ₂
01			1 ₇	
11	1 ₁₂		1 ₁₅	1 ₁₄
10	1 ₈			1 ₁₀

Fonte: Elaborado pelo autor

A organização dos mintermos e don't care states em caixa que compõe o grupo 0 é a mesma para ambos os métodos. Ao executar as devidas comparações com os mintermos do grupo 0, formam-se os implicantes que compõem os grupos 1, os mapas de Karnaugh dos grupos 1 de ambos os métodos, apresentam-se na Figura 50.

Figura 50 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 1: (a) Quine-McCluskey tradicional; (b) Quine-McCluskey modificado

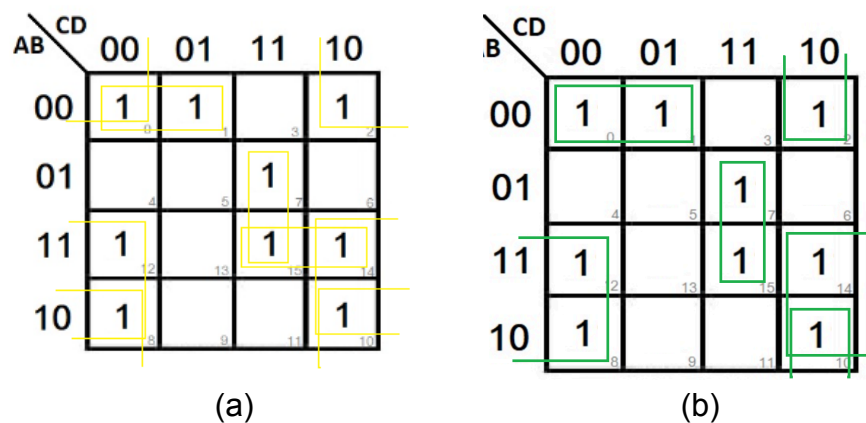


Fonte: Elaborado pelo autor

Neste caso, o método de Quine-McCluskey modificado obtém no grupo 1 uma forma irredundante de implicantes. Nota-se que alguns implicantes são perdidos na primeira etapa, não formando todos os agrupamentos possíveis, logo a obtenção de um solução mínima é comprometida.

Executando-se as devidas comparações com os implicantes do grupo 1, formam-se os implicantes que compõe os grupos 2, os mapas de Karnaugh dos grupos 2 de ambos os métodos, apresentam-se na Figura 51.

Figura 51 - Geração de Implicantes Primos - Mapa de Karnaugh do grupo 2: (a) Quine-McCluskey tradicional; (b) Quine-McCluskey modificado



Fonte: Elaborado pelo autor.

A partir dos implicantes primos gerados pelo método de Quine-McCluskey tradicional, é feita a etapa de cobertura dos mintermos da função na Figura 52.

Figura 52 - Cobertura dos Implicantes Primos - Quine-McCluskey tradicional.

Mintermos		0	1	2	7	8	10	12	14	15
8,10,12,14						x	x	x	x	
7,15					x					x
14,15									x	x
0,1		x	x							
0,2,8,10		x		x		x	x			

Fonte: Elaborado pelo autor.

A função minimizada é $S = B \cdot C \cdot D + \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{D} + \bar{B} \cdot \bar{D}$, correspondendo a custo igual a 14, sendo 2 AND de três entradas, 2 AND de duas entradas e 1 OR de quatro entradas.

A partir dos implicantes primos gerados pelo método de Quine-McCluskey modificado, é feita a etapa de cobertura dos mintermos da função na Figura 53.

Figura 53 - Cobertura dos Implicantes Primos - Quine-McCluskey modificado.

Mintermos		0	1	2	7	8	10	12	14	15
0,1		x	x							
8,10,12,14						x	x	x	x	
2,10				x			x			
7,15					x					x

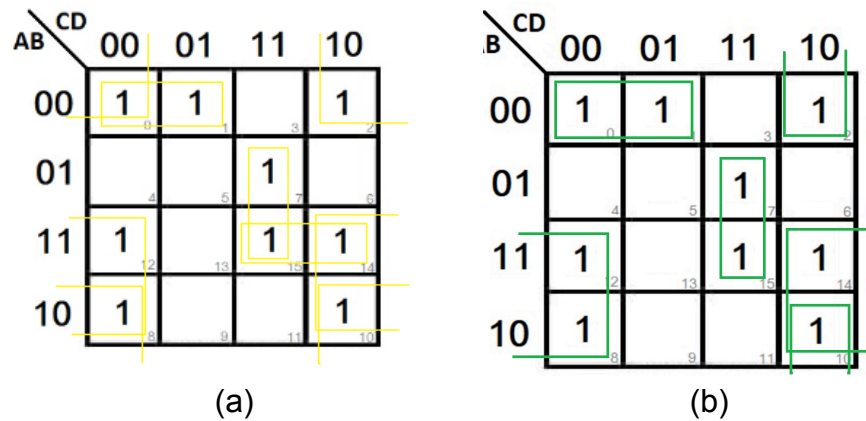
Fonte: Elaborado pelo autor.

A função minimizada é $S = B \cdot C \cdot D + \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{D} + \bar{B} \cdot C \cdot \bar{D}$, correspondendo a um custo igual a 15, sendo 3 AND de três entradas, 1 AND de duas entradas e 1 OR de quatro entradas.

Observa-se que o mintermo zero igual a 1, pelo algoritmo do método proposto, não forma par com o mintermo dois igual a 1, assim, impossibilitando a formação do implicante $\bar{B} \cdot \bar{D} = X0X0$ (0,2,8,10), termo que ocasiona na diferença de custo das

soluções. As soluções obtidas de ambos os métodos em mapas de Karnaugh apresentam-se na Figura 54.

Figura 54 - Soluções obtidas: (a) Quine-McCluskey tradicional; (b) Quine-McCluskey modificado



Fonte: Elaborado pelo autor.

Com isso é notável que o Algoritmo de Quine-McCluskey modificado, não garante a solução mínima das funções, uma vez que a geração de todos os implicantes necessários para uma simplificação que levariam ao custo mínimo pode não ocorrer, a depender da disposição dos mintermos.

6 CONCLUSÃO E SUGESTÃO PARA TRABALHOS FUTUROS

Este Neste capítulo é apresentada a conclusão do trabalho e a sugestão para trabalhos futuros nesta área de estudo

6.1 CONCLUSÃO

Este trabalho teve como objetivo desenvolver uma alternativa aos métodos existentes de simplificação de funções booleanas. Com base no método de Quine-McCluskey, foi feita uma modificação no grupo 0 da primeira etapa da fase geração de implicantes primos do método.

Essa modificação gerou implicantes em um formato com o mínimo possível de redundâncias e como consequência, reduziu também a complexidade computacional do problema de simplificação de funções booleanas.

A modificação no método foi implementada em linguagem de programação Python, utilizando de dicionários e listas para a organização dos termos em caixas e grupos.

A quantidade de implicantes gerados na etapa modificada, resultaram muito menores que no método tradicional, o que ocasionou num menor uso de memória e menor tempo de execução do algoritmo implementado, como os resultados dos testes puderam comprovar.

As soluções das funções booleanas ao aplicar o método de Quine-McCluskey modificado não tem custo mínimo garantido como as soluções pelo o método de Quine-McCluskey tradicional.

Isso se deve a redução de implicantes gerados, e a etapa de Resgate na etapa heurística, que acaba gerando um implicante dentre algumas possibilidades de implicantes que cada qual possivelmente geraria uma solução diferente.

Do ponto de vista da minimização o método proposto não apresentou resultados satisfatórios, por não garantir a solução mínima, apesar de ter um tempo de execução e uso de memória melhores que o Quine-McCluskey tradicional.

6.2 SUGESTÃO PARA TRABALHOS FUTUROS

Com o intuito de melhorar os resultados do método proposto, sugere-se a busca por uma meta-heurística. O algoritmo do método proposto encontra soluções mínimas locais e não generaliza bem para todos os casos. Uma meta-heurística aplicada a esse algoritmo traz um componente mais sofisticado ao explorar o espaço de busca para as soluções, o que pode ajudar a encontrar soluções melhores e robustas, mais próximas do mínimo global..

Mesmo que não haja garantia de que uma meta-heurística sempre encontrará a melhor solução em todos os casos, a qualidade média das soluções deve aumentar significativamente.

REFERÊNCIAS

- ADHIKARI, S. **Quine-McCluskey algorithm script in Python**. Disponível em: github.com/int-main/Quine-McCluskey. Acesso em 02/03/2024.
- BOOLE, G. **An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities**. Dover: Broadway, 424 p. 1854.
- BRAYTON, R. K. **Logic minimization algorithms for VLSI synthesis**. United States: Springer Science & Business Media, Kluwer Academic Publishers, v. 2. 54–138 p. 1984.
- BRAYTON, R. K.; HACHTEL, G. D.; HEMACHANDRA, L. A.; NEWTON, A. R.; SANGIOVANNI-VINCENTELLI, A. L. M. **A comparison of logic minimization strategies using ESPRESSO: an APL program package for partitioned logic simulation**. Proceedings of the IEEE International Symposium on Circuits and Systems, 1982. New York, New York, USA: IEEE: 42–48.
- BROWN, D.W. **A State-machine Synthesizer – SMS**. Proc 18th Design Automation Conference, Nashville, 1981.
- CHATGPT, **Generative artificial intelligence chatbot**, Disponível em : <https://openai.com/>. Acesso em 04/03/2024.
- DRAKE, C., **PyEDA a Python Library for Electronic Design Automation**, Chris Drake Revision, 2012. Disponível em: <https://pyeda.readthedocs.io/en/latest/>. Acesso em 12/12/2024.
- FERNANDES, M.I.G.M.A.S. **Estudo de Algoritmos de Minimização de Funções Booleanas Utilizando Diagramas de Decisão Binária**. Dissertação (Mestrado em Engenharia Eletrotécnica e de Computadores), Universidade do Porto, Porto, 1991.
- GUHA, S.; MITRA, U.; DEY, P.; SARMA, S. **In Search of a Better Heuristic Algorithm for Simplification of Switching Function - A Challenge to ESPRESSO**. 126f. International Journal of Computers & Technology - Council for Innovative Research(CIR). Jun. 2012.
- HLAVICKA, J.; FISER, P. **BOOM - A heuristic boolean minimizer**. In: COMPUTER AIDED DESIGN CONFERENCE, 2001, San Jose. United States: IEEE Press, p. 439–442.
- HONG, S. J.; CAIN, R. G.; OSTAPKO, D. L. MINI: **A Heuristic approach for logic minimization**. IBM Journal Of Research And Dev., New York, v.1, p. 443-458. set. 1974.
- JUPYTER. **Project Jupyter Documentation**. Disponível em: <<https://jupyter.org/>>. Acesso em: 12/12/2024.
- KOHAVI, Z. **Switching and Finite Automata Theory**, McGraw-Hill, 1970.
- MCCLUSKEY, E. J. **Minimization of boolean function**. The Bell System Technical Journal, Kansas, v. 35, pp. 1417-1444, Nov. 1956.
- MCCLUSKEY, E. J. **Minimal sums for boolean functions having many unspecified fundamental products**. In: **SWITCHING CIRCUIT THEORY AND LOGICAL**

DESIGN, 1961, Chicago. Symposium... United States: IEEE Computer Society, 1961. p. 10–17.

QUINE, W. **The problem of simplifying truth functions**. American Mathematical Monthly, New York, v. 59, p. 521-531. 1952.

ROSSUM G. V., & DRAKE, F. L. **Python Tutorial, Release 2.0**. Python Software Foundation, 2001.

RUTENBAR, R. A. **VLSI CAD Part I: Logic** by University of Illinois Urbana-Champaign, 2013 Disponível em [\[https://www.coursera.org/learn/vlsi-cad-logic\]](https://www.coursera.org/learn/vlsi-cad-logic). Acesso em: 11/10/2024.

SANCHES A. de P. **Emprego do método de Quine-McCluskey Estendido para gerar circuito mínimo com estruturas ESOP (XOR-XNOR)**. 105 f. Tese (Mestrado em Engenharia Elétrica) - Universidade Estadual Paulista, Ilha Solteira, 2017.

SILVA, A. C. R. da. **Contribuição a minimização e simulação de circuitos lógicos**. 145 f. Tese (Mestrado em Engenharia Elétrica) - Universidade Estadual de Campinas, Campinas, 1989.

SHANNON, C. E. **A Symbolic analysis of relay and Switching Circuits**, Trans.A.I.E.E., 57, pp. 713-723, 1938.

TOCCI, R.J.; WIDMER, N.S.; MOSS, G.L. **Sistemas digitais princípios e aplicações.11.ed**. São Paulo: Pearson, 2011. 817 p.

APÊNDICE A - CÓDIGO COMENTADO

O programa faz alterações e adiciona funcionalidades, com auxílio de inteligência artificial (CHATGPT, 2022), ao código do algoritmo de Quine-McCluskey de Suman Adhikari (ADHIKARI, 2018), para implementar o método de Quine-McCluskey modificado.

Linha 1 : import time

Biblioteca para monitorar o tempo de execução do programa.

Linha 2 : import psutil

Biblioteca para monitorar recursos do sistema.

Linha 3 : import random

Biblioteca para gerar entradas de forma randômica.

Linha 4 : import os

Biblioteca para interações com o sistema operacional.

Linha 5 : def disthamming(a, b):

Função que *checa se distância de Hamming entre mintermos é de apenas 1*.

Linha 6 : cont = 0

Inicializa contador de quantos bits diferem.

Linha 7 : for i in range(len(a)):

Loop que itera pelo range da string (a) recebida.

Linha 8 : if a[i] != b[i]:

Compara se os bits de índice i das strings recebidas (a e b) diferem.

Linha 9 : indexediff = i

Guarda valor do índice i que difere.

Linha 10 : cont += 1

Soma 1 para cada bit que difere.

Linha 11 : if cont > 1:

Confere se mais de um bit difere.

Linha 12 : return (False, None)

Se mais de um bit difere, retorna valor lógico falso, a simplificação não é possível.

Linha 13 : return (True, indexediff)

Retorna valor lógico verdadeiro e o bit que difere.

Linha 14 : def multpet(a, b):

Função que realiza dois parâmetros para efetuar uma multiplicação de Petrick.

Linha 15 : result = []

Inicializa uma lista para armazenar o resultado da multiplicação.

```

Linha 16 : for i in a:
# Iteração sob a primeira lista (a)
Linha 17 : for j in b:
# Iteração sob a segunda lista (b)
Linha 18 : espelhop = mul(i, j)
# Chama procedimento mul para multiplicar termos
Linha 19 : result.append(espelhop) if len(espelhop) != 0 else None
# Adiciona o resultado da multiplicação se este não for vazio
Linha 20 : return result
# Retorna a lista final dos termos multiplicados
Linha 21 : def mul(a, b):
# Função que multiplica termos via método de Petrick
Linha 22 : resultm = []
# Inicializa lista de multiplicação
Linha 23 : for i in a:
# Iteração sob a primeira lista (a)
Linha 24 : if i+"" in b or (len(i) == 2 and i[0] in b):
# verifica se 'i' está em b) está em b, ou se i já é um complemento (len(i) == 2 verifica
se i é x' e i[0] in b verifica se x está em b
Linha 25 : return []
# Retorna uma lista vazia se houver essa contradição  $A \cdot A'$ 
Linha 26 : else:
# Senão houver contradição
Linha 27 : resultm.append(i)
# Adiciona termo a lista resultante
Linha 28 : for i in b:
# Iteração sob a segunda lista
Linha 29 : if i not in resultm:
# Evitando valores duplicados no resultado
Linha 30 : resultm.append(i)
# Adiciona termo a lista resultante
Linha 31 : return resultm
# Retorna a lista final de termos
Linha 32 : def removedontcare(mescla, Idontcares):
# Remove os termos don't care da lista mesclada
Linha 33 : resultr = []

```

```

# Inicializa uma lista
Linha 34 : for i in mescla:
# Iteração sobre os termos mesclados
Linha 35 : if int(i) not in Idontcares:
# Se o termo não é um don't care
Linha 36 : resultr.append(i)
# Adiciona termo à lista de resultado
Linha 37 : return resultr
# Retorna lista
Linha 38 : def ipe(a):
# Função que identifica Implicantes Primos Essenciais, dados implicantes
Linha 39 : result = []
# Inicializa lista de resultado # Initialize list
Linha 40 : for i in a:
# Iteração sobre lista de implicantes # Loop start
Linha 41 : if len(a[i]) == 1:
# Checa se há apenas uma cobertura para um mintermo
Linha 42 : result.append(a[i][0]) if a[i][0] not in result else None
# Adiciona ao resultado, se já não estiver presente
Linha 43 : return result
# Retorna lista de implicantes primos essenciais
Linha 44 : def montasop(a):
# Converte expressão binária para expressão em Soma de Produtos
Linha 45 : lista_var = []
# Inicializa lista de variáveis
Linha 46 : for i in range(len(a)):
# Itera sobre a string de números binários
Linha 47 : if a[i] == '0':
# Checa se o bit é 0
Linha 48 : lista_var.append(chr(i+65)+""")
# Se o bit é 0, é adicionado o complemento da variável
Linha 49 : elif a[i] == '1':
# Checa se o bit é 1
Linha 50 : lista_var.append(chr(i+65))
# É Adicionada a variável na forma normal
Linha 51 : return lista_var

```

```

# Retorna lista de variáveis em forma SoP
Linha 52 : def unif(a):
# Função para unificar listas de listas(dicionários)
Linha 53 : unificados = []
# Inicializa lista vazia de unificados
Linha 54 : for i in a:
# Itera sobre a chave do dicionário
Linha 55 : unificados.extend(a[i])
# Concatena os valores à lista unificada
Linha 56 : return unificados
# Retorna lista unificada
Linha 57 : def mintermosdc(a):
# Gera os mintermos correspondentes a um don't care
Linha 58 : gaps = a.count('X')
# Conta número de bits 'X'
Linha 59 : if gaps == 0:
# Checa se não há bits 'X'
Linha 60 : return [str(int(a, 2))]
# Converte de binário e retorna representação decimal em forma de lista
Linha 61 : x = [bin(i)[2:].zfill(gaps) for i in range(pow(2, gaps))]
# Gera todas as possíveis combinações derivadas dos don't cares
Linha 62 : aux = []
# Inicializa lista auxiliar
Linha 63 : for i in range(pow(2, gaps)):
# Itera sobre todas as combinações
Linha 64 : aux2, ind = a[:], -1
# Cria cópia da string e inicia index
Linha 65 : for j in x[0]:
# Itera sobre as combinações derivadas dos don't cares
Linha 66 : if ind != -1:
# Checa se índice não está no último valor
Linha 67 : ind = ind+aux2[ind+1:].find('X')+1
# Atualiza índice para encontrar próximo 'X'
Linha 68 : else:
# Se não está no último valor
Linha 69 : ind = aux2[ind+1:].find('X')

```

```

# Obtém índice do 'X' em aux2
Linha 70 : aux2 = aux2[:ind]+j+aux2[ind+1:]
# Substitui 'X' encontrado com valor da variável j
Linha 71 : aux.append(str(int(aux2, 2)))
# Converte aux2 para um valor decimal e adiciona esse valor à lista
Linha 72 : x.pop(0)
# Remove e retorna primeiro elemento da lista de combinações derivadas de 'X'
Linha 73 : return aux
# Retorna a lista de números decimais
Linha 74 : def removeterms(tabelar, terms):
# Função que remove os mintermos já cobertos por implicantes primos essenciais
Linha 75 : for i in terms:
# Iteração sobre os implicantes primos essenciais
Linha 76 : for j in mintermosdc(i):
# Iteração sobre os termos cobertos do implicante
Linha 77 : try: # tente
Linha 78 : del tabelar[j]
# Deleta-se o mintermo da tabela
Linha 79 : except KeyError:
# Se o mintermo não estiver na tabela
Linha 80 : pass
# Não fazer nada
Linha 81 - 89 : Trecho utilizado para testar funções randômicas
Linha 81 : mintermos = []
# Inicia a variável de mintermos
Linha 82 : dontcares = []
# Inicia a variável de don't cares
Linha 83 : entradas = int(input("Quantidade de Entradas"))
# Lê a quantidade de variáveis de entrada da função
Linha 84 : qtd = random.randint(1, 2**entradas)
# Quantidade de mintermos de 1 a  $2^n$ , sendo n a quantidade de variáveis de entrada
da função
Linha 85 : while len(mintermos) < qtd:
# Enquanto a lista de mintermos for menor que a quantidade preestabelecida de
mintermos, continua a execução
Linha 86 : num = random.randint(0, (2 ** entradas) - 1)

```

```

# Gera um número inteiro aleatório entre 0 e  $2^n - 1$ 
Linha 87 : if num not in mintermos:
    # Checa se número já está na lista
Linha 88 : mintermos.append(num)
# Adiciona o número gera à lista de mintermos
Linha 89 : mintermos.sort()
# Ordena os mintermos em ordem crescente
Linha 90 : mintermos = [int(i) for i in input("Digite os mintermos da função:
").strip().split()]
# Leitura dos mintermos inteiros de forma manual
Linha 91 : dontcares = [int(i) for i in input("Entre com os don't cares(Se Houver):
").strip().split()]
# Leitura dos termos inteiros don't care
Linha 92 : process = psutil.Process(os.getpid())
# Cria o processo de um objeto que monitora o uso de memória
Linha 93 : minterms = mintermos + dontcares
# Combina mintermos and don't cares
Linha 94 : minterms.sort()
# Ordena os mintermos em ordem crescente
Linha 95: tamanho = len(bin(minterms[-1])) - 2
# Tamanho de bits necessários para a representação binária
Linha 96 : grupos, TodosImpP = {}, set()
# Inicializa os dicionários e o set de implicantes primos
Linha 97 : for minterm in minterms:
    # Itera sobre todos os mintermos
Linha 98 : try: # tente
Linha 99 : grupos[bin(minterm).count('1')].append(bin(minterm)[2:].zfill(tamanho))
#Agrupa os mintermos em representação binária pela quantidade de 1s em sua
representação binária
Linha 100 : except KeyError:
    # Caso grupo não exista
Linha 101 : grupos[bin(minterm).count('1')] = [bin(minterm)[2:].zfill(tamanho)]
# Cria novo grupo baseado na quantidade de 1s da representação binária do termo
Linha 102 : gp = 0
# Inicializa contador de grupo
Linha 103 : print("Grupo", gp)

```

```

# Imprime nome do grupo
Linha 104: print("Caixa.\tMintermos\tMinterms em Bin\n%s" % ('=' * 50))
# Imprime cabeçalho do grupo
Linha 105: for i in sorted(grupos.keys()):
# Itera sobre as chaves do dicionário de grupos(Caixa i)
Linha 106: print("%5d:" % i)
# Imprime número do grupo
Linha 107 : for j in grupos[i]:
# Itera sobre os termos binários no grupo(j = Caixa i + 1)
Linha 108: print("\t\t %-20d%s" % (int(j, 2), j))
# Imprime mintermo na forma decimal e na forma binária
Linha 109 : print('-' * 50)
# Imprime linha separadora
Linha 110 : combinou = 0
# Inicializa contador para saber se mintermo já combinou
Linha 111 : impgera = 0
# Inicializa contador para implicantes gerados na fase modificada
Linha 112 : m = 0
# Inicializa contador de caixas
Linha 113 : mem_antes = process.memory_info().rss / (1024 ** 2)
# Momento de fotografia do uso de memória
Linha 114 : inicio = time.perf_counter()
# Início da contagem do tempo de execução
Linha 115 : while True:
# Loop para encontrar implicantes primos, até não haver mais combinações
Linha 116 : gp += 1
# Incremento no contador de grupo
Linha 117 : espelho = grupos.copy()
# Cria uma cópia dos grupo atuais
Linha 118 : grupos, m, flag, parar = {}, 0, set(), True
# Reseta as variáveis de grupos, caixas, marcações, e condição de parada
Linha 119 : l = sorted(list(espelho.keys()))
# Ordena lista de chaves dos grupos
Linha 120 : for i in range(len(l) - 1):
# Itera sobre os grupos
Linha 121 : for j in espelho[l[i]]:

```

```

# Itera sobre os termos da caixa [ i ]
Linha 122 : for k in espelho[l[i + 1]]:
# Itera sobre os termos da caixa [ i + 1 ]
Linha 123: if gp >1:
# Se grupo não for grupo 0
Linha 124 : result = disthamming(j, k)
# Chama a função que checa se a distancia de Hamming é igual a 1
Linha 125 : if result[0]:
# Se a distância de Hamming é 1
Linha 126: try: # tente
Linha 128: grupos[m].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] + 'X' +
j[result[1] + 1:] not in grupos[m] else None
# Adiciona novo termo com 'X' ao grupo, se inédito
Linha 129 : except KeyError:
# Se grupo não existe
Linha 130 : grupos[m] = j[:result[1]] + 'X' + j[result[1] + 1:]
# Cria-se grupo
Linha 131 : parar = False
# Há combinações possíveis
Linha 132 : flag.add(j)
# Marca-se o termo da caixa [ i ] que foi combinado
Linha 133 : flag.add(k)
# Marca-se o termo da caixa [ i + 1 ] que foi combinado
Linha 134 : else: # senão
Linha 135: result = disthamming(j, k)
# Chama a função que checa se a distância de Hamming é igual a 1
Linha 136 : combinou = 0
# Checa se mintermo já combinou
Linha 137 : if result[0] and j not in flag and k not in flag:
# Se distância de Hamming = 1 e j e k, não foram marcados
Linha 138 : try: # tente
Linha 139 : grupos[m].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] + 'X' +
j[result[1] + 1:] not in grupos[m] else None
# Atribui don't care('X') ao bit que parecia e o adiciona a caixa correspondente
Linha 140 : combinou = 1
# Termo combinou

```

```

Linha 141 : if gp == 1:
# Se grupo é grupo 0
Linha 142 : impgera += 1
# Incremento para implicante gerado na fase modificada
Linha 143: except KeyError:
# Se grupo não existe
Linha 144 : grupos[m] = [j[:result[1]] + 'X' + j[result[1] + 1:]]
# Se grupo ainda não existe, é criado, atribui don't care('X') ao bit que parecia e o
adiciona ao grupo
Linha 145: combinou = 1
# Termo combinou
Linha 146: if gp == 1:
# Se grupo é grupo 0
Linha 147 : impgera += 1
# Incremento para implicante gerado na fase modificada
Linha 148 : parar = False
# Há combinações possíveis
Linha 149 : flag.add(j)
# Marca-se o termo da caixa [ i ] que foi combinado
Linha 150 : flag.add(k)
# Marca-se o termo da caixa [ i + 1 ] que foi combinado
Linha 151 : if combinou == 0:
# Se termo não combinou
Linha 152 : for v in espelho[l[i - 1]]:
# Itera sobre os termos da caixa [ i - 1 ]
Linha 153 : result = disthamming(j, v)
# Chama a função que checa se a distância de Hamming é igual a 1
Linha 154 : combinou = 0
# Termo não combinou
Linha 155 : if result[0] and j not in flag:
# Se distância de Hamming = 1 e j não foi marcado
Linha 156 : try: # tente
Linha 157 : grupos[m - 1].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] + 'X' +
j[result[1] + 1:] not in grupos[m - 1] else None
# Atribui don't care('X') ao bit que parecia e o adiciona a caixa correspondente
Linha 158 : combinou = 1

```

```

# Termo combinou
Linha 159: except KeyError:
# Se grupo não existe
Linha 160 : grupos[m - 1] = [j[:result[1]] + 'X' + j[result[1] + 1:]]
# Se grupo ainda não existe, é criado, atribui don't care('X') ao bit que parecia e o
adiciona ao grupo
Linha 161 : combinou = 1
# Termo combinou
Linha 162 : parar = False
# Há combinações possíveis
Linha 163 : flag.add(j)
# Marca-se o termo da caixa [ i ] que foi combinado
Linha 164 : flag.add(v)
# Marca-se o termo da caixa [ i - 1 ] que foi combinado
Linha 165 : if combinou == 0:
# Se termo não combinou
Linha 166 : for v in espelho[l[i + 1]]:
# Itera sobre os termos da caixa [ i + 1]
Linha 167 : result = disthamming(j, v)
# Chama a função que checa se a distancia de Hamming é igual a 1
Linha 168 : combinou = 0
# Termo não combinou
Linha 169 : if result[0] and j not in flag:
# Se distância de Hamming = 1 e j não foi marcado
Linha 170 : try: # tente
Linha 171 : grupos[m + 1].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] + 'X' +
j[result[1] + 1:] not in grupos[m + 1] else None
# Atribui don't care('X') ao bit que parecia e o adiciona a caixa correspondente
Linha 172 : combinou = 1
# Termo combinou
Linha 173: except KeyError:
# Se grupo não existe
Linha 174 : grupos[m + 1] = [j[:result[1]] + 'X' + j[result[1] + 1:]]
# Se grupo ainda não existe, é criado, atribui don't care('X') ao bit que parecia e o
adiciona ao grupo
Linha 175 : combinou = 1

```

```

# Termo combinou
Linha 176 : parar = False
# Há combinações possíveis
Linha 177 : flag.add(j)
# Marca-se o termo da caixa [ i ] que foi combinado
Linha 178 : m += 1
# Incremento no contador de caixas
Linha 179 : local_Nflag = set(unif(espelho)).difference(flag)
# Lista de termos não marcados
Linha 180 : TodosImpP = TodosImpP.union(local_Nflag)
# Adiciona termos não combinados ao conjunto de implicantes primos
Linha 181 : print("Implicantes Primos da tabela:", None if len(local_Nflag) == 0 else ',
'.join(local_Nflag))
# Imprime implicantes primos atuais
Linha 182 : if parar:
# Se não há mais combinações possíveis
Linha 183 : print("\n\nTodos Implicantes Primos: ", None if len(TodosImpP) == 0 else ',
'.join(TodosImpP))
# Imprime-se todos os implicantes primos
Linha 184 : break # Quebra de loop
Linha 185 : print("Grupo", gp)
# Imprime nome do grupo
Linha 186 : print("Caixa", "\tMintermos\tMintermos em Binário\n%s" % ('=' * 50))
# Imprime cabeçalho do grupo
Linha 187: for i in sorted(grupos.keys()):
# Itera sobre as chaves do dicionário de grupos(Caixa i)
Linha 188: print("%5d:" % i)
# Imprime número do grupo
Linha 189 : for j in grupos[i]:
# Itera sobre os termos binários no grupo(j = Caixa i + 1)
Linha 190: print("\t\t %-24d%s" % (mintermosdc(j)), j)
# Imprime mintermo na forma decimal e na forma binária
Linha 191 : print('-' * 50)
# Imprime linha separadora
Linha 192 : tamanho = len(str(mintermos[-1]))
# Tamanho do mintermo com maior número de caracteres

```

```

Linha 193 : Tabela = {}
# Inicializa tabela de implicantes primos
Linha 194 : print('\n\nTabela de Implicantes Primos:\n\n Mintermos    |%s\n%s' % (
    ''.join(' ' * (tamanho - len(str(i)))) + str(i) for i in mintermos),
# Cabeçalho da tabela de implicantes primos, mintermos em colunas
Linha 195 : '=' * (len(mintermos) * (tamanho + 1) + 16)))
# Linha divisória
Linha 196 : for i in TodosImpP:
# Itera sobre todos os implicantes primos
Linha 197 : MesclaMintermos, y = mintermosdc(i), 0
# Mintermos que farão parte da cobertura, Ex. don't cares
Linha 198 : print("%-16s|" % ''.join(MesclaMintermos), end=")
# Imprime mintermos da cobertura
Linha 199 : for j in removedontcare(MesclaMintermos, dontcares):
# Itera sobre a mescla, exceto don't cares
Linha 200 : x = mintermos.index(int(j)) * (tamanho + 1)
# Posição do termo
Linha 201 : print(' ' * abs(x - y) + ' ' * (tamanho - 1) + 'x', end=")
# Imprime espaços necessários para que 'x' seja impresso na posição do termo
Linha 202 : y = x + tamanho
# Posição atual
Linha 203 : try: # tente
Linha 204 : Tabela[j].append(i) if i not in Tabela[j] else None
# Adiciona implicante à variável Tabela
Linha 205 : except KeyError:
# Se posição j da tabela não existe
Linha 206 : Tabela[j] = [i]
# Cria Tabela[j] e atribui i
Linha 207 : print('\n' + '-' * (len(mintermos) * (tamanho + 1) + 16))
# Imprime linha divisória
Linha 208 : ImpPE = ipe(Tabela)
# Busca implicantes primos essenciais
Linha 209 : print("\nImplicantes Primos Essenciais: " + ' '.join(str(i) for i in ImpPE))
# Imprime implicantes primos essenciais
Linha 210 : fim = time.perf_counter()
# Fim da contagem de tempo de execução

```

```

Linha 211 : tempoexec = fim - inicio
# Subtração tempo final - tempo inicial
Linha 212 : print(f'Tempo de execução QMC mod.: {tempoexec:.4f} segundos')
# Imprime tempo de execução
Linha 213 : removeterms(Tabela, ImpPE)
# Remove termos cobertos da tabela
Linha 214 : if len(Tabela) == 0:
# Checa se todos termos foram cobertos
Linha 215 : result_final = [montasop(i) for i in ImpPE]
# Monta soma de produtos com implicantes primos essenciais
Linha 216 : else: # senao
Linha 217 : P = [[montasop(j) for j in Tabela[i]] for i in Tabela]
# Cria lista com todas as combinações possíveis
Linha 218 : while len(P) > 1:
# Reduz a lista até uma combinação sobrar
Linha 219 : P[1] = multpet(P[0], P[1])
# Método de Petrick para minimizar
Linha 220 : P.pop(0)
# Remove combinação
Linha 221 : result_final = [min(P[0], key=len)]
# Termo de P com mínimo de variáveis
Linha 222 : result_final.extend(montasop(i) for i in ImpPE)
# Adiciona soma de produtos com implicantes primos essenciais à solução
Linha 223 : print('\n\nSolução: F = ' + ' + ' + '.join("".join(i) for i in result_final))
# Imprime Soma de produtos
Linha 224 : mem_dps = process.memory_info().rss / (1024 ** 2)
# Fotografia final do uso de memória
Linha 225 : print('Implicantes gerados na fase heurística', impgera)
# Total de implicantes gerados na fase modificada
Linha 226 : print(f'Memória Utilizada: {mem_dps - mem_antes:.3f} MB')
# Imprime diferença da fotografia final e inicial de uso de memória

```

APÊNDICE B - CÓDIGO INDENTADO

As alterações feitas no código do algoritmo de Quine-McCluskey de Suman Adhikari (ADHIKARI, 2018), para implementar o método de Quine-McCluskey modificado estão destacadas em cor alaranjada no código indentado.

```
import time
import psutil
import os
import random
```

```
def disthamming(a, b): # Checa se distância de Hamming entre mintermos é igual a 1
    cont = 0
    for i in range(len(a)):
        if a[i] != b[i]:
            indexediff = i
            cont += 1
            if cont > 1:
                return (False, None)
    return (True, indexediff)
```

```
def multpet(a, b): # Multiplicação Petrick
    result = []
    for i in a:
        for j in b:
            espelhop = mul(i, j)
            result.append(espelhop) if len(espelhop) != 0 else None
    return result
```

```
def mul(a, b): # Multiplicação Mintermos
    resultm = []
    for i in a:
        if i+"" in b or (len(i) == 2 and i[0] in b):
            return []
        else:
            resultm.append(i)
    for i in b:
```

```

    if i not in resultm:
        resultm.append(i)
    return resultm

```

```

def removedontcare(mescla, ldontcares): # Remove os don't cares de uma lista
    resultr = []
    for i in mescla:
        if int(i) not in ldontcares:
            resultr.append(i)
    return resultr

```

```

def ipe(a): # Implicantes primos essenciais da lista de implicantes primos
    result = []
    for i in a:
        if len(a[i]) == 1:
            result.append(a[i][0]) if a[i][0] not in result else None
    return result

```

```

def montasop(a): # Montagem SOP(soma de produtos), Converte para ABCD
    lista_var = []
    for i in range(len(a)):
        if a[i] == '0':
            lista_var.append(chr(i+65)+"")
        elif a[i] == '1':
            lista_var.append(chr(i+65))
    return lista_var

```

```

def unif(a): # Unificando lista
    unificados = []
    for i in a:
        unificados.extend(a[i])
    return unificados

```

```

def mintermosdc(a): # Remontagem dos mintermos dos don't care
    gaps = a.count('X')
    if gaps == 0:
        return [str(int(a,2))]

```

```

x = [bin(i)[2:].zfill(gaps) for i in range(pow(2, gaps))]
aux = []
for i in range(pow(2, gaps)):
    aux2, ind = a[:, -1]
    for j in x[0]:
        if ind != -1:
            ind = ind+aux2[ind+1:].find('X')+1
        else:
            ind = aux2[ind+1:].find('X')
        aux2 = aux2[:ind]+j+aux2[ind+1:]
    aux.append(str(int(aux2, 2)))
    x.pop(0)
return aux

```

```

def removeterms(tabelar, terms): # Remoção dos mintermos já cobertos

```

```

    for i in terms:
        for j in mintermosdc(i):
            try:
                del tabelar[j]
            except KeyError:
                pass

```

```

mintermos = [int(i) for i in input("Digite os mintermos da função: ").strip().split()]
dontcares = [int(i) for i in input("Entre com os don't cares(Se Houver): ").strip().split()]
process = psutil.Process(os.getpid())

```

```

#Etapa opcional para testes randômicos

```

```

=====

```

```

#testes

```

```

mintermos = []
dontcares = []
entradas = int(input("Quantidade de Entradas"))

```

```

qtd = random.randint(1, 2**entradas)

```

```

while len(mintermos) < qtd:

```

```

num = random.randint(0, (2 ** entradas) - 1)
if num not in mintermos: #Checa se valor já está na lista
    mintermos.append(num)
mintermos.sort()
print(mintermos)

while len(dontcares) < 0:
    num = random.randint(0, 7)
    if num not in mintermos and num not in dontcares: #Checa se valor já está na lista
        dontcares.append(num)
"""

minterms = mintermos+dontcares
minterms.sort()
tamanho = len(bin(minterms[-1]))-2
grupos,TodosImpP = {}, set()
for minterm in minterms:
    try:
grupos[bin(minterm).count('1')].append(bin(minterm)[2:].zfill(
tamanho)
    except KeyError:
        grupos[bin(minterm).count('1')] = [bin(minterm)[2:].zfill(tamanho)]
gp = 0
print("Grupo", gp)
print("Caixa.\tMintermos\tMinterms em Bin\n%s" % (' '*50))
for i in sorted(grupos.keys()):
    print("%5d:" % i)
    for j in grupos[i]:
        print("\t\t %-20d%s" % (int(j, 2), j))
    print('-'*50)

combinou = 0
impgera = 0
m = 0
mem_antes = process.memory_info().rss / (1024 ** 2)
inicio = time.perf_counter() # tempo inicial

```

```

while True:
    gp += 1
    espelho = grupos.copy()
    grupos, m, flag, parar = {}, 0, set(), True
    l = sorted(list(espelho.keys()))
    for i in range(len(l)-1):
        for j in espelho[l[i]]: # Caixa atual
            for k in espelho[l[i+1]]: # Caixa+1
                if gp > 1:
                    result = disthamming(j, k)
                    if result[0]: # Se distância de Hamming = 1
                        try:
                            grupos[m].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] + 'X'
+ j[result[1] + 1:] not in grupos[m] else None # Atribui dontcare('X') ao bit que parecia e
o adiciona a caixa correspondente
                        except KeyError:
                            grupos[m] = [j[:result[1]] + 'X' + j[result[1] + 1:]] # Se caixa ainda não
existe, é criada, atribui dontcare('X') ao bit que parecia e o adiciona à caixa
                            parar = False
                            flag.add(j)
                            flag.add(k)
                else:
                    result = disthamming(j, k)
                    combinou = 0
                    if result[0] and j not in flag and k not in flag: # Se distância de Hamming = 1
                        try:
                            grupos[m].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] + 'X'
+ j[result[1] + 1:] not in grupos[m] else None # Atribui dontcare('X') ao bit que parecia e
o adiciona a caixa correspondente
                            combinou = 1
                            if gp == 1:
                                impgera += 1
                        except KeyError:
                            grupos[m] = [j[:result[1]] + 'X' + j[result[1] + 1:]] # Se caixa ainda não
existe, é criada, atribui dontcare('X') ao bit que parecia e o adiciona à caixa
                            combinou = 1

```

```

        if gp == 1:
            impgera += 1
        parar = False
        flag.add(j)
        flag.add(k)
    if combinou == 0:
        for v in espelho[l[i - 1]]: # Caixa-1
            result = disthamming(j, v)
            combinou = 0
            if result[0] and j not in flag: # Se distância de Hamming = 1
                try:
                    grupos[m - 1].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] +
                    'X' + j[result[1] + 1:] not in grupos[m - 1] else None # Atribui dontcare('X') ao bit que
pareia e o adiciona a caixa correspondente
                    combinou = 1
                except KeyError:
                    grupos[m - 1] = [j[:result[1]] + 'X' + j[result[1] + 1:]] # Se caixa ainda
não existe, é criada, atribui dontcare('X') ao bit que pareia e o adiciona à caixa
                    combinou = 1
            parar = False
            flag.add(j)
            flag.add(v)
    if combinou == 0:
        for v in espelho[l[i + 1]]: # Caixa+1
            result = disthamming(j, v)
            combinou = 0
            if result[0] and j not in flag: # Se distância de Hamming = 1
                try:
                    grupos[m].append(j[:result[1]] + 'X' + j[result[1] + 1:]) if j[:result[1]] + 'X'
                    + j[result[1] + 1:] not in grupos[m] else None # Atribui dontcare('X') ao bit que pareia e
o adiciona a caixa correspondente
                    combinou = 1
                except KeyError:
                    grupos[m] = [j[:result[1]] + 'X' + j[result[1] + 1:]] # Se caixa ainda não
existe, é criada, atribui dontcare('X') ao bit que pareia e o adiciona à caixa
                    combinou = 1

```

```

        parar = False
        flag.add(j)

    m += 1

    local_Nflag = set(unif(espelho)).difference(flag)
    TodosImpP = TodosImpP.union(local_Nflag) # + ImpP lista geral
    print("Implicantes Primos da tabela:", None if len(local_Nflag) == 0 else ',
'.join(local_Nflag))
    if parar: # Se mintermos não podem mais ser combinados
        print("\n\nTodos Implicantes Primos: ", None if len(TodosImpP) == 0 else ',
'.join(TodosImpP))
        break

    print("Grupo", gp)
    print("Caixa", "\tMintermos\tMintermos em Binário\n%s" % (' '*50))
    for i in sorted(grupos.keys()):
        print("%5d:" % i)
        for j in grupos[i]:
            print("\t\t%-24s%s" % (''.join(mintermosdc(j)), j))
        print('-'*50)

tamanho = len(str(mintermos[-1]))
Tabela = {}
print("\n\nTabela de Implicantes Primos:\n\n Mintermos    |%s\n%s" % (''.join((' ' *
(tamanho - len(str(i)))) + str(i) for i in mintermos), '=' * (len(mintermos) * (tamanho + 1) +
16)))
for i in TodosImpP:
    MesclaMintermos, y = mintermosdc(i), 0
    print("%-16s|" % ''.join(MesclaMintermos), end="")
    for j in removedontcare(MesclaMintermos, dontcares):
        x = mintermos.index(int(j)) * (tamanho + 1) # Posição 'X'
        print(' ' * abs(x - y) + ' ' * (tamanho - 1) + 'x', end="")
        y = x + tamanho
    try:
        Tabela[j].append(i) if i not in Tabela[j] else None # Imprime mintermo na tabela
    except KeyError:
        Tabela[j] = [i]

```

```

print('\n' + '-' * (len(mintermos) * (tamanho + 1) + 16))

ImpPE = ipe(Tabela) # Encontra ImpPE
print("\nImplicantes Primos Essenciais: " + ', '.join(str(i) for i in ImpPE))

fim = time.perf_counter()
tempoexec = fim - inicio
print(f'Tempo de execução QMC mod.: {tempoexec:.6f} segundos')
mem_dps = process.memory_info().rss / (1024 ** 2)
removeterms(Tabela, ImpPE) # Remoção de ImpPE

if len(Tabela) == 0: # Se não sobrarem mintermos após remoção de ImpPE
    result_final = [montasop(i) for i in ImpPE]
else:
    P = [[montasop(j) for j in Tabela[i]] for i in Tabela]
    while len(P) > 1: # Até forma SoP
        P[1] = multpet(P[0], P[1])
        P.pop(0)
    result_final = [min(P[0], key=len)] # Termo de P com o mínimo de Variáveis
    result_final.extend(montasop(i) for i in ImpPE) # ImpPEs na solução final
print("\n\nSolução: F = ' + ' + '.join("'.join(i) for i in result_final))

mem_dps = process.memory_info().rss / (1024 ** 2)
print('Implicantes gerados na fase heurística', impgera)
print(f'Memória Utilizada: {mem_dps - mem_antes:.3f} MB")

```