Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais

Campus Juiz de Fora

Engenharia Mecatrônica

Eduardo Vítor Giancoli Jabour

Projeto de Circuitos Lógicos Combinacionais via Algoritmo Genético com decomposição da Tabela Verdade utilizando Árvore de Multiplexadores

Juiz de Fora

2020

Ficha catalográfica confeccionada pela Biblioteca do *campus* Juiz de Fora

J11p Jabour, Eduardo Vítor Giancoli.

Projeto de circuitos lógicos combinacionais via algoritmo genético com decomposição da tabela verdade utilizando árvore de multiplexadores. / Eduardo Vítor Giancoli Jabour. – 2020.

60 f.: il.

Orientador: Prof. D.Sc. Francisco Augusto Lima Manfrini.

Monografia (Graduação em Engenharia Mecatrônica) – Instituto Federal do Sudeste de Minas Gerais – Juiz de Fora, 2020.

1. Hardware evolutivo. 2. Computação paralela. 3. Multiplicadores. 4. Algoritmo evolutivo.

CDD – 620.00420285

Bibliotecária Vânia Márcia de Paula – CRB6/2894

Eduardo Vítor Giancoli Jabour

Projeto de Circuitos Lógicos Combinacionais via Algoritmo Genético com decomposição da Tabela Verdade utilizando Árvore de Multiplexadores

Trabalho de Conclusão de Curso apresentado ao *Campus* Juiz de Fora do Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais como requisito parcial para obtenção do grau de Bacharel em Engenharia Mecatrônica.

Orientador: Dr. Francisco Augusto Lima Manfrini

Juiz de Fora

2020

AGRADECIMENTOS

À minha esposa e melhor amiga, que esteve intimamente próxima a mim durante todo esse processo, me acompanhando em cada momento e me tornando, a cada dia, mais capaz de alcançar meus objetivos. Lana, obrigado por ter o poder de renovar todas as minhas forças com um sorriso.

Aos meus pais, Filippe e Eugênia, a quem sou incomensuravelmente grato por abrirem mão de tanto de si pelos filhos.

Aos meus irmãos Letícia, Fábio, Débora e Alessandra, que tornam minha vida mais feliz. E aos meus cunhados, Julia e Henrique, por ajudá-los com isso.

Aos meus avós, Denize, William, Edianez e, em especial, ao meu avô Walter por ter se tornado o maior engenheiro que conheci.

Ao Leandro, Luísa, Pedro, André, Gabriel, tios e tias, pelo constante apoio e suporte, demonstrados em inúmeros momentos e de tantas formas.

Ao Lucas, Selmo e Adriana, por terem me acolhido e estendido minha família.

À Pan, Hera e Bacco, pelo companheirismo e paciência.

Ao meu orientador, Francisco, pelo auxílio e confiança.

RESUMO

Neste trabalho, é proposto e implementado um algoritmo evolutivo chamado de GAMT (Algoritmo Genético através de Árvore de Multiplexadores), para o projeto de Circuito Lógico Combinacionais (CLCs), com o objetivo de minimizar a quantidade de elementos lógicos utilizados, reduzindo o tempo de processamento necessário para solução dos problemas e possibilitando a solução de problemas mais complexos. Após receber a matriz de entrada do problema, conecta-se na saída desta matriz um multiplexador, com o intuito de decompor a entrada do problema em matrizes menores, recursivamente, gerando uma árvore de multiplexadores. Dessa forma, à medida que o programa soluciona cada um dos multiplexadores da árvore, ele deixa de processar aquela parte da entrada e concentra o processamento apenas na parcela do problema que ainda não foi resolvida. Assim, diminui-se drasticamente a quantidade de dados processados para que seja encontrada a solução do problema. O tamanho das instâncias a serem tratadas decresce exponencialmente. A versão final do sistema foi desenvolvida em Java e é totalmente parametrizada e independente do formato da tabela enunciado de entrada. Sob o ponto de vista computacional, dado que as subdivisões da tabela verdade não são dependentes entre si, pode-se explorar o paralelismo de processamento. À medida em que a matriz de entrada vai sendo subdividida, estas submatrizes vão sendo repassadas a várias linhas de execução (*threads*) que passam a ser executadas concorrentemente nos processadores e núcleos disponíveis. No algoritmo proposto, cada indivíduo é um conjunto de CLCs que gera, a partir da entrada, valores *booleanos*, que são a chave do multiplexador. O código computa um indivíduo que representa o CLC ligado a chave do MUX. Para a evolução deste indivíduo, o GAMT possui um processo de replicação e mutação. Foram desenvolvidos, também, um método de avaliação destes indivíduos e uma técnica que evita a estagnação do programa em pontos de máximo local. Foram executados testes em multiplicadores de dimensões 2×2, 3×3 e 4×4. Os resultados foram discutidos, dados de saídas foram apresentados e alguns dos circuitos gerados foram representados. O algoritmo desenvolvido se mostrou eficiente em termos de desempenho computacional, resolveu todas as instâncias propostas nas referidas dimensões e apresentou bom desempenho na fuga de regiões de máximos locais.

Palavras-chave: Hardware Evolutivo. Computação Paralela. Multiplicadores. Algoritmo Evolutivo.

ABSTRACT

In this work, an evolutionary algorithm called GAMT (Genetic Algorithm through Multiplexer Tree) is proposed and implemented for the design of Combinational Logic Circuits (CLC), with the aim of minimizing the amount of logic elements used, reducing the processing time necessary to solve problems and enabling the solution of more complex problems. After receiving the problem input matrix, a multiplexer is connected to the output of this matrix, in order to decompose the problem input into smaller matrices, recursively, generating a multiplexer tree. In this way, as the program solves each of the multiplexers in the tree, it stops processing that part of the input and focusses the processing only on the parts of the problem that have not yet been solved. Thus, the amount of data processed is drastically reduced when finding the solution to the problem. The size of the instances to be handled decreases exponentially. The final version of the system was developed in Java and is fully parameterized and independent of the configuration of the stated table of entry. From a computational point of view, given that the truth table subdivisions are not dependent on each other, processing parallelism can be explored. As the input matrix is subdivided, these sub-matrices are passed on to several threads of execution that start to be executed concurrently in the available processors and cores. In the proposed algorithm, each individual is a set of CLCs that generate, from the input, boolean values, which are the multiplexer control lines. The code computes an individual that represents the CLC linked to the MUX control inputs. For the evolution of this individual, GAMT has a process of replication and mutation. A method was also developed to evaluate these individuals and a technique that avoids stagnating the program at points of local maxima was implemented. Tests were performed on multipliers of dimensions 2×2, 3×3 and 4×4. The results were discussed, output data were presented and some of the generated circuits were represented. The developed algorithm proved to be efficient in terms of computational performance, solved all the instances proposed in the referred dimensions and presented a good performance in the escape from regions of local maxima.

Keywords: Evolutionary hardware. Parallel Computing. Multipliers. Algorithm

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACO	Otimização por Colônia de Formigas - *Ant Colony Optimization*
AE	Algoritmo Evolutivo
AG	Algoritmo Genético
API	Application Programming Interface
CLC	Circuito Lógico Combinacional
CPU	Central Processing Unit
EPS	*encapsulated postScript*
GAMT	Algoritmo Genético através de Árvore de Multiplexadores - *Genetic Algorithm through Multiplexer Tree*
MGA	Algoritmo Genético Multiobjetivo - *Multiobjective Genetic Algorithm*
MUX	Multiplexador
MUX2	Multiplexador de 2 entradas
MUX4	Multiplexador de 4 entradas
MUX8	Multiplexador de 8 entradas
NGA	Algoritmo Genético N - *N Genetic Algorithm*
PG	Programação Genética
PSO	Otimização por enxame de partículas - *Particle Swarm Optimization*
QEA	Quantum - Inspired Evolutionary Algorithm
TCC	Trabalho de Conclusão de Curso

SUMÁRIO

1 Introdução

Os sistemas digitais, desde o mais simples ao mais complexo, são compostos pelos elementos denominados portas lógicas. As portas lógicas são os blocos construtivos de um circuito digital e a partir delas podem ser implementados circuito digitais complexos (VAHID, 2006). Os circuitos lógicos dos sistemas digitais podem ser de dois tipos: circuitos combinacionais ou circuitos sequenciais.

Um circuito sequencial emprega elementos de armazenamento denominados *latches* e *flip-flops*, além de portas lógicas. Os valores das saídas do circuito dependem dos valores das entradas e dos estados dos *latches* ou *flip-flops* utilizados, diz-se que as saídas de um circuito sequencial dependem também do histórico do próprio circuito. Já um circuito combinacional é constituído por um conjunto de portas lógicas as quais determinam os valores das saídas baseando-se apenas nos valores atuais das entradas, como ilustrado na Figura 1.

Figura 1 – Diagrama genérico de um circuito combinacional.

Um CLC (Circuito Lógico Combinacional) é um circuito que possibilita encontrar a saída de um problema aplicando combinações de portas lógicas na entrada proposta. O projeto de tal circuito baseia-se nos dados de uma tabela verdade, que relaciona todas as combinações possíveis das variáveis booleanas presentes na entrada. Uma tabela enunciado é composta, então, por uma tabela verdade, como entrada, e uma tabela saída com o mesmo número de linhas. Fornecida uma determinada tabela enunciado, é possível criar um CLC que obedeça as condições propostas no problema estabelecido e apresente a saída em função da entrada. Isso é feito aplicando técnicas tradicionais de projeto e meta-heurísticas (MOORE; VENAYAGAMOORTHY, 2005).

O projeto de um circuito combinacional inicia na especificação do problema e culmina no diagrama do circuito (ou no conjunto de equações que o descrevem). Um procedimento genérico para o projeto envolve os seguintes passos:

1. A partir da especificação do problema, determinar a tabela verdade (caso ela já não faça parte da especificação do problema).
2. Obter as expressões booleanas simplificadas.
3. Mapear o circuito para a biblioteca de portas disponível (se for o caso).
4. Representar a equação booleana e o circuito final.

O projeto de um CLC é uma atividade que exige considerável conhecimento humano e criatividade (COELLO; CHRISTIANSEN; AGUIRRE, 1999), pois mesmo após a utilização dos métodos tradicionais, um projetista experiente, pode otimizar o resultado manipulando a expressão booleana encontrada através da aplicação de teoremas booleanos. Este tipo de problema pode ser considerado um problema de otimização discreta (ALBA et al., 2007) e devido ao fato do ótimo global (circuito ideal) ser desconhecido, o uso de meta-heurísticas é indicado (HERNANDEZ-AGUIRRE; COELLO, 2004).

Uma meta-heurística é um método heurístico para resolver de forma genérica problemas de otimização e são geralmente aplicadas a problemas para os quais não se conhece algoritmo eficiente. Por definição, em ciência da computação, algoritmo genético (AG) é uma meta-heurística inspirada em processos de seleção natural, hereditariedade, mutação e recombinação, que pertence a uma classe mais ampla denominada algoritmos evolutivos (AE). Seu objetivo é encontrar soluções em problemas de otimização e busca. Existem propostas na literatura do uso de AG para o projeto de circuitos combinacionais. Na literatura contemporânea, a utilização de técnicas evolutivas no projeto de CLCs é conhecida como hardware evolutivo (LINDEN, 2012).

Alguns termos importantes utilizados ao longo deste trabalho são brevemente descritos a seguir.

O indivíduo é um portador do código genético, ou seja, uma representação do espaço de busca do problema a ser resolvido, em geral na forma de sequências de bits (COELHO, 1999). Neste trabalho, cada unidade de CLC proposta para controle das chaves dos multiplexadores é um indivíduo.

A partir do conceito e modelo de indivíduo, é formada a geração ou população. Uma geração é um grupo de indivíduos criados a partir do mesmo pai. A cada iteração do algoritmo genético, uma nova população será gerada. Assim, pai é o indivíduo através do qual serão criados novos indivíduos. Filho é cada um dos indivíduos gerados diretamente a partir do indivíduo pai (Srinivas; Patnaik, 1994)

Os filhos são gerados a partir de mutações inseridas no código genético do pai. Mutação é um dos estágios mais importantes dos algoritmos genéticos, devido ao seu impacto na exploração do espaço de busca da solução (HASSANAT et al., 2016).

A cada iteração do algoritmo genético, uma nova população será gerada.

Mutação são modificações aplicadas em um indivíduo pai visando a criação de um ou mais indivíduos filhos e, consequentemente, de uma nova geração.

O mapa de Karnaugh é um método gráfico que encontra um CLC correspondente a uma tabela verdade de maneira simples e para problemas de tamanho reduzido, com no máximo seis entradas (TOCCI, 1988). O método de McCuskey teoricamente é utilizado para um número ilimitado de variáveis, mas, por ser um método exaustivo na prática, acaba não sendo computacionalmente interessante, por não ser bem otimizado.

No projeto de CLCs, o tamanho do espaço de busca (tabela enunciado) cresce exponencialmente à medida que o número de variáveis de entrada do circuito aumenta e, também, proporcionalmente à medida que a dimensão da saída (número de colunas na matriz enunciado) aumenta. Quando aplicada na concepção de circuitos digitais o problema da escalabilidade tem limitado a obtenção de circuitos mais complexos, sendo apontado como o maior problema em hardware evolutivo (MANFRINI, 2017). E, como se deve satisfazer a todas as linhas dessa tabela enunciado, o problema pode ser considerado como tendo um grande número de restrições de igualdade. A complexidade de um CLC cresce com o número de componentes do circuito. Assim, uma medida geral de otimização do circuito é o número total de portas lógicas utilizadas (ALBA et al., 2007). Miller Miller et al. (1999) não faz distinção entre os custos de um MUX e o de portas lógicas, considerando ambos como apenas um elemento lógico, o que justifica a utilização de multiplexadores para resolver CLCs.

O principal objetivo deste trabalho é propor e testar um algoritmo que solucionará multiplicadores $N \times N$ através da divisão da tabela de entrada em uma árvore de multiplexadores. A criação dessa nova heurística construtiva para o projeto de CLCs visa minimizar a quantidade de elementos lógicos utilizados, reduzindo o tempo de processamento necessário para solução de problemas mais simples e possibilitando a solução de tabelas enunciado mais complexas.

Os objetivos específicos serão:

- Desenvolver um novo algoritmo para a construção de CLCs.
- Testar o algoritmo implementado no projeto de circuitos multiplicadores N x N.
- Criar CLCs e uma árvore de multiplexadores para resolver o problema.
- Cada um destes CLCs será a chave de um multiplexador da árvore.
- Evoluir os CLCs através de Algoritmo Genético Evolutivo.
- Travar portas ou repassar para os próximos níveis.

Portanto, o restante deste trabalho está organizado da seguinte forma: A Seção 2 apresenta trabalhos relacionados; a Seção 3 apresenta detalhadamente a proposta, as notações utilizadas e descreve os algoritmos e programas computacionais desenvolvidos. A Seção 4 apresenta os testes realizados e discute os resultados. Por fim, na Seção 5 , são apresentadas as conclusões e indicados os trabalhos futuros.

2 Trabalhos Relacionados

Diversos algoritmos bioinspirados têm sido propostos para o projeto de CLCs (MANFRINI, 2017), (SILVA et al., 2018) e (ALBA et al., 2007). Coello (COELLO; CHRISTIANSEN; AGUIRRE, 1999) implementou um AG com uma codificação matricial bidimensional representando o circuito. Na proposta, cada elemento, chamado de nó da matriz, representa um elemento lógico (portas AND, NOT, OR, XOR e WIRE) com suas entradas. Dessa forma, é possível representar qualquer CLC (COELLO; CHRISTIANSEN; AGUIRRE, 1999). Um ponto importante da codificação apresentada foi a utilização de um alfabeto de cardinalidade N, onde N refere-se ao número de linhas da matriz. Por isso, esse AG foi denominado NGA. A cardinalidade N permitiu a manipulação de cadeias cromossômicas mais curtas diminuindo a complexidade da codificação. O AG implementado funciona em duas etapas. Na primeira, apenas a validade das saídas é considerada e o AG explora o espaço de busca até que uma solução satisfatória apareça. Nessa primeira etapa, não é importante o quão eficiente aquele circuito realmente é em encontrar as saídas a partir da entrada. Na segunda etapa então o AG modifica o circuito buscando aprimorar e otimizar o circuito já estabelecido através de pequenas mudanças. Em (COELLO; CHRISTIANSEN; AGUIRRE, 1999), Coello propõe também um algoritmo genético multiobjetivo (MGA), onde o objetivo é resolver restrições de igualdade geradas a partir das saídas do circuito.

Analogamente, neste trabalho, foram utilizadas as portas AND, OR, NAND, NOR e XOR para a construção do CLC, com as mesmas funcionalidades do trabalho mencionado acima. A codificação utilizada não é sensível ao tamanho da cadeia cromossômica e seu tamanho tem pouco impacto no desempenho dos programas desenvolvidos.

Em Moore e Venayagamoorthy (2005), foi proposto um algoritmo evolutivo híbrido, baseado no algoritmo *Quantum - Inspired Evolutionary Algorithm (QEA)* (HAN; KIM, 2002) e na técnica de otimização por enxame de partículas (PSO). Essa implementação se mostrou eficiente e os circuitos produzidos foram equivalentes aos obtidos no MGA. Este trabalho utiliza o conceito de *quantum* bit e superposição de estados. Em lugar de lógica binária discreta clássica, QEA usa Q-bits, representados por uma função de probabilidade. Os indivíduos são cadeias de Q-bits e a sobreposição de estados gera uma maior diversidade dentro das populações. Os resultados obtidos foram eficientes na redução do número de portas existentes no CLC projetado, mas os testes realizados foram feitos com tabelas verdade muito menores (3 ou 4 entradas e uma saída) do que as usadas neste trabalho.

Otimização por Colônia de Formigas (ACO, Ant Colony Optimization) também tem sido aplicada ao projeto de CLCs. Em alguns exemplos mostrados em Coello e Mendoza (2002), o ACO superou o AG. Mostafa propõe em Abd-El-Barr et al. (2003) um ACO modificado. Pois comenta que no ACO, além dos agentes (Formigas) cooperando entre

si, características de outras heurísticas também podem ser facilmente incorporadas para melhorar a solução. O ACO apresenta uma vantagem sobre o algoritmo genético quando o grafo muda dinamicamente. A colônia de formigas pode mudar várias vezes e se adaptar às mudanças em tempo real. Entretanto, para a solução de tabelas enunciado estáticas o AG é menos complexo e se mostrou eficiente na solução dos problemas investigados neste trabalho.

Em Coello et al. (2003) foi feito um estudo comparativo com heurísticas seriais e paralelas usadas para projetar CLCs. Constatou-se que a hibridização de um AG com um algoritmo de busca local é benéfico e que a paralelização não apenas introduz um aumento da velocidade, mas permite melhorar as soluções encontradas. O algoritmo híbrido implementado foi denominado GASA2. O trabalho aqui proposto explora efetivamente o paralelismo de processamento e a divisão do trabalho em tarefas de menor dimensão, como será descrito adiante na Seção 3.1.

Programação genética (PG) também foi aplicada para projetos de CLCs por Karakatič em Karakatič, Podgorelec e Hericko (2013). A vantagem de utilizar a PG sobre o AG consiste em evitar a convergência prematura. E os resultados encontrados foram competitivos com o NGA e MGA. Segundo Fogel em Fogel (1994) e Fogel (1995), AG podem estacionar em uma solução local que pode não ser a solução ótima e PG tem maior chance de encontrar o ótimo global. E, ainda, um AG pode encontrar uma solução correta (máximo local), mas poderia evoluir ainda mais, de modo a otimizar o circuito projetado. Conforme será discutido ao longo deste trabalho, a estagnação em soluções locais não ótimas foi enfrentada com modificações no grau de mutações a serem introduzidos em uma dada geração (Seção 3.2.3), obtendo bons resultados. Com relação à busca do máximo global, o que seria a solução ótima para o CLC que resolve a tabela enunciado, não se trata de objetivo deste trabalho. Na medida em que se buscou solucionar problemas de maior dimensão e complexidade e, ainda, em tempo computacional viável, um máximo local, que já é uma solução correta para o problema, já se constitui em um importante resultado e contribuição deste trabalho.

Além das lógicas apresentadas, um CLC pode ser projetado com o uso de multiplexadores (MUX) que possuem entradas de dados e um entrada chave para a seleção. Uma vez que qualquer função booleana pode ser implementada utilizando um MUX, este é considerado um módulo universal (ERCEGOVAC; LANG; MORENO, 1999). Uma forma de implementação de uma função booleana utilizando apenas MUX é através da decomposição de Shannon (JR, 1978). PG foi aplicada para sintetizar funções lógicas utilizando somente multiplexadores de duas entradas (MUX2) em Hernandez-Aguirre e Coello (2004). As técnicas tradicionais de projeto como Mapa de Karnaugh, Algoritmo de Quine McCluskey e a decomposição de Shannon não utilizam MUX e portas lógicas simultaneamente, mas Miller Miller et al. (1999) projetou CLCs aritméticos utilizando

multiplexadores de duas entradas (MUX2) e portas lógicas com alguns algoritmos evolutivos. O presente trabalho utilizou multiplexadores de 4 entradas, mas a codificação utilizada é consistente para multiplexadores de mais entradas.

3 Modelagem do Problema

3.1 Descrição geral

O projeto de CLCs é altamente sensível à codificação utilizada, bem como ao grau de interconectividade entre portas (COELLO et al., 2003). A determinação de uma geometria apropriada para um problema booleano não é trivial. Uma geometria grande aumenta a quantidade de dados processados desnecessariamente, enquanto uma pequena geometria pode não ser suficiente para resolver determinado problema proposto (ALBA et al., 2007). Uma estrutura de tamanho variável é interessante, mas pode gerar um crescimento descontrolado do número de multiplexadores utilizados na solução. Alba (ALBA et al., 2007) propõem uma solução gerando uma matriz na qual o número de colunas e linhas pode ser aumentado gradativamente, até que um circuito factível seja encontrado.

Diversos trabalhos publicados para o projeto de CLC através de computação evolucionista usam uma matriz bidimensional para representar um CLC. (COELLO; CHRISTIANSEN; AGUIRRE, 1999), (MILLER et al., 1999), (COELLO; CHRISTIANSEN; AGUIRRE, 1999), (KALGANOVA; MILLER, 1999), (ABD-EL-BARR et al., 2003), (COELLO; MENDOZA, 2002), (MOORE; VENAYAGAMOORTHY, 2005), (COELLO; LUNA; HERNANDEZ-AGUIRRE, 2003).

Na etapa de revisão sistemática deste trabalho, não identificou-se nenhuma proposta, utilizando AG, que tenha sido aplicada ao projeto de multiplicadores de dimensão 5×5 ou superior. A maioria dos trabalhos, quando aplicados a circuitos multiplicadores, se restringem a entradas 2×2.

Em função de problemas de escalabilidade, o projeto evolutivo de circuitos lógicos não tem sido competitivo diante de vários problemas de projeto. Do ponto de vista de escalabilidade de representação, o problema é que os cromossomos longos, que são geralmente necessários para representar soluções complexas, implicam em grandes espaços de busca que são normalmente difícil de pesquisar. Em muitos casos, mesmo um algoritmo evolutivo bem ajustado não consegue encontrar uma solução inovadora em um prazo razoável (MILLER, 2011). Segundo (STEPNEY; ADAMATZKY, 2018), existe um problema complexo a ser resolvido quanto a escalabilidade do problema, quanto mais complexo for o indivíduo solução proposto, mais trabalhoso será testá-lo. Como consequência disso, apenas uma fração do espaço de busca pode ser explorado em um tempo viável. O uso de multiplexadores é interessante pois ele decompõe o espaço de busca para que este possa ser processado parte por parte (OLIVEIRA; MANFRINI, 2020).

Neste trabalho, é proposta uma nova estrutura para o projeto de CLC via algoritmos genéticos evolutivos. Após receber a matriz de entrada do problema, conecta-se na saída

desta matriz um multiplexador com o intuito de decompor a entrada do problema em matrizes menores, recursivamente, gerando uma árvore de multiplexadores. Dessa forma, a medida que o programa soluciona cada um dos multiplexadores da árvore, ele deixa de processar aquela parte da entrada e concentra o processamento apenas na parcela do problema que ainda não foi resolvida. Assim, diminui-se drasticamente a quantidade de dados processados para que seja encontrada a solução do problema. O tamanho das instâncias a serem tratadas decresce exponencialmente. O algoritmo projetado para executar a tarefa proposta foi batizado de GAMT (Genetic Algorithm through Multiplexer Tree - Algoritmo Genético através de Árvore de Multiplexadores), e ele será descrito em detalhes no decorrer desta monografia.

O presente trabalho foi inspirado em outro projeto em desenvolvimento no IF Sudeste MG. Este projeto apresentou um algoritmo genético para a solução de CLCs. A técnica utiliza a divisão do circuito solução, com o uso de multiplexadores de duas entradas. A heurística construtiva proposta foi testada com verificador de paridade ímpar de três e quatro bits (OLIVEIRA; MANFRINI, 2020).

Além disso, sob o ponto de vista computacional, dado que as subdivisões da tabela verdade não são dependentes entre si, pode-se explorar o paralelismo de processamento. A medida em que a matriz da tabela verdade vai sendo subdividida, estas submatrizes vão sendo passadas como parâmetro a várias linhas de execução (*threads*) que passam a ser executadas concorrentemente nos processadores e núcleos disponíveis.

Multithreading é a capacidade de uma Unidade Central de Processamento (CPU) (ou de um único núcleo em um processador de vários núcleos) de fornecer várias linhas de execução ou processamento (*threads*) simultaneamente, com suporte do sistema operacional. A linguagem Java disponibiliza este nível de concorrência ou paralelismo para o programador através de sua API (*Application Programming Interface*). Este recurso fornece capacidades poderosas para o programador Java (DEITEL; DEITEL; PEARSON, 2016).

Sistemas de multiprocessamento incluem várias unidades de processamento completas e um ou mais núcleos em cada unidade. A tecnologia *multithreading* visa aumentar a utilização de um único núcleo e do conjunto de núcleos, usando paralelismo no nível de *thread* (linhas de execução distintas dentro de um processo), bem como paralelismo no nível de instrução. Como as duas técnicas são complementares, às vezes são combinadas em sistemas com várias CPUs *multithreading* e sistemas com CPUs com vários núcleos cada uma. A linguagem Java explora toda a capacidade do sistema operacional de escalonar múltiplas *threads*. Ela aloca as diversas linhas de execução em todos os núcleos da CPU e em todas as linhas de execução *threads* existentes em cada núcleo ((PATTERSON; HENNESSY, 2007) e (TANENBAUM, 2010)).

Em sua API para programação concorrente, Java disponibiliza o modificador

synchronized, que coordena o acesso de múltiplas linhas de execução a trechos de código e objetos que não podem ser acessados simultaneamente. Este modificador foi muito utilizado, de modo a manter a consistência de dados globais gerenciados pelo coordenador das *threads* e acessados concorrentemente pelas linhas de execução paralelas (*threads* filhas) (DEITEL; DEITEL; PEARSON, 2016).

A entrada do problema é uma matriz que representa uma tabela verdade com N entradas e S saídas. Estes valores N e S são definidos previamente, como parâmetros do problema, podendo assumir quaisquer valores desejados.

Considere, como exemplo, um circuito multiplicador de duas palavras com N_1 e N_2 bits, respectivamente. Deste modo, a tabela verdade terá $N = 2^{(N_1+N_2)}$ linhas e $N_1 + N_2$ colunas. Já a saída, terá o mesmo número N de linhas e $N_1 + N_2$ colunas. A Tabela 1 ilustra a tabela verdade do circuito multiplicador para $N_1 = 2$, $N_2 = 3$ e $S = 5$.

A primeira divisão de tarefas do algoritmos proposto ocorre ao se resolver cada uma das M colunas da saída separadamente. Sejam cada uma dessas colunas denotadas por S_i $(0 \leq i < S)$. É gerado um circuito digital de portas lógicas e multiplexadores para cada coluna S_i da saída desejada.

Uma vez tomada uma coluna S_i a ser solucionada, cada linha da matriz será ligada a uma das entradas de um multiplexador (MUX). O número de portas deste MUX é parâmetro configurável nos programas desenvolvidos. Neste trabalho, todos os testes foram feitos com MUX de 4 entradas ($I_0,\ I_1,\ I_2\ e\ I_3$) e, consequentemente, 2 bits de chaveamento ($C_0\ e\ C_1$). Um dos pontos que difere este trabalho da proposta apresentada em (OLIVEIRA; MANFRINI, 2020), é a possibilidade da utilização de multiplexadores de mais entradas (4, 8, etc) e o uso efetivo, nos testes, dos de quatro entradas.

No algoritmo proposto, um indivíduo é um conjunto de CLCs que gera, a partir da entrada, valores booleanos que são a chave do MUX. O número de circuitos do indivíduo será igual à quantidade de bits da chave do MUX, um CLC responsável por gerar cada bit da chave. Cada MUX da árvore possui um indivíduo solução distinto ligado a sua chave.

Para criação do CLC do indivíduo (Figura 3), é necessário definir qual será a matriz topologia utilizada no problema. Tomamos cada um dos bits de N_1 e N_2 como um nó de entrada e definimos uma malha $m \times n$ com o número de nós complementares desejados, como na Figura 2 ($N_1 = N_2 = 3$).

O indivíduo é representado por uma matriz, sendo cada linha da matriz um CLC que terá como resposta um dos bits da chave do MUX. Cada uma dessas linhas é composta por uma posição de ID; N grupos de 3 posições (um grupo para cada nó da matriz topologia, contendo dois nós anteriores aleatórios e a porta lógica que combinará esses nós); uma posição saída do circuito, contendo o valor do bit que será ligado à posição C_i da chave do MUX; e uma posição com a avaliação daquele indivíduo.

Figura 2 – Como é formado um indivíduo

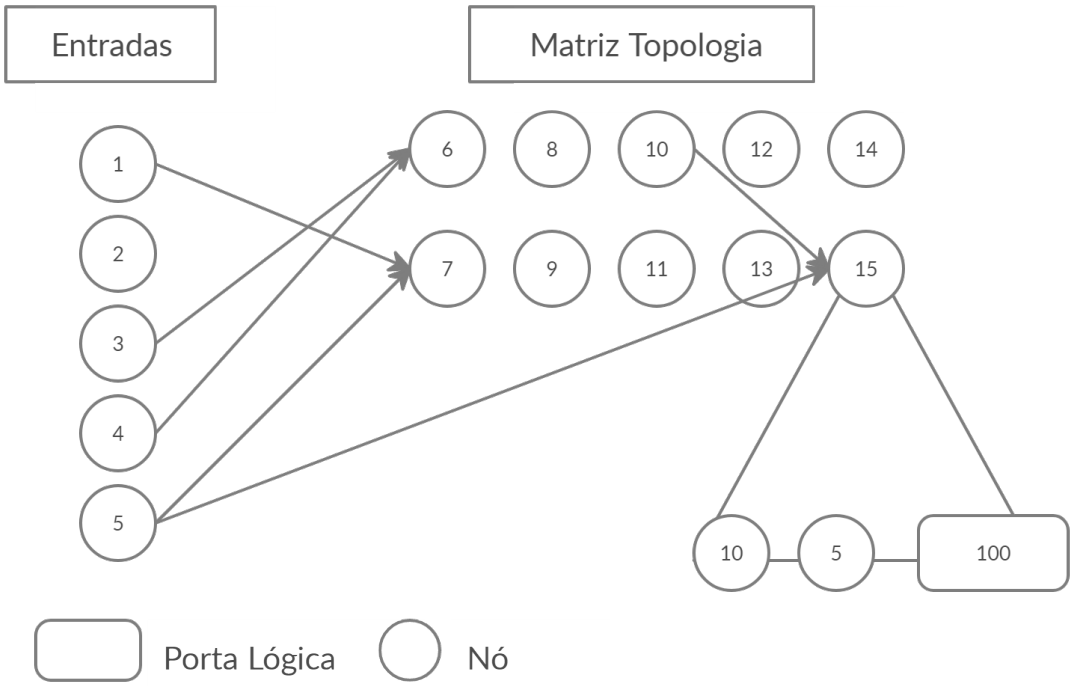


Figura 3 – Representação de um CLC de um indivíduo

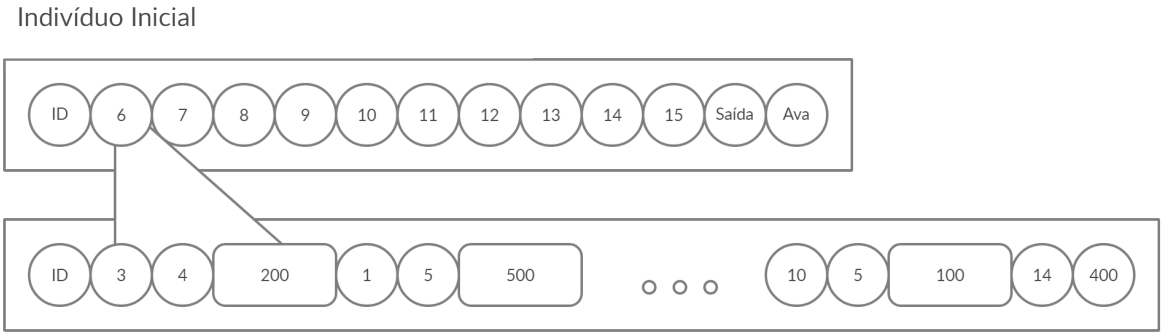


Tabela 1 – Tabela verdade do multiplicador 2×3

N_1		N_2			S				
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	1
0	1	0	1	0	0	0	0	1	0
0	1	0	1	1	0	0	0	1	1
0	1	1	0	0	0	0	1	0	0
0	1	1	0	1	0	0	1	0	1
0	1	1	1	0	0	0	1	1	0
0	1	1	1	1	0	0	1	1	1
1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	0
1	0	0	1	0	0	0	1	0	0
1	0	0	1	1	0	0	1	1	0
1	0	1	0	0	0	1	0	0	0
1	0	1	0	1	0	1	0	1	0
1	0	1	1	0	0	1	1	0	0
1	0	1	1	1	0	1	1	1	0
1	1	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	1	1
1	1	0	1	0	0	0	1	1	0
1	1	0	1	1	0	1	0	0	1
1	1	1	0	0	0	1	1	0	0
1	1	1	0	1	0	1	1	1	1
1	1	1	1	0	1	0	0	1	0
1	1	1	1	1	1	0	1	0	1

Por exemplo, na Figura 2, o nó 7 da matriz topologia tem como entradas os nós 1 e 5 e a porta lógica **OR**, representada pelo número 200 (1, 5 e 300, na linha da matriz indivíduo). Ainda na mesma figura, observa-se que a saída do circuito está no nó 14, que está na penúltima posição da linha da matriz indivíduo. Durante a execução do algoritmo, o indivíduo tem a sua avaliação gravada da última posição da linha.

A Tabela 2 representa uma matriz indivíduo de um circuito multiplicador 3×3, uma matriz topologia 2×2, com uso de multiplexadores de quatro entradas (chave de dois bits, por isso a matriz tem duas linhas).

O objetivo do indivíduo é gerar o circuito que, através dos bits de cada linha da

Tabela 2 – Exemplo de um indivíduo

ID	Nó	Nó	Porta	Nó	Nó	Porta	Nó	Nó	Porta	Nó	Nó	Porta	Saída	Avaliação
1	6	3	200	2	3	400	5	8	300	7	8	300	5	3256
1	5	1	100	1	4	300	7	3	200	5	2	100	9	3256

tabela verdade de entrada, gera uma resposta correspondente a uma das chaves possíveis para o MUX, 00, 01, 10 e 11, no caso de um MUX de quatro entradas (MUX4). Este circuito gerado pelo indivíduo se conecta diretamente à chave do MUX, travando em cada uma das entradas (I_0, I_1, I_2 *e* I_3) do MUX o bit necessário para alcançar a saída desejada. Dessa forma, a matriz entrada do problema é subdividida pela quantidade de entradas de dados que tivermos no MUX (2 para um MUX de duas entradas, 4 para um MUX de quatro entradas, etc...).

Para solucionar a primeira coluna da saída da Tabela 1 (S_0), foi gerada uma nova tabela apenas com a coluna que desejamos trabalhar naquele momento, representada na Tabela 3. Um novo indivíduo é gerado e, através dos CLCs contidos nele, são gerados valores dos dois bits de chave para cada uma das linhas da matriz problema. Esta tabela é, então, subdividida, através do valor da chave, gerando quatro matrizes menores representadas nas Tabelas 4, 5, 6 e 7.

É possível observar que, nas Tabelas 4, 5 e 6, a saída S_0 tem valor 0 independente dos valores da entrada. Isso nos possibilita dizer que para o indivíduo gerado, sempre que tivermos a chave em 00, 01 ou 10 a resposta do problema será 0, ou seja, podemos travar um 0 nas entradas I_0, I_1 e I_2 do MUX4. Por outro lado, ao analisar a Tabela 7, podemos notar que ainda temos linhas com $S_0 = 0$ e linhas com $S_0 = 1$, gerando assim necessidade da aplicação de mais um MUX4 nessa entrada. É gerado um novo indivíduo para este novo MUX4, apresentando novas combinações de chave e dividindo esta Tabela 8 mais uma vez em até quatro outras tabelas (Tabelas 9, 10 e 11).

Nas Tabelas 9, 10 e 11, podemos notar que $S_0 = 0$ para a chave 00, $S_0 = 1$ para a chave 10 e $S_0 = 0$ para a chave 11. É importante dizer que o número de chaves que será utilizado pode não ser igual ao número de chaves disponíveis. Nesse segundo MUX4 criado, podemos notar que a chave 01 não foi utilizada. Isso quer dizer que o CLC do indivíduo nunca apresentará valor igual a 01 e, portanto, essa chave deste MUX4 não será utilizada. Travamos então, nas portas de entradas desse segundo MUX4 criado, 0 na porta I_0, X (irrelevante) na porta I_1, 1 na porta I_2 e 0 na porta I_3.

Observa-se também que a medida que as tabelas são decompostas e travamos o bit 0 ou 1 nas entradas de cada MUX4, o programa fica com matrizes cada vez menores a serem trabalhadas. Isso possibilita diminuir a quantidade de dados sendo processada em um dado instante no decorrer do programa, pois não é necessário reavaliar toda a matriz entrada para cada novo indivíduo formado.

Tabela 3 – Tabela verdade do multiplicador 2×3

N_1		N_2			C_0	C_1	S_0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0
0	0	0	1	0	0	1	0
0	0	0	1	1	1	1	0
0	0	1	0	0	1	1	0
0	0	1	0	1	0	0	0
0	0	1	1	0	0	1	0
0	0	1	1	1	0	0	0
0	1	0	0	0	0	0	0
0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0
0	1	0	1	1	1	1	0
0	1	1	0	0	1	1	0
0	1	1	0	1	1	1	0
0	1	1	1	0	1	0	0
0	1	1	1	1	0	0	0
1	0	0	0	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0
1	0	0	1	1	1	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	0	1	1	1	0	1	0
1	1	0	0	0	0	1	0
1	1	0	0	1	0	0	0
1	1	0	1	0	0	0	0
1	1	0	1	1	1	1	0
1	1	1	0	0	0	0	0
1	1	1	0	1	0	1	0
1	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1

Tabela 4 – Subtabela para a chave 00

N_1		N_2			C_0	C_1	S_0
0	0	0	0	0	**0**	**0**	0
0	0	1	0	1	**0**	**0**	0
0	0	1	1	1	**0**	**0**	0
0	1	0	0	0	**0**	**0**	0
0	1	1	1	1	**0**	**0**	0
1	0	0	0	1	**0**	**0**	0
1	0	1	0	0	**0**	**0**	0
1	1	0	0	1	**0**	**0**	0
1	1	0	1	0	**0**	**0**	0
1	1	1	0	0	**0**	**0**	0

Tabela 5 – Subtabela para a chave 01

N_1		N_2			C_0	C_1	S_0
0	0	0	1	0	**0**	**1**	0
0	0	1	1	0	**0**	**1**	0
1	0	0	1	0	**0**	**1**	0
1	0	1	1	1	**0**	**1**	0
1	1	0	0	0	**0**	**1**	0
1	1	1	0	1	**0**	**1**	0

Tabela 6 – Subtabela para a chave 10

N_1		N_2			C_0	C_1	S_0
0	1	0	0	1	**1**	**0**	0
0	1	0	1	0	**1**	**0**	0
0	1	1	1	0	**1**	**0**	0
1	0	0	0	0	**1**	**0**	0
1	0	0	1	1	**1**	**0**	0
1	0	1	1	0	**1**	**0**	0

Tabela 7 – Subtabela para a chave 11

N_1		N_2			C_0	C_1	S_0
0	0	0	0	1	**1**	**1**	0
0	0	0	1	1	**1**	**1**	0
0	0	1	0	0	**1**	**1**	0
0	1	0	1	1	**1**	**1**	0
0	1	1	0	0	**1**	**1**	0
0	1	1	0	1	**1**	**1**	0
1	0	1	0	1	**1**	**1**	0
1	1	0	1	1	**1**	**1**	0
1	1	1	1	0	**1**	**1**	1
1	1	1	1	1	**1**	**1**	1

Deste modo, problemas com tempo de execução extremamente elevado, com o uso do GAMT, passam a apresentar considerável redução do tempo computacional. Já outros problemas, podem nunca convergir para uma solução, enquanto o GAMT sempre irá

Tabela 8 – Tabela de entrada para o MUX4 secundário

N_1		N_2			C_2	C_3	S_0
0	0	0	0	1	1	1	0
0	0	0	1	1	0	0	0
0	0	1	0	0	1	1	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	0	0
0	1	1	0	1	1	1	0
1	0	1	0	1	0	0	0
1	1	0	1	1	1	1	0
1	1	1	1	0	1	0	1
1	1	1	1	1	1	0	1

Tabela 9 – Subtabela para a chave 00

N_1		N_2			C_2	C_3	S_0
0	0	0	1	1	**0**	**0**	0
0	1	0	1	1	**0**	**0**	0
0	1	1	0	0	**0**	**0**	0
1	0	1	0	1	**0**	**0**	0

Tabela 10 – Subtabela para a chave 10

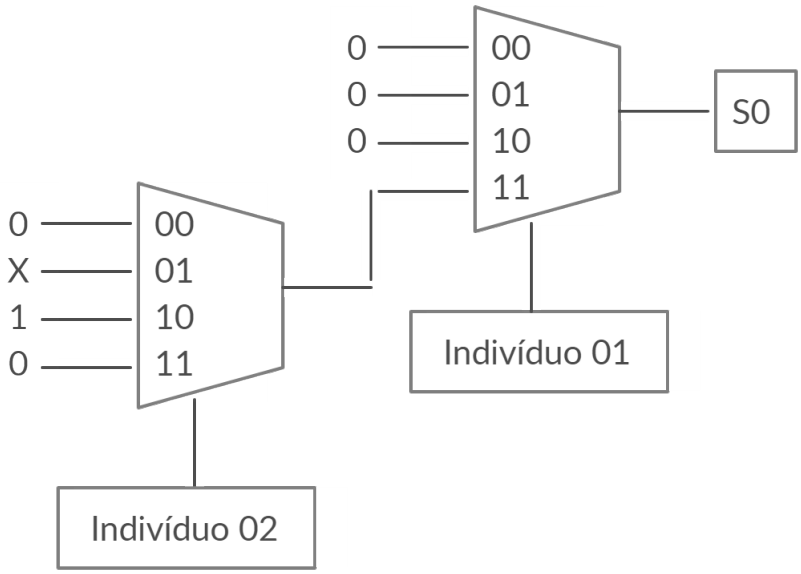
N_1		N_2			C_2	C_3	S_0
1	1	1	1	0	**1**	**0**	1
1	1	1	1	1	**1**	**0**	1

Tabela 11 – Subtabela para a chave 11

N_1		N_2			C_2	C_3	S_0
0	0	0	0	1	**1**	**1**	0
0	0	1	0	0	**1**	**1**	0
0	1	1	0	1	**1**	**1**	0
1	1	0	1	1	**1**	**1**	0

convergir para uma solução correta, utilizando tantos multiplexadores quanto necessário.

Figura 4 – Árvore de MUX resposta do problema exemplo



3.2 Evolução do Indivíduo

Na seção anterior, foi possível entender como o AG se comporta na subdivisão da matriz enunciado em uma árvore de multiplexadores, a partir dos bits da chave do primeiro MUX (C_0 e C_1) e do MUX secundário (C_2 e C_3). Entretanto, estes valores utilizados foram gerados de forma aleatória, exclusivamente para compreensão do problema. Nesta seção,

será explicado o processo de evolução de cada um desses indivíduos e, consequentemente, o processo de obtenção dos valores de chave que satisfaçam o problema proposto.

Como dito anteriormente, cada MUX da árvore possui um indivíduo diferente responsável pela solução daquele MUX. Cada um desses indivíduos é representado por uma matriz bidimensional. Cada linha desta matriz representa um CLC distinto (responsável por um bit da chave daquele MUX) e contém as instruções necessárias para construir esse CLC (topologia, portas lógicas e saída).

Para cada MUX, o GAMT cria um novo indivíduo e segue uma rotina padrão (Figura 5), por um número arbitrário de gerações, com o objetivo de encontrar a melhor combinação possível de chaves para aquela etapa.

3.2.1 Avaliação do Indivíduo

Diferentes formas de avaliar os indivíduos e selecionar o melhor deles são encontradas na literatura. A seleção do indivíduo mais eficiente, entre um grupo de candidatos, é importante, pois é através dela que as técnicas evolutivas se movem pelo espaço de busca (SILVA et al., 2018). Por isso, a implementação de um processo de avaliação específico para a heurística utilizada é importante.

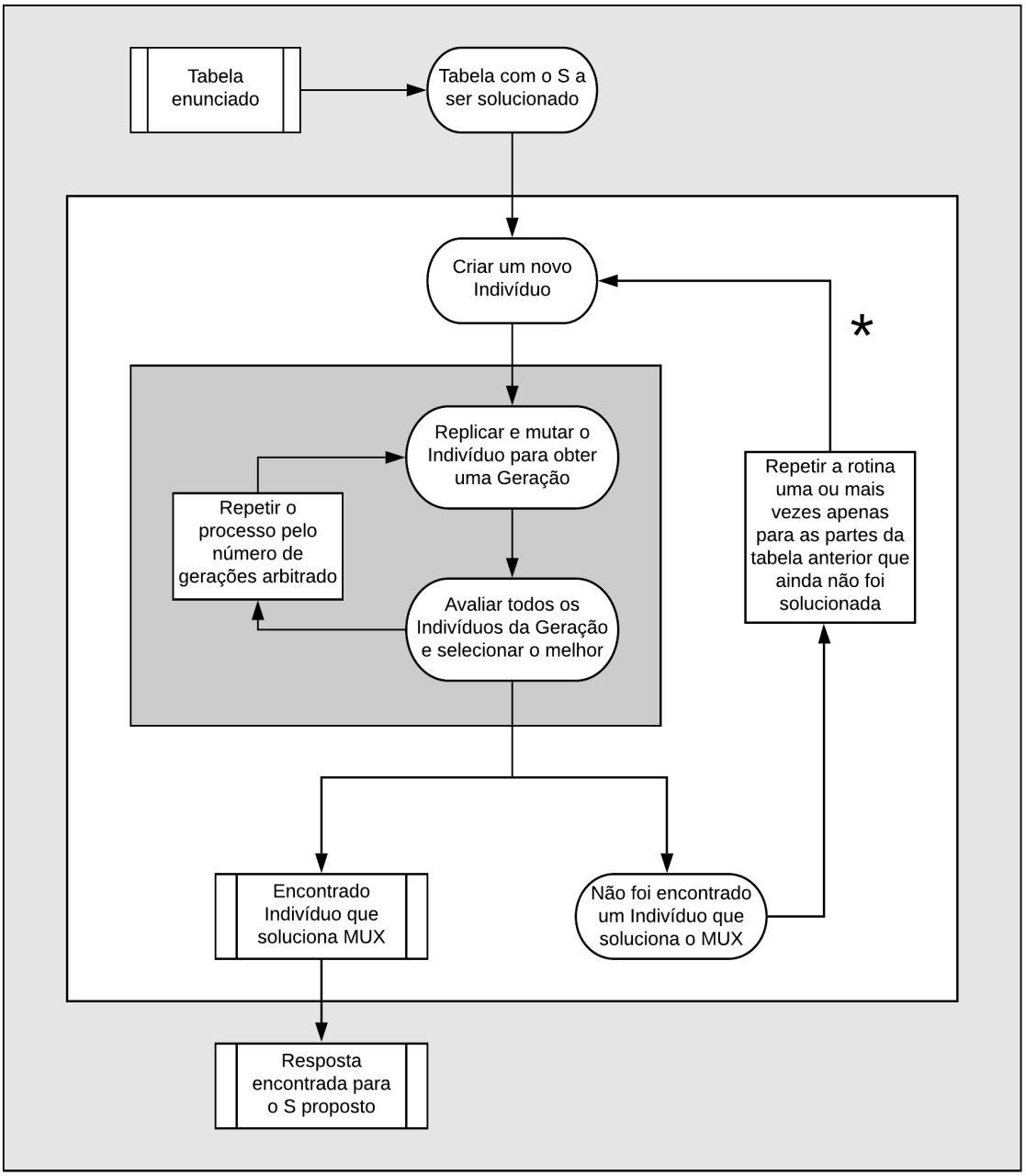
Dadas as particularidades do algoritmo proposto neste trabalho e às poucas referências similares encontradas na literatura, o processo de avaliação é novo e foi ajustado e refinado ao longo da evolução do desenvolvimento.

De cada geração, é extraído o indivíduo com melhor pontuação e toda a próxima geração será fruto de modificações neste indivíduo. Para o caso excepcional de um empate, o indivíduo mais novo tomará a posição de melhor indivíduo e criará a nova geração. Isso possibilita uma maior variabilidade no processo de evolução.

O método de avaliação elaborado para se ajustar a heurística proposta neste trabalho foi desenvolvido totalmente parametrizado nas grandezas que conduzem o código. Para avaliar um indivíduo, é necessário gerar seu circuito e analisar os valores de chave gerados por ele, separando a matriz em partes. Uma vez que o objetivo do indivíduo é travar portas do multiplexador em 1 ou em 0, em cada submatriz, busca-se, na coluna solução S_i, a quantidade de vezes que o valor 1 aparece e a quantidade de vezes que o valor 0 aparece. Existem então quatro possibilidades de resposta para cada submatriz:

1. Ocorrência de ambos os bits como resposta, com um número de zeros diferente do número de uns.
2. Ocorrência de ambos os bits como resposta, com um número de zeros igual ao número de uns.
3. Ocorrência de apenas um dos bits como resposta. Apenas zeros ou apenas uns.

Figura 5 – Fluxograma da Rotina de evolução do Indivíduo



4. Não há ocorrência dos bits. Chave não utilizada.

Para que um indivíduo seja bem sucedido, ele deve conseguir travar um bit, 0 ou 1, nas entradas do MUX. Para conseguir travar algum desses bits na entrada do MUX, precisamos que, na coluna S dessa submatriz, apareçam exclusivamente zeros ou exclusivamente uns. Quanto mais entradas este indivíduo for capaz de travar, melhor será seu desempenho, e, consequentemente, melhor será sua avaliação. Para cada uma das três possibilidades de resposta citadas na lista acima, existe uma pontuação que será atribuída ao indivíduo.

Deste modo, a Avaliação parte de zero e é modificada da seguinte maneira:

1. Ocorrência de ambos os bits como resposta, com um número de zeros diferente do número de uns.

$$\text{Avaliação} = \text{Avaliação} + 2 \times \left(\frac{QTDBitMaisFrequente}{QTDBitMenosFrequente}\right) \tag{3.1}$$

   A avaliação é incrementada segundo a relação de predominância de um dos bits em relação ao outro. Quanto mais predominante for um dos bits, maior será esta relação e melhor deve ser a avaliação do Indivíduo. Isto indica que ele está evoluindo para travar aquela porta. Este valor é multiplicado por dois para distanciá-lo da avaliação do caso 2, uma vez que ela é consideravelmente mais interessante para a resolução do problema.

2. Ocorrência de ambos os bits como resposta, com um número de zeros igual ao número de uns.

$$\text{Avaliação} = \text{Avaliação} + 1 \tag{3.2}$$

   Para o caso de o número de zeros e de uns ser o mesmo, constata-se que o CLC do indivíduo não está próximo de travar nenhum dos bits naquela porta do MUX. Este é o pior caso possível. Dessa forma, a avaliação será incrementada em apenas uma unidade.

3. Ocorrência de apenas um dos bits como resposta. Apenas zeros ou apenas uns.

$$\begin{aligned}\text{Avaliação} = \text{Avaliação} &+ \left(QTDEntradasMUX^2\right) \\ &+ \left(\left(\frac{QTDLinhasMatrizAtual}{N1 * N2}\right) * QTDBitPresente\right)\end{aligned} \tag{3.3}$$

   Neste caso, temos a ocorrência de apenas um dos bits. Isso mostra que o indivíduo teve sucesso em travar este bit na porta do MUX. É necessário, então, avaliar bem este indivíduo. Uma vez que a quantidade de linhas da matriz é um valor elevado quando comparado com quaisquer outras constantes do problema, ele é um bom parâmetro

para premiar o indivíduo. Mas é vantajoso limitar este prêmio em função da dimensão da entrada, por isso é aplicada a razão $QTDLinhasMatrizAtual/(N1 \times N2)$. Por outro lado, quanto maior a quantidade total do bit travado, mais linhas da matriz estão sendo solucionadas. Por isso, o fator multiplicador ao final da expressão acima. Por fim, é somado o valor da quantidade de portas do MUX utilizado ao quadrado. Isso impede uma avaliação pequena para quando a matriz tem poucas linhas.

4. Não há ocorrência dos bits. Chave não utilizada.

$$\text{Avaliação} = \text{Avaliação} + \sqrt[3]{NúmeroColunasTabelaVerdade} \tag{3.4}$$

Para o último caso, não existe ocorrência de nenhum dos bits. É possível concluir que a chave em questão não está sendo utilizada. A raiz cúbica do número de colunas é um valor sempre maior que 1 e sempre pequeno. Este é o valor adequado a ser somado à avaliação, pois é significativamente pior do que o caso ideal 3. É melhor do que o caso 2, pois é preferível não utilizar uma porta do MUX do que utilizá-la de forma ineficiente. Já quando comparado ao caso 1, para valores próximos de $QTDBitMaisFrequente$ e $QTDBitMenosFrequente$, o caso 4 terá avaliação superior. E quanto mais distantes forem os valores de $QTDBitMaisFrequente$ e $QTDBitMenosFrequente$, melhor será a avaliação do caso 1 em relação ao caso 4.

3.2.2 Mutação

Para criar uma nova geração, um processo de reprodução e mutação é aplicado no melhor indivíduo da geração anterior. Essa mutação feita é um processo onde uma posição da matriz bidimensional do indivíduo é selecionada aleatoriamente e essa posição é modificada, dentro dos padrões do programa. Caso seja uma posição contendo uma porta lógica, outra porta lógica será sorteada para tomar o seu lugar. Caso seja um nó de origem de dados, outro nó será selecionado para que substitua o antigo. E caso seja um nó de saída, um novo nó indicando a saída do CLC será sorteado. Suponhamos que em dada geração tenhamos 3 indivíduos (Ind_1, Ind_2 e Ind_3, avaliados em 20, 50 e 80 pontos, respectivamente). O indivíduo Ind_3, por possuir a melhor pontuação, será selecionado para ser o pai da próxima geração. A nova geração será composta por réplicas modificadas deste Ind_3 e pelo próprio Ind_3. Próxima geração: Ind_3, $Ind_3.1$ e $Ind_3.2$, onde $Ind_3.1$ e $Ind_3.2$ são modificações do Ind_3.

É importante manter uma cópia idêntica do melhor indivíduo da geração anterior (Ind_3), pois, dessa forma, é possível garantir que não haja retrocesso no potencial do indivíduo. Caso as mutações resultem em indivíduos com avaliações ruins, o melhor indivíduo da geração será novamente Ind_3 e novas modificações serão feitas para obter ainda mais uma geração. Caso surja um filho com avaliação melhor do que a do pai, este será o pai da nova geração e assim por diante.

É determinado um valor antes da execução do programa, que será o número máximo de mutações por indivíduo. Para um número máximo de mutações 3, o GAMT randomiza, para cada indivíduo, um número entre 1 e 3 e executa aquela quantidade de mutações naquele indivíduo. Por conta disso, enquanto alguns indivíduos são muito parecidos com o indivíduo do qual foram gerados, outros resultam em CLCs mais distantes. Isso possibilita que, em caso de estarmos trabalhando com um bom indivíduo, apenas alguns ajustes sejam feitos. E, no caso de estarmos trabalhando com um indivíduo com um desempenho ruim, mudanças mais consideráveis sejam feitas acelerando o aprimoramento do indivíduo.

Nas Tabelas 12 e Tabelas 13 é possível identificar um caso hipotético de três mutações aplicadas em um indivíduo. Observa-se, na segunda linha da matriz, a segunda porta do primeiro nó sofreu uma modificação aleatória de 1 para 3. Também a porta lógica do terceiro nó foi mutada de 300 (NAND) para 400 (NOR) e por fim, a terceira mutação modificou o valor da porta de saída de 9 para 10. Ao avaliar este novo indivíduo é gerado um novo *rank* que no caso se mostrou ser maior que o anterior.

Tabela 12 – Exemplo de um indivíduo

ID	Nó	Nó	Porta	Nó	Nó	Porta	Nó	Nó	Porta	Nó	Nó	Porta	Saída	Avaliação
1	6	3	200	2	3	400	5	8	300	7	8	300	5	3200
1	5	1	100	1	4	300	7	3	200	5	2	100	9	3200

Tabela 13 – Exemplo de um indivíduo

ID	Nó	Nó	Porta	Nó	Nó	Porta	Nó	Nó	Porta	Nó	Nó	Porta	Saída	Avaliação
2	6	3	200	2	3	400	5	8	**400**	7	8	300	5	**3500**
2	5	**3**	100	1	4	300	7	3	200	5	2	100	**10**	**3500**

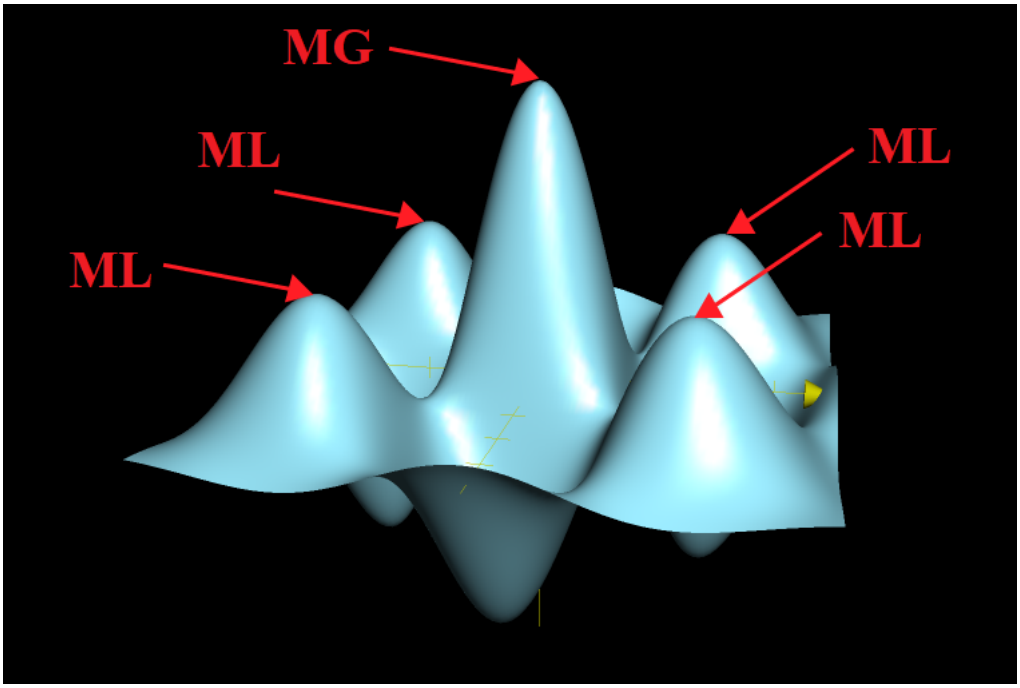
3.2.3 Evitando máximos locais

O processo explicado garante que, a cada geração, o indivíduo se aprimore, ou permaneça o mesmo. Dessa forma, o algoritmo caminha para uma solução. Entretanto, aplicando essa técnica, um problema de estagnação em pontos de máximo local pode ser encontrado. Tratam-se de indivíduos com boa avaliação até o momento, mas que não solucionam o problema. Deste modo, as mutações podem ser insuficientes para gerar indivíduos melhores, mesmo que em um número grande de gerações.

Caso o ponto de máximo local encontrado não produza uma avaliação do indivíduo elevada o suficiente para solucionar o MUX, o algoritmo não irá convergir para uma resposta que solucione a tabela enunciado naquele ponto da árvore (naquele MUX).

Para solucionar este problema, foi proposta e implementada uma técnica que possibilita a criação de novos indivíduos com um número superior de mutações. Em outras palavras, mutações que gerem um maior afastamento das regiões de máximo local, de

Figura 6 – Gráfico genérico de pontos de Máximo Local e Global



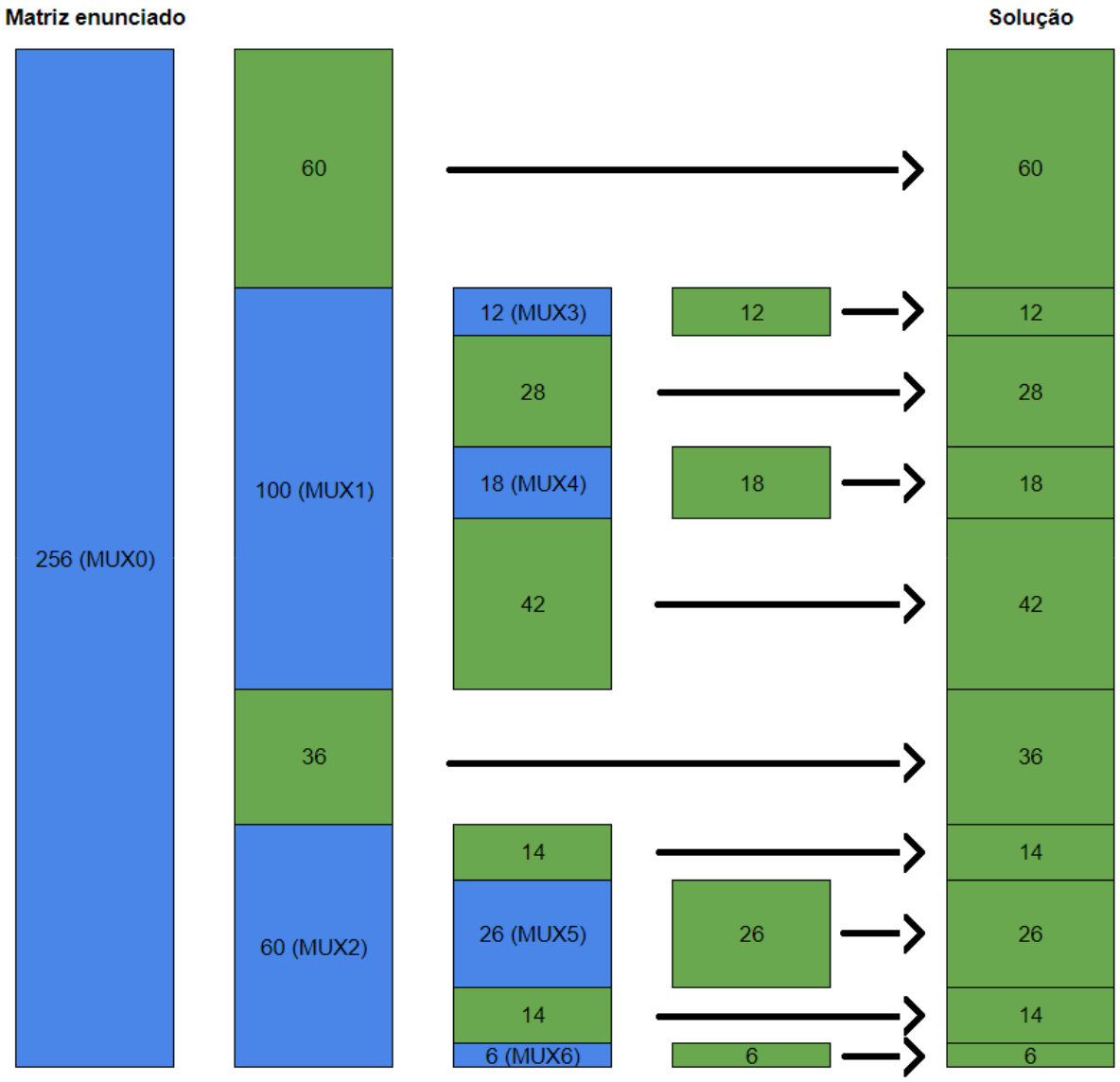
modo a aumentar a probabilidade de gerar um nova sequência de evoluções. Em caso de muitas gerações sem nenhum indivíduo que supere o melhor até o momento, o número de mutações efetuadas é incrementado linearmente, durante uma certa quantidade de gerações. Para um caso hipotético de 3 mutações na criação de cada novo indivíduo, o GAMT responderia usando 6 mutações durante um número predeterminado de novas gerações, depois 9, 12, etc... O algoritmo retorna para o número padrão de mutações após um número predefinido de gerações ou assim que um indivíduo for capaz de vencer o máximo local.

A busca de uma solução global para toda a tabela enunciado, para problemas de maiores dimensões, pode estacionar indefinidamente em máximos locais. O GAMT, após um longo período de evoluções e, mesmo aplicando esta técnica de fuga do máximo local descrita, tem um limite de exploração do espaço de soluções. Ao atingir este limite, o algoritmo proposto incorpora a solução para a maior faixa da tabela verdade encontrada até o momento, insere um ou mais MUX e reinicia o processo em um ou mais problemas menores.

A Figura 7, extraída de um dos testes reais executados, mostra esta evolução da árvore, com base nas linhas travadas (em verde) e nas linhas repassadas (em azul) para os próximos multiplexadores. No multiplicador 4×4, o primeiro MUX recebe todas as linhas (256). No caso ilustrado, duas portas do multiplexador foram travadas (uma com 60 linhas e a outra com 36). As demais 160 linhas foram repassadas a dois novos multiplexadores (MUX1 e MUX2). O MUX1 também travou duas portas (28 e 42 linhas) e repassou 12 e

18. MUX2 travou duas de 14 e repassou 26 e 6. Na extrema direita da figura, estão os linhas da solução completa e de qual multiplexador saem as soluções.

Figura 7 – Linhas resolvidas (travadas) e linhas repassadas aos próximos MUX



No próximo capítulo, serão descritos e discutidos os programas desenvolvidos e os experimentos realizados.

4 Estrutura do Sistema, Testes e Resultados

Este capítulo descreve a estrutura dos códigos do GAMT. Em seguida, são analisadas as execuções de instâncias dos multiplicadores de dimensões 2×2, 3×3 e 4×4.

4.1 Desenvolvimento do GAMT

O algoritmo projetado para resolver o problema da forma proposta no Capítulo 3 foi desenvolvido em Java e foi implementado em duas etapas. A princípio, foi desenvolvido um algoritmo capaz de subdividir a matriz de entrada apenas uma vez. Posteriormente, na segunda etapa, foi aplicada a técnica de recursividade, que permitiu a repetição da primeira etapa diversas vezes gerando a árvore de multiplexadores. Na primeira etapa, diversas funcionalidades do sistema foram observadas, melhoradas e confirmou-se a indicação da necessidade de novos MUX nas camadas seguintes da árvore de multiplexadores.

Em virtude da estrutura de árvore do GAMT, em diversos estágios do programa, há a possibilidade de se iniciar duas ou mais tarefas independentes. Deste modo, assim que um MUX é resolvido, todos os MUX que serão ligados a ele, adiante na árvore, têm suas soluções iniciadas imediatamente, em paralelo. Por este motivo, o programa foi implementado na linguagem Java. Assim, foi possível explorar o processamento em múltiplas linhas de execução concorrentes (*multithread*). O desenvolvimento da programação do GAMT em Java levou em consideração essa possibilidade de paralelismo. A estrutura de alto nível do sistema é descrita a seguir.

Na Figura 8, é apresentado o pseudocódigo do Coordenador (classe principal). Quando iniciada a execução, um arquivo de configuração **setup.txt** é lido. Ele contém todos os parâmetros necessários à execução. Seus campos são descritos na Seção 4.2. Em seguida, é solicitado ao usuário que informe qual coluna S_i da saída deve ser resolvida. Então, a matriz enunciado é gerada com a tabela verdade da entrada e apenas a coluna a ser resolvida. Uma nova instância da classe Ajudante (descrita a seguir) é iniciada e o contador de MUX sendo executados (MUXES_EM_ANÁLISE) é incrementado. Como será visto a seguir, os próprios Ajudantes vão chamando outros, recursivamente, e incrementando, no Coordenador, o valor de MUXES_EM_ANÁLISE. Cada Ajudante, ao concluir sua execução, decrementa este contador. Ao identificar que este valor chegou a zero (linha 8), o Coordenador sabe que o problema foi resolvido, salva os dados gerais da execução e termina.

O pseudocódigo do Ajudante está representado na Figura 9. A primeira instância da classe Ajudante recebe do Coordenador a tabela enunciado com a coluna S_i a ser solucionada. Este Ajudante inicia o processo de solução dessa matriz através da criação de um novo indivíduo e de uma primeira geração. O código, então, entra em um laço (linha 5), do qual só sairá após o número máximo de gerações ser alcançado ou o MUX

Figura 8 – Pseudocódigo do Coordenador

```
1
2  Início
3      Ler setup.txt
4      Ler Si
5      Gerar Matriz Enunciado
6      Iniciar Ajudante para resolver MUX0
7      Incrementa MUXES_EM_ANALISES
8      Enquanto MUXES_EM_ANALISES > 0 {
9          Aguarda Ajudantes...
10     }
11     Grava dados gerais da execução
12 Fim
13
```

ser solucionado. Dentro deste laço, a geração atual sofre mutação, cada indivíduo que ela contém é avaliado e é encontrado o indivíduo com maior *rank* (melhor indivíduo da geração). Este indivíduo, na próxima rodada do laço, da origem à próxima geração. Ainda dentro deste laço, o Ajudante avalia a necessidade de utilizar a técnica que evita os máximos locais. Após ter obtido o indivíduo solução do MUX, existem dois caminhos que podem ser seguidos. O primeiro se restringe a informar ao Coordenador que não existe nenhuma linha da tabela recebida sem solução (Coordenador decrementa MUXES_EM_ANÁLISE). O segundo caminho ocorre quando ainda existem linhas pendentes. Neste caso, o Ajudante calcula quantos novos MUX são necessários para a próxima etapa e fragmenta o restante da tabela ainda não resolvido em novas tabelas que serão repassadas para os novos objetos Ajudante. Após informar ao coordenador que o problema foi resolvido ou chamar novas instâncias de Ajudante, os dados são gravados e os recursos utilizados por esta *thread* são liberados. Somente após um destes dois passos, o Ajudante é terminado.

Figura 9 – Pseudocódigo do Ajudante

```
1
2  Início
3      Cria Indivíduo
4      Cria Geração
5      Enquanto (numeroGeracao <= NUMERO_MAX_GERACOES && Solução não encontrada) {
6          Muta Geração
7          Avalia
8          Identifica melhor Indivíduo da Geração
9          Decide se evita máximo local
10     }
11     Calcula quantos novos Ajudantes serão iniciados ( [0, QTD_ENTRADAS_MUX] )
12     Inicia de 0 a QTD_ENTRADAS_MUX Ajudantes
13     Decrementa Coordenador.MUXES_EM_ANALISES
14     Grava dados do MUX
15     Libera recursos
16 Fim
17
```

Três exemplos foram escolhidos para verificar a eficácia da técnica proposta.

4.2 Teste Multiplicador 2x2

Este primeiro exemplo foi testado para verificar o desempenho do GAMT para um problema Multiplicador 2×2.

Em várias etapas do projeto, tanto no protótipo quanto na versão final em Java, vários indivíduos foram representados na forma de circuitos lógicos e a saída foi computada fora do sistema. Estes testes validaram a precisão do algoritmo em calcular a saída do circuito lógico de forma correta. Nesta seção, será apresentado o circuito lógico final de uma solução para ilustrar o processo de validação.

A tabela abaixo contém os dados de entrada que são utilizados pelo algoritmo em cada execução (arquivo setup.txt). O sistema foi todo desenvolvido de forma a se adaptar a eles . Observa-se pelas linhas 1 e 2 que a tabela enunciado do problema é composta por uma tabela verdade de 4 colunas, pois $N_1 = N_2 = 2$. O número máximo de mutações aplicadas em cada indivíduo foi limitado a 3 (linha 3), rodando um número máximo de 1.000.000 de gerações em cada MUX (linha 4). As linhas 5 e 6 contém os dados da técnica utilizada responsável por evitar máximos locais. O número máximo de indivíduos por geração, quatro neste caso, está determinado na linha 7. E foram utilizados MUX4 (4 portas de entrada e dois bits de chave) (linha 8). Por fim, a Matriz Topologia teve tamanho definido de 2×5 (linhas 9 e 10).

setup.txt

```
1  N1=2;
2  N2=2;
3  MAXIMO_DE_MUTACOES=3;
4  NUMERO_MAXIMO_GERACOES=1.000.000;
5  GERACOES_SEM_MUDANCA=100.000;
6  PERSISTENCIA=2.500;
7  INDIVIDUOS_POR_GERACAO=4;
8  QTD_ENTRADAS_MUX=2;
9  LINHAS_MATRIZ_TOPOLOGIA=2;
10 COLUNAS_MATRIZ_TOPOLOGIA=5;
```

Para um Multiplicador 2×2, existem quatro colunas solução, denominadas S_0, S_1, S_2 e S_3. O GAMT foi aplicado 20 vezes neste problema, obtendo solução em tempo de processamento inferior a 1 segundo para todos os S_i, utilizando apenas um MUX4 na árvore. É importante observar que, para tabelas enunciado desta dimensão, apenas um MUX4 é necessário para encontrar a solução de cada S_i. A medida que os testes forem

executados em problemas mais complexos, a necessidade de uma árvore mais extensa aparece, como veremos nas subseções seguintes.

Embora apenas um MUX4 seja necessário para encontrar a solução do Multiplicador 2×2, existem outros indicadores do tempo e esforço computacional que mostram qual das colunas resposta foram mais trabalhosas. Verifica-se, na Tabela 14, o número da geração que deu origem ao Indivíduo Solução daquele MUX4. Dessa forma, é possível identificar que o número médio de gerações para encontrar a resposta do S_2 é consideravelmente maior que o do S_3.

Tabela 14 – Tabela de gerações do Indivíduo Solução

Multiplicador 2 x 2			
S_0	S_1	S_2	S_3
6	224	15	35
28	6.447	3.857	75
1.195	12	21.121	5
805	81	1.127	18
1.010	4	2.262	2
121	615	8.487	17
1.068	10	44	24
25	55	246	10
187	1	5.642	128
160	12	377	271
8	22	11.208	117
66	127	36.658	53
4	2.727	2.362	34
1517	33	48	24
22	99.271	3.943	59
29	626	706	5
177	26	59.129	5
6	10	20	10
644	32.665	1.163	59
1.844	77	23.593	2
Médias			
446,10	7.152,25	9.100,40	47,65

Vejamos a resposta do GAMT para uma das instâncias do problema. A instância 19, por se tratar do caso com o menor desvio padrão, foi escolhida para exemplificar a leitura dos dados gravado pelo programa.

O programa gera um arquivo **Si_Coordenador.txt**, com o S_i sendo solucionado, com as informações de *setup* da execução, com a matriz topologia em uso, o valor atual de quantos MUX estão em análise, com toda a história da solução (MUX criados, MUX encerrados e suas características finais). Na seção 4.3, será apresentado arquivo de saída deste tipo, com os comentários pertinentes.

Além do arquivo geral, o sistema gera um arquivo de evolução para cada MUX criado. Este arquivo é de especial importância para acompanhar a evolução dos indivíduos, os procedimento de fuga de máximos locais, a evolução do *rank* (avaliação) e a solução final daquele MUX (quantos novos MUX são criados e quantas linhas são repassadas para cada um deles).

A seguir, observa-se uma saída típica gerada pelo programa para um MUX. Neste caso, o único da árvore solução.

```
 LINHAS =====>>> 16
 Indivíduo inicial:
[1, 3, 2, 100, 2, 4, 400, 2, 5, 200, 2, 5, 200, 5, 2, 400, 2, 6, 500,
    9, 8, 300, 9, 3, 500, 12, 1, 500, 9, 2, 200, 1, 102]
[1, 3, 1, 400, 1, 2, 100, 1, 2, 500, 2, 6, 300, 4, 8, 100, 2, 7, 300,
    1, 7, 100, 4, 5, 300, 2, 7, 100, 12, 6, 100, 2, -1]


 Evolução!!!!  Geração: 1
 Melhor rank até esta geração: 102
 Valor do Multiplicador de Mutações: 1

[1, 3, 2, 100, 2, 4, 400, 2, 5, 200, 2, 5, 200, 5, 2, 400, 2, 6, 500,
    9, 8, 300, 9, 3, 500, 12, 1, 500, 9, 2, 200, 1, 102]
[1, 3, 1, 400, 1, 2, 100, 1, 2, 500, 2, 6, 300, 4, 8, 100, 2, 7, 300,
    1, 7, 100, 4, 5, 300, 2, 7, 100, 12, 6, 100, 2, -1]


 Evolução!!!!  Geração: 584
 Melhor rank até esta geração: 106
 Valor do Multiplicador de Mutações: 1

[1, 3, 2, 100, 2, 4, 400, 1, 5, 200, 2, 5, 200, 5, 2, 400, 2, 6, 500,
    9, 8, 300, 9, 3, 500, 12, 1, 500, 9, 2, 200, 1, 106]
[1, 3, 1, 400, 3, 2, 100, 1, 2, 500, 2, 6, 300, 4, 8, 100, 2, 7, 300,
    1, 7, 100, 4, 5, 300, 2, 7, 100, 12, 6, 100, 8, -1]


 Encontrado na geração: 644
 Com Rank: 128
 Temos um Indivíduo que soluciona a saída S0:
```

```
[1, 3, 2, 100, 2, 4, 400, 1, 5, 200, 2, 5, 200, 5, 8, 400, 2, 6, 500,
9, 8, 300, 9, 3, 500, 12, 1, 500, 9, 2, 200, 1, -128]
[1, 3, 1, 400, 3, 4, 100, 1, 2, 500, 2, 6, 300, 5, 8, 100, 2, 7, 300,
1, 7, 100, 4, 5, 300, 2, 7, 100, 12, 6, 100, 8, -1]

 Portas Mux: [0, 0, 1, 0]
 Ocorrência de chaves: [1, 7, 1, 7]

FIM
```

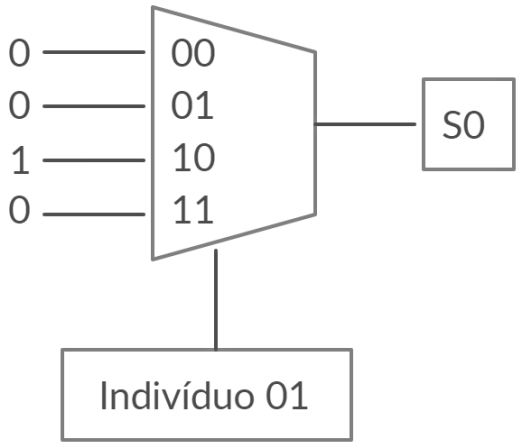
É possível observar que no início do arquivo existe uma linha dedicada a informar quantas linhas que serão trabalhadas naquele Ajudante (ou seja, naquele multiplexador). Neste caso, temos o número 16, pois este é o primeiro multiplexador da árvore solução de um Multiplicador 2×2, que possui 16 linhas. Logo em seguida, é impressa a matriz relativa ao CLC do indivíduo primordial, que dará origem a todos os outros indivíduos daquela linha evolutiva. Cada vez que ocorre um aprimoramento do indivíduo, e sua avaliação se torna a melhor até o momento, este novo melhor indivíduo é registrado, juntamente com o seu *rank* e com a geração em que ele foi encontrado. É registrado também o valor do Multiplicador de Mutações no momento em que este novo indivíduo foi encontrado (caso esse Multiplicador de Mutações seja diferente de 1 isso indica que este novo indivíduo foi encontrado enquanto o GAMT estava aplicando a técnica para evitar máximos locais). No caso apresentado, observa-se que logo na primeira geração houve uma melhora do indivíduo inicial, levando a avaliação do indivíduo a 106. Depois há outra melhora, levando o melhor *rank* de 102 para 106, na geração 584. Por fim, na geração 644, com avaliação 128 foi encontrado e gravado um indivíduo que soluciona o MUX4 em questão. São gravados no arquivo também a configuração das portas deste multiplexador e a quantidade de linhas que foi direcionada para cada uma das suas portas. Nas portas I_0, I_1 e I_3 temos como entrada o valor 0, e com, respectivamente, 1, 7 e 7 linhas sendo trabalhadas em cada uma dessas portas e na porta I_2 o valor 1, com apenas uma linha de entrada.

Podemos notar também que, para o individuo solução, no campo destinado a armazenar informações relativas a avaliação, encontramos o *rank* negativo, -128, ao invés de 128, seu *rank* real. Isso ocorre por uma particularidade da programação do GAMT. Quando um indivíduo soluciona um MUX, ele é avaliado e recebe sua avaliação multiplicada por -1, indicando que este indivíduo não só superou seu antecessor, mas que também dispensa sucessores.

Na Figura 10 é possível observar uma representação gráfica da resposta obtida para o S_0 da instância 19. E logo em seguida a representação gráfica do circuito lógico combinacional contido no indivíduo que solucionou o MUX. Neste circuito é possível

observar duas saídas, uma para cada bit da chave do multiplexador em questão, uma vez que estão sendo trabalhados multiplexadores de 4 entradas.

Figura 10 – Árvore Solução do S_0 instância 19 - Multiplicador 2×2



Abaixo, observa-se a matriz bidimensional que representa o CLC contido em "Indivíduo 01". E em seguida a representação gráfica deste circuito.

```
[1, 3, 2, 100, 2, 4, 400, 1, 5, 200, 2, 5, 200, 5, 8, 400, 2, 6, 500,
9, 8, 300, 9, 3, 500, 12, 1, 500, 9, 2, 200, 1, -128]
[1, 3, 1, 400, 3, 4, 100, 1, 2, 500, 2, 6, 300, 5, 8, 100, 2, 7, 300,
1, 7, 100, 4, 5, 300, 2, 7, 100, 12, 6, 100, 8, -1]


Legenda para as portas lógicas utilizadas:
    100 - AND
    200 - OR
    300 - NAND
    400 - NOR
    500 - XOR
```

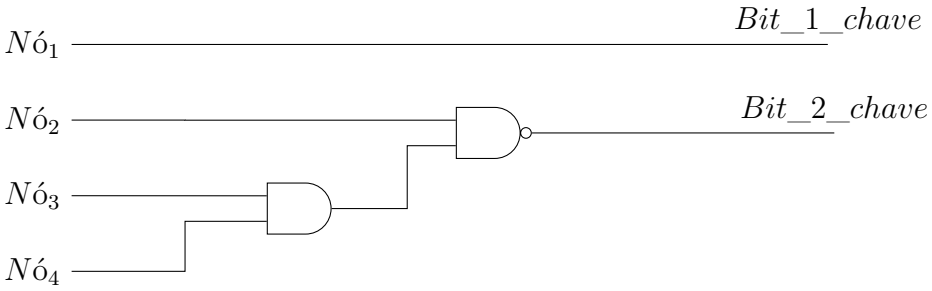


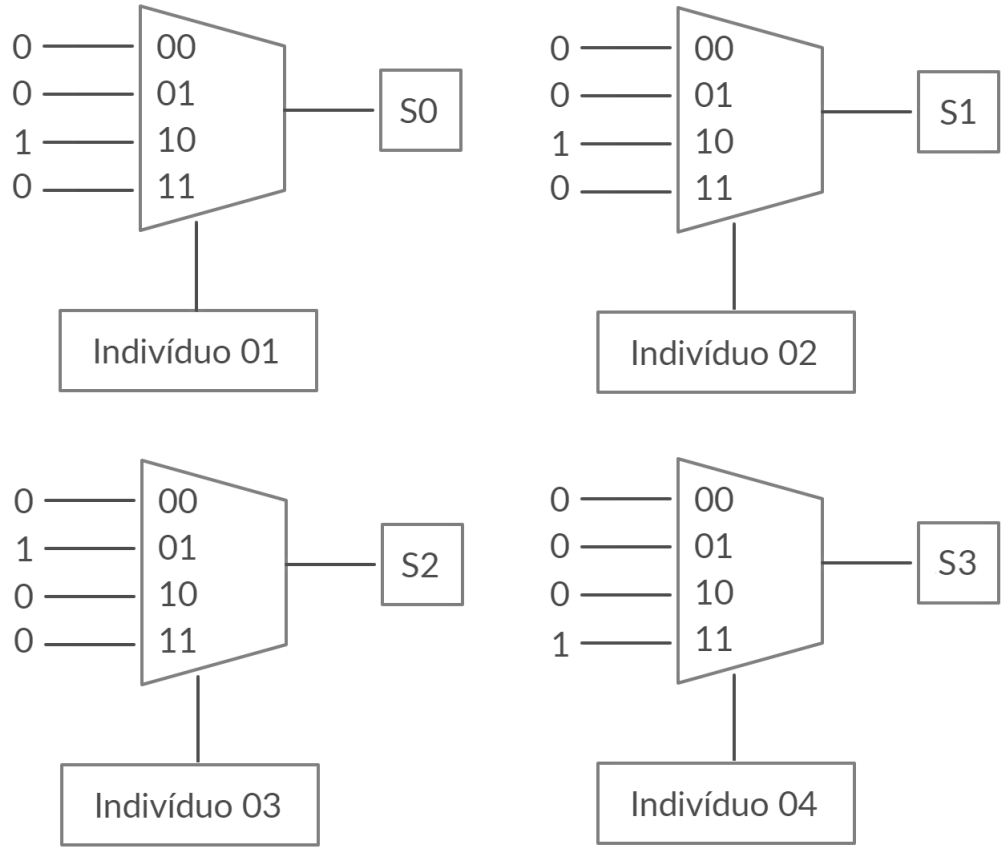
Figura 11 – CLC do Indivíduo 01 - Multiplicador 2×2

Considere a décima linha da matriz enunciado do problema, para verificar a solução proposta para o S_0. Como entrada, temos dois bits de N_1 (Nó$_1$ e Nó$_2$) e dois bits de N_2 (Nó$_3$ e Nó$_4$). A tabela verdade, na linha 10, possui como valor dos nós de 1 a 4 − | 1 | 0 | 0 | 1 | e como saída | 0 |. Ao aplicarmos estes valores dos nós de entrada na representação gráfica do Indivíduo 01 acima, obtemos $Bit\_1\_chave = 1$ e também $Bit\_2\_chave = 1$. Os bits encontrados indicam como valor de chave 11, que, quando aplicado ao multiplexador da Figura 10, liga o valor da entrada da porta $I_3 = 0$ na a saída S_0, que é a saída correta do problema.

Aplicando agora décima sexta linha da tabela enunciado à solução proposta. Temos como valores dos nós de 1 a 4 − | 1 | 1 | 1 | 1 | e como saída | 1 |. As variáveis Bit_1_chave e Bit_2_chave resultam em 1 e 0. Para a chave 10 o multiplexador entregará na saída o valor contido na porta $I_2 = 1$.

Observa-se, na Figura 12, a resposta completa da instância 19, para S_0, S_1, S_2 e S_3, acompanhada do circuito contido em cada um dos indivíduos das árvores (Figuras 13, 14 e 15).

Figura 12 – Solução da instância 19 de execução - Multiplicador 2×2



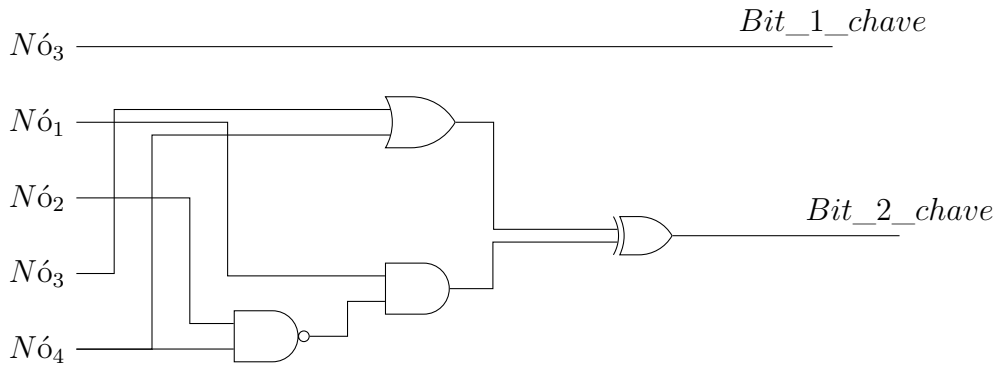


Figura 13 – CLC do Indivíduo 02 - Multiplicador 2×2

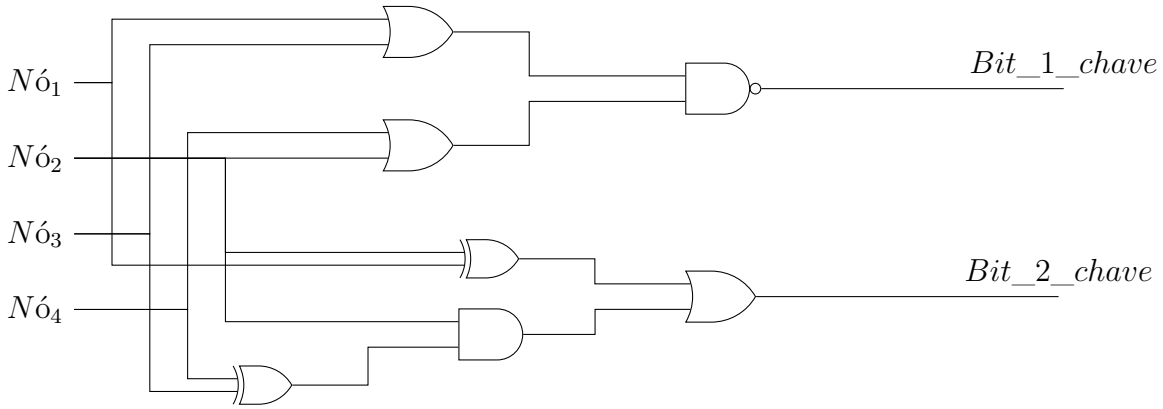


Figura 14 – CLC do Indivíduo 03 - Multiplicador 2×2

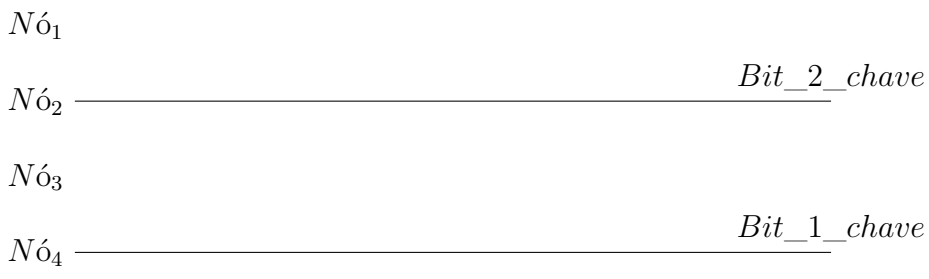


Figura 15 – CLC do Indivíduo 04 - Multiplicador 2×2

4.3 Teste Multiplicador 3x3

Para esta segunda bateria testes, foram utilizados os mesmos dados do Teste multiplicador 2×2, fazendo uma única modificação na dimensão do problema. Para solucionar um multiplicador 3×3, o recurso computacional cresce consideravelmente, pois o número de colunas a serem resolvidas cresce de 4 para 6 e toda a matriz enunciado passa a ter 64 linhas, ao invés das 16 do exemplo anterior. A matriz enunciado de um Multiplicador 3×3 é seis vezes maior e mais complexa que a de um Multiplicador 2×2. Pode-se notar no *setup.txt* que, para gerar a matriz enunciado com as novas proporções, os novos valores de N_1 e N_2 devem ser ambos 3, resultando nas matrizes de entrada e de saída com seis colunas cada.

setup.txt

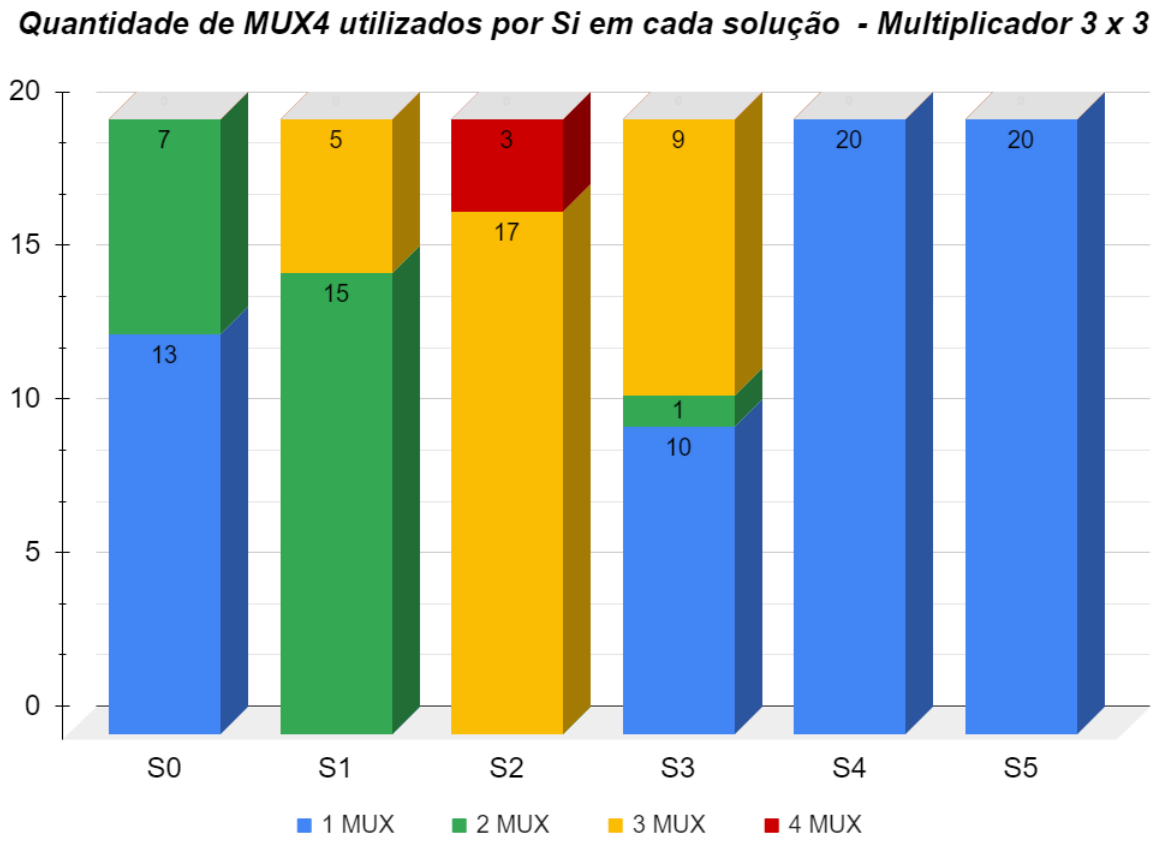
```
1  N1=3;
2  N2=3;
3  MAXIMO_DE_MUTACOES=3;
4  NUMERO_MAXIMO_GERACOES=1.000.000;
5  GERACOES_SEM_MUDANCA=100.000;
6  PERSISTENCIA=2.500;
7  INDIVIDUOS_POR_GERACAO=4;
8  QTD_ENTRADAS_MUX=2;
9  LINHAS_MATRIZ_TOPOLOGIA=2;
10 COLUNAS_MATRIZ_TOPOLOGIA=5;
```

Para este exemplo, o GAMT apresenta soluções com mais de um multiplexador na árvore de cada S_i. Foram feitas novamente 20 execuções do algoritmo, para analisar as diferentes respostas e verificar as diferenças do exemplo anterior. No gráfico abaixo é possível ver uma relação da quantidade de MUX4 usados para solucionar cada coluna da solução.

O número de multiplexadores utilizados na árvore é um indício da complexidade da resposta. O S_2 foi a coluna mais complexa, utilizando uma média de 3,15 multiplexadores em cada árvore de solução. A coluna S_1 usou uma média de 2,25 multiplexadores por árvore. As colunas S_3 e S_0 utilizaram, em média, 1,95 e 1,35, respectivamente. Estas foram as colunas que exigiram mais recurso computacional e tempo de execução.

Pode-se notar que as colunas de saída S_4 e S_5 foram resolvidas com apenas um MUX4 na árvore. Mesmo em tabelas enunciado mais complexas, é possível encontrar colunas com solução trivial através da técnica de árvore de multiplexadores. E o GAMT encontra automaticamente a solução mais simples, utilizando apenas o MUX4 inicial. A média de gerações necessárias para encontrar o S_4 do multiplicador 3×3 foi 43.329,7.

Figura 16 – Quantidade de MUX4 por S_i para 20 execuções - Multiplicador 3×3



Apesar de resolver com apenas um MUX4, nota-se que foi mais de quatro vezes mais trabalhoso que o mais complexo dos S_i do multiplicador 2×2 (ver Tabela 14). Já o S_5 foi tão facilmente encontrado quanto os demais S_i do multiplicador 2×2. É possível notar, portanto, que há mais complexidade nas colunas solução centrais da tabela saída dos multiplicadores. Esta tendência de complexidade é um fenômeno observado não só nos casos expostos até agora, mas também para multiplicadores de maiores dimensões.

Foi tomada como exemplo a instância 9 de execução, novamente por se tratar de um caso próximo ao desvio padrão, propiciando uma análise mais precisa do comportamento do GAMT ao resolver problemas desta proporção e nestas condições.

Abaixo está representado o arquivo de saída (**S2_Coordenador.txt**) da instância 9 de execução.

```
S2
N1=3
N2=3
MAXIMO_MUTACOES=3
NUMERO_MAX_GERACOES=1000000
```

```
GERACOES_SEM_MUDANCA=100000
PERSISTENCIA=2500
INDIVIDUOS_POR_GERACAO=4
QTD_CH=2
LINHAS_NOS=2
COLUNAS_NOS=5
LINHAS 64
COLUNAS=6
Topologia:
[7, 9, 11, 13, 15]
[8, 10, 12, 14, 16]

1 mux(es) em análise
2 mux(es) em análise
3 mux(es) em análise

Mux 0. Ligado ao Mux -1 na porta -1
Indivíduo solução:
[1, 6, 4, 300, 1, 2, 400, 8, 5, 400, 6, 2, 500, 7, 9, 100, 10, 4, 300,
1, 3, 100, 10, 2, 200, 2, 11, 500, 8, 11, 200, 16, 280]
[1, 5, 4, 500, 5, 1, 200, 8, 6, 200, 1, 3, 300, 10, 5, 200, 7, 5, 200,
4, 11, 100, 2, 12, 100, 2, 6, 200, 12, 3, 300, 14, -1]
Situação das portas: [2, 2, 0, 1]
Ocorrência das chaves [10, 20, 30, 4]

2 mux(es) em análise

Mux 1. Ligado ao Mux 0 na porta 0
Indivíduo solução:
[1, 4, 3, 500, 2, 1, 300, 3, 8, 300, 6, 3, 100, 3, 8, 300, 4, 10, 100,
3, 11, 100, 12, 6, 400, 5, 7, 200, 5, 6, 200, 12, -74]
[1, 3, 2, 400, 2, 6, 400, 4, 7, 200, 1, 5, 200, 8, 5, 400, 2, 4, 400,
6, 10, 500, 4, 2, 400, 4, 10, 300, 10, 12, 200, 11, -1]
Situação das portas: [1, 0, 0, 1]
Ocorrência das chaves [7, 1, 1, 1]

1 mux(es) em análise
2 mux(es) em análise
```

```
Mux 2. Ligado ao Mux 0 na porta 1
Indivíduo solução:
[1, 6, 1, 300, 1, 3, 200, 1, 2, 500, 2, 3, 500, 7, 8, 400, 4, 10, 100,
8, 2, 400, 1, 12, 500, 12, 3, 500, 9, 12, 500, 16, 86]
[1, 3, 6, 300, 4, 1, 300, 1, 7, 400, 1, 6, 100, 8, 1, 300, 4, 3, 300,
9, 10, 400, 5, 3, 400, 12, 2, 100, 2, 6, 300, 13, -1]
Situação das portas: [0, 1, 2, 0]
Ocorrência das chaves [4, 6, 5, 5]

1 mux(es) em análise

Mux 3. Ligado ao Mux 2 na porta 2
Indivíduo solução:
[1, 5, 6, 100, 2, 6, 200, 3, 2, 400, 5, 7, 100, 8, 6, 200, 7, 10, 300,
12, 2, 400, 7, 4, 500, 12, 10, 500, 4, 14, 100, 3, -49]
[1, 1, 5, 300, 5, 4, 500, 8, 1, 400, 1, 5, 200, 8, 5, 200, 1, 4, 300,
1, 5, 400, 2, 8, 500, 2, 7, 100, 10, 1, 300, 9, -1]
Situação das portas: [1, -1, 1, 0]
Ocorrência das chaves [2, 0, 2, 1]

0 mux(es) em análise
S2 solucionado. Programa encerrado
```

Nas 13 primeiras linhas do arquivo de saída do coordenador acima, são exibidas as configurações daquela execução do programa. Neste caso, os dados exibidos são os mesmos já comentados no início desta subseção para o *setup.txt* do teste do multiplicador 3×3. Além destes dados, é apresentada a matriz topologia utilizada. Em seguida, temos mensagens do Coordenador, de caráter informativo, dizendo quantos multiplexadores estão em análise em um dado momento de execução do programa. Este valor é incrementado e decrementado ao longo do arquivo de saída toda vez que um Ajudante é iniciado ou finaliza suas atividades, respectivamente.

Além disso, são gravadas as respostas de cada um dos multiplexadores, assim que o Ajudante responsável por eles termina suas atividades. Analisando o arquivo, é possível perceber que a árvore gerada neste caso possui quatro multiplexadores, pelo fato de que quatro respostas foram gravadas no arquivo de saída. Cada uma destas respostas, de cada Ajudante, possui o melhor indivíduo obtido pelo Ajudante, o seu multiplexador pai, a porta do pai onde ele está ligado e a situação de cada uma de suas portas (onde, eventualmente, serão conectados os próximos multiplexadores). Por fim, ao encontrar o número de multiplexadores em análise igual a zero, o Coordenador informa que o S_i em

questão (no caso S_2) foi solucionado e encerra a execução.

Segue uma breve análise do padrão usado para gravar o estado de cada uma das portas do MUX recém resolvido:

```
Situação das portas: [0, 1, 2, 0]
Ocorrência das chaves [4, 6, 5, 5]

Legenda para a situação das portas:
    0 = porta travada em zero
    1 = porta travada em um
   -1 = porta não utilizada
    2 = porta com necessidade de mais um MUX
```

A partir das informações acima, da "Situação das portas", pode-se concluir que a primeira e a quarta portas do MUX serão travadas em 0, a segunda porta será travada em 1 e na terceira porta será conectado um novo multiplexador. Pela "Ocorrência das chaves", conclui-se que serão repassadas 5 linhas para o MUX que será conectado na terceira porta resolver. Os demais valores, indicam quantos zeros e uns havia nas linhas da tabela enunciado que foram resolvidas. Inclusive, este somatório $4 + 6 + 5 + 5$ deve resultar exatamente no número de linhas que foram repassadas a este MUX pelo seu pai.

As informações deste arquivo de saída (**S2_Coordenador.txt**) são utilizadas para construir toda a árvore de MUX da solução da coluna.

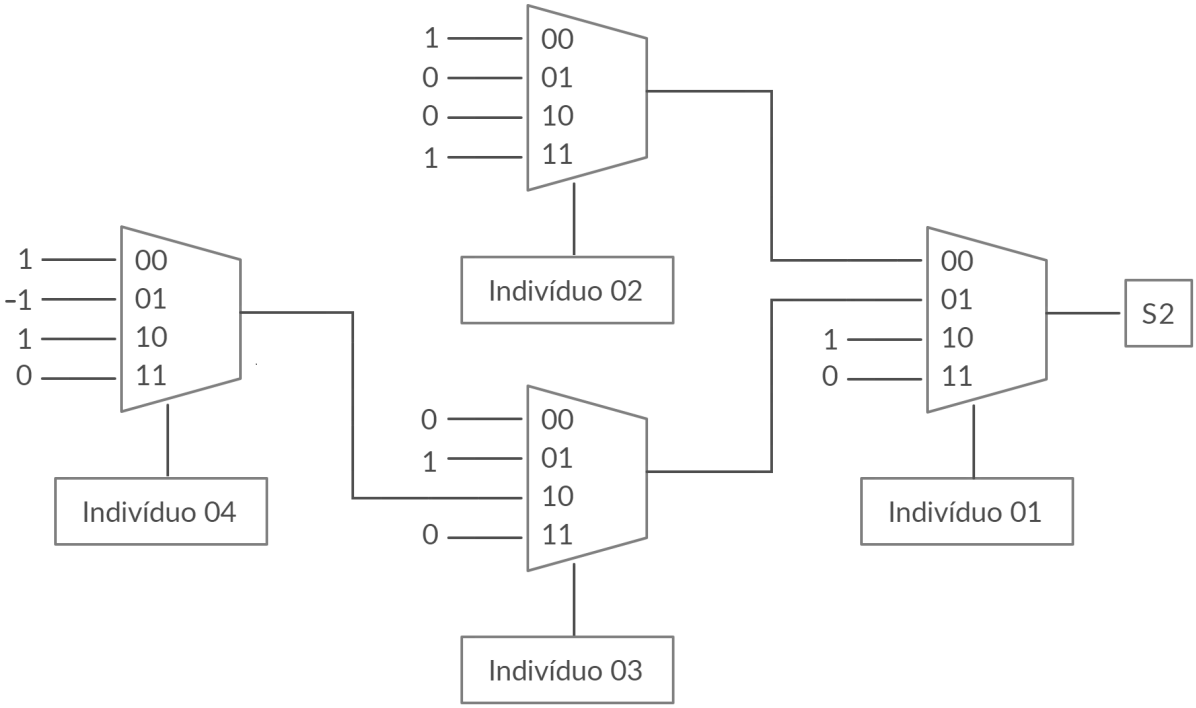
Na Figura 17, podemos observar a árvore de resposta gerada neste caso.

Todos os quatro indivíduos representados na árvore acima estão no arquivo de saída (**S2_Coordenador.txt**) que foi comentado acima. O circuito lógico combinacional de um deles, o "indivíduo 01", está na Figura 18 abaixo, seguido da sua representação gráfica em circuitos lógicos.

```
[1, 6, 4, 300, 1, 2, 400, 8, 5, 400, 6, 2, 500, 7, 9, 100, 10, 4, 300,
1, 3, 100, 10, 2, 200, 2, 11, 500, 8, 11, 200, 16, 280]
[1, 5, 4, 500, 5, 1, 200, 8, 6, 200, 1, 3, 300, 10, 5, 200, 7, 5, 200,
4, 11, 100, 2, 12, 100, 2, 6, 200, 12, 3, 300, 14, -1]
```

Neste caso, temos três bits para N_1 (Nó$_1$, Nó$_2$ e Nó$_3$) e mais três para N_2 (Nó$_6$, Nó$_5$ e Nó$_6$), por se tratar de um multiplicador 3×3. Observa-se que, para obter os valores de chave deste multiplexador, foram utilizadas 8 portas lógicas, sendo elas duas portas NOR, duas OR, duas AND, uma NAND e uma XOR. Uma particularidade desta resposta foi que o Nó$_3$ não foi utilizado para a obtenção de nenhum dos dois bits da chave e que o Nó$_5$ foi o nó mais utilizado.

Figura 17 – Árvore Solução do S_2 instância 9 - Multiplicador 3×3



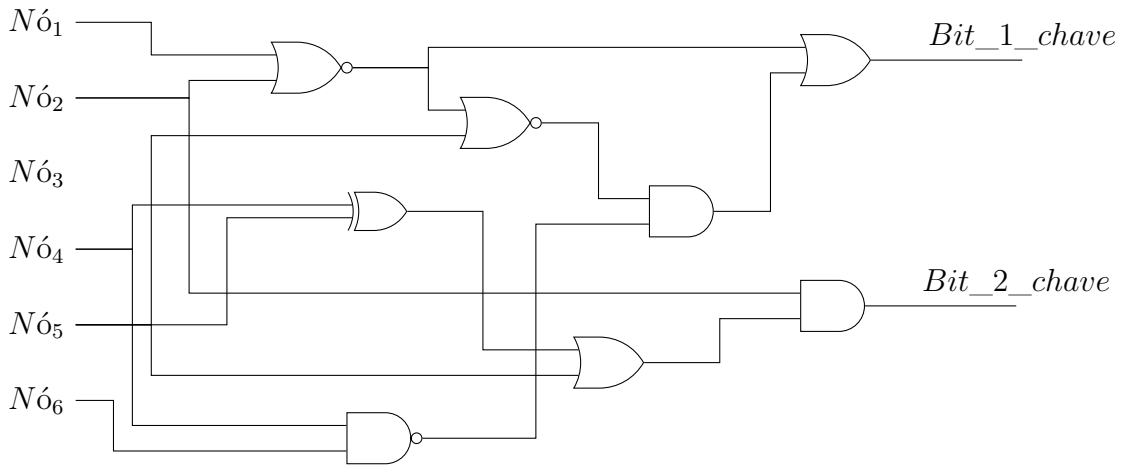


Figura 18 – CLC contido na estrutura Indivíduo 01 - Multiplicador 3×3

4.4 Teste Multiplicador 4x4

Para o último teste, foi trabalhado o multiplicador 4×4. Este possui oito colunas na tabela verdade de entrada e oito na tabela saída, ambas com 256 linhas. A tabela enunciado de um multiplicador 4×4 é mais de cinco vezes maior que a tabela enunciado de um multiplicador 3×3. Ela é, portanto, 32 vezes maior que a tabela enunciado de um multiplicador 2×2.

Foi atribuído o valor 4 às variáveis N_1 e N_2 (*setup.txt* linhas 1 e 2), e mantidos inalterados os outros parâmetros do programa. Dessa forma, foi possível determinar o desempenho do GAMT para um multiplicador 4×4.

setup.txt

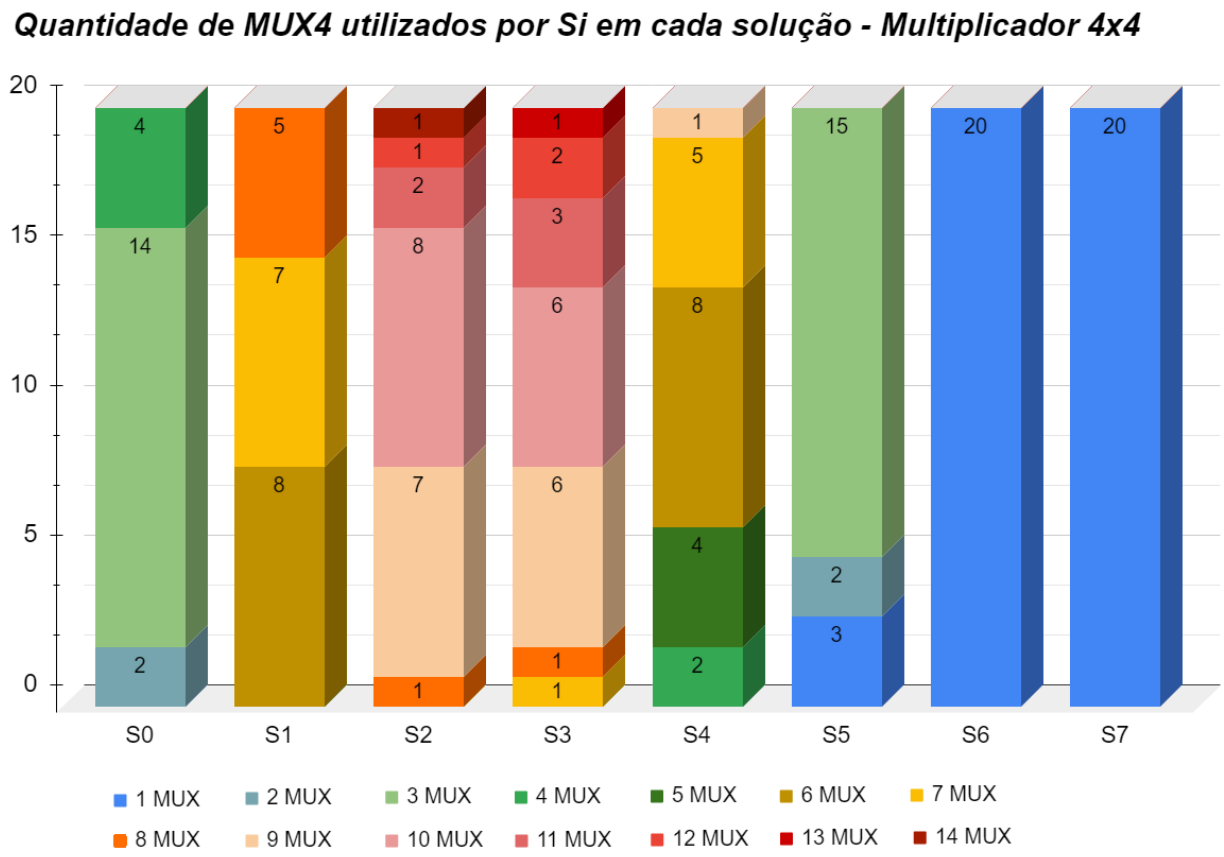
```
1  N1=4;
2  N2=4;
3  MAXIMO_DE_MUTACOES=3;
4  NUMERO_MAXIMO_GERACOES=1.000.000;
5  GERACOES_SEM_MUDANCA=100.000;
6  PERSISTENCIA=2.500;
7  INDIVIDUOS_POR_GERACAO=4;
8  QTD_ENTRADAS_MUX=2;
9  LINHAS_MATRIZ_TOPOLOGIA=2;
10 COLUNAS_MATRIZ_TOPOLOGIA=5;
```

Nestes teste, o GAMT começou a explorar de forma mais notável a recursividade do programa, gerando árvores mais extensas e complexas. Ele leva em média uma hora para encontrar uma resposta completa, incluindo todas as árvores solução, desde o S_0 ao S_7. Novamente, foram computadas 20 respostas, e será analisado o gráfico que mostra a quantidade de MUX4 utilizados em cada árvore criada como indicador da complexidade do problema e do desempenho do algoritmo projetado.

Observa-se que as soluções mais trabalhosas foram relativas às colunas S_2 e S_3 que apresentaram ambas uma média de $9,95$ MUX4 por árvore. As colunas S_1 e S_4 tiveram complexidade média para resolução, com $6,85$ e 6, respectivamente. As colunas S_0 e S_5 tiveram médias mais baixas, $3,1$ e $2,6$, respectivamente. Por fim, para resolver as colunas S_6 e S_7, o GAMT utilizou apenas um multiplexador, mostrando serem colunas com solução trivial. É interessante observar que, apesar do tamanho da tabela enunciado, para estas colunas, as soluções foram encontradas quase que instantaneamente (1 segundo, aproximadamente). Observa-se, nesta terceira bateria de testes, que foi mantido o mesmo padrão no qual as colunas centrais dos multiplicadores oferecem maior dificuldade.

Na Figura 20, é possível visualizar a árvore de multiplexadores solução da instância

Figura 19 – Quantidade de MUX4 por S_i para 20 execuções - Multiplicador 4×4



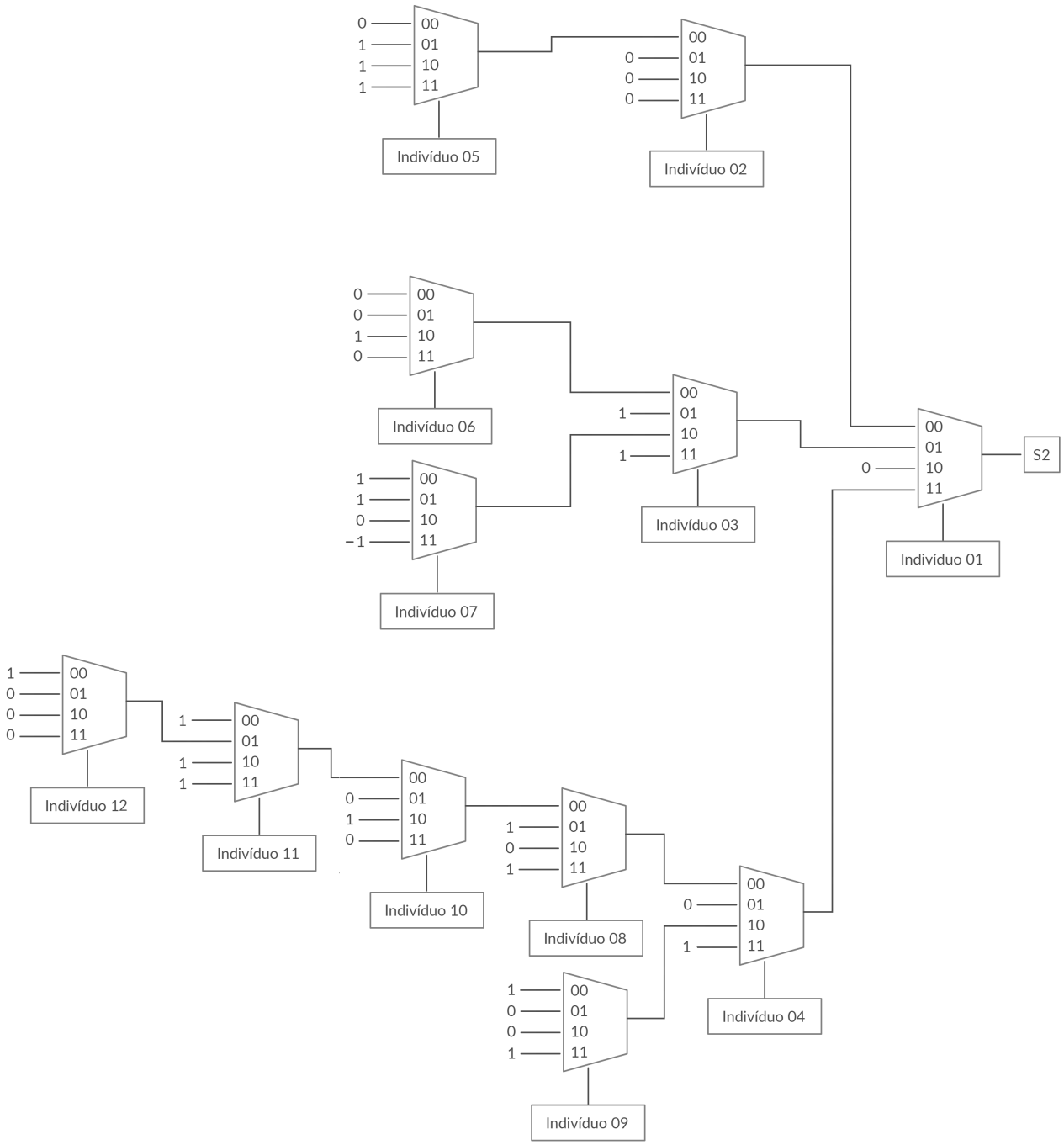
7, para o S_2, do multiplicador 4×4.

Nota-se que a dimensão da árvore cresce de forma considerável neste caso. Já de início, no primeiro multiplexador da árvore, 66 linhas da tabela foram travadas em zero e 190 linhas foram repassadas para três novos Ajudantes. Embora cada um desses ramos tenha dado origem a, no mínimo, um outro multiplexador, a parte mais considerável da árvore foi resolvida a partir do ramo ligado à última porta do MUX4 principal (chave 11), pois este recebeu o maior número de linhas dentre todos os MUX4 secundários, 134. Vejamos o registro de conclusão da solução do primeiro multiplexador da árvore.

```
Situação das portas: [2, 2, 0, 2] (2 = novo MUX; 0 = porta travada em zero)
Ocorrência das chaves [30, 26, 66, 134] (quantidade de linhas em cada porta)
```

À medida que o algoritmo avança na árvore da solução e mais portas da matriz enunciado vão sendo travadas, menos linhas são processadas por cada Ajudante. Deste modo, as matrizes diminuem de tamanho e a complexidade assintótica da execução diminui. Deste modo, o custo de aumentar o número máximo de gerações diminui e se torna vantajoso fazê-lo. Na versão atual do GAMT, a cada nova camada de MUX na árvore, o número máximo de gerações dobra. Assim, sem aumento significativo no

Figura 20 – Árvore Solução do S_2 instância 7 - Multiplicador 4×4



tempo de processamento, mais evoluções são feitas e isso possibilita soluções melhores. Observemos este exemplo: Na busca da solução do oitavo MUX4 da árvore solução de um S_3 de um multiplicador 4 x 4, temos a seguinte evolução:

```
Evolução!!!!  Geração: 4105867
Melhor rank até esta geração: 114
Valor do Multiplicador de Mutações: 1
```

Enquanto o número máximo de gerações é de 1.000.000 no MUX raiz da árvore, a mutação acima só ocorreu na geração 4.105.867, ou seja, houve ganho de qualidade na solução, sem relevante custo de tempo e computação. O MUX4 em questão, em função de sua profundidade na árvore, foi resolvido por um Ajudante programado para rodar 32.000.000 gerações. Este Ajudante havia recebido apenas 43 linhas da tabela enunciado para solucionar.

Neste outro exemplo, o quinto MUX4 da árvore de um S_3, 4×4, desta vez, utilizando o método para evitar máximos locais e trabalhando um total de 87 linhas:

```
Evolução!!!!  Geração: 12605764
Melhor rank até esta geração: 196
Valor do Multiplicador de Mutações: 3
```

Observa-se pelo valor da variável Multiplicador de Mutações, que o método para evitar máximos locais estava ativo no momento em que foi encontrada esta evolução.

A seguir será feita uma análise da eficiência do método que evita máximos locais (Tabela 15).

A Tabela 15 exibe informações de cada instância de execução do GAMT para um multiplicador 4×4. A primeira coluna apenas informa a instância em questão. A segunda coluna (Qtd_Apl) informa a quantidade de vezes que o método para evitar máximos locais foi aplicado naquele multiplexador. A terceira coluna informa a quantidade de vezes que a intervenção feita por este método resultou em uma evolução ou cadeia de evoluções. E, por fim, a quarta coluna oferece uma razão da quantidade de sucessos obtidos pelo número de aplicações do método, ou seja, a eficiência do método. Como média das eficiências, foi obtido o valor 58%, tendo um desvio padrão de 39,5%. Em outras palavras, quando o algoritmo tendia a estacionar em um máximo local e passar mais linhas para os próximos ramos da árvore, a técnica possibilitou a exploração de novas regiões do espaço de soluções, deu origem a novas mutações favoráveis e reduziu o trabalho dos ramos seguintes. Isso justifica a aplicação da técnica para a otimização e aperfeiçoamento das respostas obtidas.

A Tabela 16 foi ordenada de acordo com a quantidade de linhas da matriz enunciado repassadas do primeiro MUX4 da árvore para a segunda camada de MUX4. Observa-

Tabela 15 – Tabela com a eficácia do método que evita máximos locais - S_3

Instância	Qtd_Apl	Sucessos	Eficiência
01	0	0	-
02	5	1	20%
03	0	0	-
04	0	0	-
05	9	1	11%
06	0	0	-
07	8	1	13%
08	0	0	-
09	0	0	-
10	0	0	-
11	0	0	-
12	3	3	100%
13	3	2	67%
14	1	1	100%
15	0	0	-
16	5	1	20%
17	1	1	100%
18	1	1	100%
19	6	3	50%
20	0	0	-

se uma relação da eficiência do método que evita máximos locais, com o número de linhas repassadas. Isso propicia a visualização do impacto do método que evita máximos locais. Os casos que utilizaram o método e obtiveram valores de eficiência mais altos, estão localizados, predominantemente, na metade superior da tabela, repassando menos linhas. Enquanto isso, as instâncias com valores baixos de eficiência, ou mesmo que não aplicaram o método, se localizam, predominantemente, na metade inferior da tabela. Isso demonstra uma capacidade do método de simplificar e otimizar mais rapidamente o problema, repassando menos linhas a diante e facilitando a solução do problema.

Mais um teste foi realizado para avaliar o desempenho do método que evita máximos locais. Uma vez que a coluna se solução S_3 se mostrou ser a mais difícil durante a resolução do multiplicador 4×4, foram executados 20 testes sem o método, apenas para esta coluna. Após obter os resultados, foi montada a Tabela 17 com a quantidade de multiplexadores utilizados em ambos os casos. Ao observar as médias da tabela é possível concluir que, com a ausência do método, as árvores resposta e o tempo de processamento são maiores.

Tabela 16 – Tabela ordenada pela quantidade de linhas repassadas pelo multiplexador inicial - S_3

Instância	Eficiência	Linhas Repassadas
02	20%	178
16	20%	181
03	-	182
20	-	184
19	50%	190
17	100%	190
10	-	190
14	100%	192
18	100%	192
13	67%	196
12	100%	196
01	-	196
04	-	196
06	-	196
08	-	196
09	-	196
11	-	196
15	-	196
07	13%	198
05	11%	200

Tabela 17 – Tabela estudo de casos: quantidade de multiplexadores de S_3

Contagem	Utilizando o Método	Não utilizando o Método
01	11	11
02	10	9
03	13	10
04	10	10
05	12	12
06	11	12
07	10	12
08	12	10
09	9	10
10	10	14
11	9	10
12	8	13
13	9	10
14	10	12
15	9	12
16	7	15
17	10	11
18	9	12
19	9	15
20	11	14
Média	9,95	12,55

5 Conclusões e Trabalhos Futuros

5.1 Conclusões

Este trabalho apresentou um técnica robusta de projeto de circuitos lógicos. Partindo de uma estratégia de divisão e conquista, foi desenvolvido, implementado, testado e validado um algoritmo genético para o referido projeto.

O GAMT é um algoritmo genético que soluciona problemas através da criação de uma árvore de multiplexadores, subdividindo a tabela de entrada em pequenos problemas, possibilitando que o processamento seja dedicado às partes mais complexas do problema. Este algoritmo incorpora técnicas clássicas de algoritmos evolutivos, uma nova técnica de fuga de máximos locais e tem uma arquitetura intrinsecamente paralela. O algoritmo contém um novo método de avaliação de indivíduos que se mostrou eficiente e adequado à estratégia de divisão dos problemas em uma árvore de multiplexadores. Os experimentos mostraram que os resultados das equações de avaliação dos indivíduos conduzem eficientemente os indivíduos para uma propagação reduzida de linhas sem solução. Isso reduz o tamanho da árvore de multiplexadores e da solução em si.

A matriz proposta é segmentada, e o GAMT busca solucionar (travar) cada uma destas partes ou repassá-las para um novo multiplexador. Algumas portas dos multiplexadores podem ter seu valor simplificado (travado em zero ou um). Esse travamento de portas dos multiplexadores representa a solução de uma parte do problema proposto. As partes do problema que ainda não foram solucionadas são novamente subdivididas e repassadas para novos multiplexadores. Dessa forma, o GAMT segue particionando a matriz, diminuindo o tamanho e complexidade do problema.

O algoritmo projetado para resolver o problema da forma proposta no Capítulo 3 foi desenvolvido em Java. A versão final aplica a tecnologia *multithread* e implementa o processamento de várias tarefas simultaneamente. Este paralelismo possibilitou mais uma otimização no tempo de processamento, uma vez que o GAMT poderia processar todas as ramificações da árvore de acordo com a demanda e disponibilidade de processadores e núcleos no sistema em uso.

A filosofia de travamento de portas e subdivisão da matriz em pequenas partes juntamente com a capacidade de paralelismo do código, possibilitaram uma diminuição drástica no tempo de processamento dos problemas e também a resolução de problemas consideravelmente maiores. Sempre que a matriz é subdividida, o problema é particionado em tarefas menores, e estas podem ser trabalhadas simultaneamente. Dessa forma, o algoritmo deixa de processar dados relativos às parcelas já solucionadas e foca seus recursos no que ainda está pendente.

Mais um dos diferenciais do GAMT consiste na aplicação de uma técnica que

evita máximos locais. Essa técnica previne a estagnação do indivíduo em uma região onde a solução para aquele multiplexador não possa ser encontrada. A avaliação atual do indivíduo já tem um bom *rank* (boa avaliação) e as mutações têm mais dificuldade de tirar o indivíduo desta região do espaço de soluções. A referida técnica força o deslocamento do indivíduo para regiões mais distantes e, em quase 60% dos casos, deu origem a novas sequências de evoluções nas gerações. Execuções com e sem esta técnica, apresentaram, em instâncias mais complexas do multiplicador 4×4, uma redução em torno de 20% no número de multiplexadores usados na solução final. É importante mencionar que esta técnica é aplicada somente quando são detectadas barreiras no progresso (pontos de máximo local caracterizados por um longo período sem evoluções), visando otimizar a árvore e simplificar a solução final.

O código do GAMT é altamente parametrizado, o que permite sua utilização com diversas variações das dimensões e características da tabela enunciado; variações na dimensão dos multiplexadores utilizados; variações topológicas nos indivíduos; variações no número máximo de gerações; variações no processo de mutação; no tamanho da população de cada geração; diferentes parâmetros da fuga de máximos locais; e fator de crescimento do número máximo de gerações. Basta criar um arquivo de configurações setup.txt com os parâmetros desejados e o sistema se ajustará dinamicamente a todos eles.

O sistema demonstrou uso eficaz da técnica de divisão em multiplexadores, na medida em que fez o referido uso somente quando a complexidade aumentava. Todas as soluções de dimensão 2×2 foram feitas sem acrescentar nenhum novo MUX ao circuito solução. Colunas de solução trivial em instâncias maiores também não utilizaram mais de um multiplexador. Por outro lado, as colunas com características mais complexas nas instâncias 3×3 e 4×4, conforme descrito ao longo do trabalho, inseriram várias camadas de multiplexadores, resolvendo os problemas em tempo computacional viável.

O algoritmo desenvolvido se mostrou eficiente em termos de desempenho computacional, resolveu todas as instâncias propostas nas referidas dimensões e apresentou bom desempenho na fuga de regiões de máximos locais. Várias colunas da tabela enunciado foram resolvidas quase que instantaneamente. Outras, mais complexas, atingiram a ordem de grandeza acima de 100.000.000 de gerações (ao todo) e 64.000.000 nos multiplexadores mais profundos na árvore, e nunca ultrapassaram algumas poucas horas de processamento. Vale ressaltar que, nos níveis mais profundos da árvore, o número de gerações era aumentado com o intuito de produzir melhores indivíduos, ao mesmo tempo em que o número de linhas (e consequentemente tamanho das matrizes) processado diminuía, reduzindo muito o tempo de processamento daquele Ajudante (ou seja, daquele MUX).

5.2 Trabalhos Futuros

Como possíveis trabalhos futuros, pode-se apontar:

- Aplicar um número progressivo para a quantidade de indivíduos por geração e na quantidade de gerações por Ajudante. A medida que os Ajudantes recebem matrizes menores, pode-se potencializar a possibilidade de encontrar um indivíduo ótimo mais rápido, aumentando o número preestabelecido de indivíduos em cada geração e o número de gerações total de cada multiplexador.

- Outra aplicação interessante está em condicionar a dimensão do multiplexador utilizado, baseando-se no tamanho da matriz a ser resolvida. De modo que quanto maior a matriz recebida pelo Ajudante, mais portas o multiplexador tenha. Desta forma a matriz tende a ser decomposta em tarefas pequenas independente do seu tamanho original.

- Como demonstrado, indivíduos que solucionam o problema nem sempre utilizam todas as portas disponíveis no multiplexador. Contudo, é possível impossibilitar a não utilização de uma ou mais portas. De modo que, utilizando um MUX4, sejam obtidas, no mínimo, quatro submatrizes. Este método impõe ao indivíduo certas restrições, que podem comprometer o tempo de processamento, mas que podem se mostrar vantajosas para a obtenção de uma árvore solução mais simplificada.

- Mais uma possibilidade seria aplicar, rotineiramente, uma mutação no campo saída do indivíduo, uma vez que esta posição tem maior influência na modificação das respostas de dado indivíduo. É importante determinar um valor específico nesta situação, pois esta mutação forçada induz o indivíduo a mudanças mais bruscas, possibilitando uma convergência antecipada. Em contrapartida, caso essa modificação seja feita em excesso é possível não obter uma relação de parentesco sólida, impossibilitando a obtenção de uma solução.

REFERÊNCIAS

ABD-EL-BARR, M. et al. A modified ant colony algorithm for evolutionary design of digital circuits. In: *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.* [S.l.: s.n.], 2003. v. 1, p. 708–715 Vol.1.

ALBA, E. et al. Comparative study of serial and parallel heuristics used to design combinational logic circuits. *Optimization Methods and Software*, v. 22, p. 485–509, 06 2007.

COELHO, L. dos Santos Coelho e A. A. R. Algoritmos evolutivos em identificaÇÃo e controle de processos: Uma visÃo integrada e perspectivas. *SBA Controle e Automação*, v. 10, 04 1999.

COELLO, C. et al. Comparing different serial and parallel heuristics to design combinational logic circuits. In: *NASA/DoD Conference on Evolvable Hardware, 2003. Proceedings.* [S.l.: s.n.], 2003. p. 3–12.

COELLO, C.; LUNA, E.; HERNANDEZ-AGUIRRE, A. Use of particle swarm optimization to design combinational logic circuits. In: . [S.l.: s.n.], 2003. v. 2606, p. 398–409.

COELLO, C.; MENDOZA, B. An approach based on the use of the ant system to design combinational logic circuits. *Mathware and soft computing, ISSN 1134-5632, Vol. 9, Nº. 3, 2002, pags. 235-250*, 01 2002.

COELLO, C. A. C.; CHRISTIANSEN, A. D.; AGUIRRE, A. H. Use of evolutionary techniques to automate the design of combinational circuits. 1999.

DEITEL, H.; DEITEL, P.; PEARSON, E. *Java®: COMO PROGRAMAR.* PEARSON BRASIL, 2016. ISBN 9788543004792. Disponível em: <https://books.google.com.br/books?id=2gNRswEACAAJ>.

ERCEGOVAC, M.; LANG, T.; MORENO, J. Introduction to digital system. 01 1999.

FOGEL, D. B. Asymptotic convergence properties of genetic algorithms and evolutionary programming: Analysis and experiments. *Cybernetics and Systems*, Taylor and Francis, v. 25, n. 3, p. 389–407, 1994. Disponível em: <https://doi.org/10.1080/01969729408902335>.

FOGEL, D. B. A comparison of evolutionary programming and genetic algorithms on selected constrained optimization problems. *SIMULATION*, v. 64, n. 6, p. 397–404, 1995. Disponível em: <https://doi.org/10.1177/003754979506400605>.

HAN, K.-H.; KIM, J.-H. Quantum-inspired evolutionary algorithm for a class of combinatorial optimization. *IEEE Transactions on Evolutionary Computation*, v. 6, n. 6, p. 580–593, 2002.

HASSANAT, A. et al. Enhancing genetic algorithms using multi mutations. *PeerJ Computer Science*, 06 2016.

HERNANDEZ-AGUIRRE, A.; COELLO, C. Using genetic programing and multiplexers for the synthesis of logic circuits. *Engineering Optimization - ENG OPTIMIZ*, v. 36, p. 491–511, 08 2004.

JR, G. L. A decomposition chart technique to aid in realizations with multiplexers. *Computers, IEEE Transactions on*, C-27, p. 157 – 159, 03 1978.

KALGANOVA, T.; MILLER, J. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In: *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*. [S.l.: s.n.], 1999. p. 54–63.

KARAKATIč, S.; PODGORELEC, V.; HERICKO, M. Optimization of combinational logic circuits with genetic programming. *Elektronika ir Elektrotechnika*, v. 19, 09 2013.

LINDEN, R. *Algoritmos Genéticos (3a edição)*. [S.l.]: Ciencia Moderna, 2012. ISBN 9788574523736.

MANFRINI, F. A. L. *Estratégias de Busca no Projeto Evolucionista de Circuitos Combinacionais*. Tese (Doutorado) — Universidade Federal de Juiz de Fora (UFJF), 2 2017.

MILLER, J. et al. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, 10 1999.

MILLER, J. F. Cartesian genetic programming. In: _____. *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 17–34. ISBN 978-3-642-17310-3. Disponível em: <https://doi.org/10.1007/978-3-642-17310-3_2>.

MOORE, P.; VENAYAGAMOORTHY, G. Evolving combinational logic circuits using a hybrid quantum evolution and particle swarm inspired algorithm. In: . [S.l.: s.n.], 2005. v. 2005, p. 97– 102. ISBN 0-7695-2399-4.

OLIVEIRA, H. A. de; MANFRINI, F. A. L. Proposta de uma heurística construtiva para o projeto de circuitos combinacionais. In: *XIII Encontro Acadêmico de Modelagem Computacional*. Petrópolis, RJ, Brasil: LNCC/MCTIC, 2020. p. 39–46.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0123706068.

SILVA, J. E. da et al. Biased mutation and tournament selection approaches for designing combinational logic circuits via cartesian genetic programming. In: *Anais do XV Encontro Nacional de Inteligência Artificial e Computacional*. Porto Alegre, RS, Brasil: SBC, 2018. p. 835–846. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/eniac/article/view/4471>.

Srinivas, M.; Patnaik, L. M. Genetic algorithms: a survey. *Computer*, IEEE, v. 27, n. 6, p. 17–26, 1994.

STEPNEY, S.; ADAMATZKY, A. *Inspired by Nature*. 1. ed. [S.l.]: Springer, 2018.

TANENBAUM, A. *Sistemas operacionais modernos*. Prentice-Hall do Brasil, 2010. ISBN 9788576052371. Disponível em: <https://books.google.com.br/books?id=nDatQwAACAAJ>.

TOCCI, R. Digital systems: Principles and applications. 01 1988.

VAHID, F. *Digital Design*. Wiley, 2006. ISBN 9780471467847. Disponível em: <https://books.google.com.br/books?id=lg5GAQAAIAAJ>.