Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Uma abordagem de otimização guiada por contraexemplos usando solucionadores SAT e SMT

Higo Ferreira Albuquerque

Manaus – Amazonas

Março de 2019

Higo Ferreira Albuquerque

Uma abordagem de otimização guiada por contraexemplos usando solucionadores SAT e SMT

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Orientador: Lucas Carvalho Cordeiro

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Albuquerque, Higo Ferreira
A345u Uma abordagem de otimização guiada por contraexemplos usando solucionadores SAT e SMT / Higo Ferreira Albuquerque. 2019
100 f.: il. color; 31 cm.

Orientador: Lucas Carvalho Cordeiro
Coorientador: Eddie Batista de Lima Filho
Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Amazonas.

1. Verificadores de Programas. 2. Otimização. 3. OptCE. 4. Algoritmo CEGIO. I. Cordeiro, Lucas Carvalho II. Universidade Federal do Amazonas III. Título

HIGO FERREIRA ALBUQUERQUE

"UMA ABORDAGEM DE OTIMIZAÇÃO GUIADA POR CONTRAEXEMPLOS USANDO SOLUCIONADORES SAT E SMT"

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica na área de concentração Controle e Automação de Sistemas.

Aprovado em 27 de março de 2019.

BANCA EXAMINADORA

Prof. Dr. Eddie Batista de Lima Filho

TPV Technology

Prof. Dr. Raimundo da Silva Barreto

Universidade Federal do Amazonas

Prof. Dr. Renan Landau Paiva de Medeiros

Universidade Federal do Amazonas

Agradecimentos

Agradeço primeiramente a Deus por ter iluminado meu caminho, assim como ter me dado forças e saúde para seguir em frente mesmo após os muitos momentos de desânimo.

Agradeço aos meus pais Silvana Ferreira Albuquerque e Izannilson Geraldo dos Santos Albuquerque por todo suporte, incentivo como também pela cobrança, para o término deste trabalho e a conquista de mais este título.

Agradeço aos meus amigos e colegas do mestrado UFAM, em especial: Phellipe Pereira, Whilliame Rocha, João Paulo, Felipe Monteiro, Thiago Cavalcante, pela amizade e persistência nas longas horas de estudo. Agradeço aos meus amigos da graduação que em muitos momentos me deram conselhos e ânimo, em especial a Bárbara Lobato e Max Simões que estiveram mais presentes nesta etapa sempre compartilhando das minhas dificuldades.

Agradeço a minha amada Karen Pereira, que entrou em minha vida no período final deste trabalho, todavia sem o seu apoio e cobrança, jamais teria concluído esta etapa dos meus estudos. Agradeço aos nobres colegas Iury Bessa e Rodrigo Araújo, pela paciência, pelos ensinamentos, mas também pela ajuda que auxiliou minha caminhada até ao término deste trabalho. Agradeço aos professores Mikhail Ramalho e Eddie Lima, que me ajudaram na produção e defesa de artigos relacionada à pesquisa.

Em especial agradeço muitíssimo ao Prof. Lucas Cordeiro, pelo suporte, direcionamento, encorajamento e principalmente pela insistência para a realização desse trabalho. Tenho plena consciência que sem o seu apoio e perseverança, jamais teria concluído, como também espero algum dia retribuir todo o seu tempo dedicado para minha formação.

“Success consists of going from failure to failure without loss of enthusiasm.”.

Winston Churchill

Resumo

O processo de otimização exige uma formulação matemática do sistema ou problema a ser otimizado, e considerando que os métodos formais são técnicas baseadas nos formalismos matemáticos usados para a especificação, desenvolvimento e verificação de sistemas, busca-se usar métodos formais para a otimização de funções matemáticas.

Este trabalho apresenta o desenvolvimento da ferramenta de otimização guiada por contraexemplos OptCE, assim como os algoritmos de otimização indutiva guiada por contraexemplos (CEGIO). Busca-se estabelecer o uso da verificação de modelos limitados para a otimização em funções convexas e não convexas, encapsulando a metodologia em uma ferramenta, permitindo uma otimização guiada a partir de contraexemplos dos solucionadores de Satisfatibilidade Booleana (*Boolean Satisfiability* - SAT) e Teoria do Módulo de Satisfatibilidade (*Satisfiability Modulo Theories* - SMT).

Durante o trabalho é detalhado a metodologia e exemplo para uma otimização guiada por contraexemplos, assim como os algoritmos CEGIO, que têm como objetivo a localização do mínimo global em uma função. Também são apresentados o desenvolvimento, funcionalidades e avaliação da ferramenta de otimização OptCE. O usuário fornece um arquivo com a modelagem da função e suas restrições, e o OptCE usa as informações para implementar os algoritmos CEGIO, inserir propriedades, aplicar verificadores de programas para checar propriedades, gerar contraexemplos SAT e SMT, executando de forma automatizada as etapas de especificação e verificação sucessivas vezes em busca do ponto ótimo da função.

A avaliação do OptCE foi realizada a partir de funções de otimização obtidas da literatura, e foram comparadas com outras técnicas tradicionais. Os experimentos utilizaram 40 funções (convexas e não convexas) onde os resultados obtidos mostram sua capacidade de otimização, tendo melhor taxa de acerto quando comparada com as técnicas.

Palavras-chave: Verificadores de Programas, Otimização, OptCE, Algoritmo CEGIO.

Abstract

The optimization process requires a mathematical formulation of the system or problem to be optimized, and considering that the formal methods are techniques based on the mathematical formalisms used for the specification, development and verification of systems, we try to use formal methods to optimize mathematical functions.

This qualification presents the contributions to the development of the Counterexample Guided Inductive Optimization Algorithms (CEGIO) and the OptCE optimization tool. The goal is to establish the use of Bounded Model Checker for the optimization in convex and non-convex functions, encapsulating the methodology in a tool, allowing a counterexample guided optimization of the solvers Boolean Satisfiability and Satisfiability Modulo Theories.

During the work, the methodology and example for an optimization guided by counterexamples are detailed, as well as the algorithms developed CEGIO, whose goal is to locate the global minimum of a function. The development, functionalities and experimental evaluation of the OptCE optimization tool are also presented. The user provides a file with the modeling of the optimization function and its constraints, and OptCE uses the information entered to implement the CEGIO algorithms as well as to insert properties, make use of program verifiers to check properties, generate counterexamples SAT and SMT, executing in an automated way the steps of specification and verification successive times, looking for the optimal point of the function.

The experimental evaluations of the OptCE were performed from optimization functions obtained from the literature, and were compared with other traditional techniques. The experiments use 40 functions (convex and non-convex), where the results obtained, it is demonstrated their ability to optimization, having better-hit rate compared to other traditional techniques.

Keywords: Software Verification, Optimization, CEGIO Algorithm.

Índice

Índice de Figuras

Índice de Tabelas

Abreviações

BMC - ***B**ounded **M**odel **C**hecking*

CBMC - ***C B**ounded **M**odel **C**hecker*

CEGIO - ***C**ount**E**rexample **G**uided **I**nductive **O**ptimization*

CFG - ***C**ontrol **F**low **G**raph*

CNF - ***C**onjunctive **N**ormal **F**orm*

ESBMC - ***E**fficient **S**MT-Based Context-**B**ounded **M**odel **C**hecker*

GA - ***G**enetic **A**lgorithm*

LP - ***L**inear **P**rogramming*

NLP - *Non**L**inear **P**rogramming*

PSO - ***P**article **S**warm*

RT - ***R**eachability **T**ree*

SA - ***S**imulated **A**nnealing*

SAT - *Boolean **SAT**isfiability*

SMT - ***S**atisfiability **M**odulo **T**heories*

Símbolos

f - Função custo ou função objetivo

x - Vetor de n-variáveis de decisão

$x_1, ..., x_n$ - Variáveis de decisão do vetor x

x^* - Minimizador da função custo f

$f(x^*)$ - Solução ótima

Ω - Conjunto de restrições da função custo

h **e** g - Funções de Restrições que definem o conjunto de restrições Ω

Δ - Diferença entre a solução subótima e uma solução ótima

Υ - População de soluções aleatórias em um PSO

M - Sistema de transição em um BMC

S - Conjunto de estados em um BMC

s - Estado de um sistema de tranição de estados em um BMC

s_0 - Conjunto de estados iniciais em um BMC

R - Conjunto de transições em um BMC

γ - Transição de um sistema de transição de estados em um BMC

ϕ - Propriedade do sistema a ser verificado em um BMC

k - Limite de execução do sistema em um BMC

π - Cada intercalação de uma condição de verificação em um BMC

ψ - Condição de Verificação em um BMC

$p_1, p_2, ..., p_n$ - Conjunto de fórmulas em um BMC

C - Limite de contexto

SS - sinal numérico

MM - representação exata binária mantissa

E - um expoente inteiro exato

b - base da representação ponto flutuante

ε_m - precisão da máquina

p - Variável de precisão que é incrementada em cada mudança de precisão

ε - Variável de precisão desejada para solução final

n - Variável de número de casas decimais que é incrementado

η - Variável de número de casas decimais desejada para solução final

f_c - Valor Mínimo de uma Função Candidata

δ - Valor de compensação para reduzir os efeitos dos erros de truncamento

α - Taxa de aprendizagem, número de segmentação do espaço de busca - CEGIO-S

γ - Tamanho da faixa segmentada - CEGIO-S

f_m - Limitar o espaço de busca ao valor nulo - CEGIO-S

Capítulo 1

Introdução

O termo “otimização” é atribuído aos estudos de problemas onde se busca o mínimo ou máximo de uma função, através da escolha dos valores das variáveis de decisão dentre um subconjunto válido. Muitas áreas da ciência buscam modelar os seus problemas matematicamente. Problemas em economia, transporte, ciência da computação, engenharia e outros, tem sua representação por meio de funções matemáticas, o que torna possível aplicar técnicas de otimização para maximizar ou minimizar uma função definida, com o objetivo de encontrar uma solução ótima para o problema, ou seja, encontrar o melhor desempenho para o problema dadas as possibilidades existentes.

A computação e a otimização têm cooperado com grande evolução para ambas as áreas, onde grandes avanços da ciência da computação estão baseados na teoria de otimização, como por exemplo, problemas de planejamento (teoria dos jogos [1–3]) e problemas de alocação de recursos (*isto é*, *hardware/software co-design* [4]). Em contrapartida, a ciência da computação tem papel decisivo nos mais recentes estudos de otimização, entregando algoritmos eficientes, ferramentas para o controle de modelos matemáticos e análise dos resultados [5].

Existem diversos tipos de problemas de otimização que podem ser categorizados em diferentes classes. Também existem várias técnicas de otimização que podem ser aplicadas adequando-as para cada classe de problemas, como por exemplo, problemas definidos por funções convexas que podem ser otimizadas a partir de métodos que fazem uso de gradientes [6–8], ou problemas de Programação Linear (*Linear Programming* - LP) podendo ser resolvidas pelo método *Simplex* [9], ou ainda, funções não convexas tendo muitas abordagens de otimização heurísticas como: algoritmo genético (*Genetic Algorithm* -

GA) [10–12] e recozimento simulado (SA) [13, 14].

Devido os problemas de otimização terem suas representações mediante modelos matemáticos, permitem que os métodos formais possam ser uma importante ferramenta no desenvolvimento de novos métodos de otimização.

Os métodos formais são técnicas baseadas no formalismo matemático para a especificação, desenvolvimento e verificação de sistemas de *softwares* e *hardwares* [15–18]. O principal valor entregue por essas técnicas é prover um meio para examinar simbolicamente todo o espaço de estados de um sistema, além de estabelecer uma propriedade de segurança que seja verdadeira considerando todas as entradas possíveis [15–18].

Visando facilitar a aplicação do uso da verificação formal para a otimização de funções, a presente pesquisa emprega ferramentas de verificação de programas (ESBMC, CBMC), que aplicam de forma intrínseca da verificação formal denomiada Verificação de Modelos Limitados (*Bounded Model Checking* - BMC), para ajudar a estabelecer os algoritmos de otimização indutiva guiada por contraexemplos (CEGIO) [19] e o desenvolvimento de uma ferramenta de otimização (OptCE) [20].

Os verificadores de programas empregados nesta pesquisa realizam verificação formal (*Bounded Model Checking*), que refuta ou prova a exatidão de um algoritmo de acordo com uma especificação ou propriedade formal, através métodos formais matemáticos. A verificação é realizada fornecendo uma prova formal de um modelo matemático abstrato do sistema, que corresponde ao sistema real. O verificador de programas explora de forma sistemática e exaustiva o modelo matemático, percorrendo todos os caminhos possíveis.

Este trabalho tem como objetivo apresentar a ferramenta de otimização OptCE, que é baseada nos os algoritmos de otimização indutiva guiada por contraexemplos CEGIO. Durante o decorrer deste trabalho é apresentado à fundamentação teórica referente ao assunto; detalhamento dos algoritmos desenvolvidos CEGIO; detalhamento e avaliação experimental da ferramenta OptCE; como também as conclusões e propostas de melhorias do trabalho.

1.1 Descrição do Problema

Os problemas de otimização são descritos mediante modelos matemáticos podendo fazer alusão a um sistema físico real, onde as classes de funções convexas e não convexas tem grande importância para modelagem destes problemas de otimização, sendo que, as funções

não convexas apresentam maior grau de dificuldade que funções convexas perante as técnicas de otimização, podendo reportar mínimos locais em vez de mínimos globais.

As técnicas de otimização buscam o melhor desempenho para um sistema conforme sua modelagem matemática e as possibilidades existentes, entretanto muitas das técnicas de otimização proveem soluções subótimas ao invés de soluções ótimas, isso porque permanecem presas em mínimos locais confiando ter encontrado o mínimo global para a função.

Garantir a otimização global de uma função matemática é o problema que este trabalho propõe-se solucionar, utilizando técnicas de verificação de modelos limitadas de forma automatizada, localizando as variáveis de decisão que apontam para mínimo global da função.

1.2 Objetivos

Esta pesquisa visa desenvolver uma abordagem de otimização baseada em contraexemplos SAT e SMT, capaz de obter o mínimo global e suas variáveis de decisão correspondentes, a partir de funções de otimização modeladas por funções convexas e não convexas. Busca-se desenvolver uma ferramenta de otimização que encapsula a metodologia para proporcionar melhor usabilidade da proposta. Para atingir este objetivo, deverão ser contemplados os seguintes objetivos específicos:

- Desenvolver algoritmos para otimizar funções matemáticas convexas e não convexas usando os contraexemplos SAT e SMT dos verificadores de modelos limitados;
- Desenvolver uma ferramenta de otimização indutiva guiada por contraexemplos SAT e SMT;
- Validar a capacidade da ferramenta em otimizar funções convexas e não convexas;

1.3 Descrição da Solução

A abordagem proposta para otimização de funções convexas e não convexas faz uso da verificação formal e conceitos de otimização. Dessa forma, desenvolveu-se algoritmos e ferramentas correspondentes que foram avaliadas usando *benchmarks* clássicos de otimização. Para cada processo de otimização, a ferramenta implementa os algoritmos desenvolvidos

CEGIO usando os dados de entrada referente ao *benchmark*, para em seguida executar a verificação. A metodologia permite a convergência para a solução final executando sucessivas verificações dos arquivos de especificação, onde experimentos e deduções matemáticas confirmam a convergência para a solução ótima. Sua evolução é de forma interativa, encontrando soluções subótimas com precisão cada vez maior a cada verificação com resultado *failed*, e por conseguinte mais próximo do mínimo global, até a localização da solução ótima em uma verificação *success*.

A Figura 1.1 apresenta uma visão geral da abordagem estabelecida para uma otimização usando verificadores de programas. Assim como no processo de verificação de um programa, a abordagem é segmentada em três etapas: modelagem, especificação e verificação. As etapas são brevemente descritas a seguir:

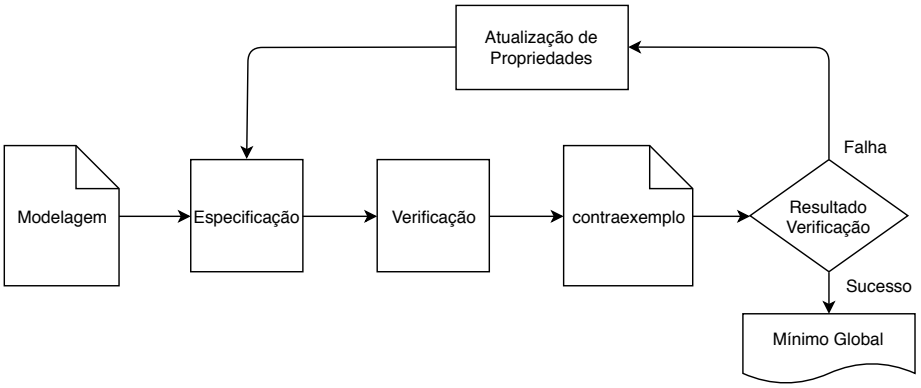

Figura 1.1: Metodologia proposta para solução.

- **Modelagem -** O processo de modelagem consiste no usuário descrever a modelagem do problema ou função matemática juntamente com suas restrições.

- **Especificação -** A ferramenta usa os dados da etapa de modelagem para gerar um arquivo de especificação. Uma propriedade é inserida em cada especificação, juntamente com dados que descrevem o sistema a ser otimizado, somado a um valor de inicialização que é gerado aleatoriamente.

- **Verificação -** A última etapa faz a verificação da propriedade existente no arquivo de especificação e gera um contraexemplo. O contraexemplo resultante indica *success* caso a propriedade não seja violada, ou *failed* caso a propriedade tenham sido violada.

Conforme os algoritmos CEGIO foram estabelecidos existem dois possíveis resultados para a verificação; caso a verificação seja *success*, indica que o mínimo global da função foi encontrado, já que em todo o espaço de busca não existe outro candidato a mínimo global que possa violar a propriedade; caso o resultado tenha sido *failed*, o verificador encontrou um candidato a mínimo global que é menor que o candidato anterior, a partir de então, o sistema atualiza a propriedade e usa o novo candidato encontrado no contraexemplo para realizar uma nova especificação. Este novo arquivo de especificação é verificado, de forma que o ciclo acontece até que seja encontrado o mínimo global da função. Vale enfatizar que a solução ótima encontrada é definida pelas restrições estabelecidas.

1.4 Contribuições

As principais contribuições desta dissertação são o estabelecimento da metodologia de otimização usando contraexemplos SAT e SMT [19], juntamente com o desenvolvimento dos algoritmos CEGIO e a criação da ferramenta de otimização guiada por contraexemplos SAT e SMT, OptCE [20].

O desenvolvimento dos algoritmos CEGIO permitiu estabelecer a abordagem de otimização que faz uso de contraexemplos SAT e SMT, sendo capaz de otimizar funções convexas e não convexas. Os algoritmos são especificados a partir da modelagem do usuário, de forma que possam ser verificados para localizar o mínimo global em funções convexas e não convexas. Foram desenvolvidos três algoritmos seguindo esta abordagem: um para funções semi-definidas positivas; outro específico para funções convexas e um terceiro que pode ser aplicado a funções convexas e não-convexas.

A concepção da ferramenta OptCE, implementa os algoritmos CEGIO desenvolvidos, contribuindo para o processo da otimização de funções convexas e não convexas, facilitando a configuração de verificadores e solucionadores. A partir da ferramenta OptCE é possível também executar outros experimentos, permitindo adaptar a abordagem para otimizar o máximo global de uma função. Uma *suíte* de testes, foi desenvolvida para auxiliar na avaliação experimental dos algoritmos, onde foram submetidas a otimização com a abordagem desenvolvida, para comparar com os resultados de otimizações dos *benchmarks* usando as técnicas de otimização: algoritmo genético, enxame de partículas, pesquisa de padrões, recozimento simulado e programação não linear.

1.5 Organização da Dissertação

O restante deste trabalho está organizado conforme descrito nos seguintes tópicos:

- **Capítulo 2** - Descreve a fundamentação teórica sobre a otimização e verificação formal. Na parte de otimização é relatado os conceitos gerais sobre otimização; o processo de modelagem de problemas; classificação e formulação matemática de problemas de otimização; diferença entre os mínimos locais e o mínimo global e breve descrição de algumas técnicas de otimização. Na parte de verificação é relatado os principais aspectos dos métodos de verificação, verificação formal, verificação de modelos limitados, verificadores de programas BMC.

- **Capítulo 3** - Detalha o desenvolvimento da abordagem de otimização indutiva guiada por contraexemplos de solucionadores, assim como os algoritmos desenvolvidos CEGIO.

- **Capítulo 4** - Expressa o desenvolvimento da primeira versão da ferramenta de otimização OptCE, que usa o método proposto de otimização guiada por contraexemplo.

- **Capítulo 5** - Salienta os resultados obtidos, as expectativas quanto ao desempenho da pesquisa, assim como as propostas de melhorias para a ferramenta OptCE.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os conceitos básicos empregados para o desenvolvimento desta pesquisa, na qual são ressaltados três itens: otimização, verificação e erros de precisão. Primeiramente é apresentado aspectos sobre a formulação matemática para problemas de otimização, a diferença básica entre mínimo local e mínimo global em uma função, bem como algumas técnicas de otimização. Em seguida, explana-se sobre a parte de verificação de uma forma abrangente e sobre os verificadores de programas, em especial sobre o ESBMC e CBMC. Por fim, apresenta-se maior detalhamento sobre erros e precisão numérica, onde é abordado aspectos acerca da representação numérica e erros de arredondamento.

2.1 Otimização

A otimização matemática, ou simplesmente otimização busca alcançar a melhor solução possível para um problema de acordo com suas restrições, como exemplo em um projeto ou construção, onde os engenheiros precisam tomar decisões. O objetivo de todas essas decisões é minimizar o esforço ou maximizar o benefício. O esforço ou o benefício, podem ser expressos por uma função, contendo um conjunto de variáveis de decisão podendo encontrar os pontos extremos desta função, ou seja, máximos e mínimos que podem ser assumidos em um dado intervalo de uma função objetivo ou função custo. Em resumo, a otimização é o processo para encontrar as variáveis que fornecem o valor máximo ou mínimo de uma determinada função [21].

No mundo real existem vários exemplos de problemas de otimização que exemplificam

e ressaltam sua importância em valores financeiros, tendo como exemplo a cidade de Montreal no Canadá, onde precisava alocar melhor sua estrutura de transporte que envolvia muitos motoristas de ônibus, de metrô, vendedores de bilhetes e guardas. Neste caso, um processo de otimização permitiu economizar cerca de quatro milhões de dólares canadenses por ano [21]. Outro exemplo que se pode resaltar, trata do departamento de polícia de São Francisco, de modo que a otimização permitiu planejar 20% mais rápido o problema de alocação de veículos policiais, economizando assim 11 milhões de dólares por ano [21]. Em outro caso, a companhia aérea *United Airlines*, a otimização permitiu resolver o problema de agendamento de tripulação economizando seis milhões de dólares ao ano. [21].

2.1.1 Modelagem dos Problemas de Otimização

Um problema de decisão enfrentado na realidade é transformado em um modelo de otimização através de algumas etapas. Precisa-se, primeiramente entender o problema, identificar os componentes essenciais, simplificar, limitar o problema, mas também quantificar as declarações qualitativas. Os dados dos problemas de otimização não são fáceis de coletar ou quantificar, pois muitas vezes há um conflito entre a resolução possível e o seu realismo [21].

A Figura 2.1 apresenta a visão geral dos procedimentos que são considerados durante o processo de modelagem de um problema de otimização, assim como as setas que indicam o ciclo deste processo.

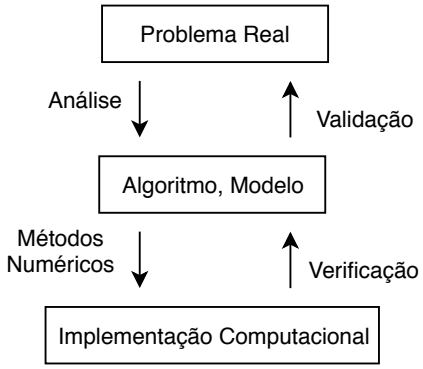

Figura 2.1: Ciclo do processo de otimização [22].

Inicialmente tem-se o problema real, composto por todos os detalhes e complexidades. Deste problema é extraído os elementos essenciais para criação de um modelo e escolha de uma técnica de otimização. Conforme se move para baixo no diagrama, existe uma perda do

realismo, do mundo real para o algoritmo ou modelo, e em seguida para uma implementação computacional [22].

No diagrama da Figura (2.1), o ciclo começa com a etapa de Análise que é importante para a otimização bem-sucedida, onde tem o trabalho de abstrair detalhes irrelevantes e focar nos principais elementos, a fim de construir o algoritmo adequado. Em seguida, tem-se a transição do algoritmo para a implementação computacional, fazendo uso de Métodos Numéricos de modo a realizar uma busca da sulução do problema dentro da região de factibilidade. Questões como precisão em computadores digitais e implementações eficientes de técnicas de inversão de matrizes são levados em consideração para esta problemática.

Após a concepção da implementação computacional é realizada a etapa de Verificação. Nessa etapa, busca-se checar se a implementação é executada conforme o planejado. Sabendo que os algoritmos funcionam como esperado, a etapa de Validação é executada, os resultados são comparados com o mundo real, para verificar se é necessário modificar a técnica aplicada. A etapa de validação é o processo que assegura se o modelo ou a técnica é apropriada para a situação em questão. Caso o modelo não atenda ao realismo necessário, o ciclo deve ser reiniciado [22].

2.1.2 Classificação dos Problemas de Otimização

Os problemas de otimização podem ser agrupados em classes conforme suas propriedades, podendo ser de diferentes maneiras, restrições, domínio de variáveis de decisão e a natureza da função de custo [23]:

- **Existência de restrições** - Um problema de otimização pode ser classificado por possuir restrições ou ser irrestrito.
- **Tipos de variáveis** - Dependendo dos valores atribuídos as variáveis de decisão, os problemas de otimização podem ser classificados como inteiros ou reais, determinísticos ou estocásticos.
- **Natureza da função** - Um problema de otimização pode ser classificado dependendo da natureza de sua função objetiva e restrições. Como por exemplo: linear, não linear, quadrática, polinomial.

- **Espaço de Busca** - Região convexa e não convexa. Este espaço é definido pelo conjunto de restrições.

Conforme o domínio e restrições da função custo, f, o espaço de busca da otimização pode ser pequeno ou grande, o que influência diretamente no desempenho dos algoritmos de otimização propostos. Será visto nas próximas seções o motivo da otimização irrestrita não ser interessante para a proposta deste trabalho.

Dado o tempo e recursos de memória disponíveis, a natureza da função custo e a resolução da solução, podem tornar complexa a otimização, sendo incapaz de solucionar alguns problemas [24].

2.1.3 Formulação do Problema de Otimização

Para um melhor entendimento sobre a formulação matemática da otimização, é usado o problema de otimização a seguir:

$$\begin{aligned} \min \quad & f(\mathbf{X}), \\ \text{s.t.} \quad & \mathbf{X} \in \Omega. \end{aligned} \tag{2.1}$$

A função $f : R^n \rightarrow R$ que deseja-se minimizar é uma função denominada de função objetivo ou função custo. O vetor X é um vetor de n-variáveis independentes, ou seja, $X = [X_1, X_2, ..., X_n]^T \in R^n$. As variáveis $X_1, ..., X_n$ são referidas como variáveis de decisão. O conjunto Ω é um subconjunto de R^n, chamado de conjunto de restrições [23, 25].

O problema de otimização descrito pode ser visto como um problema de decisão, cujo o seu objetivo é encontrar o vetor ótimo X de variáveis de decisão dentre todos os vetores possíveis existentes em Ω. Busca-se o vetor que obtém o menor valor da função objetivo, o qual é intitulado de vetor minimizador de f em Ω. Existe a possibilidade de haver mais de um minimizador, sendo suficiente encontrar apenas um para obter a solução ótima [23].

Existem casos em que se deseja encontrar o máximo de uma função, e em outros casos se deseja encontrar o mínimo. Para problemas onde se busca maximizar f, podem ser representados por seu equivalente $-f$. Portanto, pode-se considerar somente os problemas de minimização sem perda de generalidade.

O problema descrito na Equação (2.1) é uma generalização de um problema de otimização restrito, isso porque suas variáveis de decisão são definidas conforme o conjunto de

restrições Ω, onde $\Omega = x : h(x) = 0, g(x) \leq 0$, e h e g são funções de restrições. Caso fosse considerado $\Omega = R^n$, seria um problema de otimização sem restrição.

Na próxima subseção 2.1.4 é realizada uma breve discussão sobre as diferenças entre mínimos locais e mínimos globais, contudo é apresentado a seguir as definições matemáticas para o minimizador local e global.

Definição 2.1 ***Minimizador local*** *- Supondo que $f : R^n \to R$ é uma função definida em algum conjunto $\Omega \subset R^n$. Um ponto $x^* \in \Omega$ é um minimizador local de f em Ω se existe $\Delta > 0$ tal que $f(x) \geq f(x^*)$ para todos $x \in \Omega$, onde $||x - x^*|| < \Delta$.*

Definição 2.2 ***Minimizador global*** *- Um ponto $x^* \in \Omega$ é o minimizador global de f em $\Omega \iff$ $f(x) \geq f(x^*)$ para todos $x \in \Omega$.*

Caso nas definições acima, substituirmos "$\geq$" por "$>$", teremos estritamente um minimizador local e estritamente um minimizador global respectivamente.

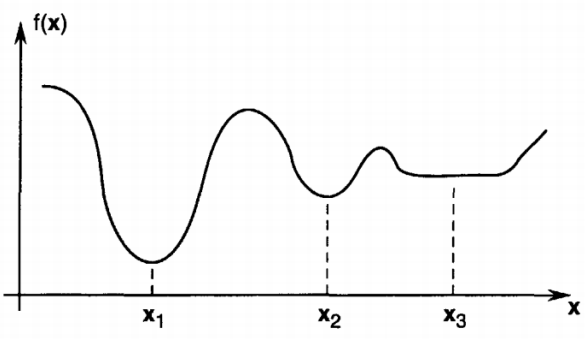

Figura 2.2: Exemplos de minimizadores: x_1 é estritamente um minimizador global; x_2 é estritamente um minimizador local; x_3 é minimizador local não estritamente [23].

A Figura 2.2 mostra graficamente as definições anteriores para $n = 1$. Dada uma função f a notação *arg min* $f(x)$ denota o argumento que minimiza a função f (um ponto no domínio de f), assumindo como ponto único. Por exemplo, se $f : R \to R$ é dado pela equação $f(x) = (x+1)^2 + 3$, então *arg min* é $f(x) = 3$. Escrevendo *arg* $min_{x \in \Omega}$, então trata-se Ω como o domínio de f. Por exemplo, para a função f acima, arg $\min_{x \geq 0} f(x) = 0$. Em geral, pode-se pensar em *arg* $min_{x \in \Omega} f(x)$ como o minimizador global de f em Ω (assumindo que existe e é único) [23].

2.1.4 Mínimo Local x Mínimo Global

Tecnicamente a melhor solução para um problema de otimização é o seu mínimo global, no entanto a solução global em geral é difícil de ser encontrada. Funções convexas, por exemplo, possuem apenas um ponto de inflexão, o que facilita as técnicas de otimização encontrarem o mínimo global. Por outro lado a otimização de problemas de funções não convexas é muito complexa, funções custo não convexas apresentam vários pontos de inflexão, como também podem direcionar o algoritmo de otimização para uma solução subótima em um mínimo local, em geral os algoritmos apontam para soluções aproximadas [23].

Recentemente, vários estudos relacionados a otimização global surgiram, contudo as soluções propostas são muito específicas para uma classe particular de problemas. Um exemplo de solução particular consiste em usar princípios de cálculo diferencial para otimizar a função convexa. Várias funções não convexas são globalmente otimizadas, transformando o problema não convexo em um conjunto de problemas convexos [26].

2.1.5 Técnicas de Otimização

Não existe um método único capaz de resolver todos os problemas de otimização de forma eficiente. Por isso, ao longo das décadas vários métodos foram desenvolvidos para solucionar diferentes classes de problemas de otimização.

As técnicas de otimização permitem alocar recursos limitados aos melhores resultados possíveis e são usados em todos os lugares, na indústria, no governo, na engenharia, assim como na ciência da computação, todos os dias essas técnicas são aplicadas em problemas reais para facilitar tomadas de decisões [22].

Muitas das técnicas de otimização usadas em larga escala, tem sua origem referida a métodos desenvolvidos durante a Segunda Guerra Mundial, que buscavam lidar com questões logísticas, limitação de suprimentos, pessoas e equipamentos [22]. Uma das primeiras técnicas práticas de otimização é o método *Simplex*, que teve seu aperfeiçoamento após a guerra, com o acesso aos primeiros computadores eletrônicos. Vale ressaltar que a maior parte de todos os cálculos nesses computadores foi dedicada à otimização por meio do método *Simplex* [9, 22].

As técnicas de otimização surgiram estimuladas por percepções em outros campos de estudos, como os algoritmos genéticos, que fazem analogia à codificação de cromossomos e seleção natural para “evoluir” boas soluções de otimização.

A seguir são brevemente apresentadas algumas técnicas de otimização que foram usadas em comparação com o método proposto.

Algoritmo Genético

O conceito de Algoritmos Genéticos (GA) faz uso de estratégias de busca de otimização padronizadas a partir de noções darwinianas de evolução e seleção natural. Durante uma otimização GA, um conjunto de soluções experimentais é escolhido e "evolui" para uma solução ótima sob a "pressão seletiva" da função objetivo [10, 27, 28].

Os otimizadores de GA procuram por soluções globais em domínios de função multimodal de alta dimensão. Os GAs diferem das técnicas tradicionais porque atuam em um grupo ou população, de soluções experimentais em paralelo. Elas operam em uma codificação dos parâmetros da função em vez dos parâmetros diretamente, usando operadores simples e estocásticos para explorar o domínio da solução [10, 27, 28].

Um simples GA deve ser capaz de executar cinco tarefas básicas: codificar os parâmetros da solução na forma de cromossomos, inicializar uma população do ponto inicial, avaliar como também atribuir valores de aptidão a indivíduos da população, além de realizar a reprodução através da seleção ponderada de indivíduos a população, e por fim realizar a recombinação e mutação para produzir membros da próxima geração [10, 27, 28].

Enxame de partículas

Semelhante ao GA, o Enxame de partículas (*Particle Swarm* - PSO) se inicia com uma população de soluções aleatórias Υ, onde ocorre a diferença, pois cada solução candidata tem uma atribuição aleatoria de velocidade denomminado de soluções de partículas [29, 30]. O método busca a melhor solução candidata interativamente de acordo com a precisão estabelecida.

Tais partículas se movem ao redor do espaço de busca, conforme o modelo matemático definido. Com o objetivo de direcionar o enxame para as melhores soluções, a posição atual conhecida é usada para determinar os movimentos das partículas, guiando-as para as posições mais conhecidas do espaço de busca, sempre atualizando as melhores posições encontradas [29, 30].

Pesquisa de Padrões

A Pesquisa de Padrões (*Pattern Search*) é um método de otimização que busca padrões heurísticos, que precisam apenas retornar o valor da função objetiva $f(x)$ para algum valor de entrada x [22]. São geralmente empregados em problemas, onde não é possível conhecer a primeira ou segunda derivada da função objetivo, como em problemas de programação não linear irrestrito [22]. Portanto, os métodos de busca de padrões são normalmente aplicados, quando as derivadas não são disponíveis.

Recozimento Simulado

A otimização de recozimento simulado (Simulated Annealing - SA) baseia-se analogamente ao processo físico de recozimento metalúrgico. Quando os metais são recozidos, em geral são controlados pelo processo de resfriamento, obtendo propriedades desejáveis como dureza, resistência, ductibilidade e outros [22]. Seu desenvolvimento ocorreu anteriormente ao desenvolvimento dos algoritmos genéticos e foi gradativamente substituído por eles em muitas aplicações [22].

Durante o processo de otimização quando o parâmetro “Temperatura” está elevado, muitas movimentações aleatórias são toleráveis. Porém, a medida que a “Temperatura” é reduzida, menos movimentos aleatórios são permitidos, até que a solução é fixada no estado final, que chamamos de “solução congelada”. Enquanto a “Temperatura” for alta o algoritmo faz ampla amostragem do espaço de soluções, gradativamente ocorre o esfriamento da “Temperatura” e o algoritmo se move para as laterais em direção à subida ou descida acentuadas, com o intuito de se desvencilhar do ótimo local obtido durante as altas temperaturas [22].

Uma importante características desta abordagem é que aceita deslocar-se para uma solução pior, conforme a probabilidade, e tal probabilidade é reduzida com o declíneo da “Temperatura”. Isso porque busca-se abandonar mínimos locais. Quando a “Temperatura” é pequena o suficiente, o algoritmo aceita apenas movimentos para soluçoes melhores, então o movimento diminui conforme a “Temperatura”, e o algoritmo converge para uma solução podendo ser ótima.

Programação não Linear

A programação não linear (*NonLinear Programming* - NLP) é composta por uma função objetiva, restrições gerais e limites das variáveis de decisão, assim como na programação linear, mas o que caracteriza o NLP, é o fato de que quando sua função objetivo não é linear, ou possui restrições não lineares, isso indica ser um problema não linear [22]. Muitos sistemas reais são inerentes de forma não linear. Por exemplo, modelando a queda na potência do sinal, conforme a distância de uma antena transmissora, por isso é importante que algoritmos de otimização sejam capazes de lidar com esta especificidade [22]. Existem muitos algoritmos que resolvem problemas de programação não linear, mas cada um deles é adaptado a um problema específico de otimização [31].

2.2 Métodos de Verificação

Os sistemas computacionais têm evoluído de forma exponencial. Nessa competição por melhores resultados, surgem sistemas cada vez mais complexos, e por sua vez em menor tempo de desenvolvimento. Agilizar o desenvolvimento de um programa poderá criar problemas com inconsistências no código, e por displicência permitir a manufatura de produtos em não conformidade, embarcando um programa com deficiências. Os projetistas utilizam métodos de validação de sistemas no processo produtivo, tais como, simulação, testes, verificação dedutiva e verificação de modelos [32].

Simulação e teste são realizados antes da implantação do sistema na prática, o primeiro abstrai um modelo do sistema para ser aplicado, enquanto que, o segundo faz uso de um protótipo que representa o sistema. Em ambos os casos, são inseridas entradas em partes do sistema e observadas às saídas. Esses métodos têm boa relação de custo-benefício para encontrar falhas, porém não são consideradas todas as interações dentro do sistema, o que não garante mapear todas as possibilidades [32].

A verificação dedutiva está relacionada ao uso de axiomas e prova de regras que validam se o sistema está correto. No início das pesquisas relacionadas à verificação, o foco da verificação dedutiva foi garantir que o sistema não apresentasse erros [33]. A importância de se ter um sistema correto era tanta, que o especialista poderia investir o tempo que fosse necessário para realização dessa tarefa [32]. Inicialmente, as provas eram todas construídas

manualmente, até que os pesquisadores perceberam que programas poderiam ser desenvolvidos para usar corretamente os axiomas e regras de prova. Tais ferramentas computacionais, também podem aplicar uma busca sistemática para sugerir várias maneiras de progredir a partir da fase atual da prova [32].

A verificação de modelos é uma técnica que se aplica a sistemas de estados finitos concorrentes, mas também pode ser usada em conjunto com outras técnicas de verificação. Essa técnica é realizada de forma automática, onde é feita uma busca exaustiva no espaço de estados do sistema para determinar se uma especificação é verdadeira ou falsa [32].

2.2.1 Métodos Formais

A verificação de sistemas complexos de *hardware* e *software* exige muito tempo e esforço. O uso de métodos formais proveem técnicas de verificação mais eficientes que reduz o tempo, minimiza o esforço e aumenta a cobertura da verificação [34].

Os métodos formais tem o objetivo de estabelecer um rigor matemático na verificação de sistemas [34], por isso são técnicas de verificação "altamente recomendadas" para o desenvolvimento de sistemas críticos de segurança do *software*, de acordo com a *International Electrotechnical Commission* (IEC) e a *European SpaceAgency* (ESA) [34]. As instituições *Federal Aviation Administration* (FAA) e *National Aeronautics and Space Administration* (NASA) reportaram resultados sobre os métodos formais concluindo que os métodos formais deveriam fazer da educação de cada engenheiro e cientista de *software* [34]

Nas últimas duas décadas as pesquisas com métodos formais, permitiram o desenvolvimento de algumas técnicas de verificação que facilitam a detecção de falhas. Tais técnicas são acompanhadas de ferramentas utilizadas para automatizar vários passos da verificação. A verificação formal mostrou que pode ser relevante ao expor defeitos como na missão *Ariane 5*, missão *Mars Pathfinder* [34] e acidentes com a máquina de radiação *Therac-25* [35].

2.2.2 Verificação de Modelos

A vericação de modelos atua em sistemas de estados finitos concorrentes, garantindo a validação de sistemas programáveis. O objetivo da técnica é provar matematicamente por meio

de métodos formais, que um algoritmo não viola uma propriedade considerando sua própria estrutura.

No início de 1980, a técnica de verificação de modelos passou por uma evolução quando os pesquisadores Clarke, Emerson e outros pesquisadores como J. P. Queille e J. Sifakis introduziram o uso da lógica temporal [34]. Observou-se que a verificação de um único modelo que satisfaz uma fórmula é menos complexo que provar a validade de uma fórmula para todos os modelos. A lógica temporal se mostra útil para especificar e verificar sistemas concorrentes, pois ela descreve a ordem dos eventos.

A aplicação da verificação de modelos consiste em três partes: modelagem; especificação e verificação [34].

- **Modelagem:** Converte o que se deseja verificar em um formalismo por uma ferramenta de verificação de modelos. Em muitos casos, isso é uma tarefa de compilação, em outros existem limitações de memória e tempo para a verificação, por isso a modelagem pode requerer o uso de abstrações e eliminar detalhes irrelevantes [34].

- **Especificação:** Antes de realizar a verificação é preciso saber quais propriedades serão verificadas. Essa especificação é geralmente realizada, através de algum formalismo lógico. Para sistemas de *hardware* e *software* é comum usar lógica temporal, podendo afirmar como se comporta a evolução do sistema ao longo do tempo. A verificação de modelos fornece meios para determinar se o modelo do *hardware* ou *software* satisfaz uma determinada especificação, contudo é impossível determinar se a especificação abrange todas as propriedades que o sistema deve satisfazer [34].

- **Verificação:** A ideia é que a verificação seja completamente automática, entretanto há casos que necessita do auxílio humano, como na análise dos resultados. Quando um resultado é negativo um contraexemplo é fornecido ao usuário, este contraexemplo pode ajudar a encontrar a origem do erro, indicar a propriedade que possui falha [34]. A análise do contraexemplo pode requerer a modificação do programa e reaplicar o algoritmo *model checking* [34]. O contraexemplo pode ser útil para detectar outros dois tipos de problemas, uma modelagem incorreta do sistema ou, uma especificação incorreta [34]. Uma última possibilidade é de que a tarefa de verificação falhe, devido ao tamanho do modelo que

pode ser grande e ter elevado consumo de memória [34], neste caso, talvez seja preciso realizar ajustes no modelo.

O maior desafio dessa abordagem são as explosões do espaço de estados, isso geralmente ocorre durante uma verificação, de mdo que muitos caminhos computacionais são traçados e verificados, consumindo todo potencial de memória da máquina que executa o teste. A medida que o número de variáveis de estado do sistema aumenta, o tamanho do espaço de estados cresce exponencialmente. Outros fatores que contribuem para esse crescimento são as interações das variáveis, a atuação em paralelo ou ainda o não determinismo presente em algumas estruturas do sistema [34].

A vantagem da verificação de modelos está em automatizar os testes, obter resultados em tempos menores, tornando-a preferível à verificação dedutiva. O processo realiza uma busca exaustiva no espaço de estados do sistema, e ao terminar é gerada uma resposta que afirma se o modelo satisfaz a especificação ou retorna um contraexemplo, onde mostra o motivo pelo qual não é satisfatório [34].

2.2.3 Verificação de Modelos Limitados - BMC

A verificação de modelos limitados (BMC) é uma técnica importante que vem apresentando bons resultados, já foi aplicada com sucesso para verificar *software* embarcado e pode descobrir erros em projetos reais [36]. Pode-se dizer que é uma extensão da verificação de modelos relatado na subseção 2.2.2, mas este tem um fator limitante aplicado a pesquisa de sua árvore de alcançabilidade (*Reachability Tree* - RT).

A ideia do BMC é verificar (a negação de) uma dada propriedade a uma dada profundidade em um sistema. Dado um sistema de transição M, uma propriedade ϕ, e o limite k, o BMC desenrola o sistema k vezes e o traduz em uma condição de verificação (*VC*) ψ, tal que ψ é satisfatível se e somente se ϕ tiver um contraexemplo de profundidade menor ou igual a k [15].

O BMC gera *VC*s que espelha o caminho exato no qual uma instrução é executada, o contexto em que uma função é chamada e descreve a representação precisa em *bit*s das expressões [37]. Para o BMC, um conjunto de fórmulas $\{p_1, p_2, ..., p_n\}$ é dito ser satisfatível se houver alguma estrutura Λ, na qual todas as suas fórmulas componentes sejam verdadeiras, ou seja, $\{p_1, p_2, ..., p_n\}$ é `SAT` se e somente se $\Lambda \models p_1 \wedge \Lambda \models p_2 ... \wedge \Lambda \models p_n$.

Um dos graves problemas enfrentados pelo BMC, herdado da verificação de modelos, é a explosão do espaço de estados. Como visto na subseção 2.2.2, isso ocorre devido o BMC gerar múltiplos caminhos de execução do programa em verificação, causando elevado consumo de memória e sobrecarregando o sistema computacional usado no processo. Outra dificuldade dessa técnica é comprovar a validade das VCs, o que implica no desempenho da verificação dos programas [37].

A técnica BMC é baseada na Satisfatibilidade Booleana (SAT) ou na Teoria do Módulo de Satisfatibilidade (SMT) [38]. O BMC baseado em SAT foi introduzido como uma técnica complementar aos diagramas de decisão, binária para auxiliar no problema da explosão do estado [18], este pode verificar se ϕ é satisfatível. Para lidar com o aumento da complexidade dos sistemas, os solucionadores SMT são usados como *backends* para resolver *VC*s geradas por instâncias do BMC [39–41].

Em SMT, predicados de várias teóricas não são codificadas por variáveis proposicionais como em SAT, mas permanecem na fórmula do problema. Essas teorias são tratadas de forma dedicada, dessa maneira no BMC baseado em SMT, faça com que ϕ seja uma propriedade em um subconjunto de lógica de primeira ordem, que é então verificada quanto à satisfatibilidade por um solucionador de SMT [15].

O uso do BMC para esta pesquisa tem influência significativa para os bons resultados alcançados quanto a taxa de acerto, que é mostrado mais adiante na seção 5. Isso ocorre pois a técnica BMC é capaz de percorrer os caminhos possíveis de execução de um código durante uma verificação. Na metodologia estabelecida que será apresentada na Seção 3.2, o espaço de estados existentes de acordo com as propriedades que são estabelecidas a partir da formulação matemática e restrições da função, são verificadas pelo BMC e garentem que as possibilidades sejam checadas exaustivamente usando solucionadores SAT e SMT, permitindo localizar o mínimo global conforme os experimentos executados.

2.3 Verificadores de Programas BMC

Como já visto na seção (2.2), as provas matemáticas do processo de verificação eram todas construídas de forma manual, até adotarem programas para verificar corretamente axiomas e regras de prova, criando um processo de verificação automatizado onde não é exigido que o usuário insira pré e pós-condições nos programas, sem a necessidade de alterar o

programa original em questão.

Relatos de trabalhos anteriores apresentam que os verificadores de programas, em especial ferramentas BMC, limitavam-se apenas as teorias de funções não interpretadas, *arrays* e aritmética linear [39, 40], a abordagem do BMC baseada em SMT não suportava a verificação de estouro aritmético e não fazia uso de informações de alto nível para simplificar a fórmula desenrolada. Os verificadores de programas começaram a usar diferentes teorias de solucionadores de SMT, para traduzir precisamente expressões de programa em fórmulas livres e aplicando técnicas de otimização, a fim de evitar sobrecarregar o solucionador, começaram a tratar problemas de estouro aritmético, operações de vetor de *bit*, matrizes, estruturas, uniões e ponteiros. [37]. A evolução dos verificadores permitiu verificar programas *mult-threads*, fator importante considerando a evolução dos sistemas computacionais que passaram a trabalhar com mais de um núcleo. Com o passar dos anos ficou comum o uso de *threads* para elaboração de códigos, e estes precisavam de uma abordagem quanto à verificação. Entre os desafios desta evolução, encontra-se o problema da explosão do espaço de estados, já que o número de intercalações cresce exponencialmente com o número de encadeamentos e instruções do programa.

A seguir, são apresentados os verificadores usados neste trabalho, ESBMC e CBMC, expondo suas competências e limitações, bem como comentando basicamente seu funcionamento e modo de operação.

2.3.1 ESBMC

A ferramenta ESBMC é um verificador de programas BMC que faz uso de solucionadores SMT para checar programas escritos em C e $C++$ [42–44]. Este é capaz de verificar programas simples ou *multi-tarefas*, programas com *arrays*, ponteiros, *structs*, *unions*, alocação de memória, tratar com aritmética de ponto fixo e flutuante. Esta ferramenta é capaz de argumentar sobre *overflows*, segurança de ponteiro, vazamento de memória, atomicidade e violação de ordens, *deadlocks* local e global, concorrência de dados, bem como *asserts* especificados pelo usuário.

Dentro do ESBMC, os programas são modelados como sistemas de transição de estados $M = (S, R, s_0)$, extraído do gráfico de fluxo de controle (*Control Flow Graph* - CFG). A variável S representa o conjunto de estados, $R \subseteq SxS$ representa o conjunto de transições e

$s_0 \subseteq S$ representa o conjunto de estados iniciais. Um estado inicial s_0 atribui a localização inicial do programa do CFG para o contador *pc*. Cada transição é identificada, $\gamma = (si, s_{i+1}) \in R$ entre dois estados s_i e s_{i+1} com uma fórmula lógica $\gamma(si, s_{i+1})$ que captura as restrições nos valores correspondentes do contador de programa e as variáveis do programa. Dado um sistema de transição M, uma propriedade de segurança ϕ, um limite de contexto C e o limite k, o ESBMC constrói uma árvore de alcançabilidade (RT) que representa o programa que se desdobra para C, k e ϕ.

Derivando então um *VC* ψ_k^π para cada intercalação dada $\pi = \{v_1, ..., v_k\}$, tal que ψ_k^π é satisfatível se, e somente se, ϕ tem um contraexemplo de profundidade k que é exibido por π, ψ_k^π é dado pela seguinte fórmula lógica:

$$\psi_k = I(S_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} (\gamma(s_j, s_{j+1}) \wedge \overline{\phi(s_i)}) \tag{2.2}$$

O conjunto de estados iniciais de M e $\gamma = (s_i, s_{i+1})$ é a relação de transição de M entre os momentos j e $j+1$. Portanto, $I(s_0) \bigwedge \bigwedge_{j=0}^{i-1} (\gamma(s_j, s_{j+1})$ representa execuções de M de comprimento i e ψ_k^π podem ser satisfeitas se, e somente se, para algum $i \leq k$ existir uma variável alcançável estado ao longo de π no instante i em que ϕ é violado. A condição de verificação ψ_k^π é uma *VC* de quantificador em um subconjunto decidível de lógica de primeira ordem, que é verificada quanto à satisfatibilidade por um solucionador de SMT. Se ψ_k^π é satisfatível, então ϕ é violado ao longo de π, e o solucionador SMT fornece uma atribuição satisfatória, a partir da qual pode-se extrair os valores das variáveis de programa para construir um contraexemplo.

O contraexemplo de uma propriedade ϕ é uma sequência de estados $s_0, s_1, ..., s_k$ com $s_0 \in S_0$, $s_k \in S$ e $\gamma(si, s_{i+1})$ para $0 \leq i < k$. Se ψ_k^π é insatisfatível, pode-se concluir que nenhum estado de erro é alcançável na profundidade menor ou igual a k ou ao longo de π. Finalmente, pode-se definir $\psi_k = \bigwedge_\pi \psi_k^\pi$ e assim usá-lo para verificar todos os caminhos. No entanto, o ESBMC combina a verificação de modelo simbólico com a exploração explícita do espaço de estado, em particular, explora explicitamente os possíveis intercalamentos enquanto trata simbolicamente cada intercalação em sua execução. O ESBMC simplesmente percorre a profundidade RT primeiro, e chama o procedimento BMC de encadeamento único na intercalação sempre que atinge um nó folha de RT.

O ESBMC foi estendido para verificar programas escritos em CUDA [45–47] e

Qt [48–50], facilitando desta forma o seu uso em problemas práticos da indústria, também sendo aplicado para checagem da satisfatibilidade quanto ao particionamento em sistemas embarcados [51, 52].

2.3.2 CBMC

CBMC é um verificador *Bounded Model Checker* para programas *C* e $C++$. Como afirmam os autores, Daniel Kroening [53], ele suporta *C*89, *C*99, grande parte do C11 e a maioria das extensões dos compiladores fornecidos pelas ferramentas *gcc* e *Visual Studio*, o CBMC verifica os limites de *array*, segurança do ponteiro e asserções especificadas pelo usuário. A verificação é realizada desenrolando os *loops* no programa e passando a equação resultante para um procedimento de decisão. O CBMC suporta solucionadores internos baseados no MiniSat [17].

Entre suas vantagens e aplicações é capaz de atuar sobre programas concorrentes, verificando programas C executados por múltiplas-*threads*, pode encontrar e mostrar a causa de um erro em um programa. Verificam programas embarcados e modelos formalizados usando código C, programas existentes como *drivers* de dispositivo Linux e Windows.

Assim como no ESBMC, o CBMC recebe como entrada um programa juntamente com as declarações de propriedades a serem satisfeitas e um limite *k*, que define o desdobramento máximo feito para os *loops* do programa.

O mecanismo de análise interno do verificador gera uma fórmula na forma formal conjuntiva (*Conjunctive Normal Form* - CNF) que descreve o programa a ser verificado, juntamente com as propriedades especificadas no programa [17].

O CBMC procura por uma atribuição que satisfaça, tanto o problema quanto a negação da propriedade, para mostrar um contraexemplo [54], ou para provar que não existe tal contraexemplo até a *k* execução limitada. O CNF resultante é então usado por um solucionador SAT, que afirma que é satisfatório ou insatisfatório, mantendo ou não a propriedade [17].

A formalização de um modelo não é tão simples e requer lidar com vários pequenos problemas, como *loops*, modelo aritmético entre outros. Portanto, um dos grandes desafios deste trabalho é formalizar modelos relacionados a otimização que possam ser usados pelos verificadores de programas, para obter uma solução provida pela técnica de verificação BMC [17].

2.4 Erro e Precisão

O sistema computacional possui formas para representar os números em uma quantidade fixa de *bits*, por isso o espaço de armazenamento de informações em um computador não é infinito. O programador tem a opção de escolher diferentes formas de representação ou diferentes tipos de dados como `float` ou `int`. Os tipos de dados podem definir o número de *bits*, assim como informações mais intrínsecas, tal qual a capacidade de armazenamento e a definição do sinal do numeral.

2.4.1 Representação Numérica

Um sistema computacional tem sua representação numérica interna realizada através de dígitos binários, que por sua vez são agrupados em conjuntos maiores, como *bytes*. Basicamente, os sistemas computacionais possuem duas categorias para representação de um número, podendo ser ponto fixo ou ponto flutuante.

O número de *bits* necessários para a precisão e o intervalo desejados, devem ser escolhidos para armazenar as partes fracionária e inteira de um número. Em uma representação de ponto fixo o número é retratado por três partes: sinal, parte inteira e parte fracionária. Quando necessita-se armazenar um valor em um sistema computacional de 32 *bits*, 1 *bit* é destinado ao sinal, outros 15 *bits* são destinados para a parte inteira, por fim, 16 *bits* para a parte fracionária. Para este caso, $2^{-16} \approx 0.00001526$ é o intervalo entre dois números de ponto fixo adjacentes. Dessa maneira, um número que exceda 32 *bits* será armazenado incorretamente em um sistema computacional de 32 *bits*, este é um problema de estouro de memória, que ocorre ao tentar escrever dados em um *buffer* ultrapassando os limites do mesmo, sobrescrevendo a memória adjacente [55].

Existe a representação alternativa que ajuda a resolver o problema de estouro de memória, representação em ponto flutuante, onde o número é representado internamente por um sinal SS (interpretado como mais ou menos), um expoente inteiro exato E, e uma representação exata mantissabinária MM, como:

$$SS \times MM \times b^{E-e}, \tag{2.3}$$

onde, b é a base na representação, geralmente com valor $b = 2$. Devido as operações de

multiplicação e divisão em 2, poderem ser realizadas por deslocamento à esquerda ou à direita dos *bits*; e é o viés do expoente, uma constante de número inteiro fixado para qualquer representação, como por exemplo para valores de 32 *bit*, o valor *float*, o expoente é representado com 8 *bit* ($e = 127$), para valores de 64 *bit* o valor *double*, o expoente é representado com 11 *bit* ($e = 1023$) [55].

Para representação de 32 *bits*, o menor número normalizado positivo é $2^{-126} \approx 1.18 \times 10^{-38}$, que é muito menor do que em uma representação de ponto fixo. Além disso, o espaçamento entre os números de ponto flutuante não é uniforme, à medida que afasta-se da origem, o espaçamento torna-se menos denso. A maioria dos processadores modernos adotam a mesma representação de dados de ponto flutuante, conforme especificado pelo padrão IEEE 754-1985 [56]. Quando a precisão absoluta é necessária, o ponto fixo é a melhor opção, mas ponto flutuante na maoria dos casos é mais apropriado.

2.4.2 Erros de Arredondamento e Truncamento

A representação de ponto flutuante em geral tem semelhanças aos números reais, mas existem inconsistências entre o comportamento de números de ponto flutuante na base 2 e números reais. Em geral as causas das inconsistências no cálculo do ponto flutuante são o arredondamento e o truncamento.

A aritmética entre os números na representação de ponto flutuante não é exata, mesmo se os operandos forem exatamente representados, ou seja, eles têm valores exatos na forma da Equação (2.3).

A precisão da máquina, ε_m, é o menor número de ponto flutuante (em magnitude), que deve ser adicionado ao número de ponto flutuante $1,0$ para produzir um resultado de ponto flutuante diferente de $1,0$. Padrão IEEE 754 *float* tem ε_m sobre $1,19 \times 10^{-7}$, enquanto *double* tem cerca de $2,22 \times 10^{-16}$.

A precisão da máquina é a precisão fracionária, na qual os números em ponto flutuante são representados correspondendo a uma mudança de 1 no *bit* menos significativo da mantissa. Quase nenhuma operação aritmética entre números de ponto flutuante deve ser considerada, como introdução a um erro fracionário adicional de pelo menos ε_m. Esse tipo de erro é chamado de erro *roundoff* ou arredondamento.

O erro de arredondamento é uma característica da capacidade do *hardware* do

computador. Existe outro tipo de erro que é uma característica do programa ou algoritmo usado, independente do *hardware* em que o programa é executado. Muitos algoritmos numéricos calculam aproximações discretas para algumas quantidades de dados contínuos. Nestes casos, existe um parâmetro ajustável, onde qualquer cálculo prático é feito com uma escolha finita, mas suficientemente grande de parâmetros. A discrepância entre a resposta verdadeira e a resposta obtida em um cálculo prático, é chamada de erro de truncamento. O erro de truncamento persiste mesmo em um computador perfeito que tenha uma representação infinitamente precisa e nenhum erro de arredondamento.

Como regra geral, não há muito o que o programador possa fazer sobre o erro de arredondamento. No entanto, o erro de truncamento está totalmente sob o controle do programador.

2.5 Resumo

Neste capítulo foi apresentado uma visão geral sobre a otimização, principalmente a formulação matemática básica de um problema desta área de pesquisa, a diferença entre mínimos locais e mínimos globais e ainda um apanhado sobre as técnicas de otimização tradicionais, que são usadas em comparação com a ferramenta OptCE. Na sequência, foi apresentado formas de verificação, em especial sobre a verificação de modelos limitados, bem como é abordado aspectos acerca de verificadores de programas BMC, em especial as ferramentas (ESBMC e CBMC) usadas para geração de contraexemplos neste trabalho. Por fim, tem-se uma visão geral sobre erros e precisão numérica que estão diretamente ligados ao bom funcionamento dos algoritmos que serão apresentados na seção 3.

Capítulo 3

Otimização Intuitiva Guiada por Contraexemplos SAT e SMT

Este capítulo apresenta a metodologia de otimização usando contraexemplos SAT e SMT, juntamente com os algoritmos CEGIO [19]. O objetivo deste capítulo é esclarecer o funcionamento da otimização guiada indutiva por contraexemplo, bem como os obstáculos e as medidas adotadas, afim de soluciona-las, detalhando os algoritmos CEGIO, explanando cada parte do algoritmo e sua finalidade para a metodologia.

3.1 Otimização baseada em Contraexemplos

O processo de otimização intuitiva guiada por contraexemplos é realizado por meio de verificadores, que são capazes de veriificar a modelagem matemática de um problema, assim como retornar `TRUE` ou `FALSE`, conforme as condições de verificação especificadas. Dentre os verificadores para linguagens $C/C++$ existem duas importantes diretivas usadas para modelar e controlar o processo de verificação, `ASSUME` e `ASSERT`. A diretiva `ASSUME` é usada para definir restrições sobre variáveis (determinísticas e não determinísticas) e a diretiva `ASSERT` é empregada para verificar uma determinada propriedade.

Essas duas instruções, estão presentes em verificadores do modelo $C/C++$, como por exemplo CBMC [17], CPAChecker [57], e ESBMC [58]), podendo ser aplicado para verificar restrições específicas em problemas de otimização, conforme descrito pela subseção 2.1.3.

O processo de verificação usa funções intrínsecas disponíveis nos verificadores para

solucionar o problema de otimização, como por exemplo no ESBMC (*__ESBMC_assume* e *__ESBMC_assert*). A verificação é repetida interativamente de forma que o espaço de estados extraído do contraexemplo gerado pelo solucionador SMT ou SAT, seja reduzido a cada interação.

Uma propriedade é especificada para garantir a convergência para o ponto mínimo em cada iteração. O valor mínimo de uma função candidata é dado por (f_c) e a especificação da propriedade é dada pela equivalência da Equação (3.1), onde a diretiva `ASSERT` verifica se l_{otimo} é satisfatível para todos os candidatos ótimos f_c, encontrados na busca do espaço de estados.

$$l_{otimo} \iff f(\mathbf{x}) > f_c. \tag{3.1}$$

A função candidata é analisada por meio da verificação da satisfatibilidade de $\neg l_{otimo}$, ou seja, a negação da propriedade. Sendo assim, se l_{otimo} não for satisfeito, então existe um $x^{(i)}$, tal que $f(\mathbf{x}^{(i)}) \leq f_c$, isto é, existe um novo candidato a mínimo global $f(\mathbf{x}^{(i)})$ que é menor que o candidato anterior f_c, onde $x^{(i)}$ será obtido por meio do contraexemplo.

Nesta etapa surgem alguns problemas, devido a aritmética de precisão finita dos verificadores, podendo ocorrer truncamentos, fato percebido principalmente em funções com elevado expoente, numerosas multiplicações e espaço de busca que permitem variáveis de decisão com valores altos. Estes problemas foram observados usando solucionadores com aritmética de ponto fixo, como o Boolector e ESBMC, mas também com a aritmética de ponto flutuante, como nos solucionadores Z3 e MathSAT usados pelo ESBMC. Sendo assim, existem problemas com erro de precisão. Para contornar este problema, o literal l_{otimo} é modificado conforme a Equação (3.2)

$$l_{otimo} \iff f(\mathbf{x}) > f_c - \delta, \tag{3.2}$$

onde, δ é um valor de compensação com objetivo de reduzir os efeitos da representação numérica e erros do truncamento nos cálculos. Contrapondo-se a isso, se $\neg l_{otimo}$ não for satisfeito, então f_c não é a função mínima, mas estará a uma distância limitada por δ do valor mínimo.

A idéia é utilizar a habilidade dos verificadores para verificar a satisfatibilidade de uma determinada propriedade, em seguida retornar um contraexemplo que contenha o rastreamento do erro. Através de sucessivas verificações de satisfatibilidade do literal $\neg l_{otimo}$, pode-se

orientar o processo de verificação para resolver o problema de otimização descrito na subseção 2.1.3

3.2 Exemplo de Otimização Indutiva Guiada por Contraexemplo

Para descrever o processo da Otimização Indutiva Guiada por Contraexemplo, é usado o exemplo abordado no periódico [19]. A função *Ursem*03 [59] é um problema de otimização não convexo e possui quatro mínimos locais espaçados regularmente posicionados em uma circunferência, com o global mínimo no centro, sua representação é realizada por uma função de duas variáveis com apenas um mínimo global em $f(0,0) = -3$. A definição de Ursem03 é apresentada na Equação (3.3), enquanto que a Figura (3.1) apresenta o gráfico da função.

$$f(x_1,x_2) = -\sin\left(2.2\pi x_1 - \frac{\pi}{2}\right)\frac{(2-|x_1|)(3-|x_1|)}{4} - \sin\left(2.2\pi x_2 - \frac{\pi}{2}\right)\frac{(2-|x_2|)(3-|x_2|)}{4} \tag{3.3}$$

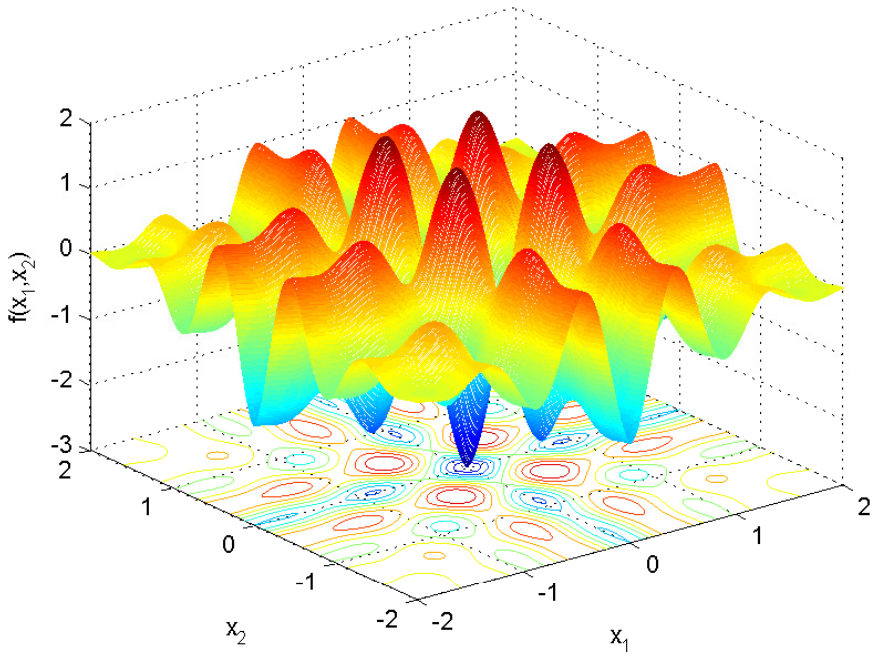

Figura 3.1: Função Ursem03

Para realizar o processo de otimização da função Ursem03, os seguintes passos são executados: modelagem, especificação e verificação, descritos nas respectivas seções, 3.2.1,

3.2.2 e 3.2.3.

3.2.1 Modelagem

Nesta etapa se define as restrições do problema e restrições das variáveis de decisão da função. Esta etapa é importante, pois reduz o espaço de estados a ser buscado, e consequentemente ajuda a evitar a explosão do espaço de estados.

A abordagem faz uso da verificação do modelo, o que torna ineficiente para otimização sem restrições, isso pois sem restrições elevaria o número de VCs a serem verificadas, de tal forma que consumiria todos os recursos de memória e processamento do ambiente de experimentos. A escolha adequada das restrições reduz consideravelmente o espaço de estados e torna a abordagem viável.

O problema de otimização apresentado na Equação (3.4) pertence à função *Ursem03* apresentada na Equação (3.3), onde as restrições são definidas por intervalos semifechados que levam para um grande domínio.

$$\begin{aligned} \min \quad & f(x_1, x_2) \\ \text{s.t.} \quad & x_1 \geq 0, \\ & x_2 \geq 0. \end{aligned} \tag{3.4}$$

As desigualdades $x_1 \geq 0$ e $x_2 \geq 0$ levam a pesquisa de espaço de estados para o primeiro quadrante; contudo, um grande espaço de estados ainda precisa ser explorado, uma vez que x_1 e x_2 podem assumir valores muito altos. O problema de otimização dado pela na Equação (3.4) pode ser melhor reescrito, como apresenta a Equação (3.5) inserindo novas restrições. Os limites são escolhidos com base no estudo descrito por Jamil e Yang [60], que define o domínio no qual os algoritmos de otimização podem avaliar as funções de *benchmark*, inclusive a função de *Ursem03*.

$$\begin{aligned} \min \quad & f(x_1, x_2) \\ \text{s.t.} \quad & -2 \leq x_1 \leq 2, \\ & -2 \leq x_2 \leq 2. \end{aligned} \tag{3.5}$$

Com o problema de otimização modelado por meio da Equação (3.5) é realizada a codificação, onde as variáveis de decisão são declaradas como variáveis não determinísticas, sendo restritas com o uso da diretiva `ASSUME`, nesse caso $-2 \leq x_1 \leq 2$ e $-2 \leq x_2 \leq 2$. Também

é descrito no código o modelo matemático da função em questão. A Figura (3.2) mostra um exemplo de modelagem da Equação (3.5) no respectivo código em *C*.

```
1  #include "math2.h"
2  float nondet_float();
3  int main() {
4    //define decision variables
5    float x1 = nondet_float();
6    float x2 = nondet_float();
7    //constrain the state-space search
8    __ESBMC_assume((x1>=-2) && (x1<=2));
9    __ESBMC_assume((x2>=-2) && (x2<=2));
10   //computing Ursem's function
11   float fobj;
12   fobj= -sin(2.2*pi*x1-pi/2)*(2-abs(x1))(3-abs(x1))/4
13   -sin(2.2*pi*x2-pi/2)*(2-abs(x2))(3-abs(x2))/4;
14   return 0;
15 }
```

Figura 3.2: Código *C* criado durante a etapa de Modelagem. Refere-se ao problema de otimização dado na Equação (3.5).

3.2.2 Especificação

Na etapa seguinte, a especificação, é descrito o comportamento do sistema e as propriedades a serem verificadas. No exemplo da função *Ursem03*, o resultado da etapa de especificação é o programa *C* mostrado na Figura (3.3), que é verificado pelos programas de verificação. As variáveis de decisão são declaradas como tipo inteiro e sua inicialização varia conforme a precisão atribuída a variável *p*, que é ajustada iterativamente quando o contraexemplo é gerado pelo solucionador. Se as variáveis de decisão fossem declaradas em ponto flutuante não-determinístico, como apresentado no código da Figura (3.2), faria o verificador realizar uma exploração de espaço de estados muito grande, por isso as variáveis de decisão são definidas como inteiras não determinísticas, reduzindo a explosão do espaço de estados, no entanto, isso reduz a precisão do processo de otimização.

A precisão é um parâmetro definido conforme o problema, sendo necessária ser elevada em alguns problemas, contudo em outros não se faz necessário. Entretanto, os algoritmos CEGIO tem capacidade para otimizar funções em qualquer precisão desejada pelo usuário, considerando os limites de tempo e memória existentes. Com o intuito de compensar a precisão e o tempo de verificação, mantendo a convergência para uma solução ótima, o procedimento

de verificação da especificação é repetido de forma iterativa, aumentando a precisão em cada execução sucessiva.

```
#include "math2.h"
#define p 1 //precision variable
int nondet_int();
float nondet_float();
int main() {
  float f_c = 100;  //candidate value of objective function
  int lim_inf_x1 = -2*p;
  int lim_sup_x1 = 2*p;
  int lim_inf_x2 = -2*p;
  int lim_sup_x2 = 2*p;
  int X1 = nondet_int();
  int X2 = nondet_int();
  float x1 = float nondet_float();
  float x2 = float nondet_float();
  __ESBMC_assume( (X1>=lim_inf_x1) && (X1<=lim_sup_x1) );
  __ESBMC_assume( (X2>=lim_inf_x2) && (X2<=lim_sup_x2) );
  __ESBMC_assume( x1 = (float) X1/p );
  __ESBMC_assume( x2 = (float) X2/p );
  float fobj;
  fobj= -sin2(2.2*pi*x1-pi/2)*(2-abs2(x1))(3-abs2(x1))/4
  -sin2(2.2*pi*x2-pi/2)*(2-abs2(x2))(3-abs2(x2))/4;
  //constrain to exclude fobj>f_c
  __ESBMC_assume( fobj < f_c );
  assert( fobj > f_c );
  return 0;
}
```

Figura 3.3: Especificação para a função de acordo com as restrições Equação (3.5).

Para compensar tanto a precisão quanto o tempo de verificação e também para manter a convergência para uma solução ótima, o procedimento de verificação de modelo subjacente deve ser chamado iterativamente, a fim de aumentar sua precisão para cada execução sucessiva.

$$p = 10^n. \tag{3.6}$$

A variável inteira p do código de especificação é declarada e iterativamente incrementada conforme a quação 3.6, de forma que n é iniciado com valor 0 e incrementa a quantidade de casas decimais relacionadas às variáveis de decisão.

Inicialmente, todos os elementos na pesquisa do espaço de estados Ω são candidatos a pontos ótimos, mas conforme as interações ocorrem, um novo valor da função objetiva $f(\mathbf{x}^{(i)})$ na i-ésima verificação não deve ser maior que o valor obtido na iteração anterior $f(\mathbf{x}^{(i-1)})$, e uma

nova restrição (linha 23 da Figura (3.3) é inserida para desconsiderar esses vários candidatos que são maiores que o mínimo atual.

Além desta restrição, a propriedade descrita pela Equação (3.2) é inserida por meio de uma assertiva que garante a convergência para o ponto mínimo, verificando se o $\neg l_{otimo}$ é satisfatível para cada valor de $f(\mathbf{x})$. Quando $\neg l_{otimo}$ não é satisfatível, isto é, se houver qualquer $\mathbf{x}^{(i)}$ para o qual $f(\mathbf{x}^{(i)}) \leq f_c$, é gerado um contraexemplo que mostra as variáveis de decisão $\mathbf{x}^{(i)}$, convergindo iterativamente $f(\mathbf{x})$ para a solução ótima $\mathbf{x}^*$.

A Figura (3.3) mostra a especificação inicial para o problema de otimização dado pela Equação (3.5). O valor inicial do candidato da função objetivo pode ser inicializado aleatoriamente. Para o exemplo mostrado na Figura (3.3), $f_c(\mathbf{x}^{(0)})$ é inicializado arbitrariamente para 100, mas o algoritmo funciona com qualquer estado inicial.

3.2.3 Verificação

Por fim, tem-se a etapa de verificação, onde o programa mostrado na Figura (3.3) é avaliado pelo verificador, e um contraexemplo é retornado com um conjunto de variáveis de decisão $\mathbf{x}$, cujo o valor da função objetivo obtida com essas variáveis converge para a solução ótima. O código C de especificação retornará um resultado de verificação bem sucedido, somente se o valor da função anterior for o ponto ótimo conforme a precisão especificada em p. Neste caso, o contraexemplo retorna as seguintes variáveis de decisão: $x_1 = 2$ e $x_2 = 0$. Essa informação é fundamental para a convergência do algoritmo, pois assim ele calcula a nova função $f(\mathbf{x}^{(i)})$ que será a nova candidata mínima $f_c(x)$ na próxima interação, que é $f(2,0) = -1,5$ menor que 100, sendo assim uma nova verificação será feita considerando o novo valor $f_c(x) = -1,5$. Este processo ocorre até que uma verificação seja bem sucedida, e a precisão é incrementada logo na sequência.

O contraexemplo gerado após a etapa de verificação é fundamental para a interação estabelecida na metodologia. Nele consta as variáveis de decisão de um novo candidato a mínimo global, este mesmo candidato é um valor menor que o candidato anterior estabelecido no início do algoritmo. Exatamente por isso que quando uma verificação é *failed*, indica que a propriedade na Equação (3.2) foi violada, pois foi encontrado um valor inferior ao descrito no início do algoritmo.

3.3 Algoritmos CEGIO

O algoritmo de otimização indutiva guiada por contraexemplos é capaz de encontrar o mínimo global em um problema de otimização, conforme a precisão e as variáveis de decisão. O tempo de busca pelo mínimo global, varia conforme o espaço de estado é descrito e o número de algarismos significativos da solução.

A abordagem geralmente possui elevado tempo de execução sendo maior que as outras técnicas tradicionais, porém a taxa de erro é menor que os outros métodos existentes, uma vez que se baseia em um procedimento de verificação completo do espaço de estados.

3.3.1 Algoritmo Generalizado CEGIO-G

O primeiro algoritmo estabelecido desta série de algoritmos, o `CEGIO-G` do inglês "*Generalized* `CEGIO` *algorithm*", é uma versão melhorada do algoritmo apresentado por Araújo [61], que apresentava uma solução de ponto fixo com precisão ajustável.

O algoritmo `CEGIO-G` necessita de três entradas: a função a ser otimizada ou função custo $f(x)$; definição do conjunto de restrições Ω; e a precisão que deseja-se obter para a solução, ε. O algoritmo após o processo de otimização retorna duas saídas: o vetor com as variáveis de decisão $\mathbf{x}^*$ e o valor mínimo da função de custo $f(\mathbf{x}^*)$.

O `CEGIO-G` repete as etapas de especificação e verificação descritas no exemplo de otimização na subseção 3.2. Sua descrição é apresentada no Algoritmo 1, possuindo dois *loops* aninhados onde o *loop* externo (`while`) está relacionado à precisão desejada, enquanto que o *loop* interno (`do-while`) está relacionado ao processo de verificação.

Nesta etapa é importante explanar sobre a definição de precisão para a solução ótima que se deseja encontrar. A precisão de uma solução ótima define a variável de precisão no algoritmo, ε. O número de casas decimais η para a solução é obtido pela Equação (3.7). Quando valor de ε for unitário, tem-se soluções inteiras, ou seja, $\eta = 0$, quando $\varepsilon = 10$ a solução terá uma casa decimal e $\eta = 1$, quando $\varepsilon = 100$ a solução terá duas casas decimais e $\eta = 2$,

$$\eta = log\varepsilon \tag{3.7}$$

Inicialmente o algoritmo CEGIO-G tem suas variáveis inicializadas e declaradas (linhas 1-3 do Algoritmo 1), o primeiro valor que é atribuído a função custo candidata é

definido de forma aleatória $f_c(\mathbf{X}^{(0)})$, a variável p será gradativamente incrementada em uma casa decimal para aprofundar a precisão durante o processo de otimização, porém inicializado com valor 1, enquanto que as variáveis auxiliares $\mathbf{x}$ são declaradas como variáveis inteiras não determinísticas, caso fossem declaradas como variáveis ponto flutuante não determinísticas, a pesquisa no espaço de estados seria muito elevada.

Algorithm 1: Algoritmo `Generalized` (CEGIO-G)

input : Uma função custo $f(\mathbf{X})$, o conjunto de restrições Ω, precisão desejada ε
output: O vetor com as variáveis de decisão ótima $\mathbf{X}^*$, e o valor ótimo da função $f(\mathbf{X}^*)$

```
1  Inicializa f_c(X^(0)) aleatoriamente e i = 1
2  Inicializa variável de precisão p = 1
3  Declara as variáveis auxiliares x como variáveis inteiras não determinísticas
4  while p ≤ ε do
5      Define limites para x com a diretiva ASSUME, de tal modo que x ∈ Ω^n
6      Descreve um modelo para função custo f(X)
7      do
8          Restringe f(X^(i)) < f_c(X^(i−1)) com a diretiva ASSUME
9          Verifica a satisfabilidade de l_otimo com a diretiva ASSERT
10         Atualiza X* = X^(i) e f_c(X*) = f(X^(i)) baseado no contraexemplo
11         Faz i = i + 1
12     while ¬l_otimo é satisfatível
13     Atualiza a variável de precisão p
14 end
15 X* = X^(i−1) e f(X*) = f_c(X^(i−1))
16 return X* e f(X*)
```

O algoritmo passa para o primeiro *loop*, onde é verificado a precisão atual p com a precisão requerida ε. Internamente ao `while` são usadas funções intrínsecas dos verificadores, `ASSUME`, para especificar o domínio de pesquisa Ω^{η} (linha 5), que é definido pelos limites inferior e superior das variáveis auxiliares $\mathbf{x}$. A variável η é usada para manipular as variáveis auxiliares $\mathbf{x}$ e assim obter o valor das variáveis de decisão X, onde $X = x/10^{\eta}$, dessa forma define-se o número de casas decimais para as variáveis de decisão X. Após a definição da restrição do problema, o modelo para a função custo $f(X)$ é definido (na linha 6), considerando as casas decimais estabelecidas para as variáveis de decisão. A variável p é atualizada ao final de cada iteração do *loop* externo, de forma que aumente o domínio das variáveis de decisão em uma casa decimal na próxima iteração do *loop*.

Quanto ao *loop* interno (`do-while`), em cada iteração a satisfatibilidade de l_{otimo} conforme apresentada na Equação (3.2) é verificada e usada para definir a permanência no *loop*. Antes disso, no entanto, uma nova restrição ($f(\mathbf{X}^{(i)}) < f_c(\mathbf{X}^{(i-1)})$) é adicionada (linha 8)

para limitar o espaço de estados levando em conta que não há necessidade de verificar valores maiores do que o valor mínimo candidato já encontrado, uma vez que o algoritmo busca o valor mínimo da função custo. Com isso o espaço de busca é reconfigurado para a precisão i-ésima e emprega os resultados anteriores do processo de otimização, $f(\mathbf{X}^{(i-1)})$.

A etapa de verificação é realizada (linha 9), onde a função candidata $f(\mathbf{X})$ é analisada por meio da verificação de satisfatibilidade de l_{otimo} dada pela Equação ((3.2)). Se houver um $f(\mathbf{X}) < f_c$ que viole a diretiva `ASSERT`, então o contraexemplo obtido a partir desta violação é capaz de apontar para um valor da função menor que o último armazenado. O vetor de variáveis de decisão, o valor mínimo da função custo e a função candidata são atualizadas (linha 10) após o incremento de i, em seguida o algoritmo retorna para remodelar novamente o espaço de estados (linha 8).

Se a diretiva `ASSERT` não for violada implica dizer que o último candidato f_c é o valor mínimo considerando a precião atual p, sendo assim p é incrementado em uma casa decimal para que η no *loop* externo `while`, busque por soluções acrescidas de uma casa decimal. O algoritmo conclui o *loop* externo atingindo o limite de precisão ε e retorna o vetor ótimo de variáveis de decisão com η casas decimais e o valor ótimo da função de custo.

3.3.2 Algoritmo Simplificado CEGIO-S

O algoritmo `CEGIO-S` do inglês "*Simplified* CEGIO *algorithm*" é apresentado no algoritmo 2. Este é adequado para funções semi-definidas positivas tal como $f(\mathbf{x}) \geq 0$. Suas modificações em relação ao `CEGIO-G` permitem lidar com classes particulares de funções, o que torna mais rápido e simples, pois não realiza buscas no espaço de estado negativo, mas o ideal é o usuário possuir algum conhecimento prévio sobre a função, indicando que a função seja semi-definida positiva para o uso do algoritmo.

O `CEGIO-S` necessita de quatro parâmetros de entradas: a função custo que se deseja otimizar $f(X)$; o conjunto de restrições Ω; a precisão que deseja-se obter para a solução ε e a taxa de aprendizagem α. Após o processo de otimização são retornados: o vetor com as variáveis de decisão $\mathbf{X}^*$ bem como o valor mínimo da função custo $f(\mathbf{X}^*)$.

As principais diferenças do CEGIO-S para o CEGIO-G, são uma nova condição estabelecida (linha 9) e possui um *loop* a mais (linhas 12-15) usado para gerar múltiplas *VCs* por meio da diretiva `ASSERT`.

Semelhante ao CEGIO-G, o CEGIO-S tem suas variáveis inicializadas e declaradas (linhas 1-4 do Algoritmo 2), a novidade é a variável f_m que é iniciada com 0, isso porque, será usada para limitar o espaço de busca até o valor nulo, evitando atuar no domínio negativo da função.

Algorithm 2: Algoritmo `Simplified` (CEGIO-S)

input : Função custo $f(\mathbf{X})$, o conjunto de restrições Ω, precisão desejada ε, e a taxa de aprendizagem α
output: O vetor com as variáveis de decisão ótima $\mathbf{X}^*$, e o valor ótimo da função $f(\mathbf{X}^*)$

```
1  Inicializa f_m = 0
2  Inicializa f_c(x^(0)) aleatoriamente e i = 1
3  Inicializa variável de precisão p = 1
4  Declara as variáveis auxiliares x como variáveis inteiras não determinísticas
5  while p ≤ ε do
6      Define limites para x com a diretiva ASSUME, de tal modo que x ∈ Ω^n
7      Descreve um modelo para função custo f(X)
8      Declara γ = (f_c(X^(i−1)) − f_m)/α
9      if (f_c(X^(i−1)) − f_m > 10^−5) then
10         do
11             Restringe f(X^(i)) < f_c(X^(i−1)) com a diretiva ASSUME
12             while (f_m ≤ f_c(X^(i−1))) do
13                 Verifica a satisfabilidade de l_otimo para cada f_m, com a diretiva ASSERT
14                 Faz f_m = f_m + γ
15             end
16             Atualiza X* = X^(i) e f_c(X*) = f(X^(i)) baseado no contraexemplo
17             Faz i = i + 1
18         while ¬l_otimo é satisfatível
19     end
20     else
21         break
22     end
23     Atualiza a variável de precisão p
24 end
25 X* = X^(i−1) e f(X*) = f_c(X^(i−1))
26 return X* e f(X*)
```

Dentro do *loop* externo (linha 5-24), as variáveis auxiliares x, variáveis de decisão X, espaço de restrição Ω^n e a modelagem para função custo $f(X)$ são definidos assim, como no primeiro algoritmo. O espaço de busca que falta perrcorrer compreende-se entre o último valor candidato a mínimo da função, encontrado $(f_c(X^{(i-1)}))$ e o limite mínimo de busca (f_m). Este espaço é segmentado em α partes e é obtida a faixa γ.

Uma condição $(f_c(X^{(i-1)}) - f_m > 10^{-5})$ (linha 9) é usada para verificar se o último

valor encontrado é bem próximo do mínimo valor de busca estabelecido, que não se faz mais necessário continuar a busca considerando a precisão atual, isso sem precisar gerar novas verificações se essa condição não se mantiver, uma vez que a solução já está no limite mínimo. O algoritmo desloca-se para a atualização da precisão (linha 23), incrementando em uma casa decimal p. Lembrando que independente do resultado da condição, ao término de cada execução do *loop* externo `while` (linhas 5-25), a precisão é atualizada.

Assim como no primeiro algoritmo `CEGIO-G`, o *loop* `do-while` é usado para definir a permanência da checagem de l_{otimo}. Porém, aqui existe um terceiro *loop* (linhas 12-15), que é responsável por gerar várias *VCs* através da diretiva ASSERT, usando o intervalo entre f_m e $f_c(X^{(i-1)})$, permitindo gerar $\alpha + 1$ *VCs* através da faixa definida por γ (linha 8).

As mudanças permitiram que o algoritmo CEGIO-S convirja mais rápido que o CEGIO-G dentre as funções semidefinidas positivas, uma vez que a chance de uma falha de verificação é maior, devido ao maior número de propriedades. No entanto, um número maior de propriedades implica em mais VCs, que podem causar em alguns casos um efeito contrário do proposto, levando a muitos processos de verificação e esgotando a memória.

3.3.3 Algoritmo Rápido CEGIO-F

O algoritmo CEGIO-F apresentado na Figura (3) é uma evolução dos algoritmos apresentados anteriormente, destina-se a funções convexas. Assim como nos demais algoritmos, o algoritmo CEGIO-F evolui aumentando a precisão das variáveis de decisão, ou seja, na primeira execução do *loop* `while`, o mínimo global obtido inicialmente é inteiro, que aqui atribui-se pela notação $X^{*,0}$. O algoritmo também possui variáveis auxiliares x, que são usadas para delimitar o espaço de busca Ω^n, que por sua vez define as variáveis de decisão X, mas sua grande diferença se dá por estabelecer um novo domínio de busca a cada resultado de verificação *failed*, neste algoritmo é atualizado os limites do conjunto Ω^n (linha 13) antes da precisão p.

Em cada execução do *loop* `while`, a solução é ótima conforme a precisão p no momento da execução. Um novo domínio de pesquisa $\Omega^n \subset \Omega^\eta$ é obtido de uma etapa do CEGIO-F aplicando Ω^{n-1}, ou seja, as extremidades da função são aproximadas para o último candidato a mínimo global, desconsiderando os espaços mais externos ao candidato, mas respeitando a proporcionalidade em todas as dimensões. A abordagem consegue manter a localização do

mínimo global, pois as funções convexas possuem apenas um ponto de inflexão, onde a solução converge para este ponto de inflexão. A definição que realiza a redução do espaço Ω^n é: $\Omega^n = \Omega^\eta \cap [x^{*,n-1} - p, x^{*,n-1} + p]$, onde $x^{*,n-1}$ é a solução com $n-1$ casas decimais.

Algorithm 3: Algoritmo `Fast` (CEGIO-F)

input : Uma função custo $f(\mathbf{X})$, o conjunto de restrições Ω, precisão desejada ε
output: O vetor com as variáveis de decisão ótima $\mathbf{X}^*$, e o valor ótimo da função $f(\mathbf{X}^*)$

Inicializa $f_c(\boldsymbol{X}^{(0)})$ aleatoriamente e $i = 1$
Inicializa variável de precisão $p = 1$
Declara as variáveis auxiliares $\boldsymbol{x}$ como variáveis inteiras não determinísticas
while $p \leq \varepsilon$ **do**
    Define limites para $\boldsymbol{x}$ com a diretiva `ASSUME`, de tal modo que $\boldsymbol{x} \in \Omega^n$
    Descreve um modelo para função custo $f(\boldsymbol{X})$
    do
        Restringe $f(\boldsymbol{X}^{(i)}) < f_c(\boldsymbol{X}^{(i-1)})$ com a diretiva `ASSUME`
        Verifica a satisfabilidade de l_{otimo} com a diretiva `ASSERT`
        Atualiza $\boldsymbol{X}^* = \boldsymbol{X}^{(i)}$ e $f_c(\boldsymbol{X}^*) = f(\boldsymbol{X}^{(i)})$ baseado no contraexemplo
        Faz $i = i + 1$
    while $\neg l_{otimo}$ *é satisfatível*
    Atualiza o conjunto de restrições Ω^n
    Atualiza a variável de precisão p
end
$\boldsymbol{X}^* = \boldsymbol{X}^{(i-1)}$ *e* $f(\boldsymbol{X}^*) = f_c(\boldsymbol{X}^{(i-1)})$
return $\boldsymbol{X}^*$ *e* $f(\boldsymbol{X}^*)$

3.3.4 Prova de convergência do CEGIO

Esta subseção apresenta a prova de convergência para os algoritmos CEGIO [19]. São demonstrados a convergência dos algoritmos CEGIO-G e CEGIO-S, que é aplicado a funções não convexas, mas que também pode ser usados para otimizar funções convexas. Em seguida é demonstrado a convergência do algoritmo CEGIO-F que é específico para funções convexas.

Inicialmente para efetuar a prova matemática da convergência do CEGIO para mínimo global, se faz necessário formalizar um problema genérico de otimização. Tendo-se um conjunto $\Omega \subset \mathbb{R}^n$, determinando $\mathbf{X}^* \in \Omega$, onde $\mathbf{X}^*$ é o vetor de variáveis de decisão que garante o menor valor para a função f a ser otimizada, tal que, $f(\mathbf{X}^*) \in \Phi$ é o menor valor da função f, onde $\Phi \subset \mathbb{R}$ é o conjunto de imagens de f, ou seja, $\Phi = Im(f)$. Os algoritmos CEGIO tem a capacidade de solucionar problemas de otimiação com η casas decimais, em que o minimizador $\mathbf{X}^*$ é um elemento do domínio racional $\Omega^\eta \subset \Omega$ tal que $\Omega^\eta = \Omega \cap \Theta$, onde

$\Theta = \{\mathbf{X} \in \mathbb{Q} | \mathbf{X} = n \times 10^{-\eta}, \forall n \in \mathbb{Z}\}$, significa que Ω^{η} é composto por racionais com η casas decimais em Ω, onde $\mathbf{X}^{*,\eta}$ é o mínimo de uma função f em Ω^{η}.

Prova de convergência do CEGIO-G e CEGIO-S

Considerando a otimiação restrita a um conjunto finito, o teorema 1 garante solucionar problemas de otimização encontrando o mínimo global de uma função, esgotando todas as possibilidades do conjunto Φ. Seguem abaixo o Lema 1, o Teorema 1 e a demonstração.

Lema 1 *Seja Φ um conjunto finito composto por todos valores de $f(\boldsymbol{X})$ em que $f(\boldsymbol{X}) < f_c^{(i)}$, onde $f_c^{(i)} \in \Phi$ é qualquer candidato mínimo e $\boldsymbol{X} \in \Omega$. O literal $\neg l_{otimo}$ Equação (3.2) não é satisfatível, se e somente se, $f_c^{(i)}$ possuir o menor valor de Φ; caso fosse o contrário, $\neg l_{otimo}$ é satisfatível ,se e somente se, existir algum $\boldsymbol{X}^{(i)} \in \Omega$ tal que $f(\boldsymbol{X}^{(i)}) < f_c^{(i)}$.*

Teorema 1 *Seja Φ_i o i-ésimo conjunto de imagem do problema de otimização restrito por $\Phi_i = \{f(\boldsymbol{X}) < f_c^{(i)}\}$, onde $f_c^{(i)} = f(\boldsymbol{X}^{*,(i-1)}) - \delta, \forall i > 0$, e $\Phi_0 = \Phi$. Existe um $i^* > 0$, tal que $\Phi_{i^*} = \emptyset$, e $f(X^*) = f_c^{(i^*)}$.*

Inicialmente é atribuído um valor de forma aleatória ao primeiro candidato a mínimo global $f_c^{(0)}$. Considerando o Lema 1, se $\neg l_{otimo}$ é satisfatível, qualquer $f(\mathbf{X}^{*,(0)})$ (do contraexemplo) é adotado como a próxima solução candidata (isto é, $f_c^{(1)} = f(\mathbf{X}^{*,(0)})$), e todo elemento de Φ_1 é menor que $f_c^{(1)}$. De forma similar, nas próximas iterações, enquanto $\neg l_{otimo}$ for satisfatível, $f_c^{(i)} = f(\mathbf{X}^{*,(i-1)})$, e todo elemento de Φ_i é menor que $f_c^{(i)}$, consequentemente o número de elementos de Φ_{i-1} é sempre menor que Φ_i. Como Φ_0 é finito, na i^*ésima iteração, Φ_{i^*} será vazio e o literal $\neg l_{otimo}$ não será satisfatível, o que leva para o Lema (1), $f(\mathbf{X}^*) = f_c^{(i^*)}$.

Prova de convergência do CEGIO-F

Um problema de otimização convexa é semelhante ao problema genérico de otimização descrito no início da seção 3.3.4, mas $f(x)$ necessita satisfazer a Equação 3.8.

$$f(\alpha x_1 + \beta x_2) \leq \alpha f(x_1) + \beta f(x_2) \tag{3.8}$$

para todo $x_i \in \mathbb{R}^n$, com $i = 1, 2$ e todo $\alpha, \beta \in \mathbb{R}$ com $\alpha + \beta = 1$, $\alpha \geq 0$, $\beta \geq 0$.

O Teorema 2 é usado para garantir a convergência do conjuto finito Ω^n para o mínimo global durante a otimização com o CEGIO específico para funções convexas, CEGIO-F.

Teorema 2 *Um mínimo local de uma função convexa f em um subconjunto convexo é sempre um mínimo global de f [62].*

O algoritmo CEGIO-G com sua própria estrutura já é capaz de convergir para o mínimo global, isso porque $\Omega^1 \subset \Omega^2 ... \subset \Omega^{n-1} \subset \Omega^n$, porém, o algoritmo CEGIO-F traslada os limites do conjunto Ω^n a partir da $(n-1)$-ésima solução candidata, permitindo otimizações mais rápidas, uma vez que o espaço de busca é reduzido em cada iteração. O CEGIO-F define um novo espaço de busca Ω^n em cada iteração, de forma que n varia entre 0 e a quantidade de casas decimais η. A convergência do algoritmo CEGIO-F é assegurada, se e somente se, Ω^{n-1} esteja contido em Ω^n.

Lema 2 *Seja $f : \Omega^n \to \mathbb{R}$ uma função convexa, como Ω^n é uma conjunto finito, o Teorema 1 garante que o mínimo, $X^{*,n}$ em Ω^n é um mínimo local na precisão p, onde $n = \log p$. Além disso, como f é uma função convexa, qualquer elemento X estando fora do intervalo $[X^{*,n} - p, X^{*,n} + p]$ tem sua imagem $f(X) > f(X^{*,n})$ garantida pela Equação ((3.8)).*

O CEGIO-F tem seu tempo de otimização inferior ao CEGIO-G, uma vez que o espaço de busca é reduzido a cada iteração. O Lema 2 garante que a solução seja um mínimo local de f, e o Teorema 2 garante que este mínimo local seja o mínimo global.

3.3.5 Desviando dos mínimos Locais

Como mencionado anteriormente, uma característica importante deste método proposto pelo CEGIO é sempre encontrar o mínimo global (Teorema 1). Muitos algoritmos de otimização podem ficar presos por mínimos locais e podem resolver incorretamente problemas de otimização. No entanto, a técnica assegura evitar esses mínimos locais, através da verificação da satisfatibilidade, que é realizada por sucessivas consultas SMT.

As Figuras (3.4) e (3.5) mostram a propriedade supracitada deste algoritmo, comparando seu desempenho com o algoritmo genético. Nessas figuras, a função de Ursem03 é adaptada para um único problema de variável sobre x_1, isto é, x_2 é considerado fixo e igual a 0.0, e a respectiva função é reduzida a um plano que cruza o ótimo global em $x_1 = -3$.

Resultados parciais após cada iteração são ilustrados pelas várias marcas nesses gráficos. O presente método não apresenta trajetória contínua desde o ponto inicial até o ponto ótimo; no entanto, sempre alcança a solução correta. A Figura (3.4) mostra que ambas as técnicas (GA e

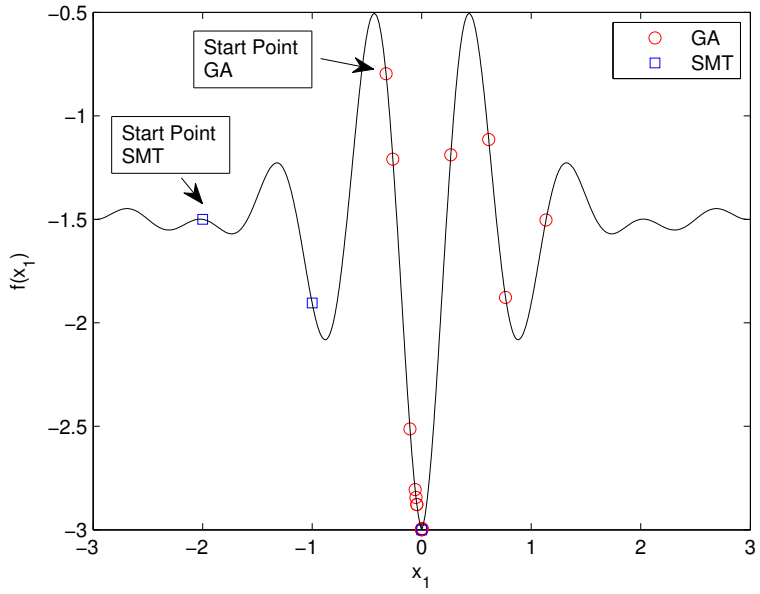

Figura 3.4: Trajetória de otimização de GA e SMT para um plano de Ursem03 em $x_2 = 0$. Ambos os métodos obtêm a resposta correta.

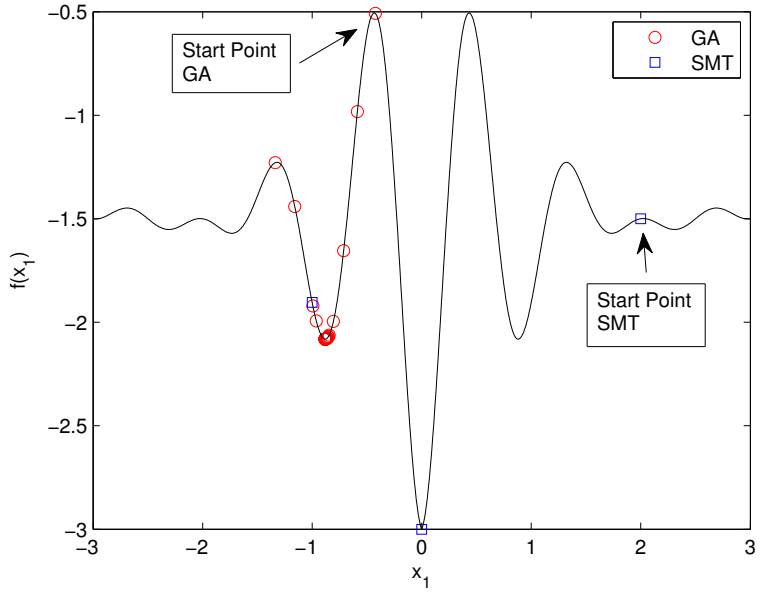

Figura 3.5: Trajetória de otimização de GA e SMT para o plano de Ursem03 em $x_2 = 0$. O GA fica preso em um mínimo local, enquanto que o SMT obtém a resposta correta.

SMT) alcançam o ótimo global. No entanto, a Figura (3.5) mostra que o GA pode ficar preso pelo mínimo local para um ponto inicial diferente. Em contraste, o método CEGIO proposto pode ser inicializado mais longe do mínimo global e, como resultado, pode encontrar o mínimo global após algumas iterações, conforme mostrado nas Figuras (3.4) e (3.5).

3.4 Resumo

Este captítulo apresentou os algoritmos de otimização desenvolvidos CEGIO. Foram desenvolvidos três algoritmos seguindo a abordagem de análise de contraexemplos para a otimização de funções, um algoritmo para funções genéricas (CEGIO-G) e outros dois algoritmos específicos, para funções convexas (CEGIO-F) e funções positivas (CEGIO-S). Os algoritmos desenvolvidos somados a abordagem de otimização guiada por contraexemplo são usados para a criação da ferramenta de otimização OptCE, que é apresentado no próximo capítulo 4.

Capítulo 4

OptCE: Um Solucionador de Otimização Indutivo Guiado por Contraexemplos SAT e SMT

Neste capítulo é apresentada a ferramenta de otimização OptCE. Esta ferramenta faz uso da metodologia de otimização guiada por contraexemplo e implementa os algoritmos CEGIO. O OptCE fornece uma interface console Linux, sendo prática quanto a configuração para realização da otimização de funções matemáticas. Durante a avaliação experimental são usadas funções convexas e não convexas para investigar sua capacidade de otimização, onde os resultados são comparados com outras técnicas de otimização.

4.1 OptCE: Solucionador Indutivo Guiado a Contraexmplo

O OptCE pode ser considerado como um *front-end* para checagem de modelos que processam programas *C* através dos algoritmos CEGIO, onde as variáveis de decisão, que são responsáveis por gerar o valor mínimo de uma função, são encontradas por meio de verificação de modelos. Essa ferramenta pode ser chamada a partir de um *shell* via linha de comando, sendo capaz de otimizar funções convexas e não convexas, onde o usuário precisa descrever as restrições e o modelo da função, através de algumas linhas de código em um arquivo com a estrutura que será mostrada a seguir. Em resumo, o OptCE é baseado na abordagem dos algoritmos CEGIO, que permite encontrar os mínimos globais, enquanto outras técnicas

geralmente permanecem fixas em mínimos locais.

4.1.1 Arquitetura do OptCE

Conforme mostrado na Figura (4.1), os usuários precisam fornecer um arquivo de entrada *.func* (vide a seção (4.1.2)) contendo os limites de restrição do espaço de estado e a descrição da função a ser otimizada, esta é a fase de modelagem. O ideal é o usuário possuir algum conhecimento sobre o problema em questão, para melhor definição do espaço de estados.

A primeira etapa que a ferramenta executa é a especificação, que recebe um arquivo de entrada e as configurações desejadas para otimização, como verificador, solucionador, tipo de algoritmo e a precisão da solução. Na Figura (4.1), η representa o número de casas decimais desejadas para a solução, que é indicada pelo usuário. Com base nas entradas fornecidas, o OptCE gera um arquivo de especificação em ANSI-C (vide Figura 3.3), denominado como `min_<function>.c`.

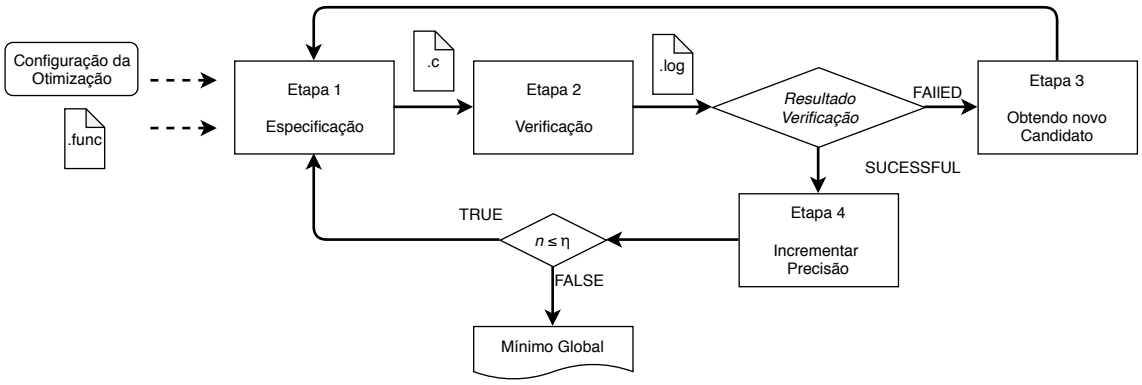

Figura 4.1: Uma visão geral da arquitetura proposta do OptCE.

Durante a primeira execução da etapa 1, a variável *n* é usada para armazenar a contagem de casas decimais ao longo do processo de otimização. Em sua inicialização é atribuído 0, o que indica uma otimização somente com soluções de precisão inteira. Além disso, um candidato mínimo arbitrário é considerado, cujo o real valor de inicialização do algoritmo, pode ser fornecido pelo usuário com a *flag* `--start-value`; caso não seja fornecido, a ferramenta gera este valor aleatoriamente.

Durante a etapa 2, ocorre a verificação, ou seja, o arquivo ANSI-C com a especificação da função é submetido por um verificador, cuja saída principal é um arquivo `.log` com o respectivo resultado da verificação. Se for obtido *"verification failed"*, significa que o mecanismo de verificação detectou uma violação de propriedade através das afirmações inseridas e consequentemente, gerou um contraexemplo. Conforme a abordagem do CEGIO, uma violação de propriedade indica que o candidato mínimo encontrado não é o mínimo global para a quantidade de casas decimais η, então o fluxo da ferramenta passa para a etapa 3.

Na Etapa 3, um arquivo `.log` com o respectivo contraexemplo é usado para obter novas variáveis de decisão, referentes a um novo candidato a mínimo global menor que o anterior, ou seja, o candidato mínimo global da última iteração será o valor de inicialização. Em seguida, obtém-se o novo candidato a valor mínimo (isto é, extraído e calculado a partir do contraexemplo) e usado para executar a Etapa 1 novamente, iniciando uma nova iteração e gerando um novo arquivo de especificação. Esse procedimento é executado iterativamente até a verificação (Etapa 2) retornar um arquivo `.log` com *"verification successfully"*, o que significa que não há variáveis de decisão capazes de encontrar um valor mínimo menor que o atual, considerando (n) casas decimais estabelecida até o momento. Quando o resultado da verificação é *"verification successfully"*, o OptCE prossegue para o Passo 4.

Na Etapa 4, n é incrementado em 1 unidade, seguida de uma checagem que avalia se a precisão é menor ou igual à precisão que define η desejada (indicada pelo usuário). Se n for maior que η (a condição $n \leq \eta$ é falsa), e o OptCE encontrou a solução (mínimo global), considerando a precisão desejada. Caso contrário, o fluxo geral do OptCE (Etapas $1-3$) é repetido com a precisão atualizada n, ou seja, o algoritmo retorna a etapa 1 e gera um novo arquivo de especificação para ser verificado.

4.1.2 Arquivo de Entrada

O OptCE suporta 2 tipos de arquivos de entrada, o primeiro é a forma matricial, onde os limites das variáveis são descritos em forma de matriz, o segundo formato permite o usuário estabelecer mais restrições, não apenas o limite superior e inferior das variáveis de decisão.

Arquivo de Entrada tipo Matricial

O arquivo de entrada matricial consiste em duas partes: a especificação da função e as restrições associadas, que são separadas por um caractere "# " isolado em uma linha. Na parte superior do arquivo de entrada, a função é descrita com atribuições de variáveis ANSI-C que termina com ";" e usa a variável `fobj` que representa a função objetivo. A Figura (4.2) resume a linguagem de entrada do OptCE.

$$
\begin{aligned}
Fml &::= Var \mid true \mid false \mid Fml \wedge Fml \mid \ldots \mid Exp = Exp \mid \ldots \\
Exp &::= Var \mid Const \mid Var[Exp] \mid Var[Exp][Exp] \mid Exp + Exp \mid \ldots \\
Cmd &::= Var = Exp \mid Var = * \mid Fml \mid sin2(Var) \mid cos2(Var) \\
&\quad \mid floor2(Var) \mid sqrt2(Var) \mid abs2(Var) \\
Prog &::= Cmd; \ldots; \#Cmd;
\end{aligned}
$$

Figura 4.2: Linguagem da entrada do programa para OptCE.

A Equação (4.1) apresenta o formato adotado para matrizes de restrição, onde o número de linhas indica a quantidade de variáveis de decisão e as colunas 1 e 2 representam os limites inferior e superior, respectivamente.

$$
\begin{bmatrix}
x_{11} & x_{12} \\
x_{21} & x_{22} \\
\multicolumn{2}{c}{\ldots} \\
x_{n1} & x_{n2}
\end{bmatrix}
\tag{4.1}
$$

As restrições de um problema de otimização como, por exemplo, a do *benchmark adjiman* [63], podem ser representadas por $A = [\ -1\ 2\ ;-1\ 1]$, e um arquivo de entrada para o mesmo *benchmark* é ilustrado na Figura (4.3).

```
1 fobj = cos2(x1)*sin2(x2) - (x1/(x2*x2+1));
2 #
3 A = [-1  2; -1  1];
```

Figura 4.3: Arquivo de entrada para a função *adjiman*.

Arquivo de Entrada tipo Extensivo

O arquivo de entrada tipo Extensivo possui duas partes assim como no matricial. Tem-se a especificação da função na parte superior, de forma que termina com ";" e usa a variável

`fobj` que representa a função objetivo. A descrição da função é separada da parte inferior por um caractere "#" isolado em uma linha, onde são descristas as restrições. A Figura (4.4) apresenta um exemplo de entrada do tipo extensivo. Percebe-se que as restrições não se limitam por definir o intervalo que as variáveis de decisão poderão assumir, mas é possível também realizar associações entre as variáveis de decisão, como apresentado em algumas restrições do problema 4.1.6 na Equação (4.2).

$$-1 \leq \frac{X1-X2}{0,1} \leq 1; -1 \leq \frac{X1-X3}{0,1} \leq 1; -1 \leq \frac{X2-X3}{0,1} \leq 1 \tag{4.2}$$

O OptCE busca as restrições descritas no arquivo de entrada 4.4 e as insere no arquivo de especificação usando a diretiva `ASSUME`, fazendo com que o verificador aceite somente as variáveis que respeitem essas restrições.

```
1  fobj = 1000*(((X1-X2)/0.1) + ((X1-X3)/0.1)) + 1200*(((X2-X3)/0.1) + ((X2
       -X1)/0.1)) + 2000*(((X3-X1)/0.1) + ((X3-X2)/0.1) + 1.8);
2  #
3  (X1 >= -PI) && (X1 <= PI)
4  (X2 >= -PI) && (X2 <= PI)
5  (X3 >= -PI) && (X3 <= PI)
6  (((X1-X2)/0.1)>= -1) && (((X1-X2)/0.1)<= 1)
7  (((X1-X3)/0.1)>= -1) && (((X1-X3)/0.1)<= 1)
8  (((X2-X3)/0.1)>= -1) && (((X2-X3)/0.1)<= 1)
9  ((((X1-X2)/0.1)+((X1-X3)/0.1))>=0) && ((((X1-X2)/0.1)+((X1-X3)/0.1))<=4)
10 ((((X2-X3)/0.1)+((X2-X1)/0.1))>=0) && ((((X2-X3)/0.1)+((X2-X1)/0.1))<=4)
11 ((((X3-X1)/0.1)+((X3-X2)/0.1)+1.8)>=0) && ((((X3-X1)/0.1)+((X3-X2)/0.1)
       +1.8)<=4)
```

Figura 4.4: Arquivo de entrada para a função *adjiman*.

4.1.3 Recursos do OptCE

O OptCE permite definir diferentes configurações em relação ao processo de otimização, isto é, algoritmo de otimização e mecanismo de verificação, que é usado para reduzir os tempos de otimização. Assim, o usuário deve adicionar *flags* adequadas durante uma chamada via linha de comando. As seguintes configurações são suportadas:

- **Configuração do BMC:** escolhe entre verificadores de modelo CBMC (`--cbmc`) e ESBMC (`--esbmc`);
- **Configuração do Solucionador:** escolhe entre os solucionadores (`--boolector`), Z3 (`--z3`), MathSAT (`--mathsat`), e MiniSAT (`--minisat`);

- **Configuração do Algoritmo:** escolhe entre os algoritmos propostos, onde a *flag* `--generalized` implementa o algoritmo CEGIO-G (*vide seção 3.3.1*), usada quando não existe conhecimento prévio sobre a função, a flag `--positive` implementa o algoritmo CEGIO-S (*vide seção 3.3.2*), usada quando a função é semi-definida positiva, e a *flag* `--convex` implementa o algoritmo CEGIO-F (*vide seção 3.3.5*), usada quando a função é convexa.

- **Inicialização:** atribui um valor mínimo inicial (`--start-value=`*value*), que é aleatório por padrão;

- **Inserir biblioteca:** os usuários podem incluir sua própria biblioteca contendo implementações de operadores e funções usadas na descrição da função objetivo (`--library=`*name-library*);

- **Timeout:** configura o limite de tempo (`--timeout=`*value*).

- **Precisão:** define a precisão desejada, ou seja, o número de casas decimais de uma solução (`--precision =`*value*).

- **Arquivo de Entrada Extensivo:** recebe o arquivo de entrada tipo extensivo (`--extensive`).

- **Otimização Máxima:** configura para otimização global máxima (`--max`).

4.1.4 Importantes mudanças no OptCE

Durante a evolução da pesquisa duas importantes mudanças foram agregadas ao OptCE com o intuito de reduzir o tempo de otimização. Foram feitas alterações no arquivo de especificação gerado pelo OptCE para facilitar a execução dos verificadores e uma lógica de interrupção foi inserida ao algoritmo, afim de evitar verificações desnecessárias.

Mudanças no arquivo de especificação

Alguns testes empíricos realizados com diversos arquivos de especificação que tiveram distintas modificações, permitiram com que o OptCE obtive-se a mesma solução para os mesmos *benchmarks*, porém com tempos de execução diferentes o suficiente para concluir que a forma com que o arquivo de especificação é descrito pode influenciar no tempo de

otimização. Verificou-se então que especificações que usavam *loops* `for` para definir variáveis ou limites da função, possuíam tempos maiores em relação a outros arquivos onde as variáveis e limites eram definidos diretamente, notou-se a importância da definição do arquivo de especificação para a técnica de otimização.

Para os verificadores em geral é oneroso lidar com estruturas de *loops* como `for` e `while`, sendo muitas vezes necessário transformar internamente a estrutura do *loop* em outra estrutura mais simples como `if-else`, para realizar a verificação.

A Figura (4.5) apresenta uma parte do arquivo de especificação gerado durante a otimização da função *cosine* [60] usando o OptCE 1.0, onde são usadas estruturas tipo `for` para atribuir o não determinismo das variáveis auxiliares e de decisão, assim como também o uso do `for` para definir os limites de busca. Neste caso, o verificador necessita desenrolar o *loop* afim de construir a representação simbólica internamente, demandando um tempo e complexidade maior do que se essas informações estivessem descritas diretamente sem o uso de estruturas de repetição.

```
1  ...
2  int x[3];
3  float X[2];
4
5  for (i = 0; i<2; i++){
6      x[i] = nondet_int();
7      X[i] = nondet_float();
8  }
9
10 int lim[4] = {-4*p, 4*p, -4*p, 4*p};
11
12 for (i = 0; i < nv; i++) {
13     __ESBMC_assume( (x[i]>=lim[2*i]) && (x[i]<=lim[2*i+1]) );
14     __ESBMC_assume( X[i] == (float) x[i]/p  );
15 }
16
17 fobj= -0.1*(cos2(5*PI*X[0]) + cos2(5*PI*X[1])) + (X[0]*X[0]+X[1]*X[1]);
18 ...
```

Figura 4.5: Arquivo de especificação para a função *cosine* usando *loops*.

O mesmo código da Figura (4.5) pode ser reescrito como mostrado na Figura (4.6), onde não se faz o uso de vetores e estruturas de *loops*. As variáveis do arquivo de especificação são declaradas e inicializadas conforme os dados extraídos do arquivo de entrada. O OptCE inspeciona a descrição da função para buscar a quantidade de variáveis da função e a usa para determinar a quantidade de variáveis auxiliares, de decisão e os limites definidos para cada

variável.

```
1  ...
2  int x1,x2;
3  float X1,X2;
4
5  x1 = nondet_int();
6  x2 = nondet_int();
7  X1 = (float) nondet_float();
8  X2 = (float) nondet_float();
9
10 __ESBMC_assume( (x1>=-4*p)&& (x1<=4*p) );
11 __ESBMC_assume( (x2>=-4*p)&& (x2<=4*p) );
12 __ESBMC_assume( X1 == (float) x1/p );
13 __ESBMC_assume( X2 == (float) x2/p );
14
15 fobj= -0.1*(cos2(5*PI*X1) + cos2(5*PI*X2)) + (X1*X1+X2*X2);
16 ...
```

Figura 4.6: Arquivo de especificação para a função *cosine* sem o uso de *loop*.

Lógica de Interrupção

Durante a avaliação experimental da ferramenta constatou-se que muitos *benchmarks* alcançavam o mínimo global em precisões menores que a precisão estabelecida ou definida por padrão. Então o valor alcançado em uma determinada precisão, era o mesmo valor obtido considerando as precisões seguintes maiores, de forma que o algoritmo incrementava as precisões e continuava a realizar verificações, até atingir a precisão definida no início da otimização.

Como exemplo, tem-se a função *Cosine* [60] que possui o mínimo global em $f(0,0) = -0,2$. Na precisão inteira o mínimo global já pode ser obtido em $-0,2$, na precisão seguinte com uma casa decimal o mesmo valor foi obtido, $-0,2$. Como o experimento foi definido com precisão para três casas decimais, o mesmo valor $-0,2$, foi encontrado mais duas vezes até que o algoritmo pudesse ser encerrado.

Em outros *benchmarks* como a função *Styblinski Tang* [64] que possui o mínimo global em $f(-2,903,-2,903) = -78.332$ considerando o domínio para duas variáveis, o valor obtido em cada casa decimal variou. Na precisão inteira o valor obtido foi de -78, para uma casa decimal foi de -78.3319 e continuou abaixando até obter -78.3322 com as variáveis de decisão $x_1 = -2,903$ e $x_2 = -2,903$.

O mínimo global da função *Styblinski Tang* variou a cada incremento de casa decimal, alcançando um valor menor após o outro, mas a função *Cosine* manteve-se com mesmo valor de mínimo global, mesmo após incrementar as casa decimal das variáveis de decisão, isso porque o valor obtido já era o menor valor para a função, não sendo necessário continuar o algoritmo.

Dessa forma adotou-se a abordagem que verifica se os valores de mínimos obtidos em cada precisão se repetem ao longo do processo. O algoritmo é interrompido nessa condição e o valor encontrado em mais de uma precisão é determinado como o valor mínimo global. Importante resaltar que a abordagem proposta pode obter falhas em alguns casos específicos, como por exemplo em um valor de mínimo global $0,003$. Neste caso a solução inteira seria 0 e a solução com uma casa decimal $0,0$, o que levaria interromper o algoritmo sem buscar pela solução 0.003.

Na seção 5.2 é apresentada a avaliação experimental do OptCE, considerando a nova padronização do arquivo de especificação sem o uso de *loops* e com a lógica de interrupção. Foram utilizados os mesmos *benchmarks* do OptCE v1.0 [63], sendo executados sob as mesmas condições experimentais.

4.1.5 Otimização do Máximo Global

Dentro da otimização nem sempre o usuário busca minimizar um recurso, ou deseja o mínimo de uma função. Existem casos em que o usuário necessita maximizar os recursos procurando pelo máximo global de uma função.

O OptCE permite localizar o máximo global de uma função considerando os mesmos algoritmos apresentados na seção 3. O usuário necessita empregar a *flag* `--max` junto com o arquivo de entrada com a descrição e as restrições da função a ser maximizada.

Considerando que o $f(x^*)$ é o valor mínimo de uma função custo f, o mesmo ponto corresponde ao valor máximo de $-f$ [21–23]. O OptCE faz uso desse artifício matemático para também poder localizar o máximo valor de uma função podendo ser convexa ou não convexa.

Para exemplificar esta funcionalidade temos as funções *Chen's V* e *Chen's Bird*, que são problemas de otimização que possuem diversos máximos locais com um único máximo global, onde o OptCE foi capaz de otimizar localizando seus máximos globais.

A definição da função *Chen's Bird* é representada na Equação (4.3) onde seu máximo global é obtido em $f(0,5;0,5) = 2000,0039$ e seu gráfico é apresentado na Figura (4.7) [65].

$$f(x_1,x_2) = \frac{0,001}{0,001^2+(x_1^2+x_2^2-1)^2} + \frac{0,001}{0,001^2+(x_1^2+x_2^2-0,5)^2} + \frac{0,001}{0,001^2+(x_1-x_2)^2} \tag{4.3}$$

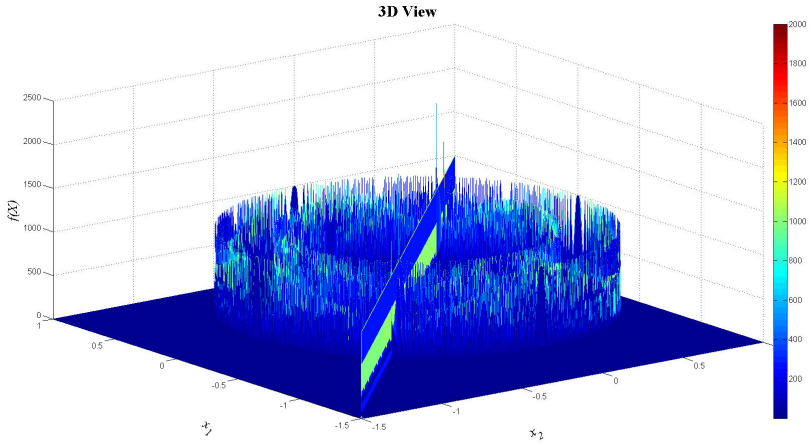

Figura 4.7: Função Chen's Bird.

A função *Chen's Bird* teve seu máximo global, $2000,0039$, localizado quando o OptCE atingiu precisão de uma casa decimal, isso porque a solução ótima para esta função é $x_i^* = (0,5;0,5)$, onde as variáveis de decisão são de uma casa decimal.

A definição da função *Chen's V* é representada na Equação (4.4) com seu máximo global em $f(0,388888888888889;0,722222222222222) = 2000$ e seu gráfico é apresentado na Figura (4.8) [66].

$$f(x_1,x_2) = \frac{0,001}{0,001^2+(x_1-0,4x_2-0,1)^2} + \frac{0,001}{0,001^2+(2x_1+x_2-1,5)^2} \tag{4.4}$$

A função *Chen's V* teve o valor 1961.54 como solução obtida para três casas decimais, sendo diferente da solução ótima que é 2000. Essa diferença se deu pela quantidade de casas decimais necessárias para determinar a solução correta, após testes considerando cinco casas decimais o resultado alcançado foi satisfatório, sendo obtido o valor de 2000 como solução.

O usuário não só pode otimizar o máximo global usando a *flag* `--max`, como também pode apenas inverter o sinal na descrição da função no arquivo de entrada, para quando o OptCE retornar a saída, inverter também o sinal do resultado. No caso da função *Chen's V*, para otimizar o máximo da função sem usar a *flag* `--max` deve-se descrever a função conforme a

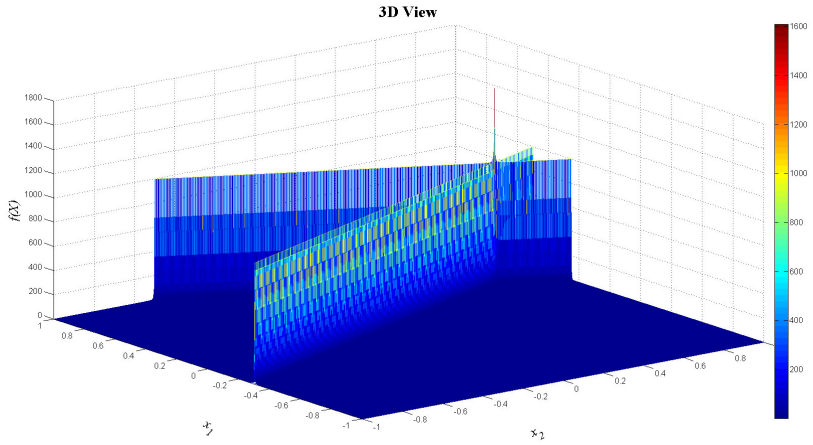

Figura 4.8: Função Chen's V.

Equação (4.5), invertendo o sinal da Equação (4.4), e após a otimização, inverter o sinal do resultado obtido.

$$f(x_1,x_2) = -\frac{0,001}{0,001^2+(x_1-0,4x_2-0,1)^2} - \frac{0,001}{0,001^2+(2x_1+x_2-1,5)^2} \tag{4.5}$$

4.1.6 Resolvendo problemas de otimização com OptCE

Usando a ferramenta OptCE, o usuário deve criar um arquivo de entrada de descrição para encontrar o mínimo global de uma função. A Figura (4.9) mostra todas chamadas possíveis do OptCE com arquivo de entrada e conjunto de propriedades. Então, emprega-se a função *adjiman* para ilustrar o uso do OptCE, considerando a entrada do arquivo mostrado na Figura (4.3).

Chamada	Configuração						
OptCE + Função	**BMC**	**Solucionador**	**Algoritmo**	**Inicialização**	**Biblioteca**	**Timeout**	**Precisão**
./optCE name.func	--esbmc --cbmc	--mathsat --boolector --z3 --minisat	--generalized --positive --convex	--start-value=?	--library=name	--timeout=?	--precision=?

Figura 4.9: Opções de configurações do OptCE.

Atualmente, o OptCE suporta dois verificadores: CBMC [17] e ESBMC [16]. A

otimização empregando CBMC como verificador de modelos (`--cbmc`) usa o MiniSAT como solucionador padrão, enquanto ESBMC (`--esbmc`) usa MathSAT. A avaliação, também tenta usar os solucionadores SMT disponíveis no CBMC, mas devido a problemas no *backend* SMT, não foi possível verificar todos os *benchmarks* da *suite* usada neste estudo. Em relação ao ESBMC, o usuário pode escolher entre solucionadores Mathsat (`--mathsat`), Z3 (`--z3`) ou Boolector (`--boolector`), no entanto, não foi avaliado outros solucionadores (*CVC4 e Yices*).

Os tempos de verificação variam de acordo com o verificador selecionado e o solucionador. Como já mencionado, o usuário tem a possibilidade de escolher configurações diferentes. Se um determinado usuário não tem certeza sobre qual verificador e solucionador selecionar, a escolha padrão da ferramenta emprega o ESBMC com MathSAT ou CBMC com MiniSAT, dado que eles normalmente apresentam os tempos de execução mais curtos.

No entanto, a avaliação experimental não demonstra conclusivamente que são as melhores configurações possíveis (dado o pequeno conjunto de *benchmark*). Pretende-se que no futuro, a ferramenta possa selecionar automaticamente o par de verificador e solucionador, usando técnicas de aprendizado de máquinas que levem em consideração funções objetivos, com um grande conjunto de *benchmarks*. Essa abordagem é semelhante ao trabalho realizado por Hutter [67], que aplicam uma ferramenta de otimização de parâmetros para melhorar os solucionadores SAT para grandes instâncias de verificação de modelos delimitadas do mundo real, através de ajuste automático da decisão de procedimentos.

Outro parâmetro importante é o tipo de algoritmo, que pode ser `--convex`, para funções convexas, `--positive`, para funções semi-definidas positiva e `--generalized`, para funções sobre as quais não se tem conhecimento prévio. Uma vez que a função não seja convexa e não seja possível garantir que não seja negativo, a configuração sugerida usa a opção `--generalized` (*optCE adjiman.func* `--generalized`).

Seguindo o fluxo de execução ilustrado na Figura (4.9), o sinalizador `--start-value` é usado para especificar (*optCE <nome>.func* `--start-value=20`) a inicialização do algoritmo proposto e, quando não é adotado, esse valor é atribuído de forma aleatória. Observa-se que as variações em relação aos valores de inicialização não influenciam significativamente os tempos de convergência, uma vez que o OptCE avalia apenas a parte inteira das soluções no início das tarefas de otimização. Além disso, verificar com valores inteiros é rápido, é normal a *"verification failed"* na primeira execução, e um resultado de *"verification failed"* geralmente é mais rápido do que um *"verification successful"*, como também experimentalmente observado

em [19, 61].

Se a função de entrada for composta por operadores aritméticos, não é obrigatório usar a *flag* `--library`; Contudo, quando as funções matemáticas estão presentes, é necessário implementá-las em ANSI-C. Tais implementações influenciam consideravelmente os resultados da verificação quanto mais simples forem, ou seja, quanto menor for o número de operações e *loops*, mais fácil é para a abordagem proposta concluir as tarefas de verificação. No caso da função *adjiman*, que usa funções matemáticas como *sin ()* e *cos ()*, a biblioteca *math2.h* foi criada, com implementação própria, que foi incluído usando a *flag* `--library` (*optCE adjiman.func* `--library=math2.h`). Esta biblioteca contém uma implementação aprimorada da biblioteca da *math.h* original, que inclui pré e pós-condições para garantir que predicados sejam verificados antes e depois da chamada de execução de uma dada função matemática.

As funções matemáticas contidas na *math2.h* têm o mesmo nome dos elementos correspondentes na biblioteca ANSI-C, exceto que foi adicionado o caractere 2 (*ex.*, *cos2 ()*, *sin2 ()*, *abs2()*).

O sinalizador `--timeout` é usado para interromper processos de otimização, caso seja atingido o limite do tempo indicado (*optCE* <*nome _function*> *.func*`--timeout = 3600`). Finalmente, o usuário tem a opção de definir a precisão da solução do OptCE, ou seja, o sinalizador `-- precision` indica o número de casas decimais de uma solução. Quando um valor de referência não é fornecido, o OptCE encontra um mínimo global com 3 casas decimas por predefinição.

Problema de Fluxo de Potência Ótimo OPF

O fluxo de potência ótimo (OPF) representa o problema em definir os melhores níveis operacionais em uma usina de energia elétrica, tendo como objetivo minimizar o custo operacional e atender às demandas de fornecimento ao longo de uma rede de transmissão. Devido a energia elétrica fluir conforme funções não lineares e podendo haver características físicas do sistema sendo modeladas por funções não convexas, isso pode se tornar um problema complexo [68].

A seguir tem-se um exemplo prático de fluxo de potência ótimo (OPF), um problema comum no planejamento e operação de sistemas de energia, onde ainda é um desafio conhecer

a solução global deste problema em grandes instâncias.

As principais características do exemplo prático apresentado são:

- Existem 3 barramentos, um com cada gerador térmico de mesma característica; Os geradores possuem limites de potência ativa de 400 MW e sua faixa de potência reativa é $[-300, 300]$ MVAr.
- Tem-se 1 carga localizada no barramento 3, com demanda $P + jQ = 180 + j30$ MVA.
- Todos os barramentos são conectados por uma única linha, com $r + jx = 0,025 + j0,1$ pu de impedância.
- O custo unitário (CVU) dos geradores térmicos são: [10, 12, 20], respectivamente.

Considerando as características apresentadas, a formulação do OPF DC para este problema é apresentado a seguir conforme a Equação (4.6).

$$f(\theta_1, \theta_2, \theta_3) = 1000P_{g1} + 1200P_{g2} + 2000P_{g3} \tag{4.6}$$

s.t.

$Pg_1 = \frac{\theta_1-\theta_2}{0,1} + \frac{\theta_1-\theta_3}{0,1}$; $-1 \leq \frac{\theta_1-\theta_2}{0.1} \leq 1$;

$Pg_2 = \frac{\theta_1-\theta_2}{0,1} + \frac{\theta_1-\theta_3}{0,1}$; $-1 \leq \frac{\theta_1-\theta_3}{0.1} \leq 1$;

$Pg_3 = \frac{\theta_1-\theta_2}{0,1} + \frac{\theta_1-\theta_3}{0,1}$; $-1 \leq \frac{\theta_2-\theta_3}{0.1} \leq 1$;

$0 \leq Pg_1 \leq 4$; $0 \leq Pg_2 \leq 4$; $0 \leq Pg_3 \leq 4$;

$-\pi \leq \theta_1 \leq \pi$; $-\pi \leq \theta_2 \leq \pi$; $-\pi \leq \theta_3 \leq \pi$;

O problema apresentado seria incapaz de ser otimizado pelo OptCE considerando somente o antigo formato de arquivo de entrada, onde era necessário inserir apenas a descrição da função e os limites de faixa de valores que as variáveis de decisão poderiam assumir. Conforme apresentado na seção 4.1.2, para estes casos pode-se usar o tipo de arquivo de entrada tipo extensivo, onde permite o usuário associar as variáveis de decição para estabelecer novas restrições. O arquivo de entrada tipo extensivo para este problema é apresentado na figura (4.10).

A função é definida usando a variável `fobj` e seguindo o padrão ANSI-C. As restrições são representadas logo depois, os valores de θ podem assumir valores igual ou entre entre $-\pi$

```
1  fobj = 1000*(((X1-X2)/0.1) + ((X1-X3)/0.1)) + 1200*(((X2-X3)/0.1) + ((X2
       -X1)/0.1)) + 2000*(((X3-X1)/0.1) + ((X3-X2)/0.1) + 1.8);
2  #
3  (X1 >= -PI) && (X1 <= PI)
4  (X2 >= -PI) && (X2 <= PI)
5  (X3 >= -PI) && (X3 <= PI)
6  (((X1-X2)/0.1)>= -1) && (((X1-X2)/0.1)<= 1)
7  (((X1-X3)/0.1)>= -1) && (((X1-X3)/0.1)<= 1)
8  (((X2-X3)/0.1)>= -1) && (((X2-X3)/0.1)<= 1)
9  ((((X1-X2)/0.1)+((X1-X3)/0.1))>=0) && ((((X1-X2)/0.1)+((X1-X3)/0.1))<=4)
10 ((((X2-X3)/0.1)+((X2-X1)/0.1))>=0) && ((((X2-X3)/0.1)+((X2-X1)/0.1))<=4)
11 ((((X3-X1)/0.1)+((X3-X2)/0.1) + 1.8)>=0) && ((((X3-X1)/0.1) + ((X3-X2)
       /0.1) + 1.8)<=4)
```

Figura 4.10: Arquivo de entrada para o problema de OPF.

e π (linhas 3,4, e 5), depois tem-se a associação de θ devendo ser igual ou entre -1 e 1, e por fim Pg deve ser igual ou entre 0 e 4.

O OptCE foi capaz de otimizar o problema e encontrar as variáveis de decisão $Pg = [1,2;0,6;0,0]$; $\theta = [0,0;-0,02;-0,1]$ e obter o mínimo global, o custo objetivo de 1920 \$/ hora. Apesar do OptCE não ser um solucionador OPF, ele pode otimizar o problema e encontrar o seu mínimo global.

4.2 Resumo

Este capítulo apresentou o desenvolvimento e evolução da ferramenta de otimização OptCE. Foram descritos: a arquitetura da ferramenta; os dados necessários para a otimização de uma função; os recursos e configurações existentes para a ferramenta e exemplos de otimização com a ferramenta.

Capítulo 5

Avaliação Experimental

Esta subseção decreve a configuração, execução e resultados dos experimentos realizados para avaliar a ferramenta OptCE. Foram avaliadas as melhorias entre as versões do optCE 1.0 e 1.5, assim como também as implementações dos algoritmos CEGIO. Por fim é avaliado a performace de tempo e taxa de acerto do OptCE em comparação com outras técnicas de otimização.

5.1 Objetivos dos Experimentos

Q1 (Validação) O OptCE é capaz de encontrar os mínimos e máximos globais em uma função de otimização?

Q2 (Configuração) A escolha das configurações entre ferramentas BMC e solucionadores influencia os resultados da otimização?

Q3 (Desempenho) Quais diferenças do OptCE em comparação com as técnicas tradicionais de otimização?

5.2 Avaliação das Mudanças no OptCE

OptCE 1.0 x OptCE 1.5

Conforme apresentado na seção 4.1.4, o OptCE sofreu algumas mudanças da versão 1.0 para versão atual 1.5, onde o arquivo de especificação não faz uso de *loops* como `for`, e

também aplica uma regra de interrupção do algoritmo, checando e comparando o valor mínimo obtido em cada precisão.

Os experimentos apresentados nesta seção 5.2 tem a finalidade de comparar os resultados do OptCE 1.0 [20] com os resultados obtidos considerando as recentes mudanças inseridas no OptCE 1.5, além de validar o bom funcionamento em localizar os mínimos globais. Para garantir a confiabilidade dos testes, os mesmos *benchmarks* mostrados na Tabela (5.1) usados para avaliar a versão 1.0 foram submetidos a otimização com a nova versão OptCE 1.5, considerando a mesma configuração do ambiente experimental, um computador com CPU Intel Core i7 – 4790 de 3.60 GHz, 16GB de RAM, configurado para 3 casas decimais.

Tabela 5.1: *Suíte* de Teste OptCE 1.0.

#	Benchmark	Domínimo	Mínimo Global
1	Alpine 1	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
2	Cosine	$-1 \leq x_i \leq 1$	$f(0,0) = -0.2$
3	Styblinski Tang	$-5 \leq x_i \leq 5$	$f(-2.903,-2.903) = -78.332$
4	Zirilli	$-10 \leq x_i \leq 10$	$f(-1.046,0) \approx -0.3523$
5	Booth	$-10 \leq x_i \leq 10$	$f(1,3) = 0$
6	Himmeblau	$-5 \leq x_i \leq 5$	$f(3,2) = 0$
7	Leon	$-2 \leq x_i \leq 2$	$f(1,1) = 0$
8	Zettl	$-5 \leq x_i \leq 10$	$f(-0.029,0) = -0.0037$
9	Sum Square	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
10	Rotated Ellipse	$-100 \leq x_i \leq 100$	$f(0,0) = 0$

Iniciamente o mais importante é garantir o bom funcionamento em localizar o mínimo global. Os mínimos globais encontrados em cada configuração do verificador e solucionador são mesmos obtidos na literatura considerando soluções com 3 casas decimais. A Tabela 5.2 apresenta os tempos de execução do OptCE 1.0 [20] e os tempos obtidos considerando as novas mudanças no OptCE 1.5 em segundos. Cada coluna da Tabela (5.2) é descrita da seguinte forma: a coluna 1 está relacionada às funções do conjunto de referência [20], as colunas 2, 3, 4 e 5 apresentam os resultados do OptCE 1.0, enquanto que as colunas 6,7,8 e 9 apresentam os resultados do OptCE 1.5. As colunas 2 e 6 dispõe os dados obtidos com a configuração do ESBMC e MathSAT, as colunas 3 e 7 com a configuração do ESBMC e Z3, colunas 4 e 8 ESBMC e Boolector (Bool) e colunas 5 e 9 CBMC e Minisat.

Todos os *benchmarks* da suíte tiveram os mínimos globais encontrados com as novas modificações OptCE 1.5, em todas as configurações de verificador e solucionador. Como visto na Tabela (5.2) a grande maioria dos experimentos sofreram importantes reduções de tempo

de execução, principalmente os *benchmarks* 1, 2 e 3 que possuiam os tempos mais elevados em todas as configurações e também todos os *benchmarks* nas configurações ESBMC-Z3 e ESBMC-Boolector.

Tabela 5.2: Avaliação comparativa da nova abordagem da especificação e lógica de interrupção

#	OptCE 1.0				OptCE 1.5			
	ESBMC			CBMC	ESBMC			CBMC
	MathSAT	Z3	Bool	MiniSAT	MathSAT	Z3	Bool	MiniSAT
1	**1068**	105192	3387	5344	623	**444**	1070	4726
2	**4130**	80481	5003	8509	**329**	552	621	1433
3	**443**	37778	2027	2438	**182**	1380	240	253
4	468	387	**190**	1143	428	353	**180**	1128
5	7	1244	4016	**2**	7	52	48	**2**
6	12	14205	6217	**4**	12	35	30	**4**
7	5	2443	212	**2**	5	5	7	**2**
8	13	753	389	**9**	13	622	322	**9**
9	18	4171	4438	**13**	14	21	17	**5**
10	3	72	39	**2**	3	33	19	**2**
ST	6167	246726	25918	17466	1616	3497	2554	7564

Essa redução no tempo de otimização como mostra o gráfico na Figura (5.1), só foi possível com a reformulação do arquivo de especificação sem o uso de *loops* e a aplicação da lógica de interrupção ao algoritmo estabelecido.

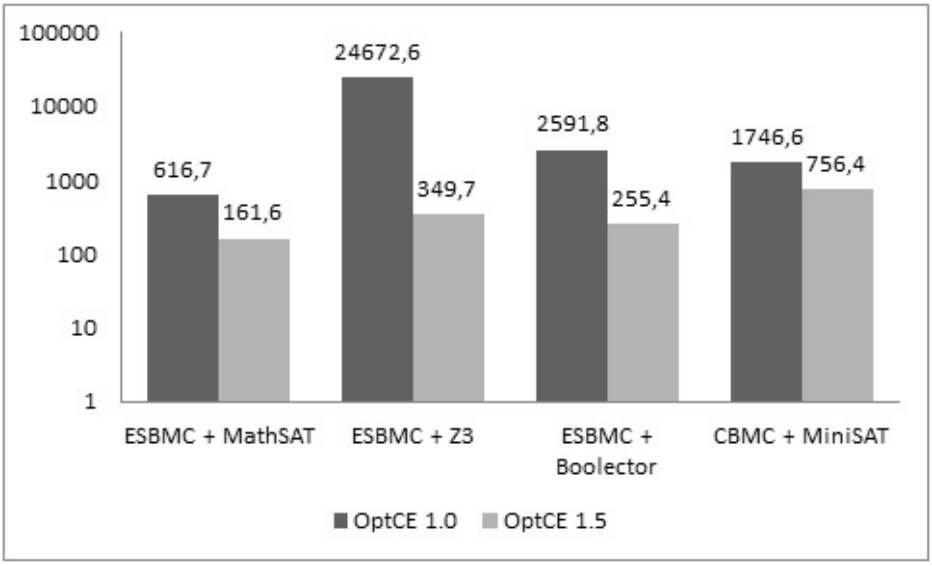

Figura 5.1: Comparação do tempo total de otimização da *suíte* de teste 5.1 entre OptCE 1.0 e OptCE 1.5, escala logarítmica.

A soma total das execuções (ST) da suíte é apresentada ao final da Tabela (5.2), onde pode-se perceber que a configuração ESBMC-Z3 foi a que obteve maior redução de tempo entre as versões OptCE 1.0 e OptCE 1.5, sendo $70,55$ vezes mais rápida, seguida pela configuração ESBMC-Boolector sendo $10,14$ vezes mais rápida que na versão anterior. As configurações CBMC-MiniSAT e ESBMC-MathSAT obtiveram as menores reduções considerando as mudanças apresentadas sendo de $2,30$ e $3,81$ vezes mais rápida que suas versões anteriores.

Por possuir mais casas decimais, em geral as verificações com precisões maiores são mais demoradas. As configurações ESBMC-Z3 e ESBMC-Boolector tomavam maior tempo para resolver estas verificações em maiores precisões se comparados às configurações CBMC-MiniSAT e ESBMC-MathSAT. Durante o processo de otimização com o ESBMC-Z3 e ESBMC-Boolector, mesmo após já terem localizado o mínimo global em uma determinada precisão, muitas outras verificações eram realizadas ao longo das demais precisões, até atingir a precisão estabelecida na execução. Com a aplicação da lógica de interrupção, todas as configurações buscam paralizar e em seguida finalizar o algoritmo, considerando o último valor de mínimo global encontrado na precisão corrente. Todas as configurações evitam realizar novas verificações, porém as melhorias foram mais notáveis nas configurações ESBMC-Z3 e ESBMC-Boolector, visto que possuíam maiores dificuldades na verificação em precisões maiores.

5.3 Avaliação do OptCE 1.5

Os resultados experimentais do OptCE 1.5 são apresentados em quatro tabelas. As Tabelas (5.5), (5.6) e (5.7) mostram aos *benchmarks* usados para avaliar as implementações dos algoritmos CEGIO definidas pelas respectivas *flags* `--generalized`, `--positive` e `--convex`. Por fim a Tabela (5.8) relata uma comparação entre a ferramenta OptCE v1.5 e outras técnicas tradicionais de otimização.

5.3.1 Descrição dos *benchmarks*

Com o objetivo de avaliar a ferramenta OptCE, foram escolhidas 40 funções (funções convexas e não convexas) de problemas clássicos de otimização para execução dos experimentos [60]. Os *benchmarks* possuem características diferentes: contínuo, diferenciável, separável, não separável, escalável, não escalável, uni-modal e multimodal, que incluem seno, cosseno, polinômios, soma e raiz quadrada.

A Tabela (5.3) apresenta a *suíte* de teste constituída com os seguintes títulos das colunas: nome do *benchmark*, domínio de otimização e mínimo global.

Todos os *benchmarks* foram usados para avaliar a *flag* `--generalized` que implementa o algoritmo CEGIO-G. Para a *flag* `--positive` que implementa o algoritmo CEGIO-S, foram utilizadas as funções semi-definidas positivas de 21 a 30 da Tabela (5.3). Por fim, foram usadas as funções convexas de 31 a 40 da Tabela (5.3) para avaliar a *flag* `--convex`, que implementa o algoritmo CEGIO-F.

Os resultados dos experimentos foram comparados com outras técnicas (algoritmo genético, enxame de partículas, pesquisa de padrões, recozimento simulado e programação não linear), onde os *benchmarks* foram executados com a *ToolBox* de Otimização do MATLAB (2016b) [69]. Os tempos apresentados nas tabelas a seguir estão relacionados ao tempo médio de 20 execuções consecutivas para cada *benchmark*.

Os experimentos foram executados utilizando 11 computador com mesma configuração, CPU Intel Core *i*7 geração 7 de 3.60 GHz, 32 GB de RAM e Linux OS Ubuntu 18.04, configurado para obter mínimos globais com 3 casas decimais.

Tabela 5.3: *Suíte* de Teste OptCE 1.5.

#	Benchmark	Domain	Global Minimum
1	Adjiman	$-1 \leq x_i \leq 1$	$f(0;0) = -2,02181$
2	Alpine 1	$-10 \leq x_i \leq 10$	$f(0;0) = 0$
3	Bohachevsky 1	$-50 \leq x_i \leq 50$	$f(0;0) = 0$
4	Bohachevsky 3	$-50 \leq x_i \leq 50$	$f(0;0) = 0$
5	Branin RCOS 01	$-5 \leq x_1 \leq 10$ $0 \leq x_2 \leq 15$	$f\{(-\pi;12,275),(\pi;2,275),$ $(9,424;2,475)\} = 0,397$
6	Camel Six	$-3 \leq x_1 \leq 3$ $-2 \leq x_2 \leq 2$	$f\{(0,089;-0,712),$ $(-0,089;0,712)\} = -1,031$
7	Camel Three	$-5 \leq x_i \leq 5$	$f(0;0) = 0$
8	Cosine	$-1 \leq x_i \leq 1$	$f(0;0) = -0,2$
9	Egg Crate	$-5 \leq x_i \leq 5$	$f(0;0) = 0$
10	Engvall	$-10 \leq x_i \leq 10$	$f(1;0) = 0$
11	Godstein Price	$-2 \leq x_i \leq 2$	$f(0;-1) = 3$
12	MC Cormick	$-2 \leq x_1 \leq 4$ $-3 \leq x_2 \leq 4$	$f(-0,547;-1,547) = -1,913$
13	Rotated Ellipse 01	$-10 \leq x_i \leq 10$	$f(0;0) = 0$
14	Scahffer 1	$-100 \leq x_i \leq 100$	$f(0;0) = 0$
15	Styblinski Tang	$-5 \leq x_i \leq 5$	$f(-2,903;-2,903) = -78,332$
16	Trecanni	$-5 \leq x_i \leq 5$	$f\{(-2;0),(0;0)\} = 0$
17	Tsoulos	$-1 \leq x_i \leq 1$	$f(0;0) = -2$
18	Ursem 1	$-2,5 \leq x_1 \leq 3$ $-2 \leq x_2 \leq 2$	$f(1,697;0) = -4.816$
19	Wayburn Seader 1	$-100 \leq x_i \leq 100$	$f\{(0,2;1),(0,42486;1)\} = 0$
20	Zirilli	$-500 \leq x_i \leq 500$	$f(-1,046;0) = -0,352$
21	Booth	$-10 \leq x_i \leq 10$	$f(1;3) = 0$
22	Dixo Price	$-10 \leq x_i \leq 10$	$f(1;0,5) = 0$
23	Himmeblau	$-5 \leq x_i \leq 5$	$f(3,2) = 0$
24	Leon	$-2 \leq x_i \leq 2$	$f(1,1) = 0$
25	Price 1	$-10 \leq x_i \leq 10$	$f(\pm 5,\pm 5) = 0$
26	Price 4	$-100 \leq x_i \leq 100$	$f\{(0,0),(2,4),$ $(1.464,-2.506)\} = 0$
27	Rosenbrock	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
28	Powell Sum	$-1 \leq x_i \leq 1$	$f(0,0) = 0$
29	Schuwefel 2.25	$-10 \leq x_i \leq 10$	$f(1,1) = 0$
30	Wayburn Seader 1	$-500 \leq x_i \leq 500$	$f\{(1,2),(1.596,0.806)\} = 0$
31	Chung Reynolds	$-100 \leq x_i \leq 100$	$f(0,0) = 0$
32	Cube	$-10 \leq x_i \leq 10$	$f(1,1) = 0$
33	Matyas	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
34	Rotated Ellipse 02	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
35	Schumer	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
36	Schwefel 2.20	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
37	Sphere	$-6 \leq x_i \leq 6$	$f(0,0) = 0$
38	Sum Square	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
39	Zakharov	$-5 \leq x_i \leq 10$	$f(0,0) = 0$
40	Zettl	$-1 \leq x_i \leq 5$	$f(-0.029,0) = -0.0037$

5.3.2 Avaliação da *flag* `--generalized` (CEGIO-G)

Para avaliar a implementação do CEGIO-G, todos os *benchmarks* foram otimizados sendo configurados com a *flag* `--generalized`. As experiências foram repetidas para diferentes combinações de verificadores e solucionadores SAT/SMT, ou seja, o ESBMC foi combinado com três solucionadores (MathSAT, Z3 e Boolector) e CBMC com o MiniSAT.

Cada coluna da Tabela (5.5) é descrita da seguinte forma: a coluna 1 está relacionada às funções do conjunto de referência, as colunas 2, 3 e 4 estão relacionadas à configuração do ESBMC com MathSAT, Z3, e os solucionadores Boolector, respectivamente, e a coluna 5 está relacionada à configuração CBMC com o MiniSat.

Para o experimento da *flag* `--generalized` o mínimo global foi encontrado em todos os *benchmarks* da suíte de teste em todas as combinações das ferramentas BMC e solucionadores. Foi configurado um limite de execução para o experimento, *timeout* de 86400 segundos, que corresponde a 24*h*. Conforme a Tabela (5.4), dentre todas as execuções houveram 6 casos de *timeout* ocorrendo para: a função *MC Cormick* nas configurações `ESBMC + MathSAT`, `ESBMC + Z3` e `CBMC + MiniSAT`; a função *Branin RCOS 01* nas configurações `ESBMC + MathSAT` e `ESBMC + Z3`; e a função *Scahffer 1* na configuração `CBMC + MiniSAT`.

Tabela 5.4: Tempos de execução para `--generalized` (CEGIO-G), em segundos.

#	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT
5	*Timeout*	*Timeout*	47035	**11830**
12	*Timeout*	*Timeout*	**2664**	*Timeout*
14	6515	1225	**510**	*Timeout*

Os tempos de otimização da Tabela (5.5) variaram significativamente o que dificulta definir uma melhor configuração de verificador e solucionador. Quando considerado os casos de *timeout* (86400s) no cálculo de tempo da execução da *suíte*, pode-se concluir conforme a Figura (5.2) que a configuração `ESBMC + Boolector` é a melhor opção na *suíte* de teste analisada com soma total de 73378 segundos, seguida da configuração `CBMC + MiniSAT` com 237138 segundos, tendo os resultados mais longos o `ESBMC + MathSAT` e o `ESBMC + Z3` obtiveram a soma total de tempo 240830 e 291778 segundos respectivamente.

Os dados analisados considerando os *timeouts* facilitam o `ESBMC + Boolector` obter melhor tempo em comparação com as outras configurações, uma vez que este não teve nenhum *timeout*, enquanto que as demais configurações obtiveram cada uma dois casos de *timeout*.

Tabela 5.5: Tempos de execução para `--generalized` (CEGIO-G), em segundos.

#	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT
1	4246	14770	**1746**	3627
2	443	**290**	657	4614
3	**1759**	4470	2466	2153
4	**3488**	22475	5891	9593
5	*Timeout*	*Timeout*	47035	**11830**
6	40133	32772	3708	**736**
7	**92**	136	134	13174
8	**293**	514	577	1062
9	959	910	**578**	3539
10	**24**	36	58	1646
11	2127	2708	2850	**2020**
12	*Timeout*	*Timeout*	**2664**	*Timeout*
13	75	50	60	**7**
14	6515	1225	**510**	*Timeout*
15	**143**	1327	223	205
16	**4**	6	11	22
17	**200**	206	258	915
18	5179	34734	**2015**	7989
19	1215	1059	722	**13**
20	417	362	**193**	1022
21	7	49	46	**2**
22	7	**4**	8	6
23	12	33	26	**4**
24	4	4	7	**2**
25	5	4	9	**4**
26	11	**6**	8	**6**
27	12	8	17	**3**
28	5	**3**	4	7
29	35	**18**	20	20
30	24	**22**	31	35
31	11	6	12	**2**
32	23	23	53	**5**
33	465	**21**	364	38
34	3	30	18	**2**
35	40	21	37	**6**
36	**3**	**3**	4	**3**
37	**2**	4	3	**2**
38	13	21	16	**5**
39	23	14	34	**10**
40	13	634	305	**8**

Fazendo uma nova análise onde os casos de *timeouts* são desconsiderados, pode-se constatar conforme a Figura (5.2) que o `ESBMC + Boolector` não apresenta a melhor soma total de tempo ficando com 73378 segundos, atrás de `ESBMC + MathSAT` com 68030 segundos e `CBMC + MiniSAT` com 64338 segundos, com o pior resultado permanece o `ESBMC + Z3` com 118978 segundos.

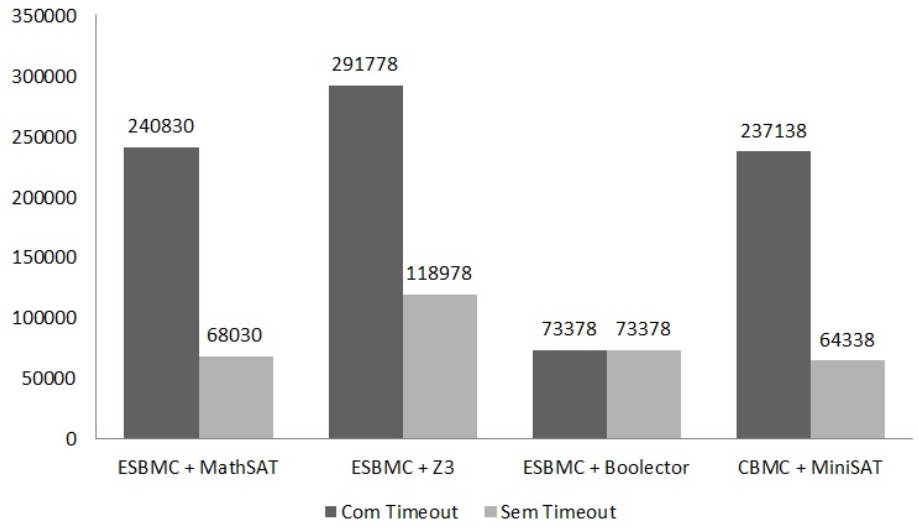

Figura 5.2: Tempo total de otimização da suíte de testes obtida na Tabela (5.3), são considerados a avaliação com e sem *timeout*, escala normal.

Além do `CBMC + MiniSAT` ter obtido a melhor soma total de tempo de execução da suíte de teste com 64338 segundos, quando desconsiderado os *timeouts*, teve também a maior quantidade de *benchmarks* resolvidos em menor tempo com 20 *benchmarks* solucionados ou 50% da *suíte* de teste, enquanto que o `ESBMC + MathSAT` pode resolver em menor tempo 10 *benchmarks* ou 25% da *suíte*, por seguinte o `ESBMC + Z3` resolveu 8 *benchmarks* ou 20% *suíte* e por fim o `ESBMC + Boolector` obteve 6 *benchmarks* resolvidos ou 15% da *suíte*.

As combinações CBMC + MiniSAT e ESBMC + Math-SAT apresentaram os melhores resultados em comparação com as outras configurações do OptCE, dado que Boolector não suporta a aritmética de ponto flutuante [70]. O solucionador MathSAT que obteve os melhores resultados, suporta a aritmética de ponto fixo e flutuante. Os resultados evidenciam que a otimização com a aritmética de ponto flutuante é melhor que a de ponto fixo, dessa maneira é principalmente sugerida a configuração `CBMC + MiniSAT` para o uso da *flag* `--generalized`.

5.3.3 Avaliação da *flag* `--positive` (CEGIO-S)

A Tabela (5.6) e Figura (5.3) apresenta os resultados para a *flag* `--positive`, que é adaptada para funções semi-definidas positivas. Foram usados os *benchmarks* #21 – 30 para este experimento, pois fazem uso de módulos com potências par, onde visualmente pode-se garantir que estas funções não podem atingir um mínimo global negativo.

Tabela 5.6: Tempos de execução para `--positive` (CEGIO-S), em segundos.

#	`--generalized`				`--positive`			
	ESBMC			CBMC	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT	MathSAT	Z3	Boolector	MiniSAT
21	7	49	46	**2**	3	9	**1**	2
22	7	**4**	8	6	5	**1**	7	5
23	12	33	26	**4**	4	2	**<1**	4
24	4	4	7	**2**	2	**1**	**1**	2
25	5	4	9	**4**	4	**<1**	4	2
26	11	**6**	8	**6**	7	**3**	**3**	**3**
27	12	8	17	**3**	4	**2**	**2**	3
28	5	**3**	4	8	4	**2**	4	7
29	35	**18**	20	20	10	**3**	4	4
30	24	**22**	31	35	**7**	**4**	5	13

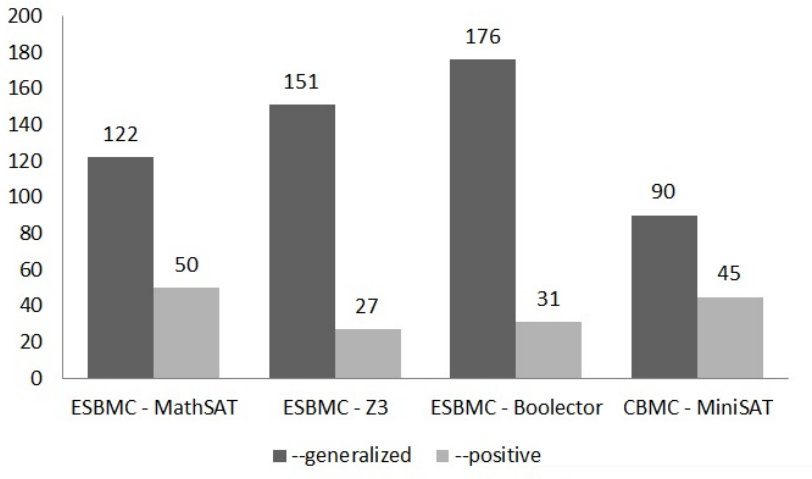

Figura 5.3: Gráfico com tempos de execução para `--positive` (CEGIO-S). Dados obtidos da Tabela (5.6).

5.3.4 Avaliação da *flag* `--convex` (CEGIO-F)

A implementação do algoritmo CEGIO-F é atribuída com a *flag* `--convex`. Para avaliar seu desempenho, foram utilizados os *benchmarks* #31 – 40 por serem funções convexas. Os resultados são apresentados na Tabela (5.7) e Figura (5.4). Os tempos de otimização utilizando o algoritmo específico para essa classe de função foram menores que os tempos apresentados pelo algoritmo generalizado. Isso acontece porque neste algoritmo, a cada verificação realizada o espaço de busca é reduzido, conforme o candidato mínimo global encontrado, reduzindo assim o tempo de verificação e conseqüentemente os tempos de otimização.

Tabela 5.7: Tempos de execução para `--convex` (CEGIO-F), em segundos.

#	`--generalized`				`--convex`			
	ESBMC			CBMC	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT	MathSAT	Z3	Boolector	MiniSAT
31	11	6	12	**2**	9	5	8	**2**
32	23	23	53	**5**	23	12	40	**3**
33	465	**21**	364	38	**7**	19	10	25
34	3	30	18	**2**	3	3	11	**2**
35	40	21	37	**6**	33	17	30	**4**
36	**3**	**3**	4	**3**	3	**2**	**2**	3
37	**2**	4	3	**2**	2	2	3	**1**
38	13	21	16	**5**	10	**3**	5	**3**
39	23	14	34	**10**	22	11	30	**7**
40	13	634	305	**8**	13	6	6	**5**

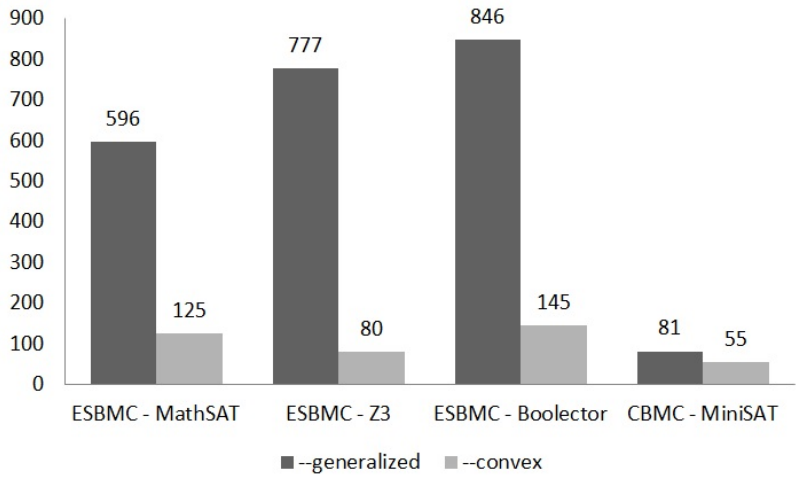

Figura 5.4: Gráfico com tempos de execução para `--convex` (CEGIO-F). Dados obtidos da Tabela (5.7).

5.3.5 OptCE x Outras Técnicas

Na Tabela (5.8) são apresentados os melhores resultados da otimização dos *benchmarks*, considerando as abordagens implementadas na ferramenta OptCE 1.5. Consta também na Tabela (5.8) os resultados da otimização com outras técnicas.

Tabela 5.8: Resultados experimentais para técnicas tradicionais e os melhores resultados com algoritmos CEGIO, em segundos.

#	OptCE			GA		ParSwarm		PatSearch		SA		NLP	
	Confing	**R%**	**T**	**R%**	**T**	**R%**	**T**	**R%**	**T**	**R%**	**T**	**R%**	**T**
1	G+3	100	1746	40	1	100	**<1**	85	**<1**	40	**<1**	100	**<1**
2	G+2	100	290	20	1	15	**<1**	15	**<1**	0	**<1**	5	**<1**
3	G+1	100	1759	100	1	100	**<1**	100	**<1**	65	**<1**	30	**<1**
4	G+1	100	3488	100	1	100	**<1**	75	**<1**	65	**<1**	45	**<1**
5	G+4	100	11830	70	1	80	**<1**	100	**<1**	75	**<1**	85	**<1**
6	G+4	100	736	55	1	55	**<1**	70	**<1**	40	**<1**	50	**<1**
7	G+1	100	92	95	1	100	**<1**	75	**<1**	90	**<1**	75	**<1**
8	G+1	100	293	100	1	100	**<1**	50	**<1**	95	**<1**	15	**<1**
9	G+3	100	578	100	1	100	**<1**	50	**<1**	95	**<1**	100	**<1**
10	G+1	100	24	100	1	100	**<1**	100	**<1**	95	**<1**	100	**<1**
11	G+4	100	2020	100	1	95	**<1**	25	**<1**	100	**<1**	60	**<1**
12	G+3	100	2664	100	1	80	**<1**	45	**<1**	80	**<1**	45	**<1**
13	G+4	100	7	100	1	100	**<1**	100	**<1**	100	**<1**	100	**<1**
14	G+3	100	510	100	1	90	**<1**	100	**<1**	100	**<1**	100	**<1**
15	G+1	100	143	100	1	100	**<1**	45	**<1**	100	**<1**	50	**<1**
16	G+1	100	4	100	1	100	**<1**	100	**<1**	100	**<1**	100	**<1**
17	G+1	100	200	100	1	95	**<1**	20	**<1**	50	**<1**	0	**<1**
18	G+3	100	2015	100	1	100	**<1**	100	**<1**	95	**<1**	45	**<1**
19	G+4	100	13	60	1	30	**<1**	0	**<1**	5	**<1**	50	**<1**
20	G+3	100	193	100	1	100	**<1**	100	**<1**	90	**<1**	70	**<1**
21	P+3	100	2	100	1	100	**<1**	100	**<1**	95	**<1**	100	**<1**
22	P+2	100	3	100	1	100	**<1**	100	**<1**	100	**<1**	100	**<1**
23	P+3	100	7	40	1	50	**<1**	20	**<1**	20	**<1**	25	**<1**
24	P+2-3	100	2	95	1	85	**<1**	5	**<1**	5	**<1**	100	**<1**
25	P+2	100	4	20	1	15	**<1**	15	**<1**	35	**<1**	15	**<1**
26	P+2-3-4	100	2	45	1	35	**<1**	45	**<1**	60	1	25	**<1**
27	P+2-3	100	1	50	1	65	**<1**	0	**<1**	10	**<1**	100	**<1**
28	P+2	100	3	100	1	100	**<1**	100	**<1**	55	**<1**	100	**<1**
29	P+3	100	7	100	1	100	**<1**	100	**<1**	90	**<1**	100	**<1**
30	P+2	100	5	45	1	70	**<1**	5	**<1**	25	**<1**	55	**<1**
31	C+4	100	2	100	1	100	**<1**	100	**<1**	90	**<1**	100	**<1**
32	C+4	100	3	35	1	25	**<1**	0	**<1**	10	**<1**	100	**<1**
33	C+1	100	7	100	1	40	**<1**	70	**<1**	15	**<1**	100	**<1**
34	C+4	100	2	100	**<1**	100	**<1**	100	**<1**	85	**<1**	100	**<1**
35	C+4	100	4	100	**<1**	100	**<1**	100	**<1**	80	**<1**	100	**<1**
36	C+2-3	100	2	100	1	100	**<1**	100	**<1**	70	**<1**	100	**<1**
37	C+4	100	1	100	1	100	**<1**	100	**<1**	100	**<1**	100	**<1**
38	C+4	100	3	100	1	100	**<1**	100	**<1**	90	**<1**	100	**<1**
39	C+4	100	7	100	1	100	**<1**	100	**<1**	95	**<1**	100	**<1**
40	C+4	100	5	100	1	100	**<1**	100	**<1**	25	**<1**	100	**<1**

A coluna `Configuração` mostra as combinações dos tipos de algoritmos (identificadas com as iniciais das *flags*, "G" para `--generalized`, "P" para `--positive` e "C" para `convex`), ferramentas BMC e solucionadores (identificadas com 1 para `ESBMC+MathSAT`, 2 para `ESBMC+Z3`, 3 para `ESBMC+Boolector` e 4 para `CBMC+MiniSAT`.). A comparação é realizada com técnicas tradicionais de otimização: algoritmo genético (GA), enxame de partículas (ParSwarm), pesquisa de padrões (PatSearch), *simulated annealing* (SA) e programação não linear (NLP). Todos os *benchmarks* avaliados foram executados 20 vezes com as técnicas tradicionais, utilizando o MATLAB (2016b), e 20 vezes com o OptCE 1.5. As repetições foram realizadas para garantir a convergência da taxa de acertos em todos os algoritmos.

Conforme a Figura (5.5), os experimentos mostram que o OptCE necessita de maior tempo e recursos de memória que outras técnicas, a fim de localizar os mínimos globais, no entanto, como pode-se ver na Figura (5.6), sua taxa de acerto é sempre superior considerando os experimentos executados, obtendo 100% de taxa de acerto com esta suíte de referência.

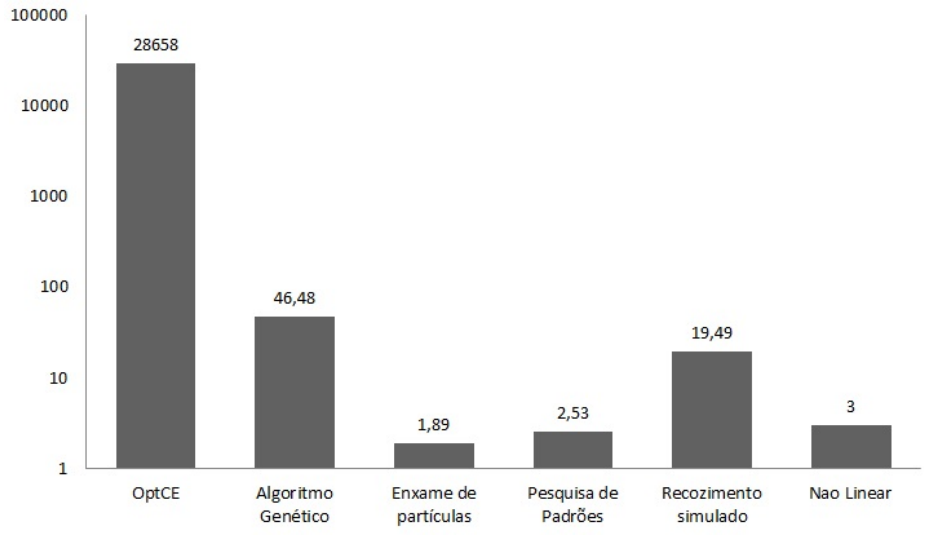

Figura 5.5: Gráfico com os melhores tempos de execução do OptCE em comparação com as técnicas de otimização: Algoritmo genético, Enxame de Partículas, Pesquisa de Padrões, Recozimento Simulado e Programação não linear. Dados obtidos da Tabela (5.8).

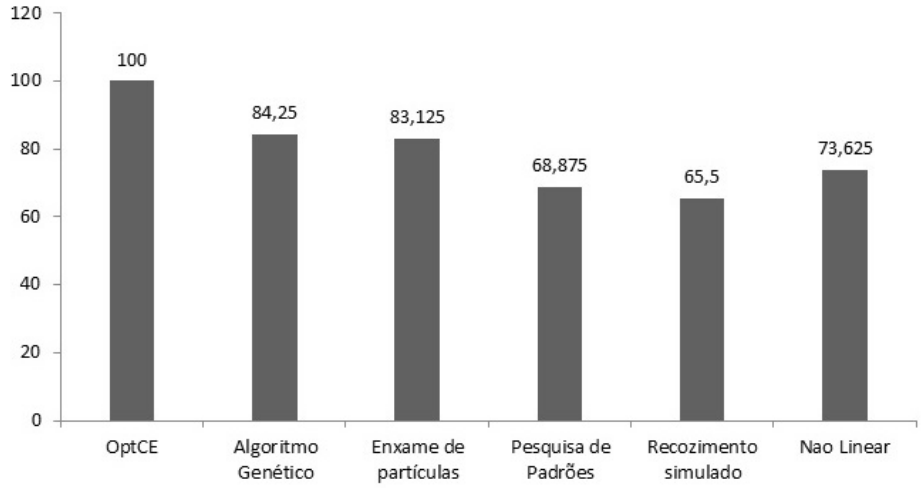

Figura 5.6: Gráfico da taxa de acerto do OptCE em comparação com as técnicas de otimização: Algoritmo genético, Enxame de Partículas, Pesquisa de Padrões, Recozimento Simulado e Programação não linear. Dados obtidos da Tabela (5.8).

Os resultados da otimização usando as *flags* `--positive` (CEGIO-S) e `--convex` (CEGIO-F) obtiveram tempos semelhantes fornecido pelas outras técnicas, mas com taxas de acerto superiores. As técnicas de otimização escolhidas para comparação, não obtiveram soluções para os *benchmarks* adotados em alguns casos, considerando a precisão estabelecida de 3 casas decimais. As técnicas tradicionais obtiveram 100% de taxa de acerto somente para as funções convexas, uma vez que essas funções não possuem mínimos locais que possam comprometer os resultados. Isso ocorre, pois as técnicas são sensíveis à não-convexidade, em muitos casos, ficam presos em mínimos locais, resultando em soluções subótimas.

A abordagem guiada a contraexemplo pode ser usada em problemas de otimização, podendo aplicar seus algoritmos conforme o problema, e assim obter melhores resultados. O usuário tem a possibilidade de usar a configuração `--convex`, especificamente para funções convexas, ou `--positive` para otimizar funções semi-definidas positivas, como funções de distância ou potência. Mas assim como as técnicas de verificação, existem restrições quanto ao tempo de verificação e o consumo de memória.

Quando analisa-se o desempenho das funções não convexas, as técnicas tradicionais ficam presas em mínimos locais e retornam soluções subótimas, o que reduz sua taxa de acerto.

O OptCE demanda um tempo um pouco acima dos seus concorrentes, mas retorna a solução ótima da função.

O desempenho do OptCE utilizando as *flags* específicas para as funções convexas e positivas se mostrou competitivo, uma vez que os tempos de execução obtidos foram muito próximos as de outras técnicas, e os mínimos globais foram encontrados em todos os casos. Dependendo do tipo de problema, o número de casas decimais da solução pode ser menor do que a quantidade usada nesta avaliação experimental. Para esses casos, os tempos de execução relativos à localização das soluções ótimas são reduzidos, uma vez que há menos casas decimais a serem verificadas, o que implica em uma quantidade menor de verificações, e consequentemente menos estados a serem considerados.

5.4 Resumo

Os resultados dos experimentos foram satisfatórios, mostrando que a implementação dos algoritmos CEGIO no OptCE é eficiente, mostrando também que as modificações na especificação a inserção de uma lógica de interrupção reduziram o tempo de otimização, por fim constatando que a ferramenta é capaz de otimizar funções convexas e não convexas. O OptCE mostrou-se vatajoso em comparação com outras técnicas de otimização conforme os resultados experimentais, pois foi capaz de obter a solução ótima em todos os *benchmarks*, enquanto que as demais técnicas obtiveram soluções subótimas, mesmo que o OptCE apresente na maioria dos resultados um maior tempo de execução.

Capítulo 6

Conclusões

6.1 Considerações Finais

Este trabalho apresentou as contribuições para o desenvolvimento da abordagem de otimização indutiva guiada por contraexemplos (Algoritmos CEGIO). Também apresentou a criação de uma ferramenta de otimização baseada nos algoritmos desenvolvidos, a ferramenta de otimização OptCE. Os 3 algoritmos de otimização que fazem uso de contraexemplos SAT e SMT são descritos, demonstrando a metodologia estabelecida para a otimização de funções convexas e não convexas. O funcionamento da ferramenta OptCE foi descrito, juntamente com suas funcionalidades implementadas, capacidades e limitações. Também é apresentado uma avaliação experimental do OptCE juntamente com as mesmas técnicas de otimização usadas para o desenvolvimento dos algoritmos CEGIO.

A avaliação experimental apresentada na seção 5.2 permitiu constata que é possível reduzir o tempo de otimização da técnica guiada por contraexemplo e consequentmente do OptCE fazendo modificações no arquivo de especificação e criando mecanismos para interromper o algoritmo caso a solução já tenha sido localizado.

Durante a avaliação experimental com o OptCE ficou evidente a influência dos solucionadores usados no processo de verificação, variando o tempo de otimização entre as configurações, mas garantem o mesmo valor de mínimo global em todas configurações. Contudo, ainda não é possível determinar explicitamente qual configuração é a melhor para cada cenário. Os resultados não convergem para um mesmo teor ou semelhança entre os *benchmarks*. Mas é possível afirmar que é os melhores resultados pertencem as configurações

`CBMC + MiniSAT` e `ESBMC + MathSAT`.

As avaliações experimentais mostraram que a abordagem desenvolvida sempre encontra o mínimo global com as configurações propostas, otimizando os *benchmarks* com o OptCE. Em contrapartida, as outras técnicas de otimização avaliadas (algoritmo genético, enxame de partículas, pesquisa de padrões, programação não linear e recozimento simulado), não foram capazes de localizar o mínimo global em todas as funções, principalmente com as funções não convexas, apresentando soluções subótimas, o que reduz sua taxa de acerto. As técnicas de otimização tradicionais são normalmente presas por mínimos locais, não sendo capazes de garantir o mínimo global, embora que ainda apresentem tempos de otimização melhor que as abordagens propostas.

As abordagens específicas para funções convexas e funções semi-definidas positivas, obtiveram ótimas taxas de acerto, assim como no algoritmo generalizado, porém com tempo de otimização menor que o algoritmo generalizado. Importante ressaltar também que a diferença entre os tempos dos algoritmos específicos para o algoritmo generalizado, eram bem maiores sem as mudanças apresentadas na seção 4.1.4, atualmente essa diferença é menor, uma vez que as mudanças propostas resultaram maior efeito sobre o algoritmo generalizado.

O OptCE é uma ferramenta de otimização que modela uma gama de problemas de otimização restrita (convexa, não-linear e não-convexa) como um problema de verificação de modelo e analisa indutivamente contraexemplos, a fim de alcançar otimização global de funções, empregando verificação SAT ou SMT. O OptCE implementa os três diferentes algoritmos CEGIO (CEGIO-G, CEGIO-S e CEGIO-F), tem como *backend* duas ferramentas BMC (CBMC e ESBMC) e quatro solucionadores SAT/SMT (MiniSAT, Boolector, Z3 e MathSAT).

6.2 Propostas para Melhorias

Esta seção apresenta propostas que visam melhorar o desempenho dos resultados desta pesquisa, sobretudo melhorias em relação ao tempo de execução e avaliação experimental.

A avaliação experimental apresentada permite verificar que os algoritmos CEGIO e a ferramenta OptCE conseguem otimizar funções convexas e não convexas com boa taxa de acerto, mas se comparado com outras técnicas de otimização, alguns *benchmarks* obtiveram

tempos de execução bem elevados, portanto, busca-se reduzir o tempo de execução da técnica e ferramenta OptCE.

Ao longo da pesquisa tentativas foram feitas para paralelizar o processo de otimização, afim de distribuir a carga e reduzir o tempo, porém os resultados iniciais não foram animadores, uma vezes que grande parte dos *benchmarks* obtiveram tempo maior que o anterior. Acredita-se que o resultado foi contrário do objetivo devido a ferramenta ter gerado maior número de condições de verificações, na mesma proporção da quantidade de núcleos paralelizado. Ainda sim, é uma alternativa que deve ser futuramente explorada e melhor estudada afim de reduzir o tempo de otimização.

Outros verificadores e solucionadores poder ser adaptados para que sejam suportados seus contraexemplos, afim de avaliar seu desempenho. Foi o caso do `ESBMC + CVC` e `ESBMC + Yices` que foram despendido tempo para sua compilação e adaptação, mas por motivos de recursos limitados de tempo e computacional, não puderam ser concluídos e inseridos até o término deste trabalho.

Por fim, existe o interesse em aplicar técnicas de *machine learning* afim de definir a melhor configuração de verificador e solucionador conforme o problema, ou ainda aplicar essas técnicas para melhor definir condições de interrupções do algoritmo proposto, quando já tenha obtido a solução global.

Referências Bibliográficas

[1] SHOHAM, Y. Computer science and game theory. *Commun. ACM*, ACM, New York, NY, USA, p. 74–79, 2008.

[2] BHOYAR, D.; YADAV, U. Review of jamming attack using game theory. *International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, IEEE, p. 1–4, 2017.

[3] CHEN, T.; WU, B. Gateway selection based on game theory in internet of things. *International Conference on Electronics Technology (ICET)*, IEEE, p. 403–406, 2018.

[4] TEICH, J. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, IEEE, p. 1411–1430, 2012. ISSN 0018-9219.

[5] DERIGS, U. *Optimization and Operations Research – Volume I:*. EOLSS Publications, 2009. ISBN 9781905839483. Disponível em: <https://books.google.com.br/books?id=WWB8DAAAQBAJ>.

[6] BARTHOLOMEW, B.; MICHAEL, B. *Nonlinear Optimization with Engineering Applications*. Boston: Springer US, 2008. ISBN 978-0-387-78723-7.

[7] CAO, M.; YANG, Y. Two classes of conjugate gradient methods for large-scale unconstrained optimization. *2011 Fourth International Joint Conference on Computational Sciences and Optimization*, p. 37–40, 2011.

[8] LIU, H. A mixture conjugate gradient method for unconstrained optimization. *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*, p. 26–29, 2010.

[9] GARFINKEL, R.; NEMHAUSER, G. *Integer programming*. Michigan: Wiley, 1972. ISBN 9780471291954.

[10] GOLDBERG, D. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Boston: Addison-Wesley Publishing Company, 1989. ISBN 9780201157673.

[11] GNANAPRASANAMBIKAI, L.; MUNUSAMY, N. Survey of genetic algorithm effectiveness in intrusion detection. *2017 International Conference on Intelligent Computing and Control (I2C2)*, p. 1–5, 2017.

[12] VARGHESE, B. M.; RAJ, R. J. S. A survey on variants of genetic algorithm for scheduling workflow of tasks. *2016 Second International Conference on Science Technology Engineering and Management (ICONSTEM)*, p. 489–492, 2016.

[13] KEIKHA, M. M. Improved simulated annealing using momentum terms. *2011 Second International Conference on Intelligent Systems, Modelling and Simulation*, p. 44–48, 2011. ISSN 2166-0662.

[14] HAJEK, B. A tutorial survey of theory and applications of simulated annealing. *1985 24th IEEE Conference on Decision and Control*, p. 755–760, 1985.

[15] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, v. 38, n. 4, p. 957–974, 2012. ISSN 0098-5589.

[16] ESBMC. *Efficient SMT-Based Context-Bounded Model Checker*. 2019. Disponível em: http://esbmc.org. Acesso em: 5 março 2019.

[17] KROENING, D.; TAUTSCHNIG, M. Cbmc – c bounded model checker. *TACAS*, Springer Berlin Heidelberg, p. 389–391, 2014.

[18] BIERE, A.; CIMATTI, A.; CLARKE, E. M. Symbolic model checking without bdds. *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Springer-Verlag, Berlin, Heidelberg, p. 193–207, 1999.

[19] ARAUJO, R. F.; ALBUQUERQUE, H. F.; BESSA, I. V. de; CORDEIRO, L. C.; FILHO, J. E. C. Counterexample guided inductive optimization based on satisfiability modulo theories. *Science of Computer Programming*, Elsevier, v. 165, p. 3–23, 2018.

[20] ALBUQUERQUE, H. F.; ARAUJO, R. F.; BESSA, I. V. de; CORDEIRO, L. C.; FILHO, E. B. de L. Optce: A counterexample-guided inductive optimization solver. In: *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*. [s.n.], 2017. p. 125–141. Disponível em: <https://doi.org/10.1007/978-3-319-70848-5_9>.

[21] PATRIKSSON, M.; ANDREASSON, N.; EVGRAFOV, A. *An Introduction to Continuous Optimization: Foundations and Fundamental Algorithms*. Dover Publications, Incorporated, 2019. (Dover Books on Mathematics Series). ISBN 9780486802879. Disponível em: <https://books.google.com.br/books?id=C9JgrgEACAAJ>.

[22] GUENIN, B.; KÖNEMANN, J.; TUNÇEL, L. *A Gentle Introduction to Optimization*. Cambridge University Press, 2014. ISBN 9781107053441. Disponível em: <https://books.google.com.br/books?id=JC4DBAAAQBAJ>.

[23] CHONG, E.; ZAK, S. *An Introduction to Optimization*. Wiley, 2013. (Wiley Series in Discrete Mathematics and Optimization). ISBN 9781118515150. Disponível em: <https://books.google.com.br/books?id=iD5s0iKXHP8C>.

[24] GALPERIN, E. A. Problem-method classification in optimization and control. *Computers & Mathematics with Applications*, v. 21, p. 1 – 6, 1991. ISSN 0898-1221.

[25] ANNA, N.; NORA, D. Convex optimization. Cambridge University Press, 2004.

[26] TORN, A.; ZILINSKAS, A. *Global Optimization*. New York, NY, USA: Springer-Verlag New York, Inc., 1989. ISBN 0-387-50871-6.

[27] JOHNSON, J. M.; RAHMAT-SAMII, Y. Genetic algorithm optimization and its application to antenna design. *Proceedings of IEEE Antennas and Propagation Society International Symposium and URSI National Radio Science Meeting*, v. 1, p. 326–329, 1994.

[28] SHEIKH, R. H.; RAGHUWANSHI, M. M.; JAISWAL, A. N. Genetic algorithm based clustering: A survey. *2008 First International Conference on Emerging Trends in Engineering and Technology*, p. 314–319, 2008. ISSN 2157-0477.

[29] EBERHART, R. C.; SHI, Y. Comparing inertia weights and constriction factors in particle swarm optimization. *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, v. 1, p. 84–88, 2000.

[30] ANANTATHANAVIT, M.; MUNLIN, M. Radius particle swarm optimization. *2013 International Computer Science and Engineering Conference (ICSEC)*, p. 126–130, 2013.

[31] REN, Y.-H.; ZHANG, L.-W. The dual algorithm based on a class of nonlinear lagrangians for nonlinear programming. *2006 6th World Congress on Intelligent Control and Automation*, v. 1, p. 934–938, 2006.

[32] CLARKE JR., E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8.

[33] PNUELI, A.; RUAH, S.; ZUCK, L. Automatic deductive verification with invisible invariants. Springer, p. 82–97, 2001.

[34] BAIER, C.; KATOEN, J.-P. Principles of model checking (representation and mind series). The MIT Press, 2008.

[35] LEVESON, N. G.; TURNER, C. S. An investigation of the therac-25 accidents. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 26, n. 7, p. 18–41, 1993. ISSN 0018-9162. Disponível em: <https://doi.org/10.1109/MC.1993.274940>.

[36] BEYER, D. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, Springer, Berlin, Heidelberg, p. 887–904, 2016.

[37] CORDEIRO, L.; FISCHER, B.; MARQUES, S. J. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transaction of Software Engineering*, v. 38, n. 6, p. 50–55, 2002.

[38] CLARK, B.; ROBERTO, S.; SESHIA, S. A.; TINELLI, C. Satisfiability modulo theories. *Handbook of Satisfiability*, v. 185, n. 1, p. 825–885, 2009.

[39] ARMANDO, A.; MANTOVANI, J.; PLATANIA, L. Bounded model checking of software using smt solvers instead of sat solvers. *Proceedings of the 13th International Conference on Model Checking Software*, Springer-Verlag, Berlin, Heidelberg, p. 146–162, 2006.

[40] GANAI, M. K.; GUPTA, A. Accelerating high-level bounded model checking. *2006 IEEE/ACM International Conference on Computer Aided Design*, p. 794–801, 2006. ISSN 1092-3152.

[41] XU, L. Smt-based bounded model checking for real-time systems (short paper). *2008 The Eighth International Conference on Quality Software*, p. 120–125, 2008. ISSN 1550-6002.

[42] CORDEIRO, L. C.; FISCHER, B. Verifying multi-threaded software using smt-based context-bounded model checking. *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, ACM, p. 331–340, 2011.

[43] RAMALHO, M. Y. G.; FREITAS, M.; RODRIGUES, F. M. S.; MARQUES, H.; CORDEIRO, L. C.; FISCHER, B. Smt-based bounded model checking of C++ programs. *20th IEEE International Conference and Workshops on Engineering of Computer Based Systems, ECBS 2013, Scottsdale, AZ, USA, April 22-24, 2013*, IEEE Computer Society, p. 147–156, 2013.

[44] RAMANHO, M. Y. G.; MONTEIRO, F. R.; MORSE, J.; CORDEIRO, L. C.; FISCHER, B.; NICOLE, D. A. ESBMC 5.0: an industrial-strength C model checker. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, ACM, p. 888–891, 2018.

[45] PEREIRA, P. A.; ALBUQUERQUE, H. F.; MARQUES, H.; SILVA, I.; CARVALHO, C.; CORDEIRO, L. C.; SANTOS, V.; FERREIRA, R. Verifying CUDA programs using smt-based context-bounded model checking. *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, p. 1648–1653, 2016.

[46] PEREIRA, P. A.; ALBUQUERQUE, H. F.; SILVA, I.; MARQUES, H.; MONTEIRO, F. R.; FERREIRA, R.; CORDEIRO, L. C. Smt-based context-bounded model checking for CUDA programs. *Concurrency and Computation: Practice and Experience*, v. 29, n. 22, 2017.

[47] MONTEIRO, F. R. S.; ALVES, E. H. S.; SILVA, I.; ISMAIL, H.; CORDEIRO, L. C.; LIMA, E. B. F. Esbmc-gpu a context-bounded model checking tool to verify cuda programs. *Science of Computer Programming*, v. 152, p. 63 – 69, 2018. ISSN 0167-6423.

[48] MONTEIRO, F. R. S.; CORDEIRO, L. C.; BATISTA, E. L. F. Bounded model checking of C++ programs based on the qt framework. *IEEE 4th Global Conference on Consumer Electronics, GCCE 2015, Osaka, Japan, 27-30 October 2015*, IEEE, p. 179–180, 2015.

[49] GARCIA, M.; MONTEIRO, F. R. S.; C., C. L.; BATISTA, E. d. L. F. Esbmc QtOM : A bounded model checking tool to verify qt applications. *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, Springer, v. 9641, p. 97–103, 2016.

[50] MONTEIRO, F. R. S.; GARCIA, M.; CORDEIRO, L. C.; BATISTA, E. L. F. Bounded model checking of C++ programs based on the qt cross-platform framework. *Softw. Test., Verif. Reliab.*, v. 27, n. 3, 2017.

[51] TRINDADE, A.; CORDEIRO, L. C. Applying smt-based verification to hardware/software partitioning in embedded systems. *Design Autom. for Emb. Sys.*, v. 20, n. 1, p. 1–19, 2016. Disponível em: <https://doi.org/10.1007/s10617-015-9163-z>.

[52] TRINDADE, A. B.; DEGELO, R. D. F.; JUNIOR, E. G. D. S.; ISMAIL, H. I.; SILVA, H. C. D.; CORDEIRO, L. C. Multi-core model checking and maximum satisfiability applied to hardware-software partitioning. *IJES*, v. 9, n. 6, p. 570–582, 2017. Disponível em: <https://doi.org/10.1504/IJES.2017.10008947>.

[53] KROENING, D. *CBMC - Bounded Model Checker*. 2019. Disponível em: https://www.cprover.org/cbmc/. Acesso em: 5 março 2019.

[54] ROCHA, H.; BARRETO, R. S.; CORDEIRO, L. C.; NETO, A. D. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, Springer, v. 7321, p. 128–142, 2012.

[55] STOKES, J. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. No Starch Press, 2007. (Ars technica library). ISBN 9781593271046. Disponível em: <https://books.google.com.br/books?id=Q1zSIarI8xoC>.

[56] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, p. 0–1, 1985.

[57] BEYER, D.; KEREMOGLU, M. E. Cpachecker: A tool for configurable software verification. *International Conference on Computer Aided Verification*, p. 184–190, 2011.

[58] MORSE, J.; RAMALHO, M. Y. G.; CORDEIRO, L. C.; NICOLE, D.; FISCHER, B. ESBMC 1.22 - (competition contribution). *TACAS*, p. 405–407, 2014.

[59] INFINITY77. *Função Ursem3*. 2019. Disponível em: http://infinity77.net/global_optimization/test_functions_nd_U.html. Acesso em: 5 março 2019.

[60] JAMIL, M.; YANG, X. S. A literature survey of benchmark functions for global optimization problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 2013.

[61] ARAÚJO, R.; BESSA, I.; CORDEIRO, L. C.; BATISTA, E. C. F. SMT-based verification applied to non-convex optimization problems. *Proceedings of VI Brazilian Symposium on Computing Systems Engineering*, p. 1–8, 2016.

[62] BOYD, S.; VANDENBERGHE, L. *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. ISBN 0521833787.

[63] POWER. *Função Adjiman*. 2019. Disponível em: https://al-roomi.org/benchmarks/unconstrained/2-dimensions/113-adjiman-s-function. Acesso em: 5 março 2019.

[64] BENCHMARKFCNS. *Função Styblinski Tang*. 2019. Disponível em: http://benchmarkfcns.xyz/benchmarkfcns/styblinskitankfcn.html. Acesso em: 5 março 2019.

[65] POWER. *Função Chen's Bird*. 2019. Disponível em: https://al-roomi.org/benchmarks/unconstrained/2-dimensions/111-chen-s-bird-function. Acesso em: 5 março 2019.

[66] POWER. *Função Chen's V*. 2019. Disponível em: https://www.al-roomi.org/benchmarks/unconstrained/2-dimensions/110-chen-s-v-function. Acesso em: 5 março 2019.

[67] HUTTER, F.; BABIC, D.; HOOS, H. H.; HU, A. J. Boosting verification by automatic tuning of decision procedures. *FMCAD*, p. 27–34, 2007.

[68] SHILAJA, C.; RAVI, K. Solution for optimum power flow with and without fact devices: A survey. *International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)*, p. 783–789, 2017.

[69] THE MATHWORKS, INC. Matlab optimization toolbox user's guide. 2016.

[70] BRUMMAYER, R.; BIERE, A. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, p. 174–177, 2009.