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

Novembro de 2018

Higo Ferreira Albuquerque

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

Qualificação 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

Higo Ferreira Albuquerque

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

Banca Examinadora

Prof. D.Sc. Eddie Batista de Lima Filho – Presidente e Co-orientador
TP Vision

Prof. Ph.D. Lucas Carvalho Cordeiro – Orientador
Escola de Ciência da Computação – UoM

Prof. M.Sc Iury Valente de Bessa – Co-orientador
Departamento de Eletrônica e Computação – UFAM

Prof. D.Sc. Raimundo da Silva Barreto
Instituto de Computação – UFAM

Prof. D.Sc. Renan Landau Paiva de Medeiros
Departamento de Eletrônica e Computação – UFAM

Manaus – Amazonas
Novembro de 2018

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, em virtude disso espero atender às expectativas com 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, amizade 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 as contribuições para o desenvolvimento dos algoritmos de otimização indutiva guiada por contraexemplos (Counterexample Guided Inductive Optimization - CEGIO) e a ferramenta de otimização OptCE. 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 desenvolvidos 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, fazer uso dos 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 10 funções (convexas e não convexas) onde os resultados obtidos comprovam sua capacidade de otimização, tendo melhor taxa de acerto se comparadas com as técnincas comparadas.

Palavras-chave: Verificadores de Programas, Otimização, 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. Our 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 (SAT) and Satisfiability Modulo Theories (SMT).

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 10 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

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*

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 fórmulas e funções matemáticas, o que torna possível aplicar técnicas de otimização para maximizar ou minimizar a função definida, com o objetivo de encontrar uma solução ótima para o problema,ou seja, é encontrado o melhor desempenho para o problema dadas as possibilidades existentes.

A computação e a otimização têm cooperado mutuamente com grande evolução para ambas as áreas. Avanços importantes da Ciência da Computação estão baseados na teoria da otimização, como por exemplo, problemas de planejamento (teoria dos jogos [2]) e problemas de alocação de recursos (*e.g.*, *hardware/software co-design* [3]). Em contrapartida, a Ciência da Computação tem papel decisivo nos mais recentes estudos de otimização, entregando algoritmos eficientes e ferramentas para o controle de modelos matemáticos e análise dos resultados [4].

Existem diversos tipos de problemas de otimização, e para as diferentes classes de problemas, muitas técnicas de otimização 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 gradientes, ou problemas de Programação Linear (Linear Programming - LP) podendo ser resolvidas por técnicas de programação linear como o método simplex, ou ainda funções não convexas tendo muitas abordagens de otimização heurísticas

como algoritmo genético (Genetic Algorithm - GA) e recozimento simulado (SA).

Como os problemas de otimização têm suas representações mediante de modelos matemáticos, os métodos formais podem ser uma importante ferramenta para o 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* [5, 6, 7, 8]. O principal valor entregue pelos métodos formais é prover um meio para examinar simbolicamente todo o espaço de estados de um sistema, e estabelecer uma propriedade de segurança que seja verdadeira considerando todas as entradas possíveis [5, 6, 7, 8].

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 a verificação formal Verificação de Modelos Limitados (*Bounded Model Checking* - BMC), para ajudar a estabelecer os algoritmos de otimização indutiva guiada por contraexemplos (CEGIO) e o desenvolvimento de uma ferramenta de otimização (OptCE).

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, ultilizando 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 documento de qualificação tem como objetivo apresentar os algoritmos de otmização indutiva guiada por contraexemplos CEGIO, assim também a ferramenta de otimização baseada nos algoritmos CEGIO, o OptCE. Durante o trabalho é apresentado à fundamentação teórica referente ao assunto; detalhamento dos algoritmos desenvolvidos CEGIO; detalhamento e avaliação experimental da ferramenta OptCE; como também por fim são apresentadas as conclusões e propostas de melhorias do trabalho.

1.1 Descrição do Problema

Esta pesquisa visa desenvolver uma abordagem de otimização, utilizando técnicas de verificação de modelos limitada de forma automatizada.

Os problemas de otimização são descritos mediante modelos matemáticos, podendo fazer alusão a um sistema físico real. As técnicas de otimização buscam a melhor performance para um sistema, conforme sua modelagem matemática e as possibilidades existentes. Enquanto que os métodos formais apresentam a capacidade de garantir o comportamento de um sistema computacional, inserindo propriedades e seguindo uma abordagem rigorosa a partir de noções de especificação. A especificação é um modelo de um sistema que contém uma descrição de seu comportamento desejado, podendo ser totalmente abstrata ou ser mais operacional [9, 10, 11].

Diate disso, busca-se otimizar funções matemáticas empregando métodos formais a partir de ferramentas de verificação de programas, reunindo todos os agentes em uma única ferramenta para o propósito da otimização.

1.2 Objetivos

Esta pesquisa tem como objetivo estabelecer o uso da verificação de modelos limitados a fim de solucionar problemas de otimização formulados por funções convexas e não-convexas. O processo para localizar o mínimo global é realizado mediante o uso de verificadores de programas que aplicam SAT e SMT. Busca-se encapsular a metodologia em uma ferramenta de otimização de interação console Linux. Para atingir este objetivo, deverão ser contemplados os seguintes objetivos específicos:

- Desenvolver a metodologia para otimizar funções matemáticas usando a verificação de modelos limitados;
- Desenvolver uma ferramenta de otimização indutiva guiada por contraexemplos SAT e SMT;
- Validar a capacidade da ferramenta OptCE 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, desenvolvemos algoritmos e ferramentas correspondentes e os avaliamos usando *benchmarks* clássicos de otmização. Para cada

otimização, a ferramenta implementa os algoritmos desenvolvidos CEGIO usando os dados de entrada referente o *benchmark*, para em seguida executar a verificação. A metodologia permite a convergência para a solução final, percorrendo por sucessivas verificações dos arquivos de especificação gerados em cada verificação *failed*. Sua evolução é de forma interativa, encontrando soluções sub-ótimas com precisão cada vez maior e mais próximo do mínimo global, até a localização do mínimo global 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:

- **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. Propriedades são inseridas juntamente com dados que descrevem o sistema a ser otimizado, somado a um valor de inicialização que é gerado aleatóriamente.
- **Verificação -** A última etapa faz a checagem das propriedades existentes do arquivo de especificação e gera um contraexemplo. O contraexemplo resultante indica *success* caso nenhuma propriedade seja violada, ou *failed* caso propriedades tenham sido violadas.

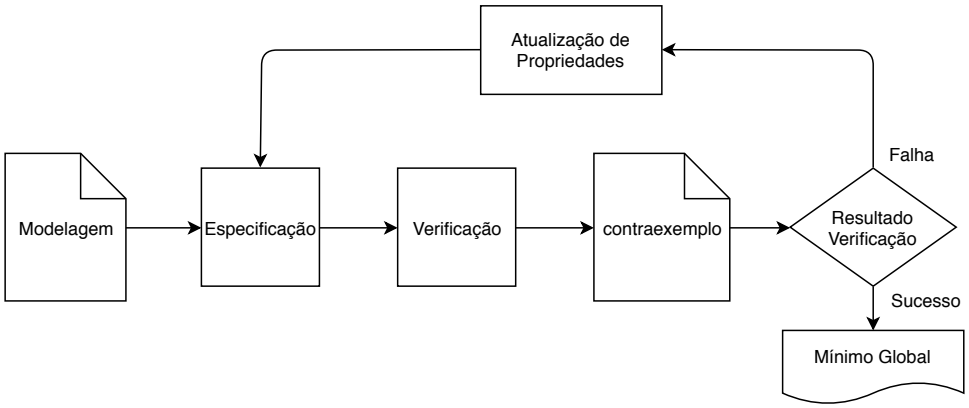

Figura 1.1: Metodologia proposta para solução.

Conforme os algoritmos CEGIO foram estabelecidos; 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 as propriedades; caso o resultado tenha sido *failed*, o verificador encontrou um candidato a mínimo global que é menor que o candidato anterior. O sistema atualiza as propriedades e usa o novo candidato a mínimo global encontrado no contraexemplo para realizar uma nova especificação. Uma nova verificação neste novo arquivo de especificação é realizada, e o cíclo acontece até que encontramos o mínimo global da função.

1.4 Contribuições

A principal contribuição desta qualificação é o desenvolvimento da metodologia de otimização usando contraexemplos SAT e SMT juntamente com os algoritmos CEGIO e o desenvolvimento de uma ferramenta de otimização guiada por contraexemplos usando solucionadores SAT e SMT.

Primeiramente foi contribuído para o desenvolvimento dos algoritmos CEGIO, uma abordagem de otimização que faz uso de contraexemplos SAT e SMT. Os algoritmos são especificados a partir da modelagem do usuário, de forma que possam ser verificados para localizar mínimos globais. Três algoritmos seguindo a abordagem foram desenvolvidos, um para funções sem classificação específica e outros dois algoritmos especificamente para funções positivas e convexas.

Em segundo lugar foi concebido a ferramenta OptCE que implementa os algoritmos CEGIO desenvolvidos na primeira etapa, permitindo assim melhor usabilidade para realização da otimização de funções, e facilitando a configuração de verificadores e solucionadores. Nesta segunda parte foram inseridas também o uso de mais um verificador de programas, o CBMC, onde foi avaliado mediante experimentos juntamente com todas as combinações possíveis de verificadores e solucionadores. Os resultados foram avaliados e comparados com os resultados de outras técnicas de otimização.

1.5 Organização da Qualificação

O trabalho está organizado conforme os seguintes tópicos:

- **Capítulo 2** - Descreve a fundamentação teórica sobre a otimização e verificação for-

mal. 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 sobre 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, são ressaltados três itens: otimização, verificação e erros de precisão. Primeiramente é abordado sobre a otimização, apresentando sobre a furmulação matemática para problemas de otmização; a diferença básica entre mínimo local e mínimo global em uma função; e algumas técnincas de otmização. Em seguida é explanado sobre a parte de verificação, primeiro de uma forma mais geral sobre os métodos de verificação e depois de forma mais específica sobre os verificadores de programas, em especial sobre o ESBMC e CBMC. Por fim uma subseção fala sobre erros e precisão numérica, é abordada sobre a 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 condições, como 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 com um conjunto de variáveis de decisão, e por meio de cálculos, a otimização encontra os pontos extremos desta função, *i.e.*,máximos e mínimos que podem ser assumidos em um dado intervalo de uma função objetiva. Em resumo, a otimização é o processo para encontrar as variáveis que fornecem o valor máximo ou mínimo de uma função [12].

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

e ressaltam sua importância em valores financeiros, tendo como exemplo no Canadá, a cidade de Montreal precisava alocar melhor sua estrutura de transporte. Envolvendo muitos motoristas de ônibus e de metrô, vendedores de bilhetes e guardas, uma otimização permitiu economizar cerca de quatro milhões de dólares canadenses por ano [13]. No departamento de polícia de São Francisco, a otimização permitiu planejar 20% mais rápido o problema de alocação de veículos policiais, economizando assim 11 milhões de dólares por ano [13]. Para a companhia aérea *United Airlines*, a otimização permitiu resolver o problema de agendamento de tripulação economizando seis milhões de dólares ao ano. [13].

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, muitas vezes há um conflito entre a resolução possível e o seu realismo [13].

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 indicam o ciclo deste processo.

Inicialmente temos 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 [14].

No diagrama da figura Fig 2.1, o ciclo começa com a etápa de **Análise**, onde tem o trabalho de abstrair detalhes irrelevantes e focar nos principais elementos, afim de construir o algoritmo, é uma parte importante para a otimização bem-sucedida. Em seguida tem-se a transição do algoritmo para a implementação computacional, que muitos chamam por **Métodos Numéricos** por geralmente ser proveniente de métodos numéricos. Questões como precisão em computadores digitais e implementações eficientes de técnicas de inversão de matrizes são levados em consideração. Por ventura o usuário necessita ter familiaridade para ajustar os parâmetros de controle dos solucionadores.

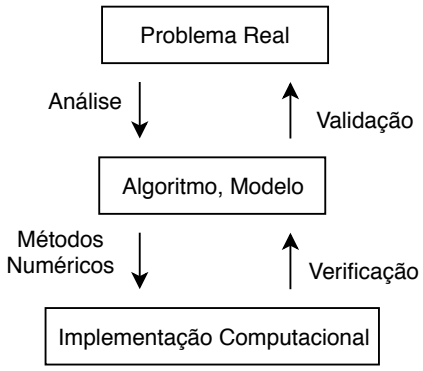

Figura 2.1: Ciclo do processo de otimização

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

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 [1]:

- **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 podem ser classificado dependendo da natureza de sua funções objetiva e restrições. Como exemplo podendo ser lineares, não lineares, quadráticas, polinomiais.

Conforme o domínio e restrições da função de custo f, o espaço da busca da otimização pode ser pequeno ou grande, o que influencia diretamente no desempenho dos algoritmos de otimização propostos. Veremos mais adiante nas próximas seções porque a otimização irrestrita não é interessante para a proposta deste trabalho.

Dado o tempo e recursos de memória disponíveis, a natureza da função de custo e a resolução da solução que deseja-se obter pode tornar complexa a otimização, sendo incapaz a resolução de alguns problemas para algumas técnicas [15]

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 que a chamamos de função objetiva ou função de custo. O vetor x é um vetor de n-variáveis independentes, *i.e.*, $x = [x_1, x_2, ..., x_n]^T \in R^n$. 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 [1, 16].

O problema de otimização descrito pode ser visto como um problema de decisão, cujo o seu objetivo é encontrar o "melhor" vetor 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 de minimizador de f em Ω. Existe a possibilidade que tenha mais de um minimizador, sendo suficiente encontrar apenas um para obter a solução ótima.

Como visto anteriormente no problema da construção, 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, podemos considerar somente os problemas de minimização sem perda por generalidade.

O problema descrito na Eq 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 considerássemos $\Omega = R^n$, seria um problema de otimização sem restrição.

Na próxima subseção 2.1.4 é realizada uma breve discussão mais esclarecedora sobre as diferenças entre mínimos locais e mínimos globais, mas a seguir temos respectivamente as definições matemáticas para o minimizador local e global, que são as variáveis de decisão que definem os mínimos local e global em uma função.

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 $\varepsilon > 0$ tal que $f(x) \geq f(x^*)$ para todos $x \in \Omega$, onde $||x - x^*|| < \varepsilon$.*

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.

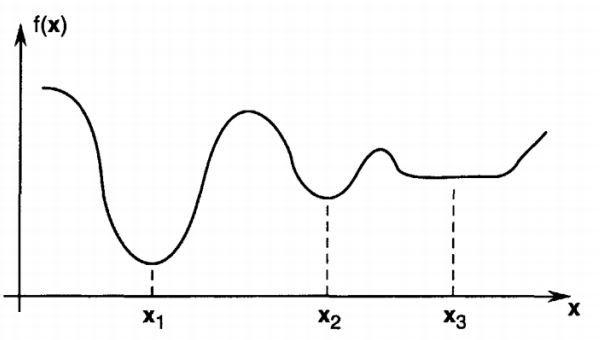

Figura 2.2: Exemplos de minimizadores: x_1 é estritamente um minimizador global; x_2 é estritamente um minimizador local; x_3 é minimizador local não estritamente. [1]

A figura Fig 2.2 mostra gráficamente 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 $f(x) = -1$. Se escrevermos *arg* $min_{x \in \Omega}$, então tratamos Ω como o domínio de f. Por exemplo, para a função f acima, arg $\min_{x \geq 0} f(x) = 0$. Em geral, podemos pensar em *arg* $min_{x \in \Omega}$ $f(x)$ como o minimizador global de f em Ω (assumindo que existe e é único) [1].

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 encontrar o mínimo global. Por outro lado à otimização de problemas não convexos são muito complexos, funções de custo não convexas apresentam vários pontos de inflexão, como também podem direcionar o algoritmo de otimização para uma solução sub-ótima em um mínimo local, em geral os algoritmos apontam para soluções aproximadas.

Alguns algoritmos geralmente apontam para soluções aproximadas [1].

Recentemente, vários estudos relacionadas à otimização global surgiram, contudo as soluções propostas são muito específicas para uma classe particular de problemas [17]. 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 [18].

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 resolver 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, como uma companhia aérea que busca melhores rotas para seus aviões e minimizar o custo de sua tripulação [14].

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, limitados suprimentos, pessoas e equipamentos [14].

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 [14].

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 é descrito sobre algumas técnicas de otimização que foram usadas em comparação com os resultados já obtidos.

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 de objeto [19].

Os otimizadores de GA procuram por soluções globais em domínios de função multimodal de alta dimensão. Os GAs diferem das técnicas tradicionais porque operam em um grupo, ou população, de soluções experimentais em paralelo, elas operam em uma codificação dos parâmetros da função em vez dos parâmetros diretamente e usam operadores simples e estocásticos para explorar o domínio da solução [19].

Um simples GA deve ser capaz de executar cinco tarefas básicas: codificar os parâmetros da solução na forma de cromossomos, inicializar uma população de ponto inicial, avaliar como também atribuir valores de aptidão a indivíduos da população, realizar a reprodução através da seleção ponderada de indivíduos a população, e realizar recombinação e mutação para produzir membros da próxima geração [19].

Enxame de partículas

Semelhante ao GA, o Enxame de partículas (Particle Swarm - PSO) se inicia com uma população de soluções aleatórias k, a diferença ocorre, pois cada solução candidata tem uma atribuição aleatoriamente de velocidade, e chamamos essas soluções de partículas [20]. 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, os movimentos

tem influência da posição atual conhecida, guiando-as para as posições mais conhecidas do espaço de busca, sempre atualizando as melhores posições encontradas [20].

Pesquisa de Padrões

O método de otimização Pesquisa de Padrões (*Pattern Searc*), busca padrões heurísticos que precisam apenas retornar o valor da função objetiva $f(x)$ para algum valor de entrada x [14]. São usados em problemas onde não é possível conhecer a primeira ou segunda derivada da função objetivo em um problema de programação não linear irrestrito [14]. 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 controlado pelo processo de resfriamento, obtendo propriedades desejáveis como dureza [14]. Seu desenvolvimento se deu antes dos algoritmos genéticos e foi gradativamente substituído por eles em muitas aplicações [14].

Dentre o processo de otimização, quando o parâmetro “Temperatura” é elevado, muitas movimentações aleatória são toleráveis, à proporção que a “Temperatura” é reduzida, menos movimentos aleatórios são permitidos, até que a solução fixa do estado final é dito “congelado”. Enquanto a “temperatura” é alta, o algoritmo faz uma ampla amostragem do espaço de soluções gradativamente com o esfriamento da “temperatura”, 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 [14].

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

Programação não Linear

A programação não linear (NonLinear Programming - NLP) é composto por uma função objetiva, restrições gerais e limites das variáveis de decisão assim como na programação linear, mas o que caracteriza o NLP é quando sua função objetiva não é linear, ou possui restrições não lineares, isso indica ser um problema não linear [14]. 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 natureza [14]. Existem muitos algoritmos que resolvem problemas de programação não lineares, mas cada um deles é adaptado a um específico problema de otimização [21].

2.2 Métodos de Verificação

Os sistemas computacionais têm evoluído de forma exponencial, nessa competição por melhores resultados, surgem sistemas cada vez mais complexos em menos tempo de desenvolvimento. Agilizar o desenvolvimento de um programa poderá criar problemas com inconsistências no código, e por displicência permitir a manufatura de um produto 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 [22].

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 observada 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 [22].

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 [23].

A importância de se ter um sistema correto era tanta, que o especialista poderia investir o tempo que fosse necessário para realização dessa tarefa [22]. 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 [22]. A verificação dedutiva tem importância para ciência da computação, pois influência no desenvolvimento dos programas. Aplica-se em sistemas de estado infinito, e nenhum limite é imposto sobre a quantidade de tempo ou memória a fim de encontrar a prova [22]. 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 pesquisa exaustiva no espaço de estados do sistema para determinar se uma especificação é verdadeira ou falsa [22].

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 eficiente, que reduz o tempo, minimiza o esforço e aumenta a cobertura da verificação [24].

Os métodos formais tem o objetivo de estabelecer um rigor matemático na verificação de sistemas [24], 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) [24]. 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 ser parte da educação de cada engenheiro e cientista de software, assim como o ramo apropriado da matemática aplicada é uma parte necessária da educação de todos os outros engenheiros [24]"

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 a missão *Ariane 5*, missão *Mars Pathfinder* [24] e acidentes com a máquina de radiação *Therac-25* [25].

2.2.2 Verificação de Modelos

A vericação de modelos atua em sistemas de estados finito concorrentes, garante 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 and J. Sifakis introduziram o uso da lógica temporal [24]. A verificação de um único modelo que satisfaz uma fórmula é mais fácil 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 de três partes: modelagem; especificação e verificação [24].

- **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 [24].

- **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 [24].

- **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 [24]. A análise do contraexemplo pode requerer a modificação do programa e reaplicar o algoritmo *model checking* [24]. O contraexemplo pode ser útil para detectar outros dois tipos de problemas, uma modelagem incorreta do sistema ou, uma especificação incorreta [24]. 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 [24], neste caso, talvez seja preciso realizar ajustes no modelo.

O maior desafio dessa abordagem são as explosões do espaço de estados, isso ocorre quando durante uma verificação, 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 [24].

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 pesquisa 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 [24].

2.2.3 Verificação de Modelos Limitados - BMC

Verificação de Modelos Limitadas (BMC), é uma técnica importante que vem apresentando bons resultados, já foi aplicada com sucesso para verificar o software embarcado e pode descobrir erros em projetos reais [26]. 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 [27].

O BMC gera VCs 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 [28]. Para o BMC, um conjnto de fórulas $\{p_1, p_2, ..., p_n\}$ é dito ser satisfatível se

houver alguma estrutura Λ na qual todas as suas fórmulas componentes sejam verdadeiras, *i.e.*, $\{p_1, p_2, ..., p_n\}$ é SAT *iff* $\Lambda \models p_1 \wedge \Lambda \models p_2 ... \wedge \Lambda \models p_n$.

Um dos graves problemas enfrentados pelo BMC, herdado a verificação de modelos, é a explosão do estado 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 sobrecarrega o sistema computacional usado no processo. Outra dificuldade dessa técnica é comprovar a validade das VCs, o que implica no desempenho na verificação dos programas [29, 28].

O técninca BMC é baseada na Satisfiabilidade Booleana (SAT) [30] ou Teoria do Módulo de Satisfatibilidade (SMT) [31]. O BMC baseado em SAT foi introduzido como uma técnica complementar aos diagramas de decisão binária para auxiliar no problema da explosão do estado [8], este podem 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 [32, 33, 34].

Em SMT, predicados de várias teóricas não são codificadas por variáveis proposicionais como em SAT, mas permanecem na fórmula do problema. Essas teorías são tratadas de forma dedicadas, dessa maneira no BMC baseado em SMT, ϕ é uma formula livre de quantificador em um subconjunto de lógica de primeira ordem, que é então verificada quanto à satisfatibilidade por um solucionador de SMT [27].

O uso da Verificação de Modelos Limitados para esta pesquisa tem influência significativa para os bons resultados alcançados quando a taxa de acerto, que é mostrado mais adiante na subseção 4.2. Isso ocorre pois a técninca BMC é capaz de percorrer os caminhos possíveis de execução de um código durante uma verificação. Na metodologia estabelecida que será apresentada no trabalho 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.

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-se programas que se desenvolvem correta-

mente 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 os verificadores de programas, em especial ferramentas BMC, limitavam-se apenas as teorias de funções não interpretadas, *arrays* e aritmética linear [32, 33], a abordagem do BMC baseada em SMT não suportava 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. [29]. A evolução dos verificadores permitiu verificar programas *mult-threads*, fator importante conciderando 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 é 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, descrevem suas competências e limitações, 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 esritos em C e $C++$ [35, 36, 37]. Este é capaz de verificar programas simples ou *multi-tarefas*, programas com *arrays*, ponteiros, *structs*, *unions*, alocação de memória, tratar com aritimética de ponto fixo e flutuante. Ele é capaz de argumentar sobre *overflows*, segurança de ponteiro, vazamento de momória, atomicidade e violação de ordens, *deadlocks* local e global, concorrência de dados, *asserts* especificados pelo usuário.

Dentro do ESBMC, os programas são modelados como sistemas de transição de estados $M = (S, R, s_0)$, extradido 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 $s \in S$ consiste no valor do contador do

programa *pc* e nos valores de todas as variáveis do programa. 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 contexto limite 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.

Derivamos 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 contra-exemplo 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 $\gamma = (si, s_{i+1})$ 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. ψ_k^π é uma fórmula livre 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 podemos extrair os valores das variáveis de programa para construir um contra-exemplo.

O contra-exemplo de uma propriedade ϕ é uma seqüência 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 insatisfável, pode-se concluir que nenhum estado de erro é alcançável na profundidade menor ou igual a k ou ao longo de π. Finalmente, podemos 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 RT. Ele para quando encontra um *bug* ou explorou sistematicamente todos os possíveis intercalamentos de RT.

O ESBMC também foi estendido para verificar programas escritos em CUDA [38, 39,

40] e Qt [41, 42, 43], facilitando desta forma o seu uso em problemas práticos da indústria.

2.3.2 CBMC

CBMC é um verificador *Bounded Model Checker* para programas *C* e $C++$. Como afirmam os autores, Daniel Kroening, ele suporta *C*89, *C*99, grande parte do C11 e a maioria das extensões dos compiladores fornecidos pelo 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 [7].

Entre suas vantagens e aplicações, é capaz de atuar sobre programas concorrentes, verificando programas C executados por múltiplas-threads, pode encontrar e mostrar 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 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 Normal Conjuntiva (Conjunctive Normal Form - CNF) que descreve o programa a ser checado, juntamente com as propriedades especificadas no programa [7].

O CBMC procura por uma atribuição que satisfaça tanto o problema quanto a negação da propriedade, para mostrar um contra-exemplo [44], 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 [7].

Claramente que a formalização de um modelo não é tão simples e requer lidar com vários pequenos problemas, como *loops*, modelo aritmético etc, esse é um dos grandes desafios neste trabalho, formalizar modelos relacionados a otimização que possam ser usandos pelos verificadores de programas, para obter uma solução provida por pela técnica de verificação BMC [7].

2.4 Erro e Precisão

O espaço de armazenamento de informações em um computador não é infinita, o sistema computacional possui uma forma para representar os números em uma quantidade fixa de *bits*. O programador tem a opção de escolher diferentes formas de representação ou diferentes tipos de dados. Os tipos de dados podem definir além do número de *bits*, podem armazenar informações mais intrínsecas, como definir se o formato é ponto fixo ou flutuante.

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 deve ser escolhido para armazenar as partes fracionária e inteira de um número.

Para a representação em ponto fixo o número tem sua representação definida por três partes: sinal, parte inteira e parte fracional. Digamos que precisamos 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 e 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, ocorre ao tentar escrever dados em um *buffer*, e ultrapassa os limites do *buffer* sobrescrevendo a memória adjacente.

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 S (interpretado como mais ou menos), um expoente inteiro exato E, e uma representação exata binária mantissa M como:

$$S \times M \times b^{E-e}, \tag{2.3}$$

Nessa expressão b é a base de na equação, 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 e é o viés do expoente, uma constante de número inteiro fixado para qualquer representação, *e.g.*, 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* ($e = 1023$).

Para representação de 32 *bitS*, o menor número normalizado positivo é $2^{-126} \approx 1.18 \times 10^{-38}$, que é muito menos do que em um representação de ponto fixo. Além disso, o espaçamento entre os números de ponto flutuante não é uniforme, à medida que nos afastamos 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 [45]. Quando 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, *i.e.*, eles têm valores exatos na forma de Eq. (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 1.19×10^{-7}, enquanto `double` tem cerca de 2.22×10^{-16}.

A precisão da máquina é a precisão fracionária no qual os números em ponto flutuante são representados, correspondendo a uma mudança de 1 no *bit* menos significativo da mantissa. Quase qualquer operação aritmética entre números de ponto flutuante deve ser considerada como introdução de 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 no hardware em que o programa é executado. Muitos algoritmos numéricos calculam aproximações discretas para alguns dados contínuos quantidade. Nestes casos, existe um parâmetro ajustável; qualquer cálculo prático é feito com uma escolha finita, mas suficientemente de grande parâmetro. A discrepância entre a resposta verdadeira e a resposta obtida em um cálculo prático é chamada de erro *truncation*. O erro de truncamento persistiria mesmo em um computador perfeito que tivesse uma representação infinitamente precisa e nenhum erro de arredondamento.

Como regra geral, não há muito que um 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 de otimização; a diferença entre mínimos locais e mínimos globais e ainda um apanhado sobre as técnincas de otimização tradicioanais que são usadas para comparar com a ferramenta OptCE. Na sequência foi apresentado sobre verificação forma, em especial sobre a verificação de modelos limitados; também é abordado sobre verificadores de programas BMC, em especial as ferramentas (ESBMC e CBMC) usadas para geração de contraexemplo 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

Counterexample Guided Inductive Optimization based on SMT

Este capítulo apresenta a metodologia de otimização usando contraexemplos SAT e SMT juntamente com os algoritmos CEGIO. O objetivo desta seção será: esclarecer o funcionamento da otimização guiada indutiva por contraexemplo, os obstáculos e as medidas adotadas afim resolver os impedimentos encontrados; também são detalhados 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 checar a modelagem matemática de um problema, assim como retorna `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 usado para definir restrições sobre variáveis (deterministicas e não deterministas) e a diretiva `ASSERT` que verifica uma determinada propriedade.

Essas duas instruções, estão presentes em verificadores do modelo $C/C++$ (*e.g.*, CBMC [7], CPAChecker [46], and ESBMC [47]), podendo ser aplicado para verificar restrições específicas em problemas de otimização, conforme descrito pela Eq. (2.1).

O processo de verificação usa funções intrínsecas disponíveis nos verificadores para

resolver o problema de otimização, como por exemplo no ESBMC (*e.g.*, *__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 é reduzido a cada interação.

O valor mínimo de uma função candidata é dada por (f_c), definida pela Eq. (2.1), então a diretiva `ASSERT` pode ser usada para verificar a satisfabilidade de l_{otimo} como

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

Sendo assim, se $\neg l_{otimo}$ não for satisfeito, então f_c é o mínimo da função, caso contrários, existe alguma região no espaço de estado definida pelas restrições do problema que viola a propriedade.

Nesta etapa surgem alguns problemas devido à aritimética de precisão finita dos verificadores, onde ocorrem um truncamento durante a restrição na linha 8 do algoritmo CEGIO-G. Temos exemplos de problemas usando aritmética de ponto fixo como o Boolector e ESBMC, mas também com a aritimética de ponto flutuante como nos solucionadores Z3 e MathSAT usados pelo ESBMC. Sendo assim existem problemas com erro de precisão.

Na tentativa de contornar este problema, l_{otimo} é modificado para $l_{subOtimo}$ dado pela Eq. (3.2)

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

onde δ deve ser suficientemente alto para reduzir os efeitos da representação numérica e erros do truncamento nos cálculos. Contrapondo-se a isso, se $\neg l_{subOtimo}$ não for satisfeito, então f_c não é a função mínima, mas estará a uma distância limitada por δ do valor mínimo. δ também pode ser usado com o objetivo de determinar a melhoria mínima em cada iteração na função de custo.

A idéia é usar a habilidade dos verificadores para verificar a satisfação de uma determinada propriedade e depois retornar um contraexemplo que contenha o rastreamento do erro. Através de sucessivas verificações de satisfabilidade do literal $\neg l_{subOtimo}$, pode-se orientar o processo de verificação para resolver o problema de otimização dado pela Eq. (2.1).

3.2 Exemplo de Otimização

Para descrever o processo da Otimização Intutiva Guiada por controexemplo, é usado o exemplo abordado do journal *Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories*. A função *Ursem*03 é um problema de otimização não convexo. 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 da equação Ursem03 é apresentada na Eq. (3.3), a Fig 3.1 apresenta o gráfico da função.

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

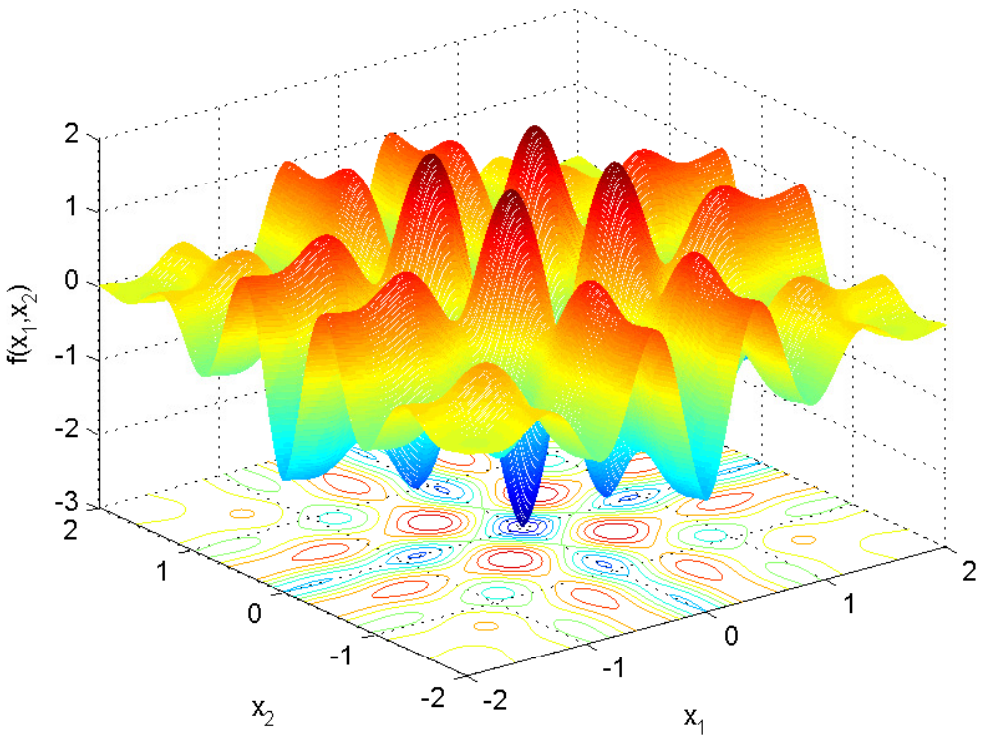

Figura 3.1: Função Ursem03

Para realizar o processo de otimização desta função é realizado os passos: modelagem, especificação e verificação.

Modelagem

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

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

O problema de otimização apresentado na Eq. (3.4) pertence à função *Ursem03* Eq. (3.3), onde as restrições são definidas por intervalos semi-fechados 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 $x_2 \geq 0$ leva 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 Eq. (3.4) pode ser melhor reescrito como a Eq. (3.5) inserindo novas restrições. Os limites são escolhidos com base no estudo descrito por Jamil e Yang [48], 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 (3.5),é realizada a codificação onde as variáveis de decisão são declaradas como variáveis não determinísticas, restritas usando a 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 Fig. 3.2 mostra o respectivo código em *C* para modelagem da Eq. (3.5).

```
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 Eq. (3.5).

Especificação

Na etapa seguinte, a especificação, é onde o comportamento do sistema e as propriedades a serem verificadas são descritas. No caso deste exemplo, a função *Ursem03*, o resultado da etapa de especificação é o programa *C* mostrado na Fig. 3.3, que é checado pelos programas de verificação.

As variáveis de decisão são declaradas como tipo inteiro e sua inicialização varia conforme a precisão atribuida a variável p, descrita na Eq. (3.6), que é ajustada iterativamente quando o contraexemplo é gerado pelo solucionador. Se as variáveis de decisão fossem declaradas em ponto flutuante não-determinístico como no código da Fig. 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, aplicando a discrição e reduzindo a exploração 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, tecnicamente, 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.

```
1  #include "math2.h"
2  #define p 1 //precision variable
3  int nondet_int();
4  float nondet_float();
5  int main() {
6    float f_c = 100;  //candidate value of objective function
7    int lim_inf_x1 = -2*p;
8    int lim_sup_x1 = 2*p;
9    int lim_inf_x2 = -2*p;
10   int lim_sup_x2 = 2*p;
11   int X1 = nondet_int();
12   int X2 = nondet_int();
13   float x1 = float nondet_float();
14   float x2 = float nondet_float();
15   __ESBMC_assume( (X1>=lim_inf_x1) && (X1<=lim_sup_x1) );
16   __ESBMC_assume( (X2>=lim_inf_x2) && (X2<=lim_sup_x2) );
17   __ESBMC_assume( x1 = (float) X1/p );
18   __ESBMC_assume( x2 = (float) X2/p );
19   float fobj;
20   fobj= -sin2(2.2*pi*x1-pi/2)*(2-abs2(x1))(3-abs2(x1))/4
21   -sin2(2.2*pi*x2-pi/2)*(2-abs2(x2))(3-abs2(x2))/4;
22   //constrain to exclude fobj>f_c
23   __ESBMC_assume( fobj < f_c );
24   assert( fobj > f_c );
25   return 0;
26 }
```

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

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

A variável inteira p é declarada e iterativamente incrementada, de forma que ε representa a quantidade de casas decimais relacionadas às variáveis de decisão, onde ε pertence ao intervalo $0 \leq \varepsilon \leq \eta$, onde η é o valor da precisão da otimização desejada. 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 Fig. 3.3) é inserida para podar esses vários candidatos que são maiores que o mínimo atual.

Além desta restrição, uma propriedade deve ser especificada para garantir a convergência para o ponto mínimo em cada iteração, isso é feito por meio de uma assertiva, que verifica se $\neg l_{subOptimo}$ fornecido na Eq. (3.2) é satisfatível para cada valor de $f(\mathbf{x})$.

Quando o procedimento de verificação para $\neg l_{subOptimo}$ não é satisfeito, ou seja, quando houver qualquer $\mathbf{x}^{(i)}$ para o qual $f(\mathbf{x}^{(i)}) \leq f_c$, é gerado um contraexemplo que mostra as var-

iáveis de decisáo $\mathbf{x}^{(i)}$, convergindo iterativamente para a solução ótima. Fig. 3.3 mostra a especificação inicial para o problema de otimização dado por Eq. (3.5). O valor inicial do candidato da função objetivo pode ser inicializado aleatoriamente.

Para o exemplo mostrado na Fig. 3.3, $f_c^{(0)}$ é inicializado arbitrariamente para 100, mas o algoritmo de otimização funciona para qualquer estado inicial e para $i > 0$, $f_c^{(i+1)} = f(\mathbf{x}^{*,(i)}) - \delta$, conforme especificado na linha 13 do algoritmo CEGIO-G.

Verificação

Por fim a etapa de verificação no processo, o programa mostrado na Fig. 3.3 é checado pelo verificador, e um contraexemplo é retornado com um conjunto de variáveis de decisão $\mathbf{x}$, cujo o valor da função objetivo obtido com essa 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 específicada em p.

Neste exemplo, o contraexemplo retorna as seguintes variáveis de decisão: $x_1 = 2$ e $x_2 = 0$. Essa informação é funcamental para a convergência do algoritmo, pois assim ele calcula a nova função candidata mínima $f_c(x)$, que é $f(2,0) = -1,5$ menor que 100, sendo assim a verificação é refeita com novo valor de $f_c^{(i+1)}$. Este processso ocorre até que uma verificação seja bem sucedida e a precissão 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 inicio 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 algotimo.

3.3 Algoritmos CEGIO

O algoritmo de otimização intuitiva 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 do número de algarismos significativo da solução.

A abordagem geralmente possui elevado tempo de execução maior que outras técnicas tradicionais, porém, taxa de erro é menor que outros métodos existentes, uma vez que se baseia em um procedimento de verificação completo dos espaços 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 *et al.* [49], que apresentava uma solução de ponto fixo com precisão ajustável.

O algoritmo CEGIO-G necessita de quatro entradas: a função a ser otimizada ou função de custo $f(x)$; definição do conjunto de restrições Ω; a precisão que deseja-se obter a solução η; e o valor de mínimo de compensação para a função candidata δ. 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 possui dois loops aninhados, onde o loop externo (`for`) está relacionado à precisão desejada e o loop interno (`while`) está relacionado ao processo de verificação. O algoritmo `CEGIO-G` usa a manipulação de precisão do número do ponto fixo para garantir a convergência de otimização.

Depois da inicialização e declaração da variável (linhas 1-2) o domínio da pesquisa Ω^{ε} é especificado na linha 4, que é definida pelos limites inferior e superior das variáveis auxiliares $\mathbf{X}$. Eles são declarados como variáveis inteiras não deterministas; Caso fossem declarados como variáveis ponto flutuante não deterministas, a pesquisa no espaço de estados seria muito elevada. Assim, os limites de Ω^{ε} são dados conforme a Eq. (3.7).

A variável ε (linha 5) é usada para manipular as variáveis auxiliares X e obter o valor das variáveis de decisão x, *i.e.*, $\mathbf{x} = \mathbf{X}/10^{\varepsilon}$. Na sequência, estas variáveis de decisão são usadas para compor a descrição da função de custo $f(x)$. A variável ε define o número de casas decimais de

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

input : Uma função de custo $f(\mathbf{x})$, o conjunto de restrições Ω, o número de casas décimais das variáveis de decisão η, valor mínimo de compensação δ

output: O vetor com as variáveis de decisão ótima $\mathbf{x}^*$, e o valor ótimo da função $f(\mathbf{x}^*)$

1 *Inicializa* $f_c^{(0)}$ *aleatoriamente e* $i = 0$
2 *Declara as variáveis auxiliares* $\boldsymbol{X}$ *como variáveis inteiras não determinísticas*
3 **for** $\varepsilon = 0 \rightarrow \eta;\ \varepsilon \in \mathbb{Z}$ **do**
4 *Define limites para* $\boldsymbol{X}$ *com a diretiva* `ASSUME`*, de tal modo que* $\boldsymbol{X} \in \Omega^{\varepsilon}$
5 *Descreve um modelo para função de custo* $f(\boldsymbol{x})$*, onde* $\boldsymbol{x} = \boldsymbol{X}/10^{\varepsilon}$
6 *Faz a variável auxiliar Check = TRUE*
7 **while** *Check* **do**
8 *Restringe* $f(\boldsymbol{x}^{(i)}) < f_c^{(i)}$ *com a diretiva* `ASSUME`
9 *Verifica a satisfabilidade de* $\neg l_{subOtimo}$ *dada pela Eq.* (3.2) *com a diretiva* `ASSERT`
10 **if** $\neg l_{subOtimo}$ *é satisfatível* **then**
11 *Atualiza* $\boldsymbol{x}^* = \boldsymbol{x}^{(i)}$ *e* $f(\boldsymbol{x}^*) = f(\boldsymbol{x}^{(i)})$ *baseado no contraexemplo*
12 *Faz* $i = i + 1$
13 *Faz* $f_c^{(i)} = f(\boldsymbol{x}^*) - \delta$
14 **end**
15 **else**
16 Check = `FALSE`
17 **end**
18 **end**
19 **end**
20 **return** $\boldsymbol{x}^*$ *e* $f(\boldsymbol{x}^*)$

x, por exemplo, quando ε é nulo obtence soluções inteiras, quando $\varepsilon = 1$ são obtidas soluções com uma casa decimal.

$$lim\{\Omega^{\varepsilon}\} = lim\{\Omega\} \times 10^{\varepsilon} \tag{3.7}$$

Inicialmente, ε é igual a zero e deve ser atualizado no final de cada iteração do *loop* externo (`for`), de modo que ele aumenta o domínio das variáveis de decisão em uma casa decimal na próxima iteração do *loop*. Quanto ao *loop* interno (`while`), a cada iteração verifica-se a satisfação de $\neg l_{subOtimo}$ dada pela Eq. (3.2). O algoritmo possui uma restrição (linha 8 - $f(\mathbf{x}^{(i)}) \leq f_c$) que limita o espaço de estados, pois não há necessidade de verificar valores maior do que o valor mínimo candidato já encontrado, uma vez que o algoritmo busca o valor mínimo da função de custo. Com isso o espaço de busca é reconfigurado para a precisão i-ésima e emprega os resultados anteriores do processo de otimização.

A etapa de verificação é realizada (linhas 9-10), onde a função candidata $f_c^{(i)}$ é analisada por meio da checagem de satisfabilidade de $\neg l_{subOtimo}$ dada pela Eq. (3.2). Se houver um $f(\mathbf{x}) \leq$

$f_c^{(i)}$ que viole a diretiva `ASSERT`, então o vetor de variáveis de decião e o valor mínimo da função de custo são atualizados com base no contraexemplo, e a função candidata é atualizada $f_c^{(i)} = f(\mathbf{x}^*) - \delta$ (linha 13), após o incremento de i o algoritmo retorna para remodelar novamente o espaço de estados (linha 8).

Se a diretiva `ASSERT` não for violada, o último candidato f_c é o valor mínimo considerando a variável de precisão ε; assim, ε é incrementado de 1 para η no *loop* `for`, adicionando uma casa decimal para a solução de otimização e o *loop* externo (`for`) é repetido. O algoritmo conclui o *loop* externo atingindo o limite definido pelo parâmetro de entrada η; assim, retorna o vetor ótimo de variáveis de decisão com η casas decimal 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 adequado para funções semi-definida positiva, como $f(\mathbf{x}) \geq 0$. O ideal é que o usuário tenha conhecimento prévio da função, saiba que a função seja semi-definida positiva para o uso do algoritmo. O algoritmo `CEGIO-S` é ligeiramente modificado para lidar com classes particulares de funções, o que torna mais rápido e simples, pois não realiza buscas no espaço de estado negativo.

A grande diferença do CEGIO-S para o CEGIO-G é a condição na linha 8, não sendo necessário gerar novas verificações caso essa condição não seja realizada, uma vez que a solução já possui o limite mínimo $f(\mathbf{x}^*) = 0$. A precisão dos valores obtidos para o mínimo condidato da função é incrementado a cada passagem *loop* externo (`for`) (linhas 4-29).

O *loop* interno (`while`) (linhas 12-15) é responsável por gerar múltiplas *VCs* através da diretiva `ASSERT`, usando o intervalo entre f_m e $f_c^{(i)}$. Este *loop* gera $\alpha + 1$ *VCs* através da etapa definida por γ na linha 7.

As mudanças permitiram que o algoritmo CEGIO-S converja mais rápido que Alg. 1 CEGIO-G dentre as funções semi-definida positiva, 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.

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

input : Uma função de custo $f(\mathbf{x})$, o conjunto de restrições Ω, o número de casas décimais das variáveis de decisão η, valor mínimo de compensação δ, e uma taxa de aprendizagem α

output: O vetor com as variáveis de decisão ótima $\mathbf{x}^*$, e o valor ótimo da função $f(\mathbf{x}^*)$

Inicializa $f_m = 0$
Inicializa $f_c^{(0)}$ *aleatoriamente e* $i = 0$
Declara as variáveis auxiliares $\boldsymbol{X}$ *como variáveis inteiras não determinísticas*
for $\varepsilon = 0 \rightarrow \eta;\ \varepsilon \in \mathbb{Z}$ **do**
    Define limites para $\boldsymbol{X}$ *com a diretiva* `ASSUME`*, de tal modo que* $\boldsymbol{X} \in \Omega^{\varepsilon}$
    Descreve um modelo para função de custo $f(\boldsymbol{x})$*, onde* $\boldsymbol{x} = \boldsymbol{X}/10^{\varepsilon}$
    Declara $\gamma = (f(\boldsymbol{x}^{(i-1)}) - f_m)/\alpha$
    if $(f_c^{(i)} - f_m > 10^{-5})$ **then**
        Faz a variável auxiliar Check = `TRUE`
        while *Check* **do**
            Restringe $f(\boldsymbol{x}^{(i)}) < f_c^{(i-1)}$ *com a diretiva* `ASSUME`
            while $(f_m \leq f_c^{(i)})$ **do**
                Verifica a satisfabilidade de $l_{subOtimo}$ *dada pela Eq.* (3.1) *para cada* f_m*, com a diretiva* `ASSERT`
                Faz $f_m = f_m + \gamma$
            end
            if $\neg l_{subOtimo}$ *é satisfeito* **then**
                Atualiza $\boldsymbol{x}^* = \boldsymbol{x}^{(i)}$ *e* $f(\boldsymbol{x}^*) = f(\boldsymbol{x}^{(i)})$ *baseado no contraexemplo*
                Faz $i = i + 1$
                Faz $f_c^{(i)} = f(\boldsymbol{x}^*) - \delta$
            end
            else
                Check = `FALSE`
            end
        end
    end
    else
        break
    end
end
return $\boldsymbol{x}^*$ *e* $f(\boldsymbol{x}^*)$

3.3.3 Algoritmo Rápido CEGIO-F

O algoritmo CEGIO-F é uma evolução dos algoritmos apresentados anteriormente, destina-se a funções convexas. Assim como nos demais algotimos, o algoritmo CEGIO-F evolui aumentando a precisão da variáveis de decisão, *i.e.*, na primeira execução do *loop* `for`, o mínimo global obtido inicialmente é inteiro desde que a precisão ε seja 0, que aqui atribuimos pela notação $x^{*,0}$. O algoritmo também possui variáveis auxiliares X que são usadas para delimitar o espaçao de busca Ω^{ε}, que são usados para definir as variáveis de decisão x. Mas sua grande diferença se dá por estabelecer um novo domínio de busca a cada resultado de verificação *failed*, a linha 13 neste algotimo atualiza os limites do conjunto Ω^{ε} antes da precisão ε.

Em cada execução do *loop* `for`, a solução é ótima conforme a precisão ε do momento. Um novo domínio de pesquisa $\Omega^{\varepsilon} \subset \Omega^{\eta}$ é obtido de uma etapa do CEGIO aplicando $\Omega^{\varepsilon-1}$, considerando que a precisão ε seja maior que zero neste estágio. O novo conjnto de espaço Ω^{ε} é definindo como seguinte: $\Omega^{\varepsilon} = \Omega^{\eta} \cap [x^{*,\varepsilon-1} - p, x^{*,\varepsilon-1} + p]$, onde p é dado pela Eq. (3.6) e $x^{*,\varepsilon-1}$ é a solução com $\varepsilon - 1$ casas decimais.

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

input : A função de custo $f(\mathbf{x})$, o conjunto de restrições Ω, e uma 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* $f_c^{(0)}$ *aleatoriamente e* $i = 0$
2 *Declara as variáveis auxiliares* $\boldsymbol{X}$ *como variáveis inteiras não determinísticas*
3 **for** $\varepsilon = 0 \rightarrow \eta;\ \varepsilon \in \mathbb{Z}$ **do**
4 *Define limites para* $\boldsymbol{X}$ *com diretivas* `ASSUME`*, de tal modo que* $\boldsymbol{X} \in \Omega^{\varepsilon}$
5 *Descreve um modelo para função de custo* $f(\boldsymbol{x})$*, onde* $\boldsymbol{x} = \boldsymbol{X}/10^{\varepsilon}$
6 *Faz a variável auxiliar Check =* `TRUE`
7 **while** *Check* **do**
8 *Restringe* $f(\boldsymbol{x}^{(i)}) < f_c^{(i)}$ *com a diretiva* `ASSUME`
9 *Verifica a satisfabilidade de* $\neg l_{subOtimo}$ *dada pela Eq.* (3.2) *com a diretiva* `ASSERT`
10 **if** $\neg l_{subOtimo}$ *é satisfeito* **then**
11 *Atualiza* $\boldsymbol{x}^* = \boldsymbol{x}^{(i)}$ *e* $f(\boldsymbol{x}^*) = f(\boldsymbol{x}^{(i)})$ *baseado no contraexemplo*
12 *Faz* $i = i + 1$
13 *Faz* $f_c^{(i)} = f(\boldsymbol{x}^*) - \delta$
14 **end**
15 **else**
16 Check = `FALSE`
17 **end**
18 **end**
19 *Atualiza os limites do conjunto* Ω^{ε}
20 **end**
21 **return** $\boldsymbol{x}^*$ *e* $f(\boldsymbol{x}^*)$

3.4 Resumo

Este captítulo apresentou os algoritmos de otimização desenvolvidos CEGIO. Foram desenvolvidos três algoritmos seguindo a abordagem de análise de contraexemplo para otimização de funções, um algoritmo para funções genéricas (CEGIO-G) e outros dois algoritmos especificos, para funções convexas (CEGIO-F) e funções positivas (CEGIO-S). Os algoritmos desenvolvidos somado a aboradagem de otimização guiada por contraexemplo são usados para a criação da ferramenta de otimização OptCE, que é apresentado na próxima seção 4.

Capítulo 4

OptCE: A Counterexample-Guided Inductive Optimization Solver

Neste capítulo é apresentado a ferramenta de otmização OptCE. Esta ferramenta faz uso da metodologia de otimização guiado por contraexmplo e implementa os algoritmos CEGIO. 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 otmização.

4.1 OptCE: Solucionador Indutivo Guiado a Contraexmplo

OptCE pode ser considerado como um front-end para checagem de modelos que processam programas *C* através dos algoritmos CEGIOs, 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 mostrado a seguir. Em resumo, o OptCE é baseado na abordagem dos algoritmos CEGIOs, 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* (*cf. Sec. 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 *verifier*, *solver*, *type de algoritmo* e *precision*. Na Figura 4.1, α representa o número de casas decimais desejadas para a solução, que é indicada pelo usuário. A variável α na ferramenta OptCE equivale a precisão desejável p descrito na Equação 3.6. Com base nas entradas fornecidas, o OptCE gera um arquivo de especificação em ANSI-C (*cf. Figura 3.3*), denominado como `min_<function>.c`.

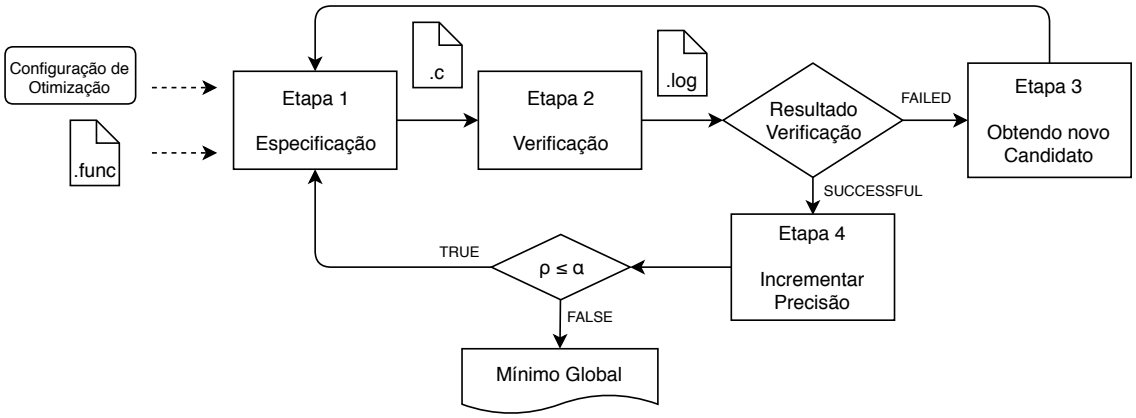

Figura 4.1: Uma visão geral da arquitetura proposta do OptCE.

Durante a primeira execução da etapa 1, variável ρ, é usada para armazenar a evolução da precisão ao longo do processo de otimização. Em sua inicialização é atribuído 0, o que indica um otimização somente com soluções de precisão inteira. Além disso, um candidato mínimo arbitrário é considerado, cujo é o real valor de inicialização do algoritmo, e 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, *i.e.*, o arquivo ANSI-C com a especificação da função é checado por um verificador, cujo a 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, conse-

quentemente, 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 precisão α, e 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, *i.e.*, o candidato mínimo global da última iteração será o valor de inicialização. Em seguida, obtém-se o novo candidato a valor mínimo (*i.e.*, extraído e calculado a partir do contra-exemplo) 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 o precisão (ρ) estabelecida até o momento. Quando o resultado da verificação é "verification successfully", o OptCE prossegue para o Passo 4.

Na Etapa 4, ρ é incrementado em 1 unidade, seguida de uma checagem que avalia se a precisão é menor ou igual à precisão desejada α (indicada pelo usuário). Se ρ for maior que α (a condição $\rho \leq \alpha$ 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 ρ, *it.e*, o algoritmo retorna a etapa 1 e gera um novo arquivo de especificação para ser verificado.

4.1.2 Arquivo de Entrada

O arquivo de entrada atual 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.

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

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

Figura 4.2: Linguagem da entrada do programa para OptCE.

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

As restrições de um problema de otimização como por exemplo a do *benchmark adjiman*, 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*.

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 um chamada via linha de comando. As seguintes configurações são suportadas:

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

- **Configuração do Algoritmo:** escolhe entre os algoritmos proposto, onde a flag `--generalized` implementa o algoritimo CEGIO-G (*cf. Sec. 3.3.1*), usada quando não existe conhecimento prévio sobre a função, a flag `--positive` implementa o algoritmo CEGIO-S (*cf. Sec. 3.3.2*), usada quando a função é semi-definida positiva, e a flag `--convex` implementa o algoritmo CEGIO-F (*cf. Sec. 3.3.3*), 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 objetiva(`--library=`*name-library*);

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

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

4.1.4 Resolvendo um problema 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. Figura 4.4 mostra todas chamadas possíveis do OptCE com arquivo de entrada e conjunto de propriedades. Aqui, empregamos a função *adjiman* para ilustrar o uso do OptCE, considerando a entrada arquivo mostrado na Figura 4.3.

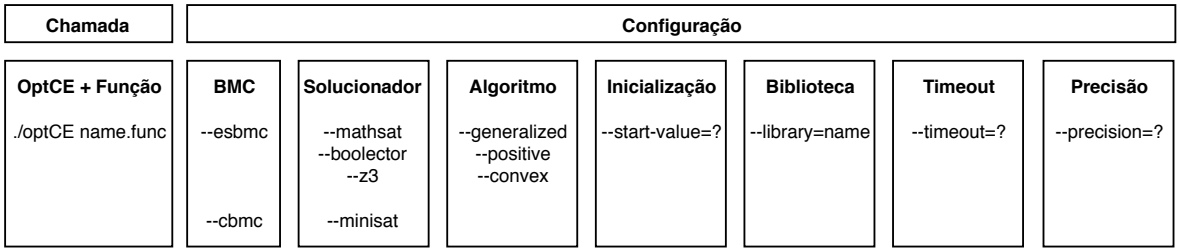

Figura 4.4: Opções de configurações do OptCE.

Atualmente, o OptCE suporta dois verificadores: CBMC [7] e ESBMC [6]. 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 back-end SMT, não foi possível verificar todos os benchmarks da suíte usada neste artigo. Em relação à ESBMC, o usuário pode escolher entre solucionadores Z3 (`--z3`) ou Boolector (`--boolector`); no entanto, não foi avaliado outros SMT 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 objetivas, com um grande conjunto de benchmarks. Essa abordagem é semelhante ao trabalho realizado por Hutter *et al.* [50], 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-definida positiva e `--generalized`, para funções sobre as quais não temos 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 4.4, 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. Observamos que as variações em relação aos valores de inicialização não influenciam significativamente os tempos de convergência, uma vez que a 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 [49, 51, 52].

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 e quanto mais simples forem, *i.e.*, 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 nossa própria implementação, que foi incluído usando a flag `--library` (*optCE adjiman.func* `--library=math2.h`). Esta biblioteca contém uma implementação aprimorada a *math.h* original, que inclui pré e pós-condições para garantir que (dado) um predicado detém antes e depois da execução de uma (dada) função matemática, respectivamente.

Nossas funções matemáticas em *math2.h* têm o mesmo nome dos elementos correspondentes na biblioteca ANSI-C, exceto que nós adicionamos o caracter 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, *i.e.*, 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.

4.2 Avaliação Experimental

Esta subseção decrevem a configuração, execução e resultos dos experiemntos realizados para avaliar a ferramenta OptCE.

4.2.1 Objetivos dos Experimentos

RQ1 (Validação) O OptCE é capaz de encontrar os mínimos globais em uma funções de otimização?

RQ2 (Configuração) A escolha das configurações entre ferramentas BMC e solucionadores influencia os resultados da otimização?

RQ3 (Performace) Quais diferenças do OptCE em comparação com as técnicas tradicionais de otimização?

4.2.2 Descrição dos *benchmarks*

Com o objetivo de avaliar a ferramenta OptCE, foram escolhidas 10 funções (funções convexas e não convexas) de problemas clássicos de otimização para execução dos experimentos [48]. 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 4.1 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-definida positiva *Booth*, *Himmelblau* e *Leon*. Por fim, foram usadas as funções *Zettl*, *Rotated Ellipse* e *Sum Square* para avaliar a *flag* `--convex`, que implementa o algoritmo CEGIO-F.

Tabela 4.1: Suíte de Testes

#	Benchmark	Domain	Global Minimum
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	$-500 \leq x_i \leq 500$	$f(0,0) = 0$

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) [53]. Os tempos apresentados nas tabelas a seguir estão relacionados ao tempo médio de 20 execuções consecutivas para cada *benchmark*.

Os experimentos foram executados em um computador equipado com CPU Intel Core $i7-4790$ de 3.60 GHz, 16 GB de RAM e Linux OS Ubuntu 14.10, configurado para obter mínimos globais com 3 casas decimais.

4.2.3 Resultados dos experimentos

Os resultados experimentais são apresentados em quatro tabelas. As tabelas 4.2, 4.3 e 4.4 mostram aos *benchmarks* usados para avaliar as implementações dos algoritmos CEGIO definidas pelas respectivas *flags* `--generalized`, `--positive` e `--convex`. A Tabela 4.5 relata uma comparação entre a ferramenta OptCE v1.0 e outras técnicas tradicionais de otimização.

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

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

Cada coluna da Tabela 4.2 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 v3.1.0 com MathSAT v.5.3.13, Z3 v4.5.0, e os solucionadores Boolector v2.2.0, respectivamente, e a coluna 5 está relacionada à configuração CBMC v4.5 com o MiniSat v2.2.0.

Tabela 4.2: Tempos de execução para `--generalized` (CEGIO-G), em segundos.

#	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT
1	**1068**	105192	3387	5344
2	**4130**	80481	5003	8509
3	**443**	37778	2027	2438
4	468	387	**190**	1143
5	7	1244	4016	**2**
6	12	14205	6217	**4**
7	5	2443	212	**2**
8	13	753	389	**9**
9	18	4171	4438	**13**
10	3	72	39	**2**

O mínimo global é encontrado em todos os *benchmarks* da suíte, considerando todas as combinações ferramentas BMC e solucionadores. Os tempos de otimização da Tabela 4.2 variaram significativamente, o que dificulta definir uma melhor configuração; entretanto, conforme o gráfico da Figura 4.5, o melhor tempo de otimização é a configuração ESBMC + MathSAT,

foi 2.8 vezes mais rápido que o segundo melhor, CBMC + MiniSAT, enquanto a configuração ESBMC + Z3 apresentou o maior tempo de execução, 40 vezes maior que o melhor caso.

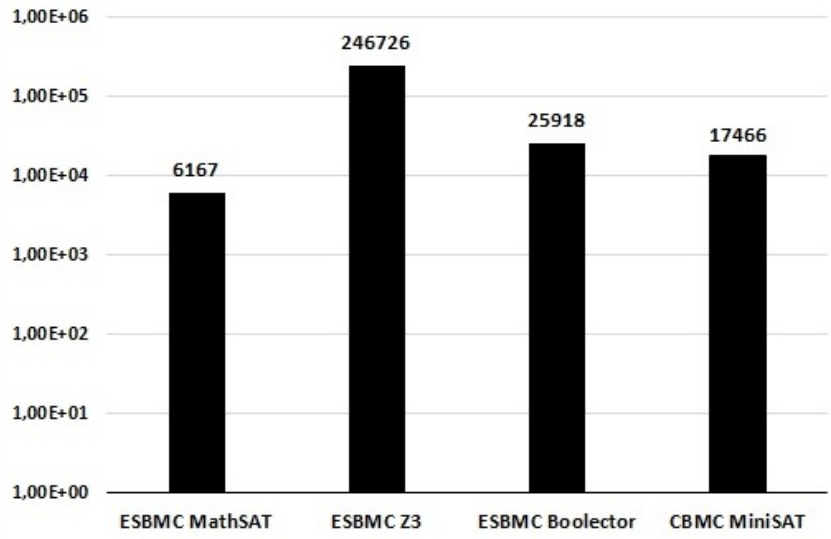

Figura 4.5: Histograma do tempo total de otimização para a suíte de testes, escala logarítmica.

Embora o CBMC + MiniSAT tenha sido o segundo melhor desempenho conforme mostra a Tabela 4.2, esta configuração teve o melhor tempo em 60% dos *benchmarks*. Os *benchmarks* $\#1-4$ são não-convexos e apresentam longos períodos quanto pesquisa de mínimos globais, considerando todas as configurações possíveis. Os *benchmarks* $\#5-7$ são funções semi-definida positiva, enquanto $\#8-10$ são convexas.

As combinações ESBMC + Math-SAT e CBMC + MiniSAT apresentaram os melhores resultados em comparação com as outras configurações do OptCE, dado que Boolector não suporta aritmética de ponto flutuante [54]. Em particular, o solucionador MathSAT que obteve os melhores resultados, suporta aritmética de ponto fixo e flutuante. Os resultados evidenciam que a otimização com aritmética de ponto flutuante é significativamente melhor que a de ponto fixo. Para a configuração com a *flag* `--generalized`, são sugeridas as configurações ESBMC + MathSAT e CBMC + MiniSAT.

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

A Tabela 4.3 apresenta os resultados para a *flag* `--positive`, que é adapitada para funções semi-definida positiva. Foram usados somente os *benchmarks* $\#5-7$ para este experimento, pois fazem uso de módulos com potências par, onde visualmente podemos garantir estas funções não podem atingir um mínimo global negativo.

A Tabela 4.3 faz uma comparação entre as *flags* `--positive` e `--generalized` nesta classe de problemas. Os resultados da otimização usando a *flag* `--positive` mostrou resultados significativamente melhores que usando a `--generalized` em todas configurações, confirmando as expectativas, pois o espaço de busca é menor, já que ignora parte negativa.

Tabela 4.3: Tempos de execução para `--positive` (CEGIO-S), em segundos.

#	`--positive`				`--generalized`			
	ESBMC			CBMC	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT	MathSAT	Z3	Boolector	MiniSAT
5	3	**<1**	1	3	7	1244	4016	2
6	4	**1**	**1**	2	12	14205	6217	4
7	3	**<1**	1	2	5	2443	212	2

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

A implementação do algoritmo CEGIO-F é atribuída com o *flag* `--convex`. Para avaliar seu desempenho, foram utilizados os *benchmarks* #8 – 10 por serem convexas. Os resultados são apresentados na Tabela 4.4. Os tempos de otimização utilizando o algoritmo específico para essa classe de função foram consideravelmente menores que os tempos apresentados pela algoritmo generalizado. Isso acontece porque nesse 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 4.4: Tempos de execução para `--convex` (CEGIO-F), em segundos.

#	`--convex`				`--generalized`			
	ESBMC			CBMC	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT	MathSAT	Z3	Boolector	MiniSAT
8	15	6	21	**5**	13	753	389	9
9	14	**3**	19	5	18	4171	4438	13
10	3	**1**	2	2	3	72	39	2

OptCE x Outras Técnicas

Na Tabela 4.5 são apresentados os melhores resultados de otimização dos *benchmarks*, considerando as abordagens implementadas na ferramenta OptCE. Consta também na tabela 4.5 os resultados de otimização com outras técnicas. A coluna *configuration* mostra as combinações de tipos de algoritmos (identificadas com as iniciais das *flags*, "G" para `--generalized`,

Tabela 4.5: Tempos de execução comparativo entre técnicas tradicionais de otimização e os melhores resultados obtidos com o OptCE, em segundos

#	OptCE			GA		ParSwarm		PatSearch		SA		NLP	
	Configuration	R%	T	R%	T	R%	T	R%	T	R%	T	R%	T
1	G + ESBMC + MathSAT	100	1068	29.1	1	22.2	3	16	4	0.4	1	4.8	9
2	G + ESBMC + MathSAT	100	4130	100	9	9.8	1	96.7	3	88.5	2	28.4	2
3	G + ESBMC + MathSAT	100	443	68.1	9	47.8	1	51.8	3	99.5	1	35.8	2
4	G + ESBMC + Boolector	100	190	95.7	9	53.9	1	98.8	3	74.4	1	62.5	2
5	P + ESBMC + Z3	100	< 1	100	10	100	2	100	6	93.5	1	100	2
6	P + ESBMC + Z3	100	1	42.4	9	43.9	1	26	3	21	1	35	2
7	P + ESBMC + Z3	100	< 1	84.4	1	80.3	2	1	7	24.3	1	100	4
8	C + CBMC + MiniSAT	100	5	100	9	48.1	1	99.8	4	26.4	1	100	3
9	C + ESBMC + Z3	100	3	100	9	71.5	1	100	4	96.9	1	100	2
10	C + ESBMC + Z3	100	1	100	9	100	2	100	7	99.8	1	100	2

"P" para `--positive` e "C" para `convex`), ferramentas BMC e solucionadores. 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 mil vezes com as técnicas tradicionais, utilizando o MATLAB, e 20 vezes com o CEGIO, utilizando o OptCE. As repetição foram realizadas para garantir a convergência da taxa de acertos em todos os algoritmos.

Os experimentos mostram que o OptCE geralmente leva mais tempo do que outras técnicas, a fim de localizar os mínimos globais, no entanto, sua taxa de acerto é sempre maior nos experimentos, sendo de 100% para esse conjunto de referência. Os resultados de 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 e, em muitos casos, ficam presos em mínimos locais, resultando em soluções sub-ótimas.

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 `--convex` especificamente para funções convexas, ou `--positive` para otimizar funções semi-definida positiva, como funções de distância ou potên-

cia. Mas assim como as técnicas verificação, existem restrições quanto ao tempo de verificação e o consumo de memória.

Quando analisamos o desempenho das funções não convexas, as técnicas tradicionas ficam presas em mínimos locais e retornam soluções sub-ótimas, o que reduz sua taxa de acerto. O OptCE demanda um tempo 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 menos verificações e menos estados a serem considerados.

4.3 Resumo

Este capítulo apresentou o desenvolvimento 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; um exemplo de otimização com a ferramenta; e a avaliação experimental. Os resultados dos experimentos foram satisfatórios, mostrando também que a implementação dos algoritmos CEGIO no OptCE é eficiente, mostrando que a ferramenta é capaz de otimizar funções matemáticas convexas e não convexas. O OptCE mostrou-se vatajoso em comparação com outras técnicas de otimização conforme foi apresentado na subseção 4.2.3 de 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 5

Conclusões

5.1 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 descrita, juntamente com suas funcionalidades implementadas, capacidades e limitações. Também é apresentado uma avaliação experimental do OptCE juntamente as mesmas técnincas de otimização usadas para o desenvolvimento dos algoritmos CEGIO.

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 ESBMC + MathSAT e CBMC + MiniSAT.

As avaliações experiemntais mostraram que a abordagem desenvlvida sempre encontra o mínimo global com as configurações propostas, otimizanado 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 simulated annealing), foram capaz de localizar o mínimo global geralmente nas funções convexas, para as funções não convexas houveram falhas, apresentando soluções sub-ótimas, 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 abaixos dos algoritmos propostos.

As abordagens específicas para funções convexas e funções semi-definida positiva, obtiveram ótima taxa de acerto assim como no algoritmo generalizado, mas a grande diferença é em relação ao tempo de otimização, que apresentou-se ser muito próximo das tempos encontrados nas técnincas tradicionais.

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).

5.2 Propostas para Melhorias

Está seção apresenta propostas que visão melhorar o desempenho dos resultados desta pesquisa, sobretudo melhorias em relação ao temnpo de execução e avaliação experimental.

A avaliação experiemntal 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 técnincas de otimização, alguns *benchmarks* obtiveram tempos de execução bem superior. Portanto, busca-se reduzir o tempo de execução da ferramenta OptCE, de uma forma prática e linear conforme a quantidade de núcleos do ambeinte de experimentos. O plano é segmentar o espaço de estados da função de custo e realizar a verificação de forma paralela em cada parte segmentada. Após o termino das verificações os valores candidatos a mínimo global seriam comparados para determinar o real mínimo global da função.

Outra melhoria que busca-se fazer é melhorar a avaliação experimental elevando o número de *benchmarks* aplicado ao OptCE.

Referências Bibliográficas

[1] CHONG, E. K.; ZAK, S. H. *An introduction to optimization*. [S.l.]: John Wiley & Sons, 2013.

[2] SHOHAM, Y. Computer science and game theory. *Commun. ACM*, ACM, New York, NY, USA, v. 51, n. 8, p. 74–79, aug 2008. ISSN 0001-0782.

[3] TEICH, J. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, v. 100, n. Special Centennial Issue, p. 1411–1430, May 2012. ISSN 0018-9219.

[4] DERIGS, U. *Optimization and Operations Research – Volume I:*. [S.l.]: EOLSS Publications, 2009. ISBN 9781905839483.

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

[6] ESBMC. *Efficient SMT-Based Context-Bounded Model Checker*. 2013. Disponível em: http://esbmc.org. Acesso em: 5 março 2013.

[7] KROENING, D.; TAUTSCHNIG, M. CBMC – C bounded model checker. In: ____. *TACAS*. [S.l.]: Springer Berlin Heidelberg, 2014. p. 389–391. ISBN 978-3-642-54862-8.

[8] BIERE, A. et al. Symbolic model checking without bdds. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 1999. (TACAS '99), p. 193–207. ISBN 3-540-65703-7. Disponível em: <http://dl.acm.org/citation.cfm?id=646483.691738>.

[9] BOWEN, J.; STAVRIDOU, V. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, IET, v. 8, n. 4, p. 189–209, 1993.

[10] YOU, J.; LI, J.; XIA, S. A survey on formal methods using in software development. IET, 2012.

[11] RUSHBY, J. *Formal methods and the certification of critical systems*. [S.l.]: Citeseer, 1993.

[12] CHONG, E. K. P.; ZAK, S. H. *An introduction to continuous optimization: foundations and fundamental algorithms*. 2. ed. [S.l.]: Wiley-Interscience, 1996. ISBN 9780471391265.

[13] MICHAEL, P.; ANDREASSON, N.; EVGRAFOV, A. *An introduction to continuous optimization: foundations and fundamental algorithms*. [S.l.]: Studentlitteratur AB, 2013. ISBN 9789144060774.

[14] CHINNECK, J. W. *Practical Optimization: A Gentle Introduction*. [S.l.]: Carleton University, 2000.

[15] GALPERIN, E. A. Problem-method classification in optimization and control. *Computers & Mathematics with Applications*, v. 21, n. 6-7, p. 1 – 6, 1991. ISSN 0898-1221.

[16] ANNA, N.; NORA, D.; MARGRIETL. *Convex Optimization*. [S.l.]: Cambridge University Press, 2004. ISBN 978-0-521-83378-3.

[17] HIRIART-URRUTY, J.-B. Conditions for global optimality. In: ____. *Handbook of Global Optimization*. Boston, MA: Springer US, 1995. p. 1–26. ISBN 978-1-4615-2025-2.

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

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

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

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

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

[23] PNUELI, A.; RUAH, S.; ZUCK, L. Automatic deductive verification with invisible invariants. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2001. p. 82–97.

[24] BAIER, C.; KATOEN, J.-P. *Principles of Model Checking (Representation and Mind Series)*. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

[25] 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, jul. 1993. ISSN 0018-9162. Disponível em: <https://doi.org/10.1109/MC.1993.274940>.

[26] BEYER, D. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In: ____. *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*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. cap. 7, p. 887–904. ISBN 978-3-662-49674-9.

[27] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 4, p. 957–974, 2012.

[28] CORDEIRO, L.; FISCHER, B. Bounded model checking of multi-threaded software using smt solvers. *arXiv preprint arXiv:1003.3830*, 2010.

[29] CORDEIRO, Lucas, FISCHER, Bernd, MARQUES-SILVA, João. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transaction of Software Engineering*, v. 38, n. 6, p. 50–55, 2002.

[30] BIERE, A. Bounded Model Checking. In: *Handbook of Satisfiability*. [S.l.]: IOS Press, 2009. p. 457–481. ISBN 978-1-58603-929-5.

[31] BARRETT, C. W. et al. Satisfiability modulo theories. In: *Handbook of Satisfiability*. [S.l.]: IOS Press, 2009. p. 825–885.

[32] ARMANDO, A.; MANTOVANI, J.; PLATANIA, L. Bounded model checking of software using smt solvers instead of sat solvers. In: *Proceedings of the 13th International Conference on Model Checking Software*. Berlin, Heidelberg: Springer-Verlag, 2006. (SPIN'06), p. 146–162. ISBN 3-540-33102-6, 978-3-540-33102-5.

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

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

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

[36] RAMALHO, M. et al. Smt-based bounded model checking of C++ programs. In: *20th IEEE International Conference and Workshops on Engineering of Computer Based Systems, ECBS 2013, Scottsdale, AZ, USA, April 22-24, 2013*. [S.l.]: IEEE Computer Society, 2013. p. 147–156.

[37] GADELHA, M. Y. R. et al. ESBMC 5.0: an industrial-strength C model checker. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. [S.l.]: ACM, 2018. p. 888–891.

[38] PEREIRA, P. A. et al. Verifying CUDA programs using smt-based context-bounded model checking. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. [S.l.: s.n.], 2016. p. 1648–1653.

[39] PEREIRA, P. A. et al. Smt-based context-bounded model checking for CUDA programs. *Concurrency and Computation: Practice and Experience*, v. 29, n. 22, 2017.

[40] MONTEIRO, F. R. et al. ESBMC-GPU A context-bounded model checking tool to verify CUDA programs. *Sci. Comput. Program.*, v. 152, p. 63–69, 2018.

[41] SOUSA, F. R. M.; CORDEIRO, L. C.; FILHO, E. B. de L. Bounded model checking of C++ programs based on the qt framework. In: *IEEE 4th Global Conference on Consumer Electronics, GCCE 2015, Osaka, Japan, 27-30 October 2015*. [S.l.]: IEEE, 2015. p. 179–180.

[42] GARCIA, M. et al. Esbmc$^{\text{qtom}}$: A bounded model checking tool to verify qt applications. In: *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*. [S.l.]: Springer, 2016. (Lecture Notes in Computer Science, v. 9641), p. 97–103.

[43] MONTEIRO, F. R. et al. Bounded model checking of C++ programs based on the qt cross-platform framework. *Softw. Test., Verif. Reliab.*, v. 27, n. 3, 2017.

[44] ROCHA, H. et al. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*. [S.l.]: Springer, 2012. (Lecture Notes in Computer Science, v. 7321), p. 128–142.

[45] IEEE. IEEE standard for binary floating-point numbers. 1985.

[46] BEYER, D.; KEREMOGLU, M. E. CPAchecker: A tool for configurable software verification. In: *CAV*. [S.l.]: Springer Berlin Heidelberg, 2011. p. 184–190. ISBN 978-3-642-22110-1.

[47] MORSE, J. et al. ESBMC 1.22 - (competition contribution). In: *TACAS*. [S.l.: s.n.], 2014. p. 405–407.

[48] JAMIL, M.; YANG, X. A literature survey of benchmark functions for global optimization problems. *CoRR*, abs/1308.4008, 2013. Disponível em: <http://arxiv.org/abs/1308.4008>.

[49] ARAÚJO, R. et al. SMT-based verification applied to non-convex optimization problems. In: *SBESC*. [S.l.: s.n.], 2016. p. 1–8.

[50] HUTTER, F. et al. Boosting verification by automatic tuning of decision procedures. In: *FMCAD*. [S.l.: s.n.], 2007. p. 27–34.

[51] ARAÚJO, R. et al. Counterexample guided inductive optimization. In: *arXiv:1704.03738 [cs.AI]*. [s.n.], 2017. p. 1–32. Disponível em: <http://arxiv.org/abs/1704.03738>.

[52] ARAUJO, R. F. et al. Counterexample guided inductive optimization applied to mobile robots path planning. In: *2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR), Curitiba, Brazil, November 8-11, 2017*. [S.l.]: IEEE, 2017. p. 1–6.

[53] THE MATHWORKS, INC. *Matlab Optimization Toolbox User's Guide*. [S.l.], 2016.

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