UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO PEDROZO AMARAL

ACELERANDO A CALIBRAGEM DO SOFTWARE SLEUTH: UM ESTUDO DE CASO DE OTIMIZAÇÃO APLICADO À SIMULAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2020

LEONARDO PEDROZO AMARAL

ACELERANDO A CALIBRAGEM DO SOFTWARE SLEUTH: UM ESTUDO DE CASO DE OTIMIZAÇÃO APLICADO À SIMULAÇÃO

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. André Koscianski

PONTA GROSSA

2020

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa

Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

ACELERANDO A CALIBRAGEM DO SOFTWARE SLEUTH: UM ESTUDO DE CASO DE OTIMIZAÇÃO APLICADO À SIMULAÇÃO

por

LEONARDO PEDROZO AMARAL

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 23 de Setembro de 2020 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Dr. André Koscianski
Orientador

_____
Prof. Dr. Augusto Foronda
Membro titular

_____
Prof. Dr. Hugo Valadares Siqueira
Membro titular

_____
Prof. MSc. Geraldo Ranthum
Responsável pelo Trabalho de Conclusão de Curso

_____
Profª. Dra. Mauren Louise Sguario
Coordenador do Curso

AGRADECIMENTOS

Infelizmente, não tenho como citar a todos que de alguma forma contribuíram para esta pesquisa, e que merecem estar aqui. No entanto, deixo evidente que essas pessoas têm a minha gratidão, e peço desculpas por não inclui-las nas palavras seguintes.

Agradeço ao meu orientador Prof. Dr. André Koscianski, por tudo que me ensinou e auxiliou nesta jornada.

Agradeço a Mylena, por sua companhia e pelo imenso apoio que me deu nestes momentos difíceis que enfrentamos.

Agradeço aos meus pais, Cintia e Cleandro, que sacrificaram muito para que eu chegasse até aqui.

Agradeço aos meus professores, por tudo que me ensinaram sobre esta ciência que tanto me fascina.

Aos meus colegas da universidade, em especial o Matheus e Andrey, que compartilharam comigo momentos memoráveis ao longo das Maratonas de Programação.

Enfim, a todos os que por algum motivo contribuíram para a realização deste trabalho.

RESUMO

AMARAL, Leonardo. **Acelerando a calibragem do software SLEUTH:** um estudo de caso de otimização aplicado à simulação. 2020. 48 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2020.

O crescimento acelerado dos espaços urbanos nas cidades exige hoje uma série de ferramentas, inclusive computacionais, para exercer adequadamente o planejamento urbano. O SLEUTH, um simulador urbano, visa atender esse propósito. Esse simulador se empenha em predizer quais locais serão futuramente urbanizados, tendo como base a urbanização histórica da região. Entretanto, ele requer uma fase prévia de calibragem que busca ajustar parâmetros e que tem uma duração muito longa. Este trabalho pretende diminuir o custo da fase de calibragem por meio de métodos de otimização disponíveis em uma biblioteca e que se adequam às características do simulador SLEUTH. Os resultados da utilização dessa biblioteca foram comparados com outras duas estratégias de calibragem presentes na literatura: força bruta e algoritmos genéticos. De maneira geral os métodos baseados em heurísticas apresentaram comportamento e vantagens semelhantes, em comparação com a calibragem padrão implementada no simulador.

Palavras-chave: Otimização. Problemas não-contínuos. Simulação. Simulação Urbana.

ABSTRACT

AMARAL, Leonardo. **Accelerating SLEUTH calibration:** a case study of optimization applied to simulation. 2020. 48 p. Work of Conclusion Course (Bachelor in Computer Science) - Federal Technology University of Paraná. Ponta Grossa, 2020.

The accelerated growth of urban spaces in cities today requires a series of tools, including computational ones, to properly exercise urban planning. SLEUTH, an urban simulator, aims to serve this purpose. This simulator aims to predict which places will be urbanized in the future, based on the historical urbanization of the region. However, it requires a previous calibration phase that seeks to adjust parameters and that has a very long duration. This work intends to reduce the cost of the calibration phase through optimization methods available in a library and that are adapted to the characteristics of the SLEUTH simulator. The results of using this library were compared with two other calibration strategies present in the literature: brute force and genetic algorithms. In general, the methods based on heuristics showed similar behavior and advantages, in comparison with the standard calibration implemented in the simulator.

Keywords: Optimization. Non- continuous problems. Simulation. Urban Simulation.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Otimização é algo fundamental na ciência, sendo empregada em diversas áreas como medicina, astronomia, economia, engenharia automobilística e engenharia civil. Qualquer situação em que seja necessário determinar a entrada que produza a melhor saída de um determinado sistema corresponde a um problema de otimização.

Problemas elementares de otimização se resumem a encontrar extremos de funções; no caso de funções matemáticas contínuas isso pode ser feito a partir da noção de gradiente. Apesar de preferível essa abordagem para buscar a entrada ótima, alguns problemas não dispõem de derivadas a serem exploradas. Consequentemente, esses problemas exigem outras técnicas para serem resolvidos. Exemplos comuns são problemas derivados de simulações, por software, de processos industriais, físicos ou sociais, baseados na execução de algoritmos e técnicas como agentes. Nesses casos não há uma função matemática que possa ser derivada, ou ainda o resultado dos cálculos de um software podem apresentar saltos descontínuos que tornam difícil fazer uma aproximação do gradiente.

Softwares de simulação, devido a sua alta complexidade, podem demandar a configuração prévia de algumas das variáveis que definem seu comportamento. A escolha ideal dessas variáveis pode não ser óbvia. Consequentemente, alguns simuladores adotam um procedimento de calibragem para a escolha desses parâmetros. Se houver um objetivo específico da simulação, como encontrar as dimensões dos pneus que reduzam o gasto de combustível de um veículo em um dado trajeto, então a tarefa de calibragem consiste na realidade em um problema de otimização.

Uma área de pesquisa que faz uso de simuladores de comportamento bastante complexo e de calibragem difícil é a simulação urbana. Ela se ocupa de criar modelos de comportamento para vários aspectos da dinâmica de cidades, como economia, tráfego e administração. A expansão de cidades é um fenômeno de interesse em função dos impactos que gera em relação à necessidade de ampliar infraestrutura, gerenciar ocupação, ou ainda ecologia. Muitos simuladores de crescimento urbano são baseados em algoritmos, e dependem de conjuntos de parâmetros que devem ser ajustados para garantir que os resultados calculados estejam em acordo com a evolução da cidade sendo estudada. Uma das ferramentas muito utilizadas nessa área é o simulador SLEUTH. A fase de calibragem desse software pode ser bastante extensa; alguns trabalhos da década de 90 apontavam tempos da ordem de meses; um estudo recente (Roth, 2019) consumiu tempos de cálculo da ordem de dois dias. O algoritmo de calibragem original de SLEUTH utilizava um algoritmo de força

bruta, e apesar da ineficiência o mesmo teria a vantagem de cobrir todas as combinações de entrada.

1.1 OBJETIVOS

Este trabalho tem como objetivo modificar a calibragem do SLEUTH para exercer um método de otimização implementado em uma biblioteca, e comparar seu desempenho com as principais estratégias de calibragem presentes na literatura. Para isso, os seguintes objetivos específicos foram atendidos:

- Modificar o código fonte do SLEUTH a fim de possibilitar a ligação com a biblioteca;
- Realizar experimentos com o método de otimzação;
- Comparar os resultados das estratégias de calibragem por força bruta e algoritmos genéticos com as do SLEUTH modificado.

2. REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta a revisão na literatura necessária para atingir os objetivos apresentados no capítulo anterior. A seção 2.1 provê uma visão geral sobre otimização, com foco em problemas derivados de simuladores e suas principais abordagens. A seção 2.2 aborda as funcionalidades do SLEUTH fundamentais para este trabalho. E a seção 2.3 discorre sobre mecanismos de comunicação entre processos.

2.1 OTIMIZAÇÃO

2.1.1 Visão Geral

Otimização é constantemente utilizada nas mais diferentes áreas do conhecimento. Nos setores de gestão é comum o interesse em minimizar gastos, maximizar a produtividade e assim maximizar o lucro. Em logística, minimizar a distância total percorrida em uma sequência de entregas pode resultar em um gasto menor de combustível. Na engenharia aeroespacial é vantajoso minimizar o peso das asas de um avião, enquanto a aerodinâmica e robustez são mantidas. Até mesmo na natureza, o processo de adaptação dos seres vivos causada por ações como a seleção natural também pode ser visto como um caso de otimização, uma vez que esse processo visa maximizar a sobrevivência de uma espécie. De fato, o excepcional desempenho da evolução dos organismos vivos inspirou uma série de algoritmos de otimização.

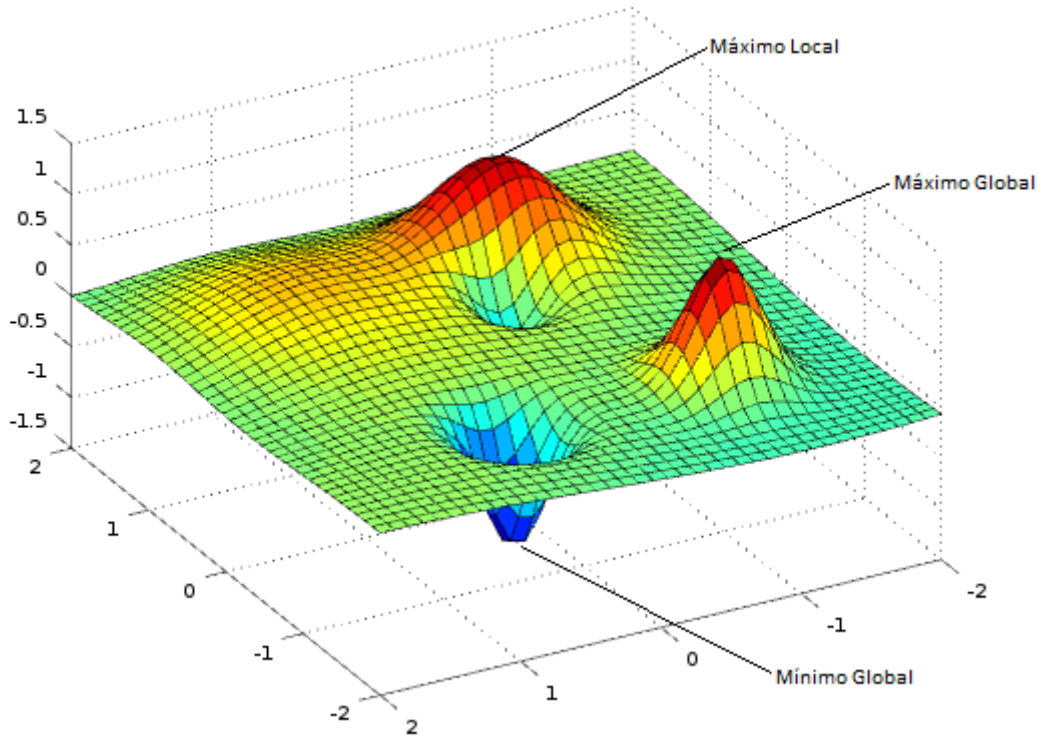
2.1.2 Função Objetivo

A função objetivo $f: \mathbb{X} \rightarrow Y$ é responsável por avaliar numericamente as soluções do alvo a otimização. Seu domínio $\mathbb{X}$ possui todas as entradas viáveis, que podem ser números reais, números inteiros, vértices de grafos e etc. Enquanto seu contradomínio $Y \subseteq \mathbb{R}$ retrata a qualidade de cada solução (WEISE, 2009). Funções objetivo são traduzidas ao mundo real como modelos matemáticos, modelos de simulação, modelos físicos e assim por diante. Nesse

contexto, o procedimento da escolha ideal dos parâmetros desses modelos é conhecido como otimização (BAYER; SENDHOFF, 2007).

Métodos de otimização se empenham em encontrar a solução ótima local ou global de um problema, que usualmente são os máximos e mínimos da função objetivo. No contexto de minimização da função objetivo, um ótimo local é uma solução que contempla o menor resultado da função objetivo entre suas soluções vizinhas. O ótimo global, o menor resultado entre todas as soluções viáveis do problema (WEISE, 2009). A figura 1 exemplifica os extremos locais e globais de uma função.

Figura 1 – Extremos de uma função bidimensional



Fonte: Autoria própria

Encontrar a solução ótima é impraticável em inúmeras situações. Salvo algumas exceções, é infactível afirmar que a solução encontrada pelo modelo de otimização é de fato ótima. Além disso, o custo computacional necessário para encontrar soluções ligeiramente superiores pode tornar a busca pelo ótimo global economicamente inviável (BAYER; SENDHOFF, 2007). Consequentemente, os algoritmos de otimização fazem uso de critérios de parada, como limitar o total de iterações da função objetivo ou interromper a otimização a partir de um limiar de erro.

2.1.3 Problemas Contínuos e Discretos

As variáveis de um problema de otimização podem ser contínuas ou discretas. Uma variável real que pode assumir qualquer valor em algum intervalo é contínua. Quando todas as variáveis de um problema são contínuas, o problema é contínuo (JEYAKUMAR; RUBINOV, 2006).

Variáveis discretas assumem valores limitados a um conjunto discreto de possibilidades, como os números inteiros. Um problema é discreto se pelo menos uma de suas variáveis é discreta (LEE, 2000). Um exemplo disso é o problema da mochila, que consiste em encontrar uma combinação de pacotes de diferentes tamanhos que possam ser guardados em um volume restrito (FRÉVILLE, 2004).

2.1.4 Gradiente

Nos casos em que a função objetivo é um modelo matemático derivável, o vetor gradiente pode ser utilizado na otimização (SHEWCHUK et al., 1994). Para todos os pontos da função, esse vetor aponta a direção que provê o maior incremento, uma informação útil para convergência aos ótimos locais de alguns algoritmos. O algoritmo Descida de Gradiente, por exemplo, faz uso de gradientes para encontrar mínimos locais de funções diferenciáveis (SHEWCHUK et al., 1994). Além disso, alguns métodos baseados no Descida de Gradiente usam estratégias para escapas dos extremos locais, como o proposto por Cursi, Ellaia e Bouhadi (2004), que busca o ótimo global a partir de perturbações aleatórias na direção do vetor gradiente.

Embora gradientes tenham se mostrado muito úteis, diversos problemas do mundo real não dispõem de derivadas na função objetivo. Consequentemente, esses problemas exigem algoritmos que dispensam o uso de qualquer derivada (AMARAN et al., 2016).

2.1.5 Métodos Determinísticos e Probabilísticos

Para um método determinístico ser adequado é necessário que o problema possua propriedades analíticas capazes de motivar, de forma incontestável, sua solução ótima. O determinismo do método se deve ao fato de não existir nenhuma característica probabilística em seu processo, com uma clara relação entre cada passo que leva à solução ótima (LIN;

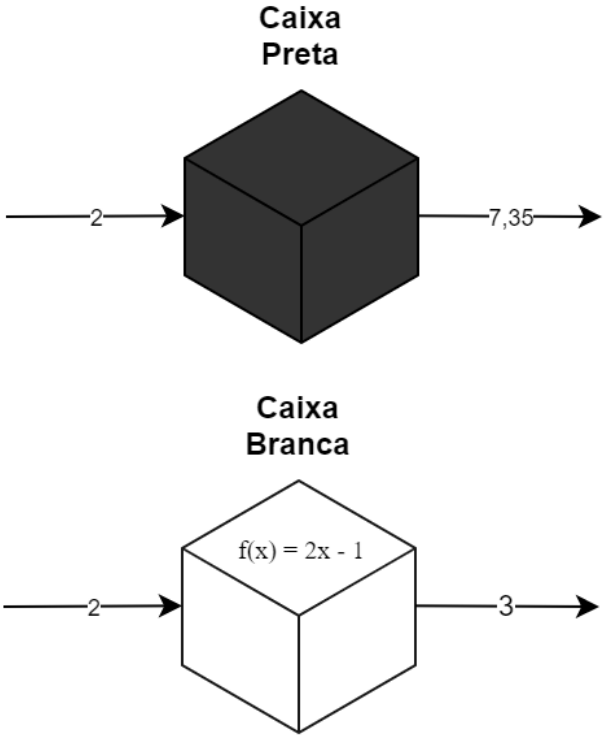
TSAI; YU, 2012, p. 1). Abordagens como programação linear e divisão e conquista são exemplos de métodos determinísticos.

Quando o domínio da função é muito extenso ou a estratégia que leva a um ótimo global é complexa ou inexistente, métodos determinísticos são inviáveis para resolver o problema, inspirando a utilização dos métodos probabilísticos. Esses algoritmos não garantem que o ótimo global será sempre encontrado, sendo esse o custo pela diminuição do tempo de otimização (WEISE, 2009, p. 22).

2.1.6 Problemas de Caixa Preta

Caixa preta é um termo usado com frequência e que se refere a um sistema complexo com um comportamento desconhecido ou difícil de prever. Problemas dessa natureza não podem ser resolvidos com base em suas características internas; as únicas informações extraíveis são as saídas correspondentes a algumas entradas (AUDET; HARE, 2017). Como a figura sugere, não há como saber qual o procedimento realizado pela caixa preta.

Figura 2 - Sistemas de caixa preta e caixa branca



Fonte: Autoria própria

A calibragem ou otimização com softwares de simulação são típicos problemas de caixa preta, no sentido de que o usuário pode controlar variáveis que guiam funcionamento sem se importar com o funcionamento interno do software (SUMATA et al., 2013). Simuladores também podem ter complicadores adicionais, tais como elevado custo de execução (e.g. tempo) ou ruído (KOLDA; LEWIS; TORCZON, 2003). Essas características dificultam significativamente o processo de otimização e, consequentemente, estimulam o desenvolvimento de solucionadores para problemas de caixa preta (AMARAN et al., 2016, p. 352).

Os algoritmos de otimização aplicáveis a problemas de caixa preta são obrigados a basearem-se somente nos resultados da função objetivo, sem disporem da informação de gradiente. As seções seguintes abordam alguns desses algoritmos.

2.1.7 Têmpera Simulada

O algoritmo Têmpera Simulada, proposto por Kirkpatrick, Gelatt e Vecchi (1983), é um algoritmo probabilístico de otimização global amplamente utilizado em problemas não diferenciáveis. Baseado no tratamento térmico de materiais em metalurgia, o algoritmo oferece uma estratégia interessante para fugir de ótimos locais e alcançar aproximações de ótimos globais (NACHTIGALL, 2015).

O algoritmo Têmpera Simulada requer inicialmente um ponto de partida e um valor real que represente a temperatura inicial. A cada iteração a temperatura é ligeiramente decrementada e o algoritmo deve decidir entre manter a solução atual, ou trocar para uma solução vizinha (i.e. uma solução viável, feita a partir de pequenas alterações na solução atual). A probabilidade de troca para soluções piores é maior em temperaturas altas, o que permite que o algoritmo escape de ótimos locais no inicio e se aproxime do ótimo global nas últimas iterações (PIRLOT; VIDAL, 1996).

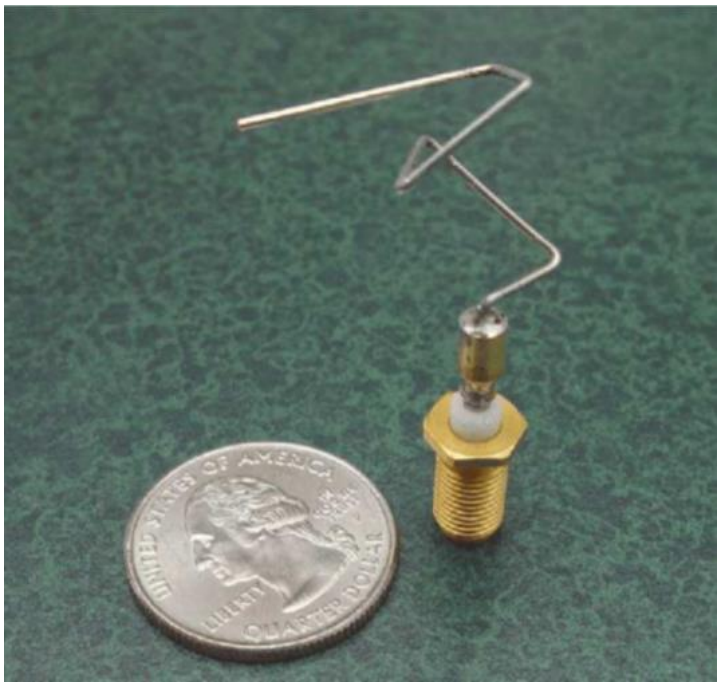
2.1.8 Algoritmos Genéticos

Assim como a Têmpera Simulada, os algoritmos genéticos também são inspirados em fenômenos naturais (AUTRIQUE; LEYRIS; DE CURSI, 1999). Esses algoritmos usam diversos mecanismos motivados pela evolução dos organismos, como mutação, recombinação, hereditariedade e seleção natural. Além disso, costumam ser particularmente

eficientes em problemas discretos, visto que desprezam um sistema de sequência entre os elementos do espaço de busca (WEISE, 2009, p. 95). Vale observar que há muitos algoritmos de otimização bio-inspirados que basicamente repetem heurísticas já existentes, mas com diferentes nomes, conforme explica Sörensen (2015).

Como exemplo de aplicação, Hornby et al. (2006) utilizaram algoritmos genéticos para buscar o design de antena que maximizava seu desempenho. Inicialmente, um conjunto de antenas com formatos aleatórios foi gerado, definindo a população inicial. As antenas passavam então por simulações para que o desempenho de cada uma fosse avaliado, com o objetivo de gerar uma nova população constituída da combinação das características das melhores antenas da população anterior e, também, de sutis modificações aleatórias (análogo às mutações dos seres vivos). Ao final do procedimento, o formato de antena obtido (figura 3) superou o desempenho das antenas criadas à mão e foi utilizado pela *National Aeronautics and Space Administration* (NASA) em microssatélites lançados no dia 22 de março de 2006.

Figura 3 - Melhor design de antena



Fonte: Hornby et al. (2006)

2.1.9 Variable Neighborhood Search

O *Variable Neighborhood Search* (VNS) foi proposto por Mladenović e Hansen em 1997, como um método para resolver problemas de otimização combinatória. Para ser

aplicado, o problema deve possuir alguma estrutura de vizinhança e um método de descida que permita encontrar soluções melhores de maneira sequencial até alcançar um ótimo local. Aliado a isso, o método foge de extremos locais com outra fase de busca, que se propõe a encontrar soluções distantes da atual com base em perturbações aleatórias. O princípio dessa perturbação é vinculado ao problema, de modo que a eficiência do VNS é diretamente proporcional à aplicação de um método de perturbação apropriado ao problema (AUDET, BÉCHARD e LE DIGABEL, 2008, p. 302).

2.1.10 Métodos de Busca Direta

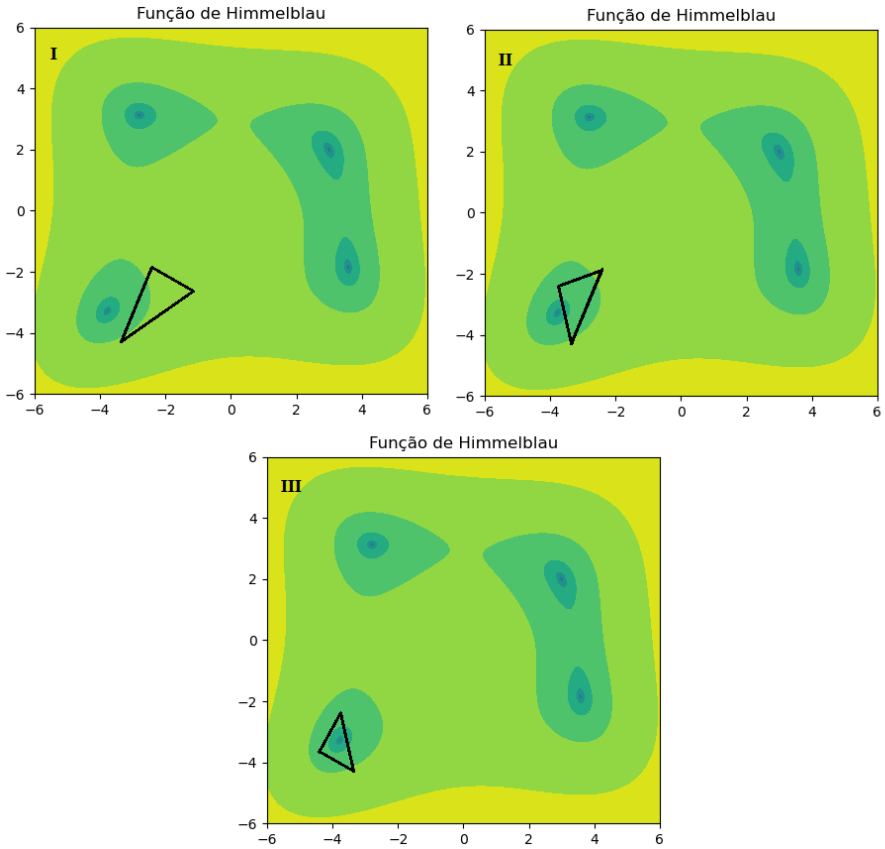
Métodos de busca direta se destacam por basearem-se somente em comparações quanto aos resultados da função objetivo (KOLDA; LEWIS; TORCZON, 2003). Em outros termos, esses métodos dispensam a necessidade de gradientes, e não é obrigatório que os valores da função objetivo sejam numéricos, desde que esses possam ser classificados de algum modo. Assim sendo, como os métodos de busca direta exigem poucas premissas, são frequentemente aplicados a problemas de caixa preta (AMARAN et al., 2016, p. 364).

2.1.10.1 Método de Nelder-Mead

Um dos mais famosos métodos de busca direta é o proposto por Nelder e Mead em 1965, que tem como princípio o uso de simplex para convergir a ótimos globais (KOLDA; LEWIS; TORCZON, 2003). Simplex são em um meio com n dimensões a generalização de um polítopo com $n + 1$ vértices. Por exemplo: o triângulo é um simplex bidimensional; tetraedro, um simplex tridimensional.

O procedimento de Nelder e Mead inicia com um conjunto de soluções que representam os vértices do simplex. A cada iteração um novo vértice é calculado para substituir o vértice que retrata a solução com o pior custo. A condição atual das soluções define se a nova solução será proveniente da reflexão, contração, encolhimento ou expansão do simplex. A figura 4 exemplifica a evolução do simplex quando aplicado à função de Himmelblau. Quando o método é bem sucedido, os vértices do simplex se aproximam até alcançar um critério de parada (KOLDA; LEWIS; TORCZON, 2003, p. 424).

Figura 4 - Evolução do Nelder-Mead na função de Himmelblau



Fonte: Autoria própria

2.1.10.1 Mesh Adaptive Direct Search

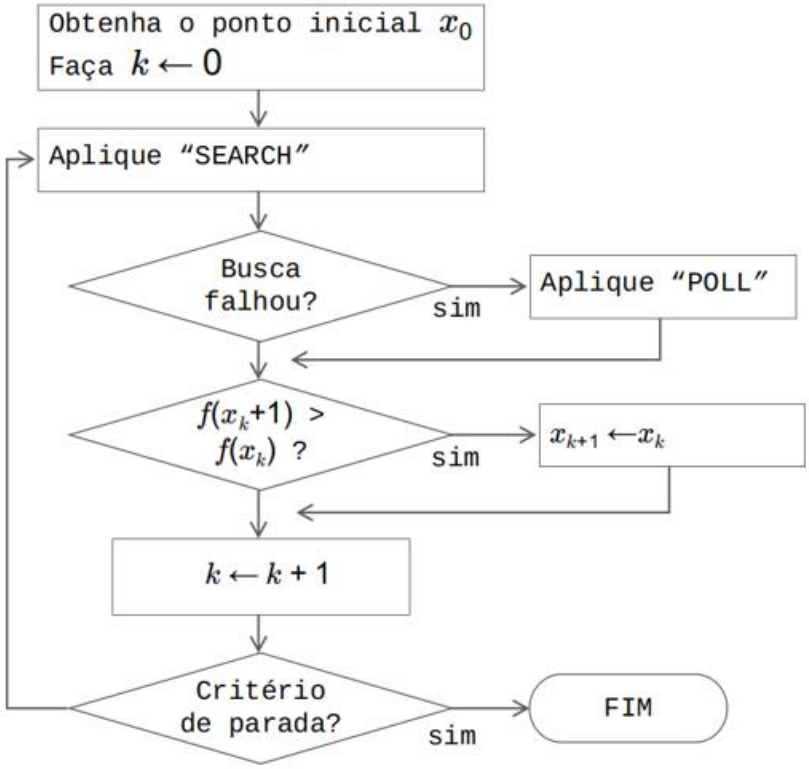
O *Mesh Adaptive Direct Search,* ou MADS, é um método de otimização interessante, pois: pode ser aplicado a problemas de caixa preta contínuos e discretos. Dedica-se a encontrar soluções ótimas com poucas execuções da função objetivo, o que é interessante para simulações de elevado custo. E pode ser facilmente combinado com outros métodos de otimização (AUDET; DENNIS, 2006). O princípio do método é a construção de uma malha conceitual de pontos (conceitual porque nunca é efetivamente construída). A cada iteração, essa malha delimita a região de soluções que podem ser verificadas e, no decorrer do algoritmo, ela pode ser expandida ou contraída (AUDET; DENNIS, 2006). Esta seção visa abordar apenas os conceitos básicos do método; os detalhes de seu funcionamento e a demonstração de sua análise de convergência não serão vistos aqui. Para essas informações, sugere-se consultar (AUDET; DENNIS, 2006).

Cada iteração do MADS possui duas fases de busca: *search* e *poll*. A metodologia da fase *search* não é definida, é permitido que qualquer técnica de busca seja aplicada (e.g.

método de Nelder-Mead), desde que todos os pontos encontrados respeitem as restrições da malha. Além disso, a fase *search* é opcional, ou seja, é possível realizar apenas a fase *poll* nas iterações. Consequentemente, a fase *search* não interfere na teoria de convergência do algoritmo, servindo apenas como um possível acelerador de otimização (AUDET; DENNIS, 2006).

Em uma iteração, a fase *search* deve ocorrer primeiro, com o propósito de encontrar alguma solução melhor que a atual. Se a busca for bem sucedida, o algoritmo pode prosseguir para próxima iteração com a solução encontrada; do contrário, a fase de *poll* deve suceder obrigatoriamente. A lógica dessa fase é rigorosamente definida, e também deve respeitar as restrições da malha. Caso *poll* falhe em encontrar uma solução melhor, a malha será encolhida para a próxima iteração; senão, o tamanho da malha pode ser mantido ou então aumentado. Com isso, o encolhimento gradual da malha auxilia a garantir a convergência do algoritmo (AUDET; LE DIGABEL; TRIBES, 2019, p. 1172). A figura 5 resume o fluxo do MADS.

Figura 5 - Fluxograma do MADS



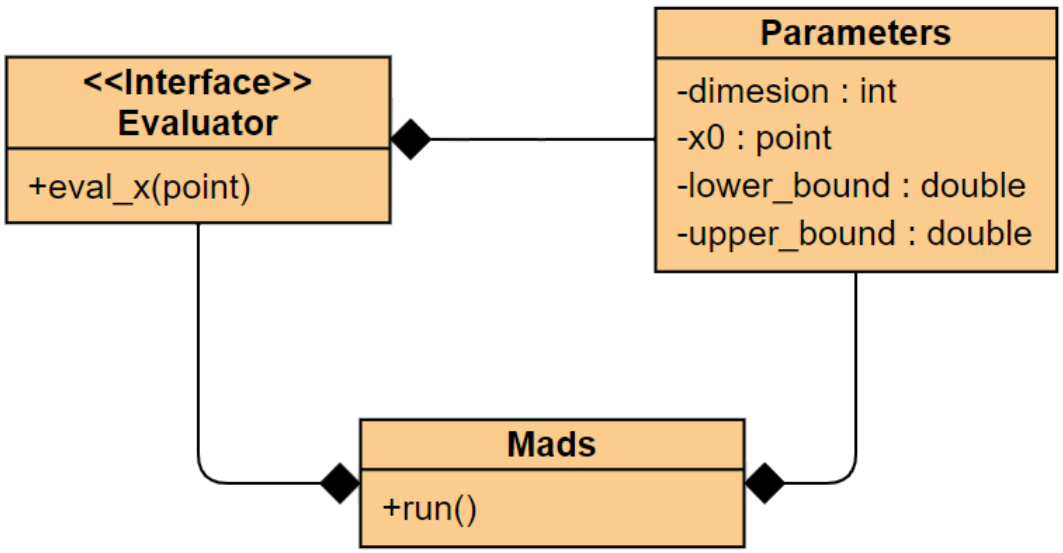
Fonte: Autoria própria

2.1.10.1.1 NOMAD

O NOMAD (*Nonlinear Optimization by Mesh Adaptive Direct Search*) é um software projetado para a otimização de problemas de caixa preta (LE DIGABEL, 2011). Mais especificamente, ele visa abordar problemas que são custosos, sem informações de gradiente e com poucas variáveis. Se o problema não apresentar essas características o software não é recomendado. O software exerce a otimização por meio do algoritmo MADS e possui os algoritmos Nelder-Mead e VNS que podem ser opcionalmente aplicados durante a fase *search* do MADS.

O uso ideal do NOMAD é como uma biblioteca para aplicações em *C++*, em que é organizado com o paradigma orientado a objetos. A figura 6 mostra de forma resumida as três classes principais para o uso do NOMAD.

Figura 6 - Diagrama de classe simplificado do NOMAD



Fonte: Autoria própria

A classe *Parameters* define as características do espaço de busca do problema de otimização, como quantidade de dimensões e limites inferiores e superiores das variáveis. *Evaluator* determina a abstrção de um problema; é preciso que alguma subclasse sua

sobrescreva o método *eval_x* para desempenhar a função objetivo. A classe *Mads* depende de um objeto do tipo *Parameters* e outro, do tipo *Evaluator*, para iniciar as iterações do algoritmo *Mesh Adaptive Direct Search* através de seu método *run*.

2.2 SLEUTH

O software SLEUTH tem por finalidade prever crescimento urbano a partir de simulações (CLARKE; GAYDOS, 1998). Ele representa a região com autômatos celulares, em que cada célula descreve uma parcela do local que poderá ser urbanizada futuramente (CLARKE, 2014). Para tal fim, o simulador requer que sejam fornecidas imagens históricas da região e cinco coeficientes em números inteiros. As seções seguintes fazem uma descrição superficial desses dados; descrições sobre outros aspectos do SLEUTH fogem do escopo deste trabalho. No entanto, caso seja de interesse, uma apresentação mais detalhada está disponível em Roth (2019).

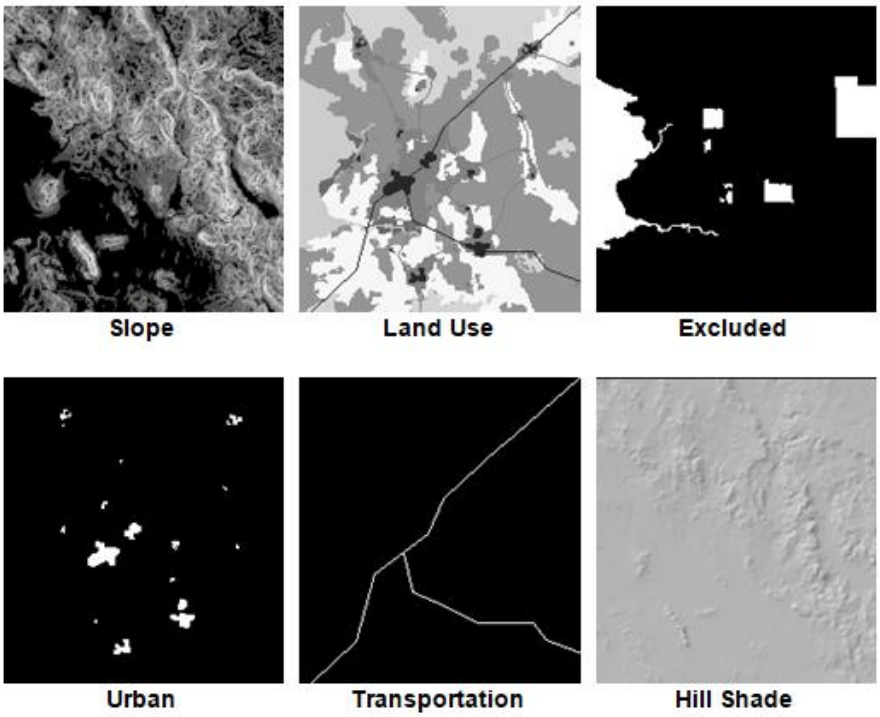
2.2.1 Dados de Entrada

O SLEUTH exige um conjunto de imagens que descrevem, durante um período histórico, o desenvolvimento urbano em uma região definida. Cada imagem ilustra em preto e branco ou em tons de cinza as condições de um determinado aspecto na região e sua evolução ao longo do tempo, totalizando seis categorias de imagens:

- *Slope*: o nível de cinza é proporcional a declividade do terreno;
- *Land Use*: tipos de uso do terreno, como florestas, áreas rurais e áreas urbanas;
- *Excluded*: locais onde a urbanização é inviável, como parques e lagos;
- *Urban*: a área efetivamente urbanizada;
- *Transportation*: a rede de transporte (e.g. estradas e linhas de trem) do local;
- *Hill Shade*: uma iluminação hipotética da superfice pelo sol, em escalas de cinza.

A figura 7 exemplifica cada tipo de imagem sobre uma mesma região.

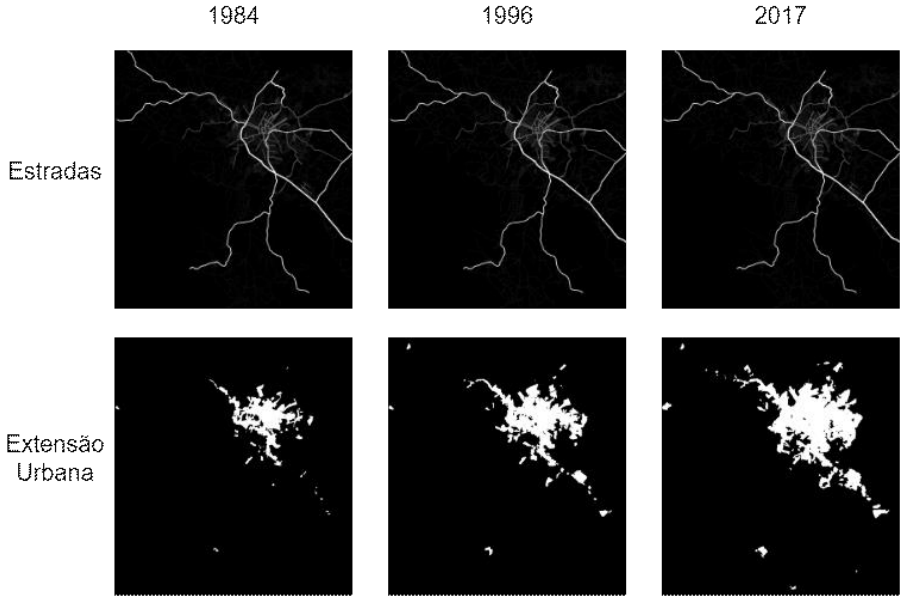
Figura 7 - Tipos de imagens de entrada do SLEUTH



Fonte: Documentação do SLEUTH

A fim de descrever a urbanização da região durantes os anos anteriores, múltiplas imagens de cada categoria devem ser fornecidas. As imagens devem ter exatamente as mesmas dimensões e corresponder a mesma região e proporção. A figura 8 exemplifica isso com imagens reais da evolução da extensão urbana e das estradas da cidade de Ponta Grossa, no Brasil:

Figura 8 - Extensão urbana e estradas da cidade de Ponta Grossa, no Brasil, de 1984 a 2017.



Fonte: Autoria própria

2.2.2 Calibragem

Além das imagens históricas vistas na seção anterior, o SLEUTH requer a configuração de cinco coeficientes:

- *Dispersion*: define a frequência em que células aleatórias são urbanizadas**
- *Breed*: controla a probabilidade de novas células urbanas serem expandidas
- *Spread*: ajusta a intensidade da expansão das células urbanas
- *Slope*: controla a sensibilidade da urbanização em relação a inclinação do terreno
- *Road gravity*: define a sensibilidade da urbanização quando distante das estradas

Cada coeficiente assume um valor inteiro entre 0 e 100 a fim descrever a intensidade de alguma característica do crescimento urbano; diferentes combinações entre esses valores influenciam a modelagem da urbanização. O objetivo da calibragem é, com base em numerosos testes, decidir quais valores de cada coeficiente descrevem melhor o urbanização na região de estudo.

Nas versões iniciais do simulador a calibragem é por força bruta. Porém, com 101^5 diferentes combinações de coeficientes é computacionalmente inviável um processo de força bruta. Para mitigar o problema é possível delimitar cada coeficiente em um intervalo e configurar também o valor de passo (a quantia que o coeficiente é incrementado). Por exemplo: se um coeficiente for delimitado aos extremos 20 e 80 e o comprimento do passo for 15, esse coeficiente vai assumir apenas os valores 20, 35, 50, 65 e 80 durante as iterações de calibragem.

Ao longo da calibragem o SLEUTH produz um arquivo que relaciona cada combinação de coeficientes com dados estatísticos e uma medida de comparação entre as imagens produzidas pelo simulador e as imagens reais da região sob estudo. Esses dados podem ser utilizados para decidir quais coeficientes melhor descrevem o crescimento urbano dessa região. Os dados são:

1. *Product*
2. *Compare*
3. *Pop*
4. *Edges*
5. *Clusters*
6. *Size*
7. *Leesalee*

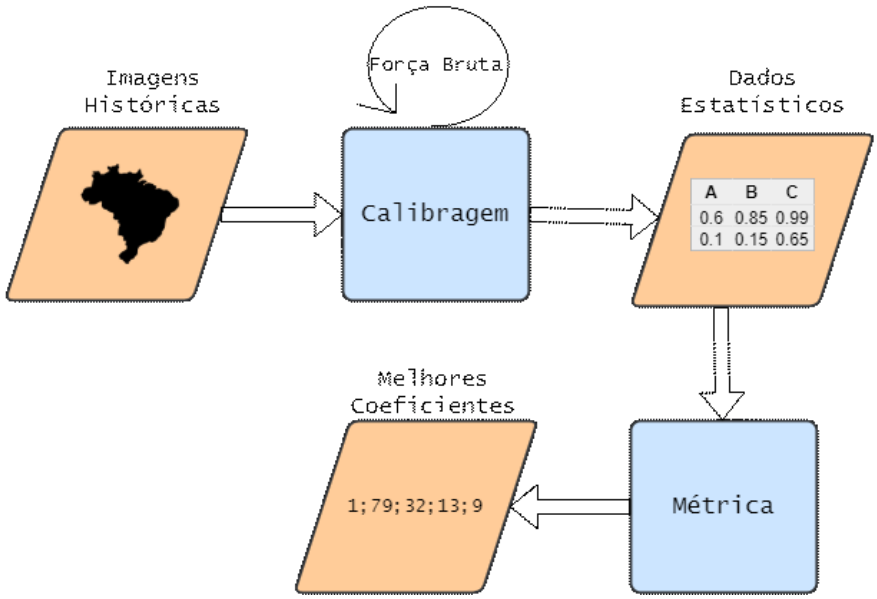
8. *Slope*
9. *%Urban*
10. *X-mean*
11. *Y-mean*
12. *Rad*
13. *Fmatch*

O valor de *Leesalee* corresponde à uma métrica que compara imagens simualadas e reais. Outros números, como "%Urban", "X-mean" e "Y-mean", podem ser interpretados por especialistas no uso do SLEUTH e usados como critério para filtrar simulações julgadas mais apropriadas.

Uma métrica denomida OSM (*Optimum SLEUTH Metric*) foi proposta por Dietzel, Charles e Clarke em 2007. Ela é calculada como o produto das variáveis: *compare, population, edges, clusters, slope, X-mean e Y-mean*. Ambas as métricas são amplamente usadas nas aplicações do SLEUTH.

A figura 9 demonstra o fluxo habitual para a descoberta dos melhores coeficientes:

Figura 9 - Fluxo do processo de seleção dos melhores coeficientes



Fonte: Autoria própria

Quando a calibragem é por força bruta, é recomendado que ela seja efetuada em três fases, denominadas *coarse, fine* e *final*. A cada fase são utilizadas imagens de resolução crescente. Na primeira fase, os coeficientes assumem os valores 0, 25, 50, 75 e 100. Já na segunda e na terceira fase, as faixas de valores dos coeficientes são determinados com base nos resultados da fase anterior: as três melhores iterações de calibragem (de acordo com

alguma métrica) são selecionadas, e o menor e maior valor que cada coeficiente assumiu nelas define o novo intervalo deste coeficiente para a próxima fase; o valor de passo, por sua vez, é calculado de modo que o coeficiente assuma de 4 a 6 valores distintos nas iterações. Isso implica que cada fase execute uma quantidade de iterações próxima de 3.125.

2.2.3 Estrutura do Código

O SLEUTH é inteiramente escrito na linguagem C e deve ser compilado para plataformas Linux através do *GNU Compile Collection* (GCC). Para este trabalho, os fragmentos de código mais importantes são os responsáveis por realizar o processo de calibragem e a persistência de seus resultados estatísticos. A força bruta da calibragem é desempenhada a partir de laços aninhados, em que cada laço é responsável por um coeficiente. Ao final de cada iteração de calibragem os resultados (descritos na seção 2.2.2) são salvos em um arquivo. A figura 10 comtempla os laços da calibragem por força bruta e a persistência dos dados resultantes.

Figura 10 - Trechos de código da calibragem e a persistência de seus resultados

main.c

```
544      * CALIBRATION AND TEST RUNS
545      */
546     proc_SetStopYear (igrid_GetUrbanYear (igrid_GetUrbanCount () - 1));
547
548     for (diffusion_coeff = coeff_GetStartDiffusion ();
549          diffusion_coeff <= coeff_GetStopDiffusion ();
550          diffusion_coeff += coeff_GetStepDiffusion ()) {
551       for (breed_coeff = coeff_GetStartBreed ();
552            breed_coeff <= coeff_GetStopBreed ();
553            breed_coeff += coeff_GetStepBreed ()) {
554         for (spread_coeff = coeff_GetStartSpread ();
555              spread_coeff <= coeff_GetStopSpread ();
556              spread_coeff += coeff_GetStepSpread ()) {
557           for (slope_resistance = coeff_GetStartSlopeResist ();
558                slope_resistance <= coeff_GetStopSlopeResist ();
559                slope_resistance += coeff_GetStepSlopeResist ()) {
560             for (road_gravity = coeff_GetStartRoadGravity ();
561                  road_gravity <= coeff_GetStopRoadGravity ();
562                  road_gravity += coeff_GetStepRoadGravity ())
```

stats-obj.c

```
1954 static void  stats_LogControlStats (FILE * fp) {
1955
1956   fprintf (fp, "%5u %8.5f %7.5f %7.5f %7.5f %7.5f %7.5f %7.5f %7.5f %7.5f %7.5f ",
1957            proc_GetCurrentRun (),
1958            aggregate.product  ,  aggregate.compare,
1959            regression.pop     ,  regression.edges,
1960            regression.clusters,  regression.mean_cluster_size,
1961            aggregate.leesalee ,  regression.average_slope,
1962            regression.percent_urban);
1963
1964   fprintf (fp, "%7.5f %7.5f %7.5f %7.5f %4.0f %4.0f %4.0f %4.0f %4.0f\n",
1965            regression.xmean          ,  regression.ymean,
1966            regression.rad            ,  aggregate.fmatch,
1967            coeff_GetSavedDiffusion (),  coeff_GetSavedBreed (),
1968            coeff_GetSavedSpread    (),  coeff_GetSavedSlopeResist (), coeff_GetSavedRoadGravity ());
```

Fonte: Código fonte do SLEUTH

2.2.4 SLEUTH GA

Uma nova versão do SLEUTH foi proposta por Goldstein (2004) substituindo o algoritmo de força bruta por uma calibragem utilizando algoritmos genéticos. Cada conjunto de coeficientes é tido como um cromossomo, e a combinação desses segue uma estratégia de escolha uniforme de genes. Os testes foram realizados na cidade de Sioux Falls, em Dakota do Sul, e, em linhas gerais, a calibragem com algoritmos genéticos resultou em uma diminuição de 80% do custo de CPU com uma acurácia superior a força bruta (CLARKE-LAUER e CLARKE, 2011, p. 25). Em contrapartida, o algoritmo genético exige a configuração prévia de alguns parâmetros, como o tamanho da população e a taxa de mutação, o que pode demandar iterações extras de calibragem. Além disso, há evidências de que o algoritmo possa estagnar em ótimos locais.

Posteriormente, a estratégia de algoritmos genéticos foi aperfeiçoada em diversos trabalhos e, em 2017, foi publicado uma versão oficial do SLEUTH que realiza a calibragem unicamente por algoritmos genéticos (CLARKE, 2017, p. 324).

2.3 COMUNICAÇÃO ENTRE PROCESSOS

Nos sistemas operacionais, é excessivamente recorrente a necessidade de processos serem capazes de comunicar-se entre si. Geralmente, com o objetivo de enviar sinais para notificar acontecimentos, compartilhar dados ou sincronizar ações (GALVIN et al., 2003). Os mecanismos responsáveis por isso são chamados de IPC (*Inter-Process Communication*) e, a fim de promover compatibilidade entre diferentes sistemas operacionais, foram padronizados no POSIX (*Portable Operating System Interface*) pela IEEE. Entretanto, outras especificações existem atualmente. Este capítulo visa abordar os principais mecanismos IPC responsáveis por transferência de dados e que estão presentes em sistemas Linux.

2.3.2 Memória Compartilhada

Processos podem se comunicar por meio do acesso a uma mesma região de memória. Ou seja, um processo pode compartilhar uma informação escrevendo-a numa região de memória que pode ser lida por outro(s) processo(s) posteriormente. Esse é um meio de

comunicação extremamente rápido; os dados podem ser recebidos imediatamente após serem transmitidos, já que não há necessidade de realizar chamadas ao sistema operacional durante a transmissão.

No entanto, o acesso simultâneo a mesma região de memória é vulnerável a condições de corrida. Pode ser necessário sincronizar seu acesso, usualmente mediante o uso de semáforos. Aliás, a sincronização pode demandar uma alta complexidade de elaboração da interface de comunicação, uma desvantagem que meios mais sofisticados de comunicação não possuem. Por conta disso, apesar de possibilitar a comunicação entre vários processos e da grande vantagem em desempenho, comunicação por meio de memória compartilhada pode ser contraindicada.

2.3.3 Pipes

A comunicação por meio de *pipes* é efetuada a partir de leituras e escritas em um buffer no sistema operacional. A comunicação é unidirecional, o processo que escreve não deve ler, e vice-versa. Porém, uma comunicação bidirecional pode ser efetuada com o uso de um par de *pipes*. Além disso, a sincronização é automática. Logo, se um processo tentar realizar uma operação de leitura e o buffer não possuir informação alguma, esse processo ira esperar até que outro processo realize uma escrita.

Embora seja possível que vários processos compartilhem um mesmo *pipe*, isso não é habitual, dado que as operações de leitura removem os dados lidos do buffer. Desse modo, a tarefa de compartilhar uma informação entre todos os processos pode ser complexa. Consequentemente, *pipes* são indicados para a comunicação de pares de processos, de forma unidirecional e, geralmente, quando um processo é pai do outro.

2.3.4 Sockets

Em comparação aos mecanismos apresentados até o momento, *sockets* são mais sofisticados e possuem um nível superior de abstração. A comunicação segue um esquema de cliente-servidor; um *socket* cliente só pode se conectar com um *socket* servidor, e vice-versa. Inclusive, é comum que quando mais de duas aplicações se comunicam, uma delas exerça o papel de servidor enquanto as outras atuem como clientes. Para utilizar um *socket* é necessário definir duas características independentes: seu tipo e o domínio em que atua.

Existem dois diferentes tipos de *sockets*: de fluxo e de datagrama. *Sockets* de datagrama não utilizam conexões, cada mensagem é enviada separadamente através de pacotes, chamados de datagramas. A integridade das mensagens não é garantida; elas podem ser duplicadas, vir fora de ordem ou sequer chegar ao destino.

Sockets de fluxo obrigam o uso de conexões. Cada conexão provê um fluxo de dados bidirecional entre os participantes, em que a comunicação é contínua, sem separação entre as mensagens. Além disso, a comunicação é confiável, pois toda a integridade dos dados é mantida. Na impossibilidade de preservar a integridade da comunicação, a conexão é interrompida.

Por meio dos *sockets*, aplicações podem se comunicar em dois diferentes domínios: através da rede (*Internet domain)* ou apenas entre processos em uma mesma máquina (*UNIX domain)*. No âmbito da rede, a comunicação exige um protocolo de rede. Normalmente, os *sockets* de fluxo fazem uso do protocolo TCP (*Transfer Control Protocol*) que mantêm as condições de integridade da comunicação ao custo do aumento de sua latência. Já o *socket* de datagrama utiliza o protocolo UDP (*User Datagram Protocol)* que, em comparação com o protocolo TCP, proporciona uma comunicação mais rápida, porém, menos confiável.

Embora os *sockets* de fluxo e de datagrama, quando no *Internet domain*, possuem um contraste de velocidade e confiança da comunicação, isso não ocorre quando restringidos ao *UNIX domain*. Nesse domínio, ambos os tipos de *sockets* operam a velocidades semelhantes e, além disso, é sempre garantido que as mensagens serão entregues integralmente ao destino (KERRISK, 2010, p. 1171).

3. MATERIAIS E MÉTODOS

Como é visto na seção 2.2.2, o SLEUTH efetua a calibragem por força bruta. Esse processo é ineficiente, pois existe uma vasta quantidade de combinações diferentes entre os valores dos coeficientes, e mesmo que adotada uma estratégia que restrinja o domínio dos coeficientes, é possível, e muitas vezes provável, que uma única calibragem não encontre coeficientes razoáveis. Este capítulo pretende substituir o método de força bruta por um algoritmo de otimização adequado, e, posteriormente, elaborar experimentos que possam avaliar essas modificações.

3.1 OTIMIZAÇÃO DO SLEUTH

3.1.1 Princípios das Modificações

Primeiramente, os laços aninhados responsáveis pela força bruta deverão ser removidos e substituídos por alguma estrutura equivalente. Contudo, como é indesejável que o funcionamento e desempenho do simulador seja prejudicado de alguma forma, as modificações devem seguir princípios de coesão e acoplamento. Em vista disso, foi implantado no SLEUTH um mecanismo de comunicação entre processos, de modo que o simulador receba de outro processo os valores dos coeficientes e devolva para ele os dados estatísticos resultantes da iteração de calibragem. A figura 11 ilustra a comunicação entre SLEUTH e um outro processo denominado Otimizador.

Figura 11 – Diagrama da comunicação entre SLEUTH e o Otimizador

Fonte: Autoria própria

Os processos serão executados paralelamente, com diferentes espaços de memória virtual, e poucas modificações no código fonte do SLEUTH, assim evitando que seu comportamento seja alterado. Ainda, essa solução não limita o algoritmo de otimização a ser escrito na mesma linguagem do simulador, em C, possibilitando utilização de outras linguagens, como C++, Java e Python, que contemplam várias bibliotecas de otimização.

3.1.2 Interface de Comunicação

Uma vez que ambos os processos estão em uma plataforma Linux, diferentes formas de comunicação entre processos são possíveis. Cada meio de comunicação tem vantagens e desvantagens que justificam as ocasiões que são aplicadas. Dessa forma, o Otimizador e o SLEUTH se comunicam através de *sockets* restringidos ao *Unix domain*. Entre os meios de comunicação entre processos, *sockets* possuem um considerável nível de abstração, são bidirecionais, possuem desempenho equiparável e podem ser escalados para a comunicação entre vários processos.

Não é interessante que as chamadas ao sistema operacional e a lógica de conexão dos *sockets* estejam diretamente no código do SLEUTH. Por conta disso, foi criado um modulo chamado *socket interface*, responsável por abstrair essas funcionalidades. O código abaixo apresenta seu cabeçalho.

Figura 12 – Cabeçalho do socket interface

```
#ifndef SOCKET_INTERFACE_H
#define SOCKET_INTERFACE_H

typedef struct sock_type *sock_t;

extern sock_t server_init();
extern sock_t client_init();
extern void sock_close(sock_t sock);
extern int sock_send(sock_t sock, const void *data, size_t size);
extern int sock_recv(sock_t sock, void *dest, size_t limit);

#endif
```

Fonte: Autoria própria

O tipo *sock_t* representa uma conexão qualquer, proveniente tanto de um cliente, quanto de um servidor, e sua estrutura interna não é definida no cabeçalho, pois assim seus

membros são encapsulados. A função *server_init* deve iniciar um servidor *socket* e, em seguida, aceitar a conexão com algum cliente. Já a função *cliente_init* estabelece a conexão com um servidor. As funções *sock_send* e *sock_recv* transmitem e recebem dados em uma conexão, respectivamente. Por fim, *sock_close* libera todos os recursos do *socket*.

A figura 13 apresenta o procedimento de calibragem com o uso do módulo da figura 12.

Figura 13 – Modificações no SLEUTH

```
/* CALIBRATION AND TEST RUNS */
sock_t sock = client_init();

if (sock == NULL) {
  printf("Can't connect to server\n");
  exit(0);
}

proc_SetStopYear (igrid_GetUrbanYear (igrid_GetUrbanCount () - 1));

for (;;) {
  int coeffs[5];

  if (!sock_recv(sock, coeffs, sizeof(coeffs))) {
    sock_close(sock);
    break;
  }

  diffusion_coeff = coeffs[0];
  breed_coeff = coeffs[1];
  spread_coeff = coeffs[2];
  slope_resistance = coeffs[3];
  road_gravity = coeffs[4];

  InitRandom (scen_GetRandomSeed ());
  drv_driver ();
  sock_send(sock, latestControlStats, sizeof(latestControlStats));

  // resto do código omitido!
}
```

Fonte: Autoria própria

Com essas modificações, o SLEUTH se comporta como um oráculo, recebendo um conjunto discreto de valores dos coeficientes e devolvendo dados estatísticos que, através de alguma métrica, resultam em um único valor real. É esperado que essas fossem as únicas informações extraíveis do SLEUTH, pois se trata de um simulador com comportamento

complexo e, logo, não há uma relação clara entre as variáveis de entrada e o resultado da métrica de avaliação.

3.1.3 Desenvolvimento do Otimizador

Como visto anteriormente, o algoritmo de otimização vai residir em uma aplicação diferente do simulador, apelidado de Otimizador. Para a construção dessa aplicação foi utilizado o NOMAD, uma biblioteca de otimização para problemas não lineares de caixa preta. A biblioteca visa encontrar as soluções com um número reduzido de execuções da função objetivo, dado que utiliza o algoritmo MADS (descrito na seção 2.1.10.1) internamente.

O NOMAD permite que o espaço de busca possua restrições e também que seja constituído de variáveis binárias, inteiras, contínuas e categóricas. Todos esses atributos estão em harmonia com as características da calibragem do SLEUTH, pois se trata de um problema de caixa preta, com coeficientes inteiros e restringidos ao intervalo de 0 á 100.

A figura 14 representa o código em C++ utilizado para a criação do objeto *Parameters* do NOMAD com as características do SLEUTH.

Figura 14 - Trecho de código de configuração do *Parameters*

```
Parameters params;

// os 5 coeficientes do Sleuth
params.set_DIMENSION(5);

// todos os coeficientes são do tipo inteiro
for (int i = 0; i < 5; i++)
    params.set_BB_INPUT_TYPE(i, INTEGER);

// apenas uma dimensão de saída
vector<bb_output_type> bbot(1);
bbot[0] = OBJ;
params.set_BB_OUTPUT_TYPE(bbot);

// ponto inicial
params.set_X0(Point(5, 0.0));

// limites inferiores e superiores de cada coeficiente
params.set_LOWER_BOUND(Point(5, 0));
params.set_UPPER_BOUND(Point(5, 100));
```

Fonte: Autoria própria

Podemos observar ainda na figura 6 (seção 2.1.10.1.1) a necessidade de elaboração de alguma classe filha de *Evaluator*. Isto posto, a classe *SleuthEvaluator* foi criada para esse propósito. A figura 15 exibe por completo seu código.

Figura 15 - Código do *SleuthEvaluator*

```
class SleuthEvaluator : public Evaluator {
  private:
    sock_t sock;

  public:
    SleuthEvaluator(const Parameters &p) : Evaluator(p) {
      this->sock = server_init();

      if (this->sock == NULL) {
        throw runtime_error("Connection failed");
      }
    }

    bool eval_x(Eval_Point &x, const Double &h_max, bool &count_eval) const {
      Double y;
      int coeffs[5];
      double result[13];

      for (int i = 0; i < 5; i++)
        coeffs[i] = (int) x[i].value();

      if (!sock_send(this->sock, coeffs, sizeof(coeffs))) {
        throw runtime_error("Fail to send coefficients to SLEUTH");
      }

      if (!sock_recv(this->sock, result, sizeof(result))) {
        throw runtime_error("SLEUTH does not respond");
      }

      y = result[6]; // Métrica Lee-Salee

      // -y pois queremos buscar um máximo global; não, um mínimo
      x.set_bb_output(0, -y);

      count_eval = true;

      return true;
    }
};
```

Fonte: Autoria própria

O construtor de *SleuthEvaluator* inicia a conexão socket com o SLEUTH, que deve estar aguardando a conexão. A partir disso, o método *eval_x* pode enviar os coeficientes e receber os resultados estatísticos, o que caracteriza uma iteração de otimização. Importante

notar, também, que a métrica *Leesalee* foi utilizada. Porém, é possível substituir por outra métrica com pouco esforço, pois todos os dados estatísticos estão disponíveis no escopo do método.

Por fim, o código abaixo cria o objeto do tipo *Evaluator* e, em seguida, cria o objeto *Mads*. Com isso, o método *run* pode ser invocado para que, finalmente, as iterações de otimização sejam iniciadas.

Figura 16 - Trecho de código que inicia o MADS

```
SleuthEvaluator evaluator(params);

Mads mads(params, &evaluator);
mads.run();
```

Fonte: Autoria própria

3.2 EXPERIMENTOS

3.2.1 Descrição dos Experimentos

Para avaliar o desempenho do SLEUTH modificado, que aqui será denominado SLEUTH NOMAD, foram realizados experimentos em três estratégias diferentes de calibragem: com força bruta, com algoritmos genéticos e o com o NOMAD. Essas três versões são calibradas com a mesma região alvo: a cidade de Ponta Grossa, no Paraná. Os arquivos de dados utilizados no experimento são os mesmos empregados por Roth (2019); a dissertação está disponível eletronicamente e descreve todos os dados, motivo pelo qual essa informação não será repetida aqui. Além disso, os parâmetros internos de configuração do simulador são idênticos em todas as execuções e, desse modo, o mesmo grupo de coeficientes exerce o mesmo resultado independente da variação.

As três fases de calibragem (*coarse, fine* e *final*) foram desempenhadas na versão com força bruta (SLEUTH clássico). No SLEUTH GA, a configuração padrão do algoritmo genético foi mantida: população com 55 indivíduos, taxa de mutação a 13% e 100 gerações.

A métrica escolhida para avaliar as calibragens foi a *Optimum SLEUTH Metric*. No entanto, outras métricas podem ser utilizadas com pouco esforço.

3.2.2 Estratégia Comparativa

Neste trabalho, o principal fator tido para definir a qualidade do processo de calibragem é a relação entre a quantidade total de iterações e os melhores coeficientes encontrados nessas iterações, pois com isso é possível analisar quais estratégias de calibragem se adequam melhor a diferentes cenários de uso do simulador.

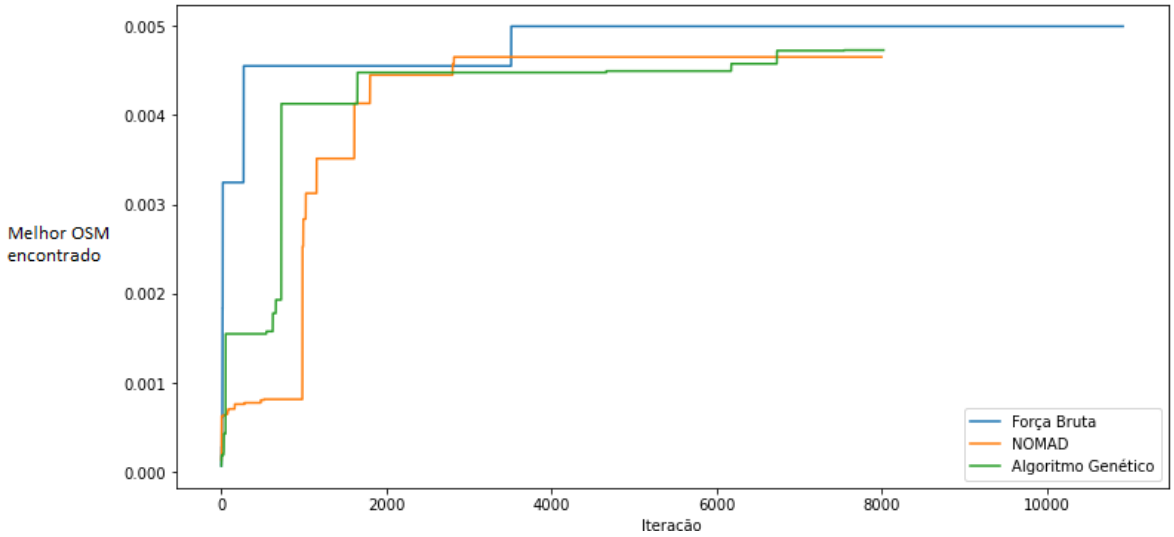
Outro aspecto interessante a ser observado é a quantidade e dispersão de soluções boas produzidas na calibragem. Coeficientes que possuem métricas boas, mas não necessariamente ótimas, podem ser avaliados de forma qualitativa a partir da análise das imagens produzidas pelas predições com esses coeficientes.

4. RESULTADOS

Para realizar os experimentos, a calibragem do software SLEUTH foi executada de três maneiras: usando força bruta para testar combinações de parâmetros (de acordo com a metodologia das três fases de calibragem); usando a versão SLEUTH equipada com um algoritmo genético; e finalmente usando a versão SLEUTH modificada neste trabalho. Em cada um desses experimentos foram gerados arquivos de log. A partir desses dados pode-se obter a métrica OSM, que oferece uma medida de distância entre o resultado calculado pelo simulador, e um mapa real da cidade para um determinado período. Quanto maior o valor dessa métrica, melhor terá sido o mapa calculado pelo SLEUTH. Os parâmetros que correspondem ao melhor valor de OSM podem ser usados, no simulador, para calcular uma previsão de futuro do crescimento da cidade.

A calibragem com a estratégia de força bruta finalizou com um total de 10.925 iterações; as calibragens com uso o NOMAD e com os algoritmos genéticos, com aproximadamente 8.000 iterações, cada um. O gráfico abaixo demonstra a métrica do melhor grupo de coeficientes encontrados ao longo das iterações (i.e. o máximo acumulado do OSM).

Figura 17 – Evolução dos métodos de calibragem

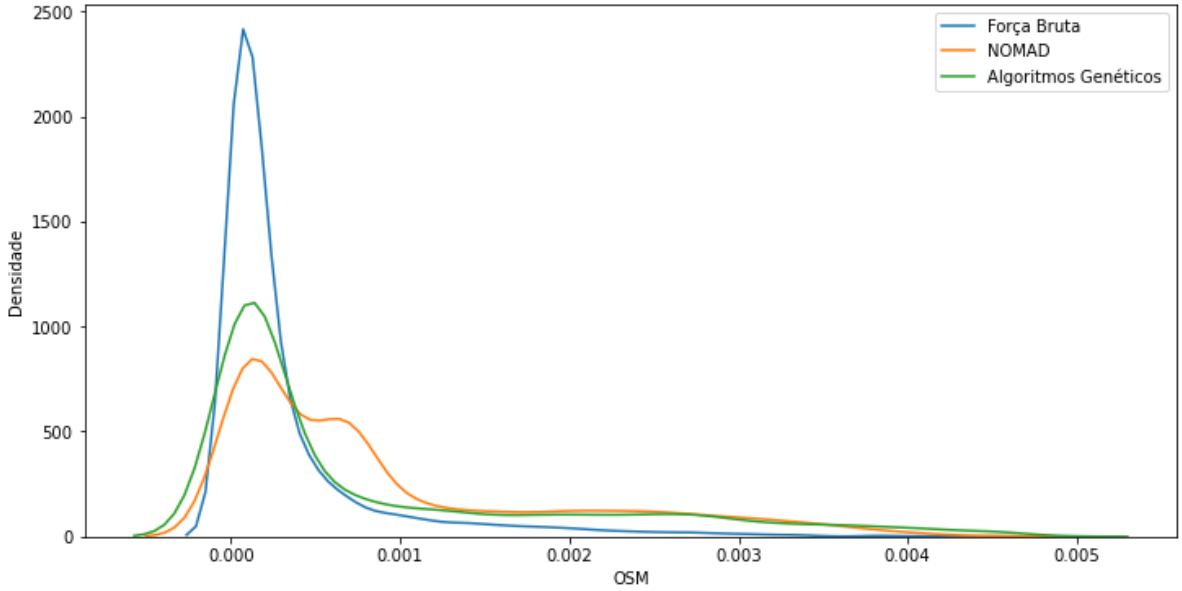


Fonte: Autoria própria

Embora a métrica de calibragem seja a fonte primária de qualidade dos coeficientes, ela pode não ser suficiente para definir quais coeficientes melhor representam a urbanização da região. Alguns autores optam por ordenar os coeficientes de acordo com a métrica e, então, avaliar algumas das melhores soluções de maneira visual a partir de predições no SLEUTH. Para que essa abordagem seja viável, é interessante que a calibragem produza diversas

soluções com alto valor da métrica. A figura 18 compara a função densidade de probabilidade entre as soluções encontradas por cada estratégia de calibragem de acordo com a métrica OSM. Assim, as estratégias mais interessantes são aquelas que apresentam uma concentração maior de soluções com um valor alto da métrica.

Figura 18 – Comparação de densidade das soluções pela métrica OSM



Fonte: Autoria própria

Outro aspecto relevante a ser analisado é a quantidade de iterações que avaliam coeficientes que já tenham sido avaliados anteriormente. A tabela a seguir mostra a razão entre as iterações repetidas pelo total de iterações da calibragem. Importante salientar que a força bruta repete coeficientes devido à estratégia das três fases (*coarse, fine* e *final*) descritos na seção 2.2.2.

Tabela 1 – Comparação entre as taxa de soluções repetidas

	Total de iterações	**Iterações repetidas**	**Razão**
Força Bruta	10.925	130	1,19%
Algoritmo Genético	8.025	1.264	15,75%
NOMAD	8.000	16	0,2%

Fonte: Autoria própria

5. CONSIDERAÇÕES FINAIS

Na literatura, diversas tentativas de diminuir o extenso custo de calibragem do SLEUTH foram praticadas, havendo uma ênfase em metodologias de força bruta e algoritmos genéticos. Este trabalho propôs uma abordagem diferente a esse problema: o algoritmo *Mesh Adaptive Direct Search* juntamente com o *Variable Neighborhood Search.* Essa aplicação foi experimentada em simulações sobre a cidade de Ponta Grossa, e os resultados sugerem que, entre as três estratégias estudadas, a calibragem por força bruta obteve o melhor desempenho na tarefa de encontrar a melhor solução (com exceção da faixa de 2.803 a 3.511 iterações, em que o SLEUTH NOMAD obteve coeficientes ligeiramente superiores). Ainda no mesmo experimento, o algoritmo genético foi consideravelmente superior ao MADS nas primeiras iterações; todavia, o desempenho de ambos se tornou equiparável a partir de 1612 iterações.

Há uma concentração maior de soluções inferiores no resultado do método de força bruta; os algoritmos de otimização, em contrapartida, oferecem mais variedade de coeficientes razoáveis. No entanto, de acordo com o experimento realizado, o algoritmo genético tende a repetir uma quantia considerável de coeficientes, o que não é interessante, pois causa trabalho desnecessário ao simulador. Esse comportamento é infrequente no SLEUTH NOMAD.

5.1 TRABALHOS FUTUROS

Evidentemente, os resultados deste trabalho refletem um único experimento realizado; conclusões melhor fundamentadas exigem uma ampla base experimental. Avaliar o desempenho da estratégia proposta em aplicações já documentadas do SLEUTH é um caminho interessante a se seguir.

Ademais, o módulo de otimização foi elaborado de forma a permitir que outros métodos de otimização fossem facilmente aplicados. Inclusive, em consequência do uso de *sockets* na comunicação entre o otimizador e o SLEUTH, é viável aplicar um algoritmo paralelo de otimização e dividir a tarefa de calibragem entre vários computadores através da rede. Anteriormente, esse privilégio era exclusivo a estratégia de força bruta.

REFERÊNCIAS

AMARAN, Satyajith et al. Simulation optimization: a review of algorithms and applications. **Annals of Operations Research**, 240, n. 1, p. 351-380, set. 2015.

AUDET, Charles; BÉCHARD, Vincent; LE DIGABEL, Sébastien. Nonsmooth optimization through mesh adaptive direct search and variable neighborhood search. **Journal of Global Optimization**, v. 41, n. 2, p. 299-318, 2008.

AUDET, Charles; DENNIS JR, John E. Mesh adaptive direct search algorithms for constrained optimization. **SIAM Journal on optimization**, v. 17, n. 1, p. 188-217, 2006.

AUDET, Charles; HARE, Warren. Derivative-free and blackbox optimization. 2017.

AUDET, Charles; LE DIGABEL, Sébastien; TRIBES, Christophe. The mesh adaptive direct search algorithm for granular and discrete variables. **SIAM Journal on Optimization**, v. 29, n. 2, p. 1164-1189, 2019.

AUTRIQUE, Laurent; LEYRIS, Jean Pierre; DE CURSI, Eduardo Souza. A genetic algorithm for global constrained optimization problem. **IFAC Proceedings Volumes**, v. 32, n. 2, p. 367-372, 1999.

BEYER, Hans-Georg; SENDHOFF, Bernhard. Robust optimization–a comprehensive survey. **Computer methods in applied mechanics and engineering**, v. 196, n. 33-34, p. 3190-3218, 2007.

CLARKE, Keith C. Cellular automata and agent-based models**. Handbook of regional science**, p. 1217-1233, 2014.

CLARKE, Keith C.; GAYDOS, Leonard J. Loose-coupling a cellular automaton model and GIS: long-term urban growth prediction for San Francisco and Washington/Baltimore. **International journal of geographical information science**, v. 12, n. 7, p. 699-714, 1998.

CLARKE, Keith C. Improving SLEUTH calibration with a genetic algorithm. **International Workshop on Geomatic Approaches for Modelling Land Change Scenarios**. SCITEPRESS, 2017. p. 319-326.

CLARKE-LAUER, M. D.; CLARKE, Keith C. Evolving simulation modeling: Calibrating SLEUTH using a genetic algorithm. **Proceedings of the 11th International Conference on GeoComputation**, London, UK. 2011. p. 20-22.

DE CURSI, JE Souza; ELLAIA, R.; BOUHADI, M. Global optimization under nonlinear restrictions by using stochastic perturbations of the projected gradient. In: Frontiers in Global Optimization. **Springer**, Boston, v.7, p. 541-561, 2004.

DIETZEL, Charles; CLARKE, Keith C. Toward optimal calibration of the SLEUTH land use change model. **Transactions in GIS**, Santa Bárbara, v. 11, n. 1, p. 29-45, 2007.

FRÉVILLE, Arnaud. The multidimensional 0–1 knapsack problem: An overview. **European Journal of Operational Research**, v. 155, n. 1, p. 1-21, 2004.

GALVIN, Peter B. et al. **Operating system concepts**. John Wiley & Sons, 2003.

GOLDSTEIN, Noah C. Brains versus brawn-comparative strategies for the calibration of a cellular automata-based urban growth model. **GeoDynamics**, p. 249-272, 2004.

HORNBY, Gregory et al. Automated antenna design with evolutionary algorithms. **American Institute of Aeronautics and Astronautics**, San Jose, p. 7242, set. 2006.

JAFARNEZHAD, Javad; SALMANMAHINY, Abdolrassoul; SAKIEH, Yousef. Subjectivity versus objectivity: comparative study between brute force method and genetic algorithm for calibrating the SLEUTH urban growth model. **Journal of Urban Planning and Development**, San Diego, v. 142, n. 3, p. 05015015, 2016.

JEYAKUMAR, Vaithilingam; RUBINOV, Alexander M. (Ed.). Continuous optimization: current trends and modern applications. **Springer Science & Business Media**, 2006.

KOLDA, Tamara G.; LEWIS, Robert Michael; TORCZON, Virginia. Optimization by direct search: New perspectives on some classical and modern methods. **SIAM review**, v. 45, n. 3, p. 385-482, 2003.

KERRISK, Michael. The Linux programming interface: a Linux and UNIX system programming handbook**. No Starch Press**, 2010.

KIRKPATRICK, Scott; GELATT, C. Daniel; VECCHI, Mario P. Optimization by simulated annealing. **Science**, v. 220, n. 4598, p. 671-680, maio. 1983.

LE DIGABEL, Sébastien. Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. **ACM Transactions on Mathematical Software (TOMS)**, v. 37, n. 4, p. 1-15, 2011.

LEE, Jon. A first course in combinatorial optimization. **Cambridge University Press**, 2004.

LIN, Ming-Hua; TSAI, Jung-Fa; YU, Chian-Son. A review of deterministic optimization methods in engineering and management. **Mathematical Problems in Engineering**, Hindawi, v. 2012, abr. 2012.

NACHTIGALL, Matheus Garcia. **Têmpera simulada aplicada no mapeamento tecnológico de FPGAs baseadas em LUTs**. 2015. Dissertação de Mestrado. Universidade Federal de Pelotas.

PIRLOT, Marc; VIDAL, Rene Victor Valqui. Simulated annealing: A tutorial. **Control and Cybernetics**, v. 25, p. 9-32, 1996.

RAFIEE, Reza et al. Simulating urban growth in Mashad City, Iran through the SLEUTH model (UGM). **Cities**, v. 26, n. 1, p. 19-26, 2009.

ROTH, Ellen Cristina Wolf. **Urban growth forecast using segmented and complete maps with the SLEUTH simulator**. 2019. Dissertação de Mestrado. Universidade Tecnológica Federal do Paraná.

SÖRENSEN, Kenneth. Metaheuristics—the metaphor exposed. **International Transactions in Operational Research**, v. 22, n. 1, p. 3-18, 2015.

SUMATA, Hiroshi et al. A comparison between gradient descent and stochastic approaches for parameter optimization of a sea ice model. **Ocean Science**, v. 9, n. 4, p. 609-630, 2013.

WEISE, Thomas. Global optimization algorithms-theory and application. **Self-Published**. Thomas Weise, 2009.