UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA MECÂNICA
ENGENHARIA AERONÁUTICA

Uma Abordagem Distribuída Aplicada ao Algoritmo de Evolução Diferencial

Autor:
Daniel Ribeiro Santiago

Uberlândia
2024

DANIEL RIBEIRO SANTIAGO

Uma Abordagem Distribuída Aplicada ao Algoritmo de Evolução Diferencial

Trabalho de Conclusão de Curso apresentado à Faculdade de Engenharia Mecânica da Universidade Federal de Uberlândia como parte dos requisitos para obtenção do título de Bacharel em Engenharia Aeronáutica.

Orientador: Prof. Dr. Aristeu da Silveira Neto
Coorientador: Prof. Dr. Fran Sergio Lobato

Banca Avaliadora:
Aldemir Aparecido Cavallini Junior
Tobias Souza Morais

Uberlândia

2024

AGRADECIMENTOS

Gostaria de agradecer à todas as pessoas que tornaram o desenvolvimento desse trabalho e da minha trajetória possíveis. Ao meu pai Lúcio que sempre despertou minha curiosidade por temas científicos e minha mãe Adriana por ser meu modelo de ser humano a ser seguido, à minha vó Irma por ter me acolhido tanto e aos meus irmãos, Rafael que sempre compartilhou comigo o seu interesse por assuntos de engenharia e computação e Gabriel pela sua criatividade inspiradora.

Também gostaria de agradecer imensamente aos meus amigos Iago, Cunha, Alfredo, Isis, Alves, Torezan, Freitas e Phillemon que me acompanharam durante a jornada da graduação e que tornaram essa fase tão leve e especial.

Meus sinceros agradecimentos a equipe Tucano Aerodesing por ter proporcionado tantos aprendizados e desafios e por ter me impulsionado e motivado tanto em vários aspectos, sem ela esse trabalho não existiria.

Esse trabalho não seria possível sem a ajuda de meu professor orientador Aristeu da Silvera Neto e o professor Fran Sergio Lobato que atuou como coorientador, sempre fornecendo instruções valiosas e sem o qual não conseguiria completar este projeto.

RESUMO

Meta-heurísticas baseadas em população estão entre as formas mais populares de resolução de problemas de otimização em larga escala, se destacando pela sua capacidade de exploração e aproveitamento do espaço de busca. Esses algoritmos também são conhecidos pela sua facilidade de paralelização, que reduzem drasticamente o tempo necessário para encontrar soluções viáveis. No entanto as implementações de versões paralelas e distribuídas podem ser pouco flexíveis quanto a forma que a função objetivo deve ser fornecida, muitas vezes forçando o usuário a escreve-la em uma linguagem de programação específica.

Esse trabalho propõe a implementação um algoritmo de evolução diferencial distribuído utilizando o modelo mestre-escravo, de forma a maximizar a flexibilidade de implementação da função objetivo e contornar possíveis falhas durante a avaliação dos indivíduos sem retardar o processo. A aplicação desenvolvida funciona como um servidor HTTP, fornecendo uma API REST para a interação com os nós trabalhadores e persiste os resultados em disco, a tornando tolerante a desligamentos e reinicializações. A metodologia foi aplicada à função de rastrigin e a um problema de ajuste de uma curva por uma spline de bézier suave e, a partir dos resultados obtidos, foi demonstrado o ganho de performance da estratégia e sua capacidade de convergência para um resultado satisfatório.

Palavras Chave: otimização, meta-heurística, evolução diferencial, distribuído.

ABSTRACT

Population-based metaheuristics are among the most popular ways of solving large-scale optimization problems, standing out for their ability to explore and take advantage of the search space. These algorithms are also known for their ease of parallelization, which drastically reduces the time needed to find viable solutions. However, implementations of parallel and distributed versions can be inflexible as to how the objective function should be provided, often forcing the user to write it in a specific programming language.

This work proposes the implementation of a distributed differential evolution algorithm using the master-slave model, in order to maximize the flexibility of implementation of the objective function and circumvent possible failures during the evaluation of individuals without slowing down the process. The application developed works as an HTTP server, providing a REST API for interaction with the worker nodes and persists the results on disk, making it tolerant of shutdowns and reboots. The methodology was applied to the rastrigin function and to a curve fitting problem using a smooth bézier spline and, based on the results obtained, the strategy's performance gain and its ability to converge to a satisfactory result were demonstrated.

Keywords: optimization, differential evolution, distributed.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

A otimização é um processo central para problemas envolvendo tomada de decisões e é de grande interesse para todas às áreas de engenharia. O processo de tomada de decisão envolve a escolha de uma solução entre várias possibilidades e é movida pelo nosso desejo de tomar a "melhor" decisão a disposição [8]. A medida de quão boa é uma decisão pode ser descrita por uma função objetivo matemática, muitas vezes chamada de função erro ou função de aptidão, que em geral é moldada para que apresente um valor menor quanto melhor a solução fornecida como entrada [28].

O problema de otimização pode então ser descrito como a busca de uma solução viável $\boldsymbol{x}^*$ que quando aplicada a função objetivo $f(\boldsymbol{x})$ retorne o menor valor possível.

A otimização está presente em muitos problemas reais de engenharia. Wu e Tseng em [41] utilizou o algoritmo de evolução diferencial para realizar uma otimização de estruturas treliçadas, He et al. em [16] utilizou um método de programação quadrática sequencial em conjunto com simulações de dinâmica dos fluidos computacional (*Computational Fluid Dynamics* - CFD) para otimizar a geometria de um aerofólio de forma robusta partindo de um círculo, Abu Abed em [1] utilizou de dados obtidos por sensores para aprimorar os resultados de uma simulação, selecionando os parâmetros através de um algoritmo de Evolução Diferencial de forma a diminuir o erro entre a resposta apresentada pelo modelo e a realidade, apenas para citar alguns casos.

O processo para se obter uma solução exata para um problema de otimização é, muitas vezes, computacionalmente proibitivo. Na prática, é comum aceitar uma solução considerada suficientemente boa para o problema. Essa solução pode ser obtida por métodos de *otimização aproximada*, que se dividem em duas classes principais: heurísticas específicas e meta-heurísticas [36, p.xvii].

Heurísticas específicas são algoritmos que utilizam princípios simples para obter uma solução aproximada sub-ótima em um menor tempo. Um exemplo de heurísticas específicas são os algoritmos gananciosos, que buscam o melhor resultado global fazendo as melhores escolhas locais a cada etapa [36, p.26].

A palavra *heurística* tem origem na palavra grega *heuriskein*, que significa a arte de descobrir novas estratégias para resolver problemas, já o sufixo *meta* do grego significa metodologia de nível superior. O termo meta-heurística foi introduzido por Glover no seu artigo [14] em 1986.

Apesar de úteis, heurísticas específicas tem seu uso limitado apenas à alguns problemas. Meta-heurísticas, por outro lado, são formas generalizadas de se aplicar heurísticas para problemas de otimização, de forma que são necessárias poucas suposições sobre o problema a ser resolvido para que possam ser utilizadas.

De acordo com Gendreau, Potvin et al. [13], meta-heurísticas são métodos que orquestram uma interação entre procedimentos de melhoras locais e estratégias de alto nível capazes de escapar de ótimos locais e efetuar uma procura robusta na região de busca de soluções.

Existem vários critérios de classificação que podem ser aplicados aos algoritmos de meta-heurísticas [36, p.25]. A Figura 1 da um panorama de algumas meta-heurísticas mais utilizadas suas formas de classificação. Dentre os critérios de classificação existentes dois são de maior interesse para este trabalho:

Meta-heurísticas determinísticas VS estocásticas: Meta-heurísticas determinísticas solucionam um problema de otimização tomando decisões determinísticas, portanto ao executa-las com as mesmas condições iniciais os mesmos resultados finais são obtidos. Já meta-heurísticas estocásticas tomam decisões baseadas em regras aleatórias e portanto execuções com as mesmas condições iniciais podem levar a resultados finais distintos.

Meta-heurísticas de solução única VS baseadas em população: Meta-heurísticas de solução única manipulam e transformam uma única solução durante sua busca e podem ser vistas como "caminhadas guiadas" pelas vizinhanças. Já meta-heurísticas baseadas em população manipulam uma população de soluções que evoluem durante o processo de busca e podem ser vistas como uma melhora iterativa em uma população de soluções.

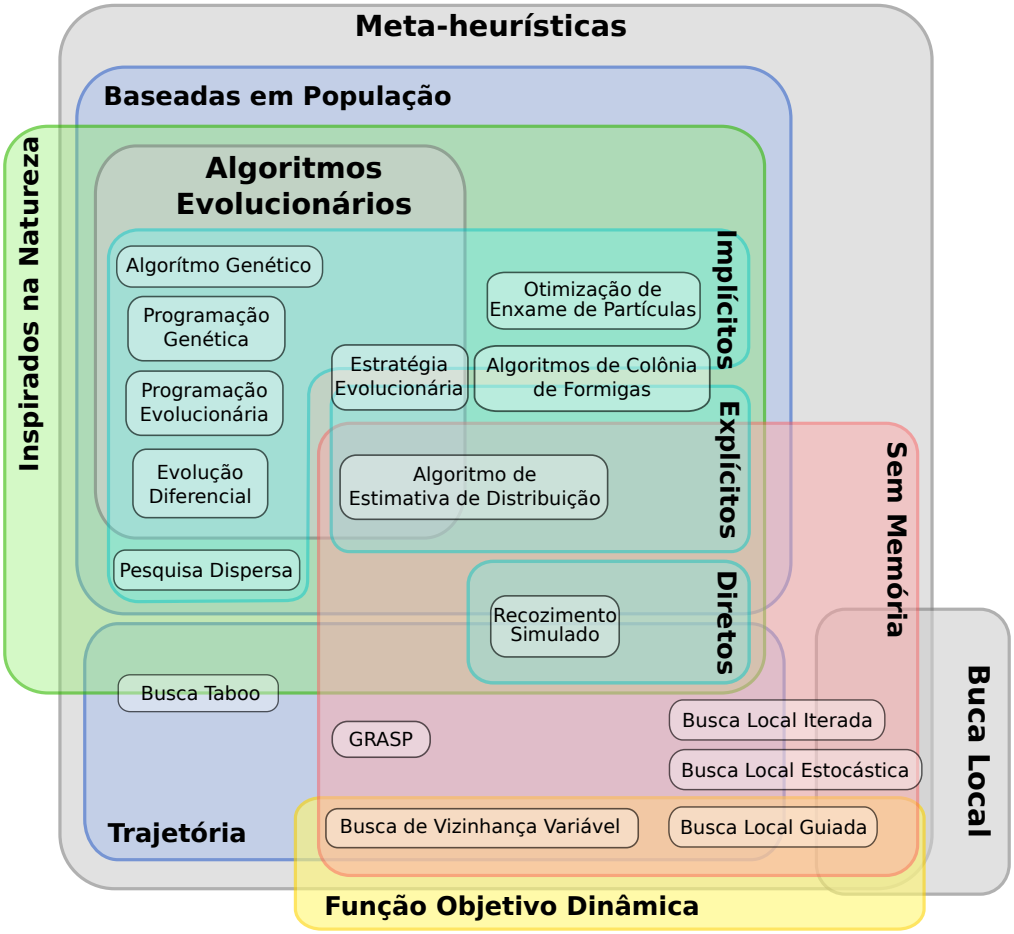

Figura 1: Classificações Meta-heurísticas.

Meta-heurísticas baseadas em população estão entre as formas mais populares de resolução de problemas de otimização em larga escala. Entre alguns dos algoritmos mais conhecidos dessa categoria estão a otimização de colonia de formigas (*Ant Colony Optimization - ACO*) desenvolvido por Kirkpatrick, Gelatt e Vecchi em [21] e a otimização de exame de partículas (*Particle Swarm Optimization - PSO*) desenvolvido por Kennedy e Eberhart em [20].

Dentre as meta-heurísticas baseadas em população existem alguns métodos que se inspiram no processo de evolução denominados algoritmos evolucionários (AE) como o algoritmo

genético (*Genetic Algorithm - GA*) popularizado por Holland em [18] e o algoritmo de evolução diferencial (*Differential Evolution - DE*) proposto por Storn e Price em [34].

Nesse tipo de abordagem cada possível solução é tida como um indivíduo de uma população que deve ter sua aptidão calculada. Após calculadas as aptidões de todos os indivíduos da população é possível utilizar essas informações para gerar a população seguinte e assim dar continuidade ao processo.

Em grande parte, a popularidade desses algoritmos surge da facilidade em que eles podem ser aplicados aos problemas de otimização, visto que diferente de outros métodos comumente usados não é necessário calcular o gradiente da função objetivo ou ser capaz de formular o problema com uma notação matemática analítica e desambigua. Na realidade muitos problemas podem ser considerados como *caixas pretas*.

No entanto, o uso dessas metodologias requerem múltiplas avaliações dos indivíduos, que podem ser computacionalmente custosas, especialmente em problemas reais onde muitas vezes são necessárias simulações para calcular o valor da função objetivo. Portanto, é de grande interesse que esses algoritmos possam ser implementados de forma eficiente e distribuída, reduzindo o tempo computacional necessário para encontrar uma solução.

Diferentes versões do algoritmo de Evolução Diferencial (ED) aparecem entre os ganhadores das competições IEEE CEC (em inglês *Congress on Evolucionary Computation*) ao longo dos anos [25], demonstrando sua versatilidade na resolução de problemas de otimização envolvendo variáveis contínuas. Por esta razão, este trabalho dará enfase a implementação deste algoritmo.

1.1 Objetivo

O objetivo desse trabalho é implementação de um algoritmo de ED distribuído utilizando o modelo de mestre-escravo. O algoritmo tem como requisito tornar o desenvolvimento da função objetivo por parte de seus usuários o mais flexível possível, não impondo nenhuma linguagem de programação específica para sua escrita, ao mesmo tempo que busca ser tolerante a falhas durante o processo de avaliação dos indivíduos, evitando que o processo seja desacelerado significativamente caso um nó trabalhador apresente erros.

A implementação deve permitir que nós trabalhadores sejam adicionados e removidos dinamicamente. Também é de interesse que o método utilizado seja capaz de se recuperar caso o nó mestre sofra alguma falha, e que seja capaz de salvar os resultados das avaliações realizadas em disco para que sejam analisadas posteriormente.

1.2 Estruturação

Este trabalho está estruturado como segue. Na Seção 2 será feita uma breve explicação sobre conceitos de otimização, uma introdução ao conceito de meta-heurísticas e algoritmos evolucionários, uma descrição em detalhes o método de ED e o estado da arte no que diz respeito a algoritmos evolucionários distribuídos. Em seguida, na Seção 3, serão descritos os detalhes da implementação do algoritmo desenvolvido. Na Seção 4 dois problemas serão propostos e utilizados para validar a capacidade de convergência e performance do método e os resultados obtidos serão analisados e discutidos. Por fim, na Seção 5, são apresentadas as

conclusões avaliando as vantagens e desvantagens da solução e sugerindo caminhos futuros para o trabalho.

2 Revisão Bibliográfica

Para compreender o modelo proposto é necessário recapitular alguns conceitos de otimização, meta-heurísticas, algoritmos evolucionários distribuídos e sistemas distribuídos.

2.1 O Problema de Otimização

Um problema de otimização pode ser classificado de diversas maneiras, quanto as características das variáveis de otimização, também chamadas de variáveis de decisão ou variáveis de projeto, e o número e funções objetivo que se deseja minimizar simultaneamente. Os problemas de otimização envolvendo variáveis contínuas são chamados de problemas de otimização contínua, já os problemas envolvendo variáveis discretas são conhecidos como problemas de otimização discreta. Quanto ao número de funções objetivo que se deseja minimizar simultaneamente temos que uma otimização que busca minimizar apenas uma função objetivo é chamada de otimização mono-objetivo, já problemas que buscam minimizar mais de uma função objetivo simultaneamente são chamadas de otimizações multi-objetivo [8, p.577-578].

Problemas de otimização também podem ser formulados de forma a incluir restrições quanto ao espaço de busca de soluções, ou seja, os valores que as variáveis de otimização podem assumir não incluem todo o conjunto $\mathbb{R}^n$, mas sim um subconjunto Ω de $\mathbb{R}^n$ que respeita uma série de equações e/ou inequações. O subconjunto que contém as soluções possíveis que respeitam as restrições impostas é chamado de conjunto viável ou espaço de busca. Problemas em que o espaço de busca é irrestrito são chamados de problemas de otimização irrestrita, enquanto problemas em que existem restrições no espaço de busca são chamados de problemas de otimização restrita [8, p.81-82].

Neste trabalho discutiremos exclusivamente problemas de otimização mono-objetivo de variáveis contínuas e restritas á um conjunto Ω convexo, que matematicamente pode ser descrito da seguinte forma:

$$\begin{aligned} minimize \quad & f(\boldsymbol{x}) \\ sujeito\ a \quad & \boldsymbol{x} \in \Omega \end{aligned}$$

Onde $f : \mathbb{R}^n \to \mathbb{R}$ é função objetivo que desejamos minimizar, $\boldsymbol{x} = [x_1, \ldots, x_n]$ é um vetor em que cada posição representa uma variável de otimização e Ω é um conjunto convexo contido em $\mathbb{R}^n$.

2.2 Meta-Heurísticas

De acordo com Talbi em [36], meta-heurísticas podem ser vistas como moldes genéricos usados como estratégias orientadoras de heurísticas subjacentes utilizadas no modelo para resolver problemas de otimização. Alguns princípios simples baseados nos valores obtidos por cada tentativa de solução são utilizados para guiar o processo para soluções melhores.

A estratégia de uma meta-heurística se baseia no equilíbrio entre dois aspectos contraditórios na busca de uma solução: a **exploração** do espaço de busca (diversificação) e o **aproveitamento** de regiões promissoras para o descobrimento de boas soluções (intensificação).

Na etapa de diversificação as regiões do espaço de busca não exploradas devem ser visitadas igualmente para garantir que a busca não esteja confinada a apenas algumas regiões. Esse comportamento ajuda a garantir que a solução apresentada não fique presa a mínimos locais.

É interessante notar que o espaço de busca aumenta exponencialmente com o número de variáveis de otimização, isso é, de dimensões. Podemos explorar um intervalo unitário em um espaço de busca unidimensional com por exemplo 10 pontos igualmente espaçados em uma linha de 0 a 1 (Figura 2). Se quisermos ter o mesmo efeito explorando um intervalo equivalente em um espaço bi-dimensional serão necessários $10^2 = 100$ pontos igualmente distribuídos em uma grade com cada combinação de valores de 0 a 1 das duas variáveis (Figura 3). Esse aumento exponencial no espaço de busca com o número de dimensões é chamado de **maldição da dimensionalidade**.

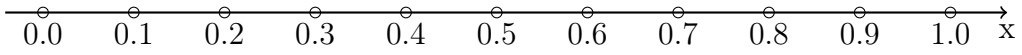

Figura 2: Pontos igualmente espaçados em um intervalo de 0 a 1.

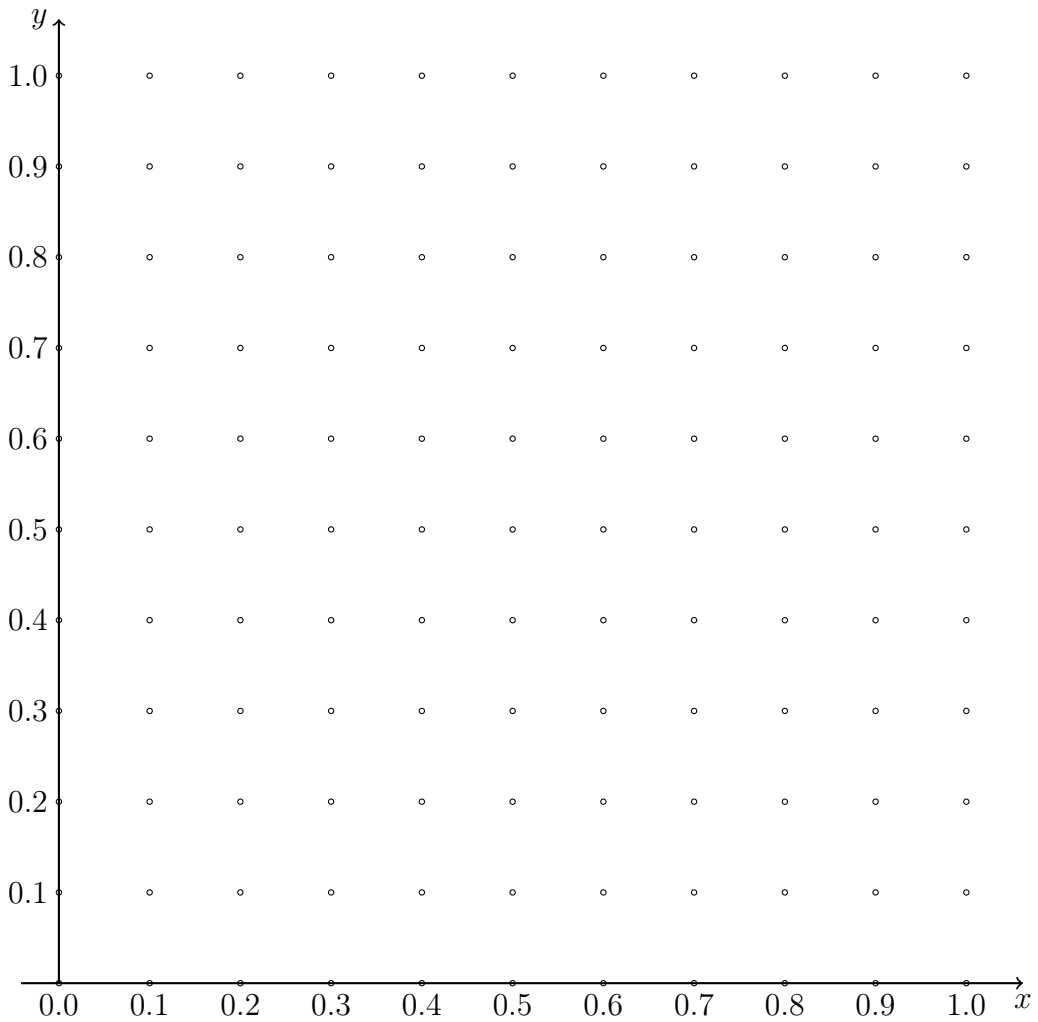

Figura 3: Pontos igualmente espaçados em um plano de 0 a 1.

Na etapa de intensificação regiões promissoras são exploradas mais rigorosamente na esperança de se encontrar melhores soluções. Um algoritmo que prioriza mais essa etapa em detrimento da exploração pode acabar preso em mínimos locais sub-ótimos[36, p.24].

Geralmente para que seja possível levar em conta essas características o algoritmo precisa de alguma forma de memória das soluções anteriores as quais ele já calculou o valor da função objetivo e, a partir destes resultados, tomar a melhor decisão para navegar no espaço de busca.

2.2.1 Algoritmos Evolucionários

Algoritmos Evolucionários (AE) são meta-heurísticas baseadas em população que se inspiram no processo de evolução das espécies, utilizando metáforas da biologia e genética no desenvolvimento de processos de otimização. As principais famílias de algoritmos evolucionários são os *algoritmos genéticos* [18], desenvolvido principalmente por J. H. Holland em 1967, *estratégias evolucionárias* [29, 30] proposto por um grupo de três estudantes, Bienert, Rochenberg e schwefel em 1965, *programação evolucionária* [11] desenvolvido por Lawrence Foge em 1966 e *programação genética* [22] desenvolvido por J.Koza na segunda metade de 1980.

Em geral AEs possuem os seguintes componentes em comum:

1. **Representação:** Na comunidade de AE, em especial a de algoritmos genéticos, a solução codificada é chamada de *cromossomo*, enquanto as variáveis de otimização em uma solução são chamadas de *genes*. A posição de um elemento em um cromossomo é chamada de *locus*.

2. **Inicialização da População:** Este é um componente comum em todas as meta-heurísticas baseadas em população. O objetivo deste componente em geral é garantir a maior diversidade possível dentro de um espaço viável de soluções.

3. **Função Objetivo:** Este é um componente comum à todas meta-heurísticas. Na comunidade de AE é usado o termo *aptidão* (*fitness*) para referenciar o valor obtido pelo cálculo da função objetivo em um indivíduo.

4. **Estratégia de Seleção:** A estratégia de seleção, ou seleção de parentes, é a forma como o AE dita quais serão os pais que gerarão descendentes para a próxima geração.

5. **Estratégia de Reprodução:** A estratégia de reprodução é o conjunto de operações de mutação e crossover desenvolvidos para a geração de novos indivíduos a partir de seus parentes.

6. **Estratégia de Substituição:** Os novos indivíduos competem com seus parentes para ocupar os lugares na próxima geração, garantindo a sobrevivência do mais bem adaptado. Este componente também pode se chamado de estratégia de seleção de sobreviventes.

7. **Critério de Parada:** Este é um componente comum a todas meta-heurísticas, no entanto alguns critérios de parada são específicos para meta-heurísticas baseadas em população, como a adoção de uma geração máxima.

O ciclo básico de um AE está representado na Figura 4 a seguir.

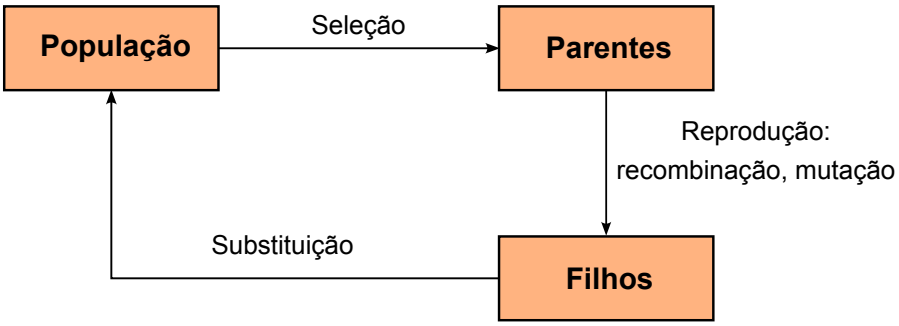

Figura 4: Ciclo de uma geração em um algoritmo evolucionário.

2.2.2 Evolução Diferencial

A evolução diferencial é uma meta-heurística baseada em população criada por Storn e Price em 1995 como um método de busca direta para resolução de problemas de otimização de variáveis reais. O método trata os indivíduos como vetores e utiliza da diferença entre eles para gerar perturbações que modificam a população em busca de novos resultados.

Por se tratar de um AE, o método possui os mesmos componentes citados na Subsubseção 2.2.1 que na versão base do método tem a seguinte forma:

Inicialização da População

A primeira geração é criada ao inicializar uma população de N_p indivíduos. Cada indivíduo $\boldsymbol{X}_{g,i}$ da geração g é tratado como um vetor de N_d elementos representando as variáveis de otimização (ver Equação 1).

$$\boldsymbol{X}_{g,i} = [x_{g,i,0}, x_{g,i,1}, \ldots, x_{g,i,j}, \ldots, x_{g,i,N_d-1}] \tag{1}$$

Os elementos $x_{0,i,j}$ de cada indivíduo da primeira geração de índice $g = 0$ são gerados aleatoriamente dentro de um intervalo fornecido pelo usuário na forma de um limite inferior $x_{j_{inf}}$ e um limite superior $x_{j_{sup}}$ para cada variável de índice j, como descrito pela Equação 2.

$$x_{0,i,j} = x_{j_{inf}} + rand() \cdot (x_{j_{sup}} - x_{j_{inf}}) \tag{2}$$

Mutação

Na etapa de mutação, para cada indivíduo da geração atual é criado um **vetor mutação** $\boldsymbol{V}_{g,i}$ ao selecionar dois indivíduos aleatórios distintos da população atual, subtrair o valor um do outro, e somar esse **vetor diferença** resultante a um outro indivíduo distinto da população após multiplica-lo por um **fator de escala** F constante fornecido pelo usuário (Equação 3). A Figura 5 ilustra o processo. A forma como é feita a escolha dos vetores $\boldsymbol{X}_{g,r_1}$, $\boldsymbol{X}_{g,r_2}$ e $\boldsymbol{X}_{g,r_3}$ é tida como a estratégia de seleção de parentes do método.

$$\boldsymbol{V}_{g,i} = \boldsymbol{X}_{g,r_1} + F \cdot (\boldsymbol{X}_{g,r_2} - \boldsymbol{X}_{g,r_3}) \tag{3}$$

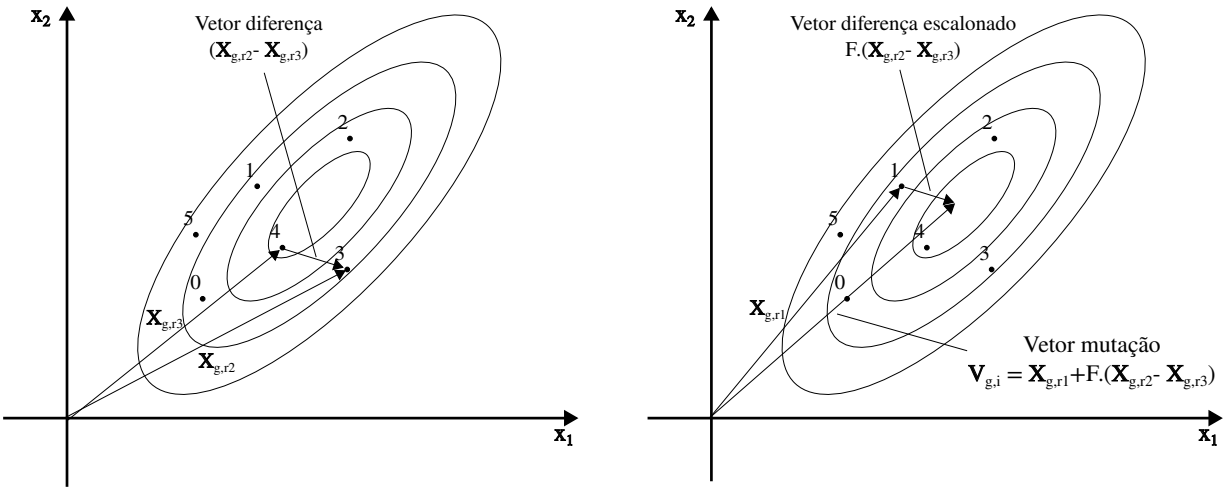

Figura 5: Processo de mutação na Evolução Diferencial.

Crossover

Na etapa de crossover é criado um **vetor tentativa** $\boldsymbol{U_{g,i}}$ para cada membro da população atual, que nesse cenário é chamado de **vetor alvo**. Para cada elemento do vetor tentativa é gerado um valor aleatório que é comparado com uma probabilidade de crossover C_r fornecida pelo usuário. Se o valor aleatório é menor que a probabilidade de crossover o elemento do vetor tentativa naquela posição é tido como o elemento de mesma posição do vetor mutação, caso contrário o valor adotado é o do elemento de mesma posição do vetor alvo. Esse processo está discrito na Equação 4 e ilustrado na Figura 6.

$$u_{g,i,j} = \begin{cases} v_{g,i,j} & \text{se } rand() \leq C_r \\ x_{g,i,j} & \text{se } rand() > C_r \end{cases} \tag{4}$$

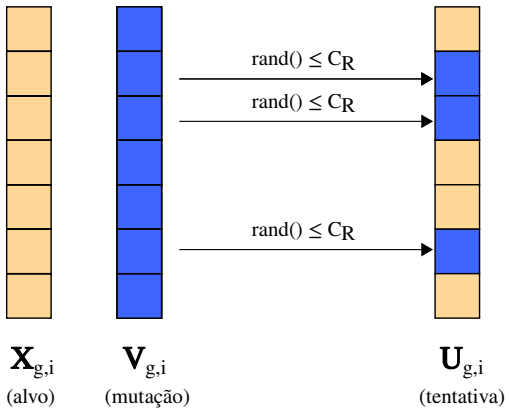

Figura 6: Processo de cruzamento binário na Evolução Diferencial.

Após criados os vetores tentativa os valores da função objetivo para cada vetor tanto da população inicial quanto seus vetores tentativa correspondentes devem ser calculados.

Assim que os resultados estão disponíveis é realizado o processo de substituição (seleção de sobreviventes).

Substituição

Na substituição são escolhidos os vetores que farão parte da geração seguinte. Isso é feito comparando cada vetor alvo com seu vetor tentativa correspondente, aquele que tiver o menor valor da função objetivo passa a fazer parte da geração seguinte (Equação 5)

$$\boldsymbol{X}_{g+1,i} = \begin{cases} \boldsymbol{X}_{g,i} & \text{se } f(\boldsymbol{X}_{g,i}) \leq f(\boldsymbol{U}_{g,i}) \\ \boldsymbol{U}_{g,i} & \text{se } f(\boldsymbol{X}_{g,i}) > f(\boldsymbol{U}_{g,i}) \end{cases} \tag{5}$$

Critério de parada

Após a criação da nova população a partir da substituição e, consequentemente, o avanço para a geração seguinte, o processo é repetido até alcançar o critério estabelecido de parada. O critério de parada pode ser tido como um número máximo de gerações g_{max} ou então um valor mínimo esperado para a função objetivo $fitness_{min}$.

Estratégias

O algoritmo descrito até então contabiliza apenas uma das estratégias possíveis de evolução diferencial, chamada de `DE/rand/1/bin`. As estratégias de evolução diferencial usam a notação `DE/X/Y/Z` em que `X` representa o método de seleção dos parentes, `Y` a quantidade de vetores diferença usado na mutação e `Z` a estratégia de crossover utilizada.

Geralmente as opções na escolha de `X` variam entre `rand` representando a escolha de um indivíduo aleatório na mutação e `best` indicando que o indivíduo $\boldsymbol{X}_{g,r_1}$ da Equação 3 passa a ser o melhor até então ($\boldsymbol{X}_{best}$). As escolhas de `Y` ficam entre 1 ou 2 vetores diferença utilizados durante a mutação e as de `Z` entre `bin` representando um cruzamento binomial e `exp` representando um cruzamento exponencial.

O pseudocódigo do algoritmo de evolução diferencial base pode ser encontrado no Algoritmo 1 a seguir.

2.3 Algoritmos Evolucionários Distribuídos

Uma propriedade que surge da independência da avaliação dos indivíduos em metaheurísticas baseadas em população é a de que essa etapa pode ser realizada paralelamente por um processo denominado paralelização de dados [26, p.31], reduzindo drasticamente o tempo total necessário para se realizar o processo de otimização.

Uma forma de paralelizar esse processo é distribuindo os indivíduos entre um conjunto de nós, que podem ser outros processos rodando em uma mesma máquina ou processos rodando em computadores distintos que se comunicam por uma interface em comum como um soquete de rede (*Network Socket*) [38, p.141-142]. A forma como dois processos se comunicam é chamada de comunicação entre processos (*Inter-Process Communication* -

Algoritmo 1 Pseudocódigo Evolução Diferencial

Input: $N_p, N_d, \boldsymbol{x}_{inf}, \boldsymbol{x}_{sup}, f_{obj}(), F, C_r, fitness_{min}, g_{max}$

`/* Inicialização da População */`
for (i em $0...N_p - 1$) **do**
  for (j em $0...N_d - 1$) **do**
    $X[0, i, j] = x_{inf}[j] + rand() * (x_{sup}[j] - x_{inf}[j])$
  end for
  $fitness[0, i] = f_{obj}(X[0, i])$
end for

$g = 0$
while ($min(fitness) > fitness_{min}$ || $g < g_{max}$) **do**
  for (i em $0...N_p - 1$) **do**

    `/* Mutação /*`
    $V[g, i] = X[g, r_1] + F * (X[g, r_2] - X[g, r_3])$

    `/* Crossover /*`
    for (j em $0...N_d - 1$) **do**
      if $rand() \leq C_r$ **then**
        $U[g, i, j] = V[g, i, j]$
      else
        $U[g, i, j] = X[g, i, j]$
      end if
    end for

    `/* Cálculo da função objetivo para Vetor Tentativa /*`
    $fitness[g, N_p + i] = f_{obj}(U[g, i])$

    `/* Substituição /*`
    if $fitness[g, i] \leq fitness[g, N_p + i]$ **then**
      $X[g + 1, i] = X[g, i]$
      $fitness[g + 1, i] = fitness[g, i]$
    else
      $X[g + 1, i] = U[g, i]$
      $fitness[g + 1, i] = fitness[g, N_p + i]$
    end if
  end for
  $g = g + 1$
end while

Output: $\boldsymbol{X}_{best}$

IPC) [38, p.72]. Até mesmo dentro de um mesmo processo é possível criar diferentes *threads* onde cada uma é responsável por uma parte do processo.

Existem diversos modelos para realizar a distribuição dos indivíduos em algoritmos evolucionários, alguns apenas buscam diminuir o tempo total de execução do método, como no modelo mestre-escravo, enquanto outros buscam alterar o método de forma a aumentar a diversidade das populações analisadas e potencialmente obter melhores soluções, como no modelo de ilha. Existem ainda outros modelos que impõe restrições quanto entre quais indivíduos um indivíduo pode competir e reproduzir com o objetivo de tornar a paralelização do processo mais eficiente como o modelo celular, entre outros. Os modelos de algoritmos evolucionários distribuídos de maior destaque serão detalhados adiante.

2.3.1 Modelo Mestre-Escravo

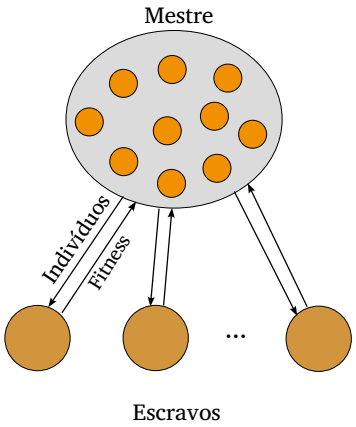

Figura 7: Modelo mestre-escravo.

O modelo de computação distribuída conhecido como mestre-escravo e está ilustrado na Figura 7. Nele um processo é tido como nó mestre e seu objetivo é realizar as operações necessárias do algoritmo evolucionário como inicialização da população, seleção, mutação e substituição, além de distribuir os indivíduos entre os outros nós, denominados nós escravos ou trabalhadores, que ficam responsáveis por fazer o cálculo da função objetivo de cada indivíduo e reportar seu resultado para o nó mestre [35]. Quando o nó mestre recebe o resultado do cálculo de todos os indivíduos ele prossegue avançando a geração e distribuindo novos indivíduos para os nós escravos.

O uso dessa estratégia implica na introdução de uma ineficiência devido ao tempo necessário para a comunicação entre os nós escravos e o nó mestre. Por conta disso o aumento teórico de velocidade conforme o aumento no número de nós escravos depende da relação entre o tempo necessário para estimar a aptidão de indivíduo e o tempo de comunicação entre o nó escravo e o nó mestre.

Quanto maior o tempo necessário para o cálculo da função objetivo com relação ao tempo de comunicação menor o desvio do aumento de velocidade esperado e o aumento de velocidade obtido de fato. Dubreuil, Gagne e Parizeau em [9] demonstrou que um problema que requer 0.25s para avaliação de um indivíduo leva a uma eficiência de 82%, mas se o tempo de avaliação aumenta para 1s enquanto o tempo de comunicação permanece o mesmo a eficiência se torna 95%.

Uma forma de mitigar a sobrecarga causada pelo tempo de comunicação é transmitir mais de um indivíduo por vez a cada nó escravo, tornando necessárias menos chamadas para distribuir os indivíduos.

2.3.2 Modelo de Ilha

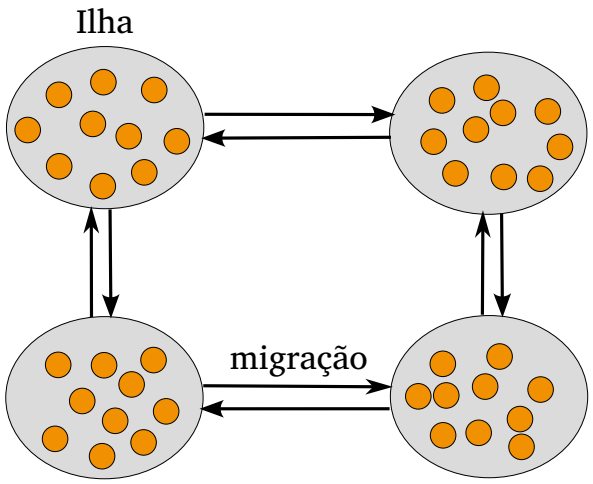

Figura 8: Modelo de ilha.

No modelo de ilha cada nó representa uma "ilha" e possui sua própria população, realizando todas as operações necessárias para o funcionamento do AE. As ilhas então se comunicam a cada intervalo pré-definido de gerações, trocando seus melhores indivíduos [27, 42].

A ideia de dividir a população em diversas ilhas é que assim há um aumento na diversidade das soluções apresentadas e, portanto, na capacidade de busca pelo ótimo global, melhorando assim o aspecto exploratório do AE e impedindo a convergência prematura em ótimos locais. Ao implantar subpopulações em ilhas isoladas é possível manter mais de um melhor indivíduo que atua como um atrator.

Muitos parâmetros podem influenciar o desempenho de um modelo distribuído por ilhas, como a homogeneidade entre as configurações dos algoritmos executados nos nós, a topologia da comunicação entre as ilhas e a frequência em que ocorre essa comunicação.

O uso de configurações heterogêneas na configuração dos AE executados nos nós em um modelo de ilha pode melhorar o balanço entre os aspectos de exploração global e intensificação local das buscas pelo espaço de soluções. Sefrioui e Périaux em [33] construíram um modelo de ilha heterogêneo hierárquico de três camadas utilizando um algoritmo genético, onde uma camada possuía configurações focadas em aumentar o aspecto exploratório, outra camada possuía configurações focadas em balancear o aspecto exploratório e de aproveitamento e a ultima camada possuía configurações que aumentavam o aspecto de aproveitamento das regiões promissoras.

2.3.3 Modelo Celular

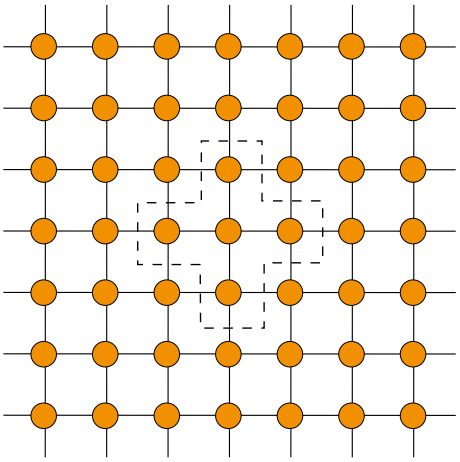

Figura 9: Modelo celular.

O modelo celular é uma estratégia de granulação fina que assim como o modelo de ilha possui uma estrutura espacial. Nele os indivíduos são alocados em uma malha, em que há idealmente um indivíduo por processo (célula) e em que as interações são ditadas por uma topologia de rede pré-definida. Cada indivíduo pode competir e reproduzir apenas com os que estão na sua vizinhança. Como as vizinhanças se sobrepõe, bons indivíduos podem propagar pela população.

A topologia toroidal é a utilizada com maior frequência, nela os nós de uma das extremidades se comunicam com os nós da extremidade oposta, a Figura 10 ajuda a visualizar a geometria toroidal formada pelos nós.

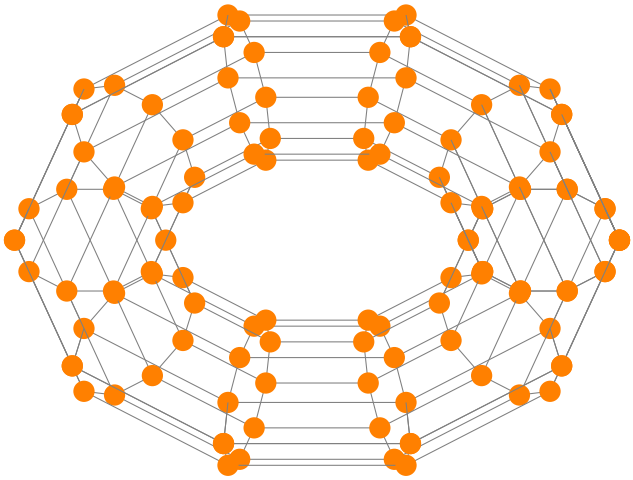

Figura 10: Malha toroidal em um modelo celular.

As atualizações das células no modelo celular podem ocorrer de forma síncrona ou assíncrona. No modelo síncrono todas as células são atualizadas simultaneamente, isso é, considerando todos os dados dos indivíduos constantes de uma geração para outra durante as operações de seleção e mutação. Já no modelo assíncrono as células podem ser atualizadas uma à uma, o que trás uma diferença entre a geração de cada célula na vizinhança durante um processo de atualização.

Outras formas de atualização assíncrona também existem [32] como por exemplo: a varredura por linha (*Line Sweep*), onde uma linha inteira da malha pode ser atualizada de cada vez, a varredura fixa aleatória (*Fixed Random Sweep*) onde uma sequência aleatória fixa de indivíduos é atualizada um à um, a varredura nova aleatória (*New Random Sweep*) onde uma sequência aleatória nova de indivíduos é atualizada a cada geração e a escolha uniforme (*Uniform Choice*) onde uma célula é atualizada aleatoriamente de cada vez, podendo haver células atualizadas múltiplas vezes e células sem nenhuma atualização.

Diferentes topologias, no que diz respeito a quantas células são utilizadas em cada direção, podem ser utilizadas. Alba e Troya em [2] comparam a performance de diferentes topologias de malhas 2D de algoritmos genéticos celulares.

2.3.4 Modelo de Piscina

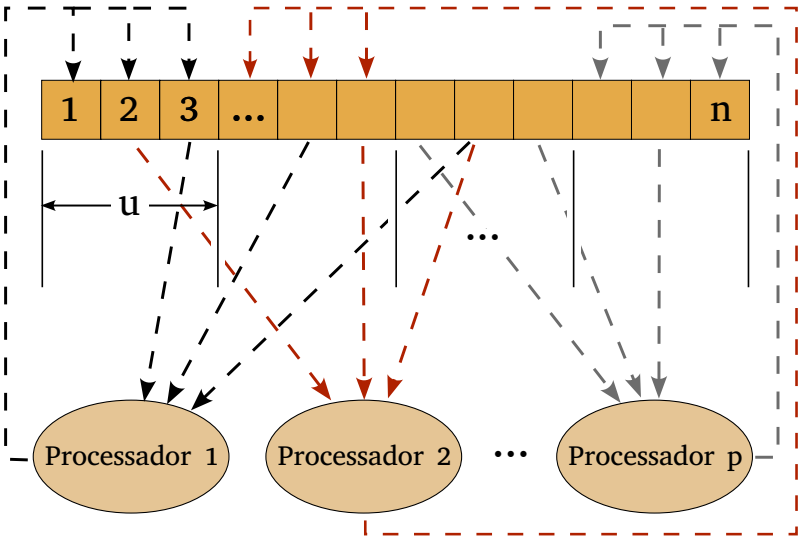

Figura 11: Modelo piscina.

No modelo de piscina vários processadores atuam em um mesmo conjunto (*pool*) de recursos compartilhados (Figura 11). Os processos são desacoplados e interagem apenas com o conjunto de recursos, provendo uma abordagem natural para lidar com a heterogeneidade entre os nós e o assincronismo [31].

A piscina é um vetor global compartilhado de tamanho n representando os indivíduos de uma população. Este vetor então é particionado em p segmentos de tamanho u que correspondem a p processadores (ou threads). Cada processador pode ler os indivíduos de qualquer posição do vetor mas apenas pode sobrescrever indivíduos em sua própria partição.

No processo de otimização cada processador escolhe aleatoriamente u indivíduos do vetor compartilhado para realizar operações genéticas e gerar u novos indivíduos, então substitui o indivíduo da posição i de sua partição caso a aptidão calculada do novo individuo seja melhor que a anterior.

As vantagens de um modelo de piscina são que como os processadores são desacoplados para trabalhar em um conjunto compartilhado de recursos, eles podem lidar com heterogeneidade e assincronismos de forma relativamente mais fácil. Nesse modelo o número de processadores pode mudar dinamicamente e o sistema continua funcionando bem mesmo que alguns processadores falhem, obtendo uma boa tolerância falhas.

2.3.5 Modelo Híbrido

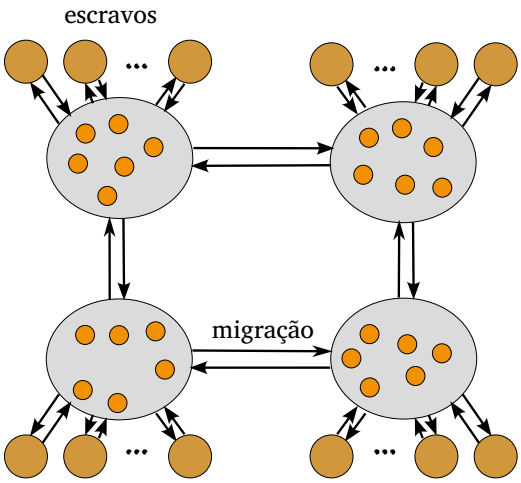

Figura 12: Modelo de hibrido ilha-mestre-escravo.

Modelos híbridos, também chamados de hierárquicos, são gerados ao juntar dos métodos distintos como ao usar um modelo de ilha em conjunto com o modelo mestre-escravo, ilustrado na Figura 12, combinando o aumento de velocidade do modelo mestre-escravo com o aumento na diversidade da população do modelo de ilha [5, 24].

Outros modelos híbridos como o ilha-celular [12] e ilha-ilha [17] existem na literatura, no entanto sua discussão fica além do escopo desse trabalho.

2.4 O Protocolo HTTP

O *Hypertext Transfer Protocol* (HTTP) é o protocolo base da comunicação na World Wide Web, sendo utilizado para transferir documentos de hipertexto e outros tipos de dados entre um cliente e um servidor. Desde sua criação, o HTTP tem evoluído para suportar uma ampla variedade de métodos e funcionalidades que o tornam um dos protocolos mais usados na internet [10].

2.4.1 Estrutura do Protocolo

O HTTP é um protocolo baseado em texto que segue um modelo de requisição-resposta. No processo típico de comunicação, um cliente, como um navegador web, envia uma requisição HTTP para um servidor, e este responde com os dados solicitados, como uma página web ou um arquivo. Cada mensagem HTTP consiste em três partes principais: a linha de requisição ou resposta, os cabeçalhos, e o corpo da mensagem [3].

A linha de requisição geralmente contém o método HTTP, o caminho do recurso solicitado e a versão do protocolo. Os métodos HTTP mais comuns são o `GET`, que solicita dados, e o `POST`, que envia dados ao servidor. Outros métodos incluem `PUT`, `DELETE`, `PATCH`, entre outros, que possibilitam interações mais complexas com os recursos do servidor [37]. Cada requisição pode incluir também cabeçalhos que fornecem informações adicionais, como tipo de conteúdo (`Content-Type`) ou detalhes de autenticação (`Authorization`).

A resposta do servidor é composta de uma linha de status, que informa o resultado da requisição por meio de códigos de status HTTP, como `200 OK` para sucesso ou `404 Not Found` quando o recurso solicitado não é encontrado [4].

2.5 Arquitetura REST

Representational State Transfer (REST) é um estilo arquitetural proposto por Roy Fielding em sua tese de doutorado, que descreve um conjunto de princípios para a construção de sistemas distribuídos, em particular, a interação entre cliente e servidor na Web [10]. A arquitetura REST baseia-se em um conjunto de restrições que garantem a simplicidade, escalabilidade, desempenho e modificação independente dos componentes de um sistema.

2.5.1 Princípios Fundamentais da Arquitetura REST

A arquitetura REST se apoia em seis princípios fundamentais que definem como os recursos devem ser acessados e manipulados na web:

- **Cliente-servidor:** A interação entre cliente e servidor é uma das bases do REST. A separação de responsabilidades permite que os dois componentes evoluam independentemente. O cliente solicita recursos, enquanto o servidor fornece esses recursos e processa as solicitações [37].
- **Sem estado:** Cada requisição HTTP de um cliente para um servidor deve conter todas as informações necessárias para que o servidor possa entender a solicitação. Isso significa que o servidor não mantém nenhum estado do cliente entre as requisições [10]. A ausência de estado melhora a escalabilidade do sistema, permitindo que os servidores tratem um número maior de solicitações de clientes de forma independente.
- **Cacheabilidade:** As respostas do servidor devem indicar explicitamente se podem ou não ser armazenadas em cache pelo cliente. Isso melhora a eficiência do sistema, permitindo que respostas comuns sejam reutilizadas, reduzindo o número de requisições desnecessárias [37].
- **Interface uniforme:** A uniformidade da interface entre os componentes é um dos aspectos mais importantes do REST. Uma interface uniforme permite que os recursos sejam manipulados de maneira previsível e padronizada, o que facilita o desenvolvimento e a manutenção de sistemas distribuídos [10]. Essa interface padronizada inclui o uso de métodos HTTP (como `GET`, `POST`, `PUT`, `DELETE`), a identificação dos recursos através de URIs, e a transferência de representações dos recursos.
- **Sistemas em camadas:** Uma arquitetura REST permite a divisão do sistema em camadas, nas quais os componentes de cada camada podem ser implementados e escalados de forma independente. Isso melhora a modularidade do sistema e facilita a adição de funcionalidades, como servidores intermediários e balanceamento de carga [3].
- **Código sob demanda (opcional):** Embora seja uma característica opcional do REST, os servidores podem fornecer scripts ou códigos executáveis ao cliente, como applets Java ou scripts JavaScript, para estender a funcionalidade do cliente temporariamente [10].

2.5.2 Manipulação de Recursos

No contexto de uma arquitetura REST, os recursos são qualquer entidade que pode ser acessada e manipulada por meio de um identificador (normalmente uma URL). Um recurso pode representar qualquer coisa, desde uma página da web até um item em uma loja online, um arquivo ou até mesmo um usuário de um sistema.

Cada interação com o recurso é realizada utilizando os métodos HTTP. As operações básicas sobre os recursos são conhecidas como CRUD (Create, Read, Update, Delete), mapeadas diretamente para os métodos HTTP da seguinte maneira:

- `GET`: Usado para obter uma representação de um recurso existente.
- `POST`: Usado para criar um novo recurso no servidor.
- `PUT`: Usado para atualizar completamente um recurso existente.
- `DELETE`: Usado para remover um recurso existente.

As respostas do servidor às operações nos recursos são comunicadas por meio de códigos de status HTTP, como o `200 OK` para sucesso, `404 Not Found` quando o recurso não existe, ou `500 Internal Server Error` para problemas no servidor [37].

2.6 Message Brokers

Message brokers são intermediários de comunicação em sistemas distribuídos, utilizados para facilitar a troca de mensagens entre diferentes serviços ou aplicativos. Eles desempenham um papel essencial na arquitetura orientada a eventos, desacoplando os remetentes e destinatários de mensagens, o que permite maior escalabilidade e resiliência no sistema [15].

2.6.1 Funcionamento dos Message Brokers

O conceito de message brokers baseia-se no uso de filas e tópicos para a comunicação assíncrona entre componentes. Um remetente (ou produtor) publica uma mensagem em uma fila ou tópico gerenciado pelo broker, e um destinatário (ou consumidor) recupera a mensagem da fila para processá-la. Isso permite que os produtores e consumidores operem de maneira independente no tempo, aumentando a flexibilidade do sistema [19].

Há dois principais padrões de troca de mensagens usados pelos message brokers:

- **Filas de Mensagens (Message Queues):** Nesse modelo, as mensagens são enviadas para uma fila específica, e cada mensagem é processada por apenas um consumidor. Esse padrão é utilizado quando é necessário garantir que uma mensagem seja processada uma única vez [40].
- **Publicação/Assinatura (Publish/Subscribe):** Nesse padrão, uma mensagem é publicada em um tópico e pode ser entregue a múltiplos consumidores. Ele é ideal para cenários onde é necessário que vários serviços ou sistemas recebam a mesma mensagem [7].

2.6.2 Exemplos Populares de Message Brokers

Entre os message brokers mais populares, destacam-se:

- **Apache Kafka:** Um sistema distribuído otimizado para o tratamento de grandes volumes de dados em tempo real, ideal para pipelines de dados e arquiteturas baseadas em eventos [23].
- **RabbitMQ:** Um broker de mensagens amplamente utilizado para o gerenciamento de filas de mensagens em sistemas distribuídos, conhecido por sua simplicidade e robustez [39].
- **IBM MQ:** Um *message broker* com forte foco em transações, usado em ambientes corporativos para garantir a entrega confiável de mensagens [19].

Vantagens dos Message Brokers

Os *message brokers* trazem uma série de vantagens para sistemas distribuídos:

- **Desacoplamento temporal:** Produtores e consumidores de mensagens não precisam estar disponíveis ao mesmo tempo para a comunicação ocorrer, o que aumenta a resiliência do sistema [15].
- **Escalabilidade:** Permitem que novos consumidores sejam adicionados facilmente, de modo que a arquitetura possa crescer conforme necessário [23].
- **Garantia de entrega:** Muitos *message brokers* oferecem garantias fortes de entrega de mensagens, como entrega exatamente uma vez (*exactly-once*) ou pelo menos uma vez (*at-least-once*) [39].

3 Metodologia

Nesta seção será descrito o funcionamento da Aplicação Mestre (AM) do modelo mestre-escravo de forma a cumprir com os objetivos apresentados. Como será observado adiante, a arquitetura proposta maximiza a flexibilidade do desenvolvimento da Aplicação Escrava (AE), responsável por calcular a função objetivo, tornando o desenvolvimento desse componente responsabilidade do usuário.

Inicialmente abordaremos alguns aspectos gerais que guiam o desenvolvimento da aplicação, como o mecanismo de comunicação entre processos a ser utilizado e os recurso REST elaborados para compor a aplicação, abordaremos brevemente as tecnologias utilizadas na construção da AM e por fim descreveremos como os componentes devem funcionar e interagir para alcançar o objetivo final.

3.1 Comunicação entre Processos

Uma das alternativas para a comunicação entre os processos é o uso de um sistema de fila, intermediado por um *message broker* como o RabbitMQ ou Amazon MQ. Essa opção é interessante visto que torna a comunicação assíncrona e indireta. A AM atuaria como uma produtora de mensagens, gerando uma mensagem para cada indivíduo e as publicando em uma fila. As AEs atuariam como consumidoras dessa fila, lendo suas mensagens de forma mutualmente exclusiva para então calcular o valor da função objetivo e publicar o resultado obtido em outra fila, a qual a AM consumiria, registrando os resultados e dando prosseguimento ao processo.

Ao utilizar uma API de push, as AEs se inscreveriam na fila de interesse e receberiam as mensagens assim que elas estivessem disponíveis, sem a necessidade de recorrer a métodos como *polling* que degradam a performance da solução.

No entanto essa solução tornaria o desenvolvimento de uma AE mais complexo, visto que algumas linguagens de programação não possuem um modelo simples para a declarar um consumidor de uma fila RabbitMQ, o que poderia diminuir a flexibilidade desse método.

Portanto, dentre as formas possíveis para estabelecer a comunicação entre os processos foi escolhido o protocolo HTTP usando uma arquitetura REST. Essa escolha se justifica pela grande flexibilidade que esse protocolo de comunicação proporciona, visto que a grande maioria das linguagens de programação possuem uma forma de realizar chamadas HTTP, tornando a integração com a AE muito mais flexível. Além disso, a disponibilidade de uma API REST permite que o usuário configure a AM e obtenha informações sobre as otimizações em curso utilizando um cliente HTTP como o Insôminia, ou Postman.

Nesse modelo de comunicação é necessário que a AE realize requisições GET para a AM repetidamente em busca de indivíduos para realizar o cálculo da função objetivo em um modelo que é chamado de *short polling*.

Caso a AM possua um indivíduo que não foi calculado ainda ela responde com um código 200 contendo um campo *optimizationStatus* com o valor `"RUNNING"` e os dados do indivíduo a ser calculado no corpo da resposta. Caso não exista nenhum indivíduo para ser calculado ainda mas a otimização ainda não foi finalizada ela responde com um código 404 para que a AE tente novamente mais tarde. E por fim, se a rodada de otimização já tenha alcançado o critério de parada, a aplicação responde com um código 200 mas contendo o

campo *optimizationStatus* com o valor `"FINISHED"`.

3.2 Recursos REST

Por se tratar de uma arquitetura REST é necessária a definição de recursos que serão representados e manipulados pelo usuário para interagir com a aplicação. Os recursos escolhidos, suas propriedades e relações podem ser observados de forma geral no diagrama UML (do ingles *Unified Modeling Language*) representado na Figura 13. Suas definições e detalhamentos serão descritas a seguir:

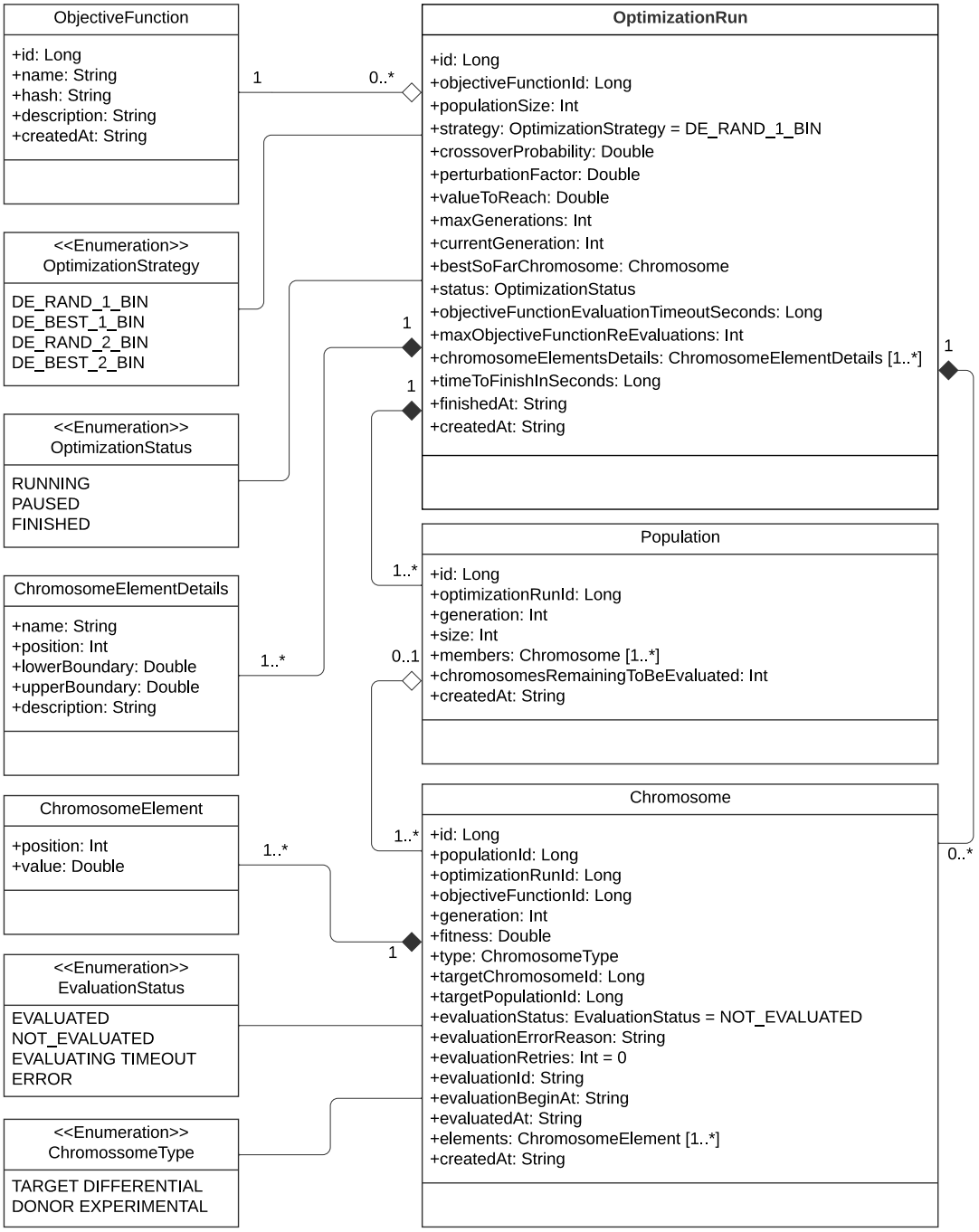

Figura 13: Diagrama de classes UML.

Função Objetivo (*Objective Function*)

Este recurso representa a função objetivo que será otimizada pelo algoritmo de ED. As propriedades do recurso estão presentes na Tabela 1, já as rotas disponíveis para interagir com esse recurso estão na Tabela 2.

Tabela 1: Atributos do recurso *Objective Function*.

Atributo	Tipo de Dado	Descrição
`name`	String	Nome da função objetivo.
`description`	String	Descrição da função objetivo.
`hash`	String	Hash da função objetivo.

Tabela 2: Rotas disponíveis para o recurso *Objective Function*.

Método HTTP	Rota	Descrição
GET	/objectiveFunction	Obtém página de funções objetivo
GET	/objectiveFunction/{id}	Obtém *Objective Function* pelo ID
POST	/objectiveFunction	Cria novo *Objective Function*
PATCH	/objectiveFunction/{id}	Altera dados de um *Objective Function*
DELETE	/objectiveFunction/{id}	Deleta um *Objective Function* pelo ID

Rodada de Otimização (*Optimization Run*)

Optimization Run é o recurso REST que representa uma rodada de otimização, portanto deve fornecer todas as propriedades necessárias para o funcionamento do método de ED, como probabilidade de crossover, fator de perturbação e estratégia utilizada. As propriedades do recurso em questão estão descritas na Tabela 3.

Tabela 3: Atributos do recurso *Optimization Run*.

Atributo	Tipo de Dado	Descrição
`id`	Long	Identificador único da rodada de otimização.
`objectiveFunctionId`	Long	Id que referencia à função objetivo utilizada na execução.
`populationSize`	Int	Tamanho das populações geradas.
`strategy`	Enum	Estratégia de ED utilizada.
`crossoverProbability`	Double	Valor de 0 a 1 que representa a probabilidade de crossover.
`perturbationFactor`	Double	Fator de pertubação adotado.
`valueToReach`	Double	Valor mínimo da função objetivo utilizado critério de parada

Continuação na próxima página

Tabela 3: Atributos do recurso *Optimization Run*. (Continuação)

Atributo	Tipo de Dado	Descrição
`maxGenerations`	Int	Máximo de gerações utilizado como critério de parada.
`currentGeneration`	Int	Geração atual da rodada de otimização.
`bestSoFarChromosome`	Chromosome	Melhor cromossomo da rodada de otimização até o momento
`objectiveFunction-EvaluationTimeout-Seconds`	Long	Tempo máximo de avaliação em segundos.
`maxObjective-FunctionRe-Evaluations`	Int	Número máximo de re-tentativas de avaliação.
`chromosomeElement-Details`	Chromosome-ElementDetails [ ]	Resultado final da execução com o cromossomo otimizado.
`status`	Enum	Status da rodada de otimização.
`createdAt`	String	Data e hora de criação da rodada de otimização.
`finishedAt`	String	Data e hora de finalização da rodada de otimização, caso esteja com status FINISHED.
`timeToFinishInSeconds`	Long	Tempo em segundos que a rodada de otimização levou para finalizar, caso esteja com status FINISHED.

O campo *strategy* é uma enumeração que representa a estratégia de ED adotada e pode receber os seguintes valores: `"DE_RAND_1_BIN"`, `"DE_BEST_2_BIN"`, `"DE_RAND_2_BIN"` e `"DE_BEST_1_BIN"`.

Para que seja configurado um critério de parada ao menos um dos campos *maxGenerations* e *valueToReach* deve ser fornecido.

O campo *status* representa os possíveis status para a rodada de otimização em questão e pode assumir os valores `"RUNNING"`, `"PAUSED"` e `"FINISHED"`.

O campo *chromosomeElementDetails* é uma lista contendo objetos que descrevem as variáveis de otimização, indicando seus nomes, posições no vetor e seus limites inferior e superior. A Tabela 4 descreve os campos em questão.

Tabela 4: Atributos do *Chromosome Element Details*.

Atributo	Tipo de Dado	Descrição
`name`	String	Nome da variável de otimização.
`description`	String	Descrição da variável de otimização
`position`	Int	Posição da variável no vetor
`lowerBoundary`	Double	Limitante inferior da variável
`upperBoundary`	Double	Limitante superior da variável

As rotas para interagir com o recurso estão representadas na Tabela 5 a seguir:

Tabela 5: Rotas disponíveis para o recurso *Optimization Run*.

Método HTTP	Rota	Descrição
GET	/optimizationRun	Obtém página de rodadas de otimização
GET	/optimizationRun/{id}	Obtém rodada de otimização pelo ID
POST	/optimizationRun	Cria nova rodada de otimização
PATCH	/optimizationRun/{id}	Altera dados de uma rodada de otimização
DELETE	/optimizationRun/{id}	Deleta uma rodada de otimização pelo ID
GET	/optimizationRun/{id}/ chromosome/notEvaluated	Obtém cromossomo não avaliado da rodada de otimização com ID fornecido
GET	/optimizationRun/{id}/ populations	Obtém página populações da rodada de otimização com ID fornecido

População (*Population*)

O recurso *population* representa uma população de indivíduos que participa de uma rodada de otimização, contendo informações como qual o id da rodada de otimização que ela faz parte, a geração que ela representa e os cromossomos membros dessa população. Os atributos do recurso estão disponibilizados na Tabela 6, já as rotas disponíveis para interação com o recurso estão na Tabela 7.

Tabela 6: Atributos do recurso *Population*.

Atributo	Tipo de Dado	Descrição
`id`	Long	Identificador único da população.
`optimizationRunId`	Long	Id que referencia à rodada de otimização a qual a população faz parte.
`generation`	Int	Geração referente a população.
`members`	Chromosome [ ]	Lista contendo os chromossomos membros da população
`chromosomes-RemainingToBe-Evaluated`	Int	Número de chromossomos associados a população que precisão ser avaliados.
`createdAt`	String	Data e hora de criação da população.

Tabela 7: Rotas disponíveis para o recurso *Population*.

Método HTTP	Rota	Descrição
GET	/population	Obtém página de populações
GET	/population/{id}	Obtém *Population* pelo ID
GET	/population/{id}/statistics	Obtém dados estatísticos da população com ID fornecido

Cromossomo (*Chromosome*)

O recurso *chromosome* representa um indivíduo de uma população ou então um vetor experimental no método de ED, portanto possui informações quanto a qual rodada de otimização e população faz parte, os elementos que representam as variáveis de otimização, seu status de avaliação, entre outros. Os atributos desse recurso são apresentados na Tabela 8.

Tabela 8: Atributos do recurso *Chromosome*.

Atributo	Tipo de Dado	Descrição
`id`	Long	Identificador único do cromossomo.
`populationId`	Long	Id que referencia a população a qual o cromossomo faz parte, se existir.
`optimizationRunId`	Long	Id que referencia à rodada de otimização a qual o cromossomo faz parte.
`objectiveFunctionId`	Long	Id da função objetivo que o cromossomo está associado.
`type`	Enum	Tipo do cromossomo.
`targetChromosomeId`	Long	Id do cromossomo alvo, caso exista.
`targetPopulationId`	String	Id da população do cromossomo alvo, caso exista.
`evaluationStatus`	Enum	Status da avaliação do cromossomo.
`evaluationErrorReason`	String	Razão do erro na avaliação do cromossomo, caso exista..
`evaluationRetries`	String	Número de retentativas na avaliação do cromossomo.
`evaluationId`	String	Id de avaliação do cromossomo.
`evaluationBeginAt`	String	Data e hora de inicio da avaliação do cromossomo.
`evaluatedAt`	String	Data e hora de do fim da avaliação do cromossomo.
`generation`	String	Geração a que o cromossomo pertence.
`fitness`	String	Valor da função objetivo calculada, caso exista.
`elements`	Double [ ]	Valores das variáveis de otimização.
`createdAt`	String	Data e hora de criação do cromossomo.

O campo *type* é uma enumeração que pode assumir os valores `"TARGET"`,`"DIFFERENTIAL"`, `"DONOR"` e `"EXPERIMENTAL"`, representando os diferentes tipos de vetores do método de ED.

Já o campo *evaluationStatus* é uma enumeração que representa os possíveis status de avaliação de um cromossomo, sendo esses: `"EVALUATED"`, `"NOT_EVALUATED"`, `"EVALUATING"`, `"TIMEOUT"` e `"ERROR"`.

A Tabela 9 a seguir contém as rotas existentes na aplicação para interagir com o recurso.

Tabela 9: Rotas disponíveis para o recurso *Chromosome*.

Método HTTP	Rota	Descrição
GET	/chromosome	Obtém página de cromossomos
GET	/chromosome/{id}	Obtém cromossomo pelo ID
GET	/chromosome/ targetPopulation/{id}	Obtém dados de cromossomos experimentais com população alvo de ID fornecido
POST	/chromosome/{id}/ evaluationResult	Publica os resultados da avaliação da função objetivo
POST	/chromosome/{id}/ evaluationErro	Publica erro durante avaliação da função objetivo

3.3 Tecnologias Utilizadas

Para desenvolver a AM foi construído um código na linguagem Kotlin, visto que ela tem um ecossistema bem desenvolvido de bibliotecas já que possui uma compatibilidade e interoperabilidade com a linguagem Java. O *framework* Spring Boot foi utilizado em conjunto com os módulos spring MVC para a criação do servidor http e spring data JPA para a interação com o banco de dados.

Por padrão a aplicação usa o banco de dados H2 no modo de persistência em arquivo, já que dessa forma não é adicionada a complexidade de gerenciar um sistema de banco de dados como um componente separado. No entanto essa configuração pode ser personalizada inicializando a variável de ambiente *ACTIVE_PROFILES* com o valor "h2-mem" caso se deseje usar o banco de dados H2 em modo de memória ou "mysql" caso se deseje que a aplicação se conecte com um banco de dados mysql.

O Git foi utilizado como sistema de controle de versão e o código construído está disponível no github. Lá é possível encontrar o arquivo de especificação *openapi.json*, que pode ser usado como entrada para ferramentas como o OpenAPI Generator para gerar um código cliente da API REST da AM em diversas linguagens de programação a escolha do usuário. Ao iniciar a aplicação, na rota `/swagger-ui.html` também é possível encontrar uma documentação interativa da interface HTTP da API.

3.4 Funcionamento de uma Rodada de Otimização

Agora que os recursos REST foram apresentados podemos detalhar como é esperado que o usuário e a AE interajam com a AM através da interface HTTP para que uma rodada de otimização possa ser concluída com sucesso.

3.4.1 Inicialização da Rodada de Otimização

Inicialmente o usuário deve fazer uma requisição POST na rota `/objectiveFunction` fornecendo as informações necessárias para criar um *Objective Function* e armazenar o id retornado.

Em seguida o usuário deve fazer uma requisição POST na rota `/optimizationRun` com as informações necessárias para o funcionamento do algoritmo de ED e fornecendo o id da

função objetivo retornado do passo anterior. No retorno dessa chamada é gerado um id da rodada de otimização, que deve ser utilizado nos passos seguintes.

Nesse momento é gerada a população inicial de cromossomos alvo e um conjunto de cromossomos experimentais associados a cada um deles. Uma contagem do número de cromossomos associados a população em questão que necessitam ser avaliados é iniciada e armazenada no campo *chromosomesRemainingToBeEvaluated* da população associada.

3.4.2 Obtenção do Cromossomo a Ser Avaliado

Com o id da rodada de otimização em mãos, a aplicação escrava pode começar a fazer requisições GET na rota `/optimizationRun/{id}/chromosome/notEvaluated` para obter os dados de um dos cromossomos a serem avaliados.

Ao fazer essa requisição o cromossomo em questão adquire status `"EVALUATING"` e deixa de estar disponível para que outras instâncias da aplicação escrava o obtenham pela mesma rota. Esse processo está sujeito ao que é chamado de condição de corrida, isso é, quando duas execuções em paralelo tentam obter e atualizar um mesmo dado. Esse problema é evitado iniciando uma transação e criando um `"LOCK"` nos dados do cromossomo durante a requisição de leitura no banco de dados. Assim as outras execuções não poderão obter este dado até que a transação finalize e o status do cromossomo seja atualizado.

No corpo da resposta é enviado um campo *evaluationId*, que se trata de um UUID único associado a aquela avaliação. O objetivo desse UUID (do inglês *Universally Unique IDentifier*) é que ele seja fornecido nas requisições que informam o resultado da avaliação, garantindo que a aplicação que está informando o resultado da avaliação é a mesma que requisitou os dados daquele cromossomo.

3.4.3 Envio do Resultado Obtido

Após calcular o valor da função objetivo a aplicação escrava deve fazer uma requisição para a rota `/chromosome/{id}/pulishEvaluation` informando o valor obtido no campo *fitness* e o *evaluationId* associado no corpo da requisição.

Se a AE encontrar algum erro mapeado durante o cálculo da função objetivo ela deve fazer uma requisição para a rota `/chromosome/{id}/publishError` com o motivo do erro no campo *reason* e o *evaluationId* associado.

A cada cromossomo que atinge um dos status finais `"EVALUATED"` ou `"ERROR"` o campo *chromosomesRemainingToBeEvaluated* da população associada é decrementado. Quando esse valor chega a 0 os critérios de parada são avaliados, se algum deles for satisfeito a otimização passa a ter o status `"FINISHED"` e o processo é finalizado, caso contrário é iniciado um processo para avançar para a próxima geração, que envolve o processo de seleção, mutação e crossover.

Um cuidado especial deve ser tomado no processo de atualização e leitura do campo *chromosomesRemainingToBeEvaluated* já que esse processo está sujeito a condições de corrida. Para evitar isso a atualização do campo é feita através de comandos SQL (do inglês *Structured Query Language*) `"UPDATE"` que independem da ordem, e são feitas juntamente com a leitura do resultado no contexto de uma transação no banco de dados.

3.4.4 Processo de Re-tentativa e Timeout

Se a AE demorar mais tempo do que o determinado no campo *objectiveFunctionEvaluationTimeoutSeconds* para enviar o resultado da avaliação e o campo *maxObjectiveFunctionReEvaluations* fornecido na criação da rodada de otimização for maior que 0 o cromossomo automáticamente passa para o status `"TIMEOUT"` e o valor de seu campo *evaluationRetries* é incrementado.

Após isso o cromossomo passa a estar novamente disponível para ser obtido pela rota `/optimizationRun/{id}/chromosome/notEvaluated` para tentar ser avaliado novamente. Esse ciclo se repete enquanto o campo *evaluationRetries* do cromossomo é menor que o campo *maxObjectiveFunctionReEvaluations* da rodada de otimização, após isso o cromossomo passa para o status `"ERROR"` e é descartado.

Se tanto o cromossomo alvo quanto o cromossomo experimental associado atingem o status `"ERROR"`, um novo cromossomo experimental associado a aquele cromossomo alvo é criado e fica disponível para avaliação. Esse processo se repete até que ao menos um cromossomo experimental seja avaliado com sucesso.

3.4.5 Regra de Seleção

No processo de seleção, caso ambos os cromossomos alvo e seu cromossomo experimental associado estejam no status `"EVALUATED"`, aquele que possuir o menor valor no campo *fitness* passa a compor a população da próxima geração.

Caso entre os cromossomos alvo e experimentais associados haja apenas um com status `"EVALUATED"` enquanto o outro possui o status `"ERROR"` apenas o cromossomo que foi avaliado com sucesso passa a compor a população da próxima geração.

Após gerada a nova população o processo de mutação ocorre de acordo com a estratégia fornecida no campo *strategy* e o crossover de acordo com a probabilidade de crossover fornecida no campo *crossoverProbability*. Ao final dessa etapa um novo conjunto de cromossomos experimentais não avaliados é gerado e a contagem contida no campo *chromosomesRemainingToBeEvaluated* da população é iniciada com o tamanho desse conjunto.

3.4.6 Resiliência da Aplicação

A aplicação desenvolvida não mantém os dados das rodadas de otimização a todo momento na memória e avança o processo de otimização apenas durante a execução de cada requisição. A cada requisição os dados pertinentes a ela são obtidos do banco de dados e, conforme o trabalho requisitado é executado, as mudanças feitas nos recursos são salvas no novamente no banco. Isso garante que a aplicação é do tipo *stateless* e as requisições são totalmente independentes umas das outras.

Essa característica garante a aplicação alguns comportamentos desejáveis, como a resiliência a eventuais desligamentos e reinicializações. A qualquer momento a aplicação pode ser desligada e reiniciada e os dados do processo não serão perdidos. As AEs podem parar de funcionar totalmente e voltar a qualquer momento sem danos a rodada de otimização.

Outras características adquiridas são as de que várias rodadas de otimização podem acontecer simultaneamente sem nenhum prejuízo e mais de uma instância da aplicação pode

ser inicializada e responder as requisições ao mesmo tempo, desde que estejam conectadas ao mesmo banco de dados.

4 Resultados

4.1 Validação da Metodologia

A fim de validar a capacidade da metodologia de convergir para uma solução razoável de um problema de otimização foram criadas algumas rodadas de otimização a fim de encontrar o mínimo da função de rastrigin de duas variáveis. Essa função foi escolhida por conter vários mínimos locais, possibilitando avaliar a capacidade do algoritmo de escapar deles e se aproximar do mínimo global.

A função rastrigin de duas variáveis é descrita pela Equação 6 a seguir e pode ser visualizada na Figura 14. O mínimo global da função é 0 e é obtido na posição $\boldsymbol{x}^* = [0, 0]$.

$$f(x_1, x_2) = 20 + x_1^2 + x_2^2 - 10\cos(2\pi x_1) - 10\cos(2\pi x_2) \tag{6}$$

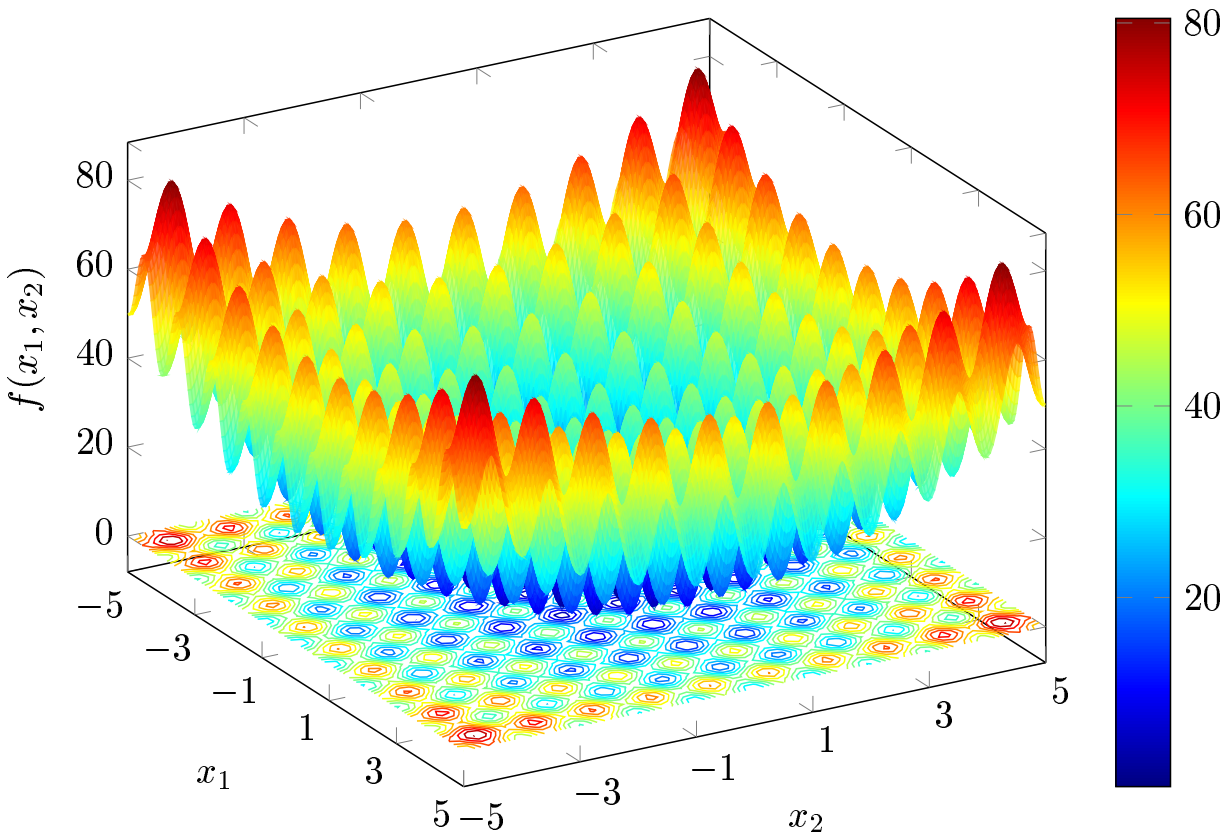

Figura 14: Função de rastrigin de duas variáveis.

Para fazer o papel da AE foi escrito um código em kotlin que está disponível no repositório do github. Como o cálculo da função objetivo é muito rápido a aplicação não obtém nenhum ganho de velocidade ao utilizarmos várias instâncias ao mesmo tempo, isso ocorre pois a aplicação fica limitada ao tempo de resposta das chamadas HTTP que faz a AM. Por conta disso apenas uma instância da AE foi utilizada durante as rodadas de otimização realizadas.

Foram criadas 15 rodadas de otimização a fim de se obter uma média estatística dos resultados, visto que, por ser um método estocástico, resultados diferentes são obtidos a

cada rodada, mesmo com os mesmos parâmetros iniciais. As rodada de otimização foram criadas utilizando o seguinte corpo de requisição na rota `/optimizationRun`:

Listagem 1 Corpo da requisição para criação das rodadas de otimização da função de rastrigin

```
{
    "objectiveFunctionId": 1,
    "populationSize": 10,
    "strategy": "DE_RAND_1_BIN",
    "crossoverProbability": 0.8,
    "perturbationFactor": 0.5,
    "maxGenerations": 100,
    "objectiveFunctionEvaluationTimeoutSeconds": 30,
    "chromosomeElementDetails": [
        {
            "name": "Variável de rastrigin 1",
            "position": 0,
            "lowerBoundary": -5.12,
            "upperBoundary": 5.12,
            "description": "Primeira variável da função de rastrigin
            representando a coordenada x"
        },
        {
            "name": "Variável de rastrigin 2",
            "position": 1,
            "lowerBoundary": -5.12,
            "upperBoundary": 5.12,
            "description": "Segunda variável da função de rastrigin
            representando a coordenada y"
        }
    ]
}
```

Após a finalização das rodadas de otimização os valores da função objetivo e variáveis de otimização dos melhores indivíduos, além do tempo de duração das rodadas, foram coletados e suas médias e desvios padrões calculados. Os resultados estão presentes na Tabela 10 a seguir.

Resultado	x_1	x_2	**tempo (segundos)**
$1,06747 \times 10^{-10}$ $\pm$ $2,63951 \times 10^{-10}$	$-1,03002 \times 10^{-8}$ $\pm$ $3,83767 \times 10^{-7}$	$3,83767 \times 10^{-7}$ $\pm$ $6,30083 \times 10^{-7}$	$4,5 \pm 0,63245$

Tabela 10: Resultados na rodada de otimização da função de rastrigin.

Os resultados do melhor, pior e a média dos indivíduos de cada geração obtidos durante uma das rodadas de otimização utilizada como exemplo podem ser observados na Figura 15. Como pode ser visto, o algoritmo converge rapidamente para o mínimo global em uma escala logarítmica.

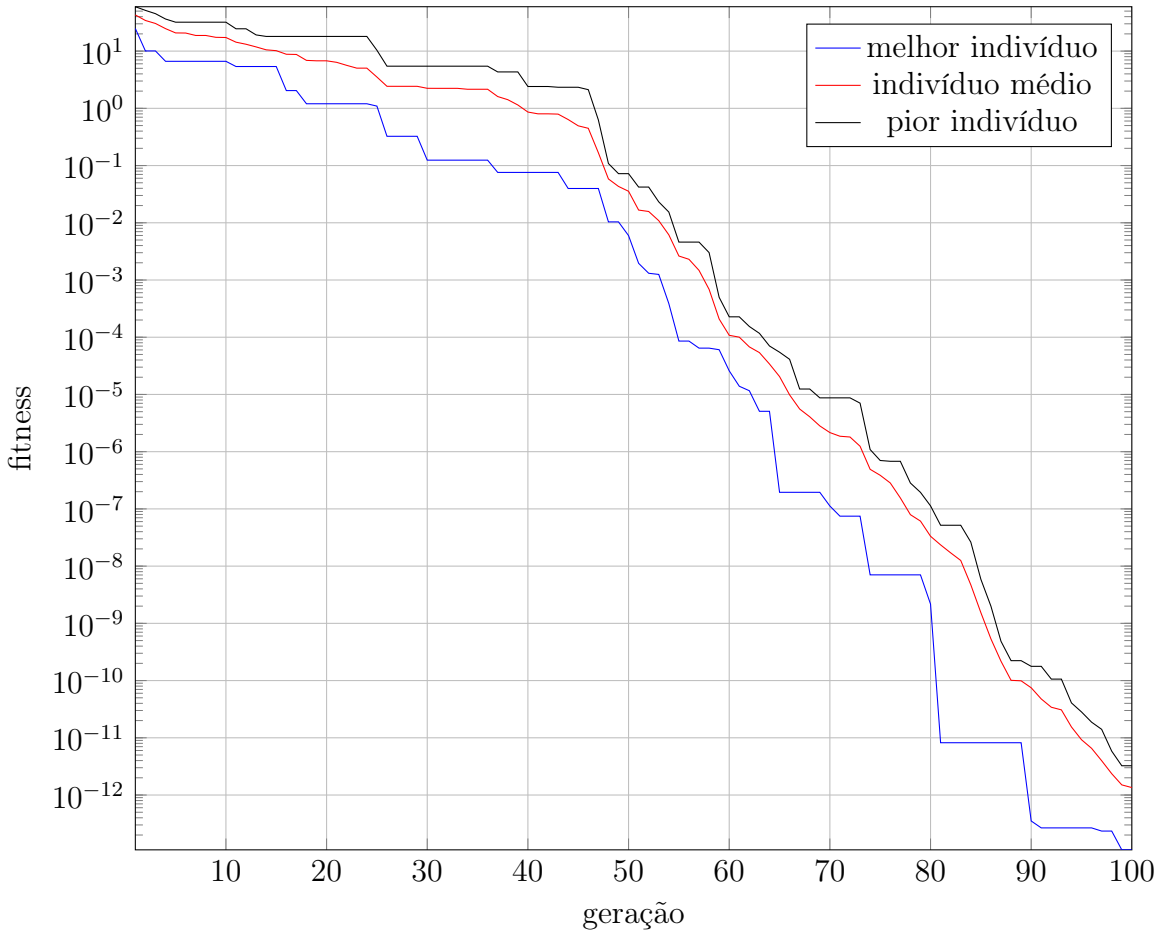

Figura 15: Resultado rodada de otimização da função rastrigin.

4.2 Ajuste de Curva

A maior motivação por de trás do uso de uma arquitetura distribuída no algoritmo de ED é o ganho de velocidade que a abordagem pode proporcionar. O potencial ganho de velocidade da abordagem se torna mais acentuado quanto maior o tempo necessário para o cálculo da função objetivo. Isso ocorre pois nessas situações a duração da comunicação entre os processos se torna menos significativa em comparação com o tempo total de avaliação de um indivíduo.

Portanto para demonstrar o ganho de velocidade do algoritmo desenvolvido em relação a uma abordagem sequencial tradicional foi elaborada uma função objetivo que torna possível avaliar essas características.

A função criada busca aproximar uma curva gerada por uma spline suave de bézier cúbica a parte superior da curva de um aerofólio clark y (Figura 16).

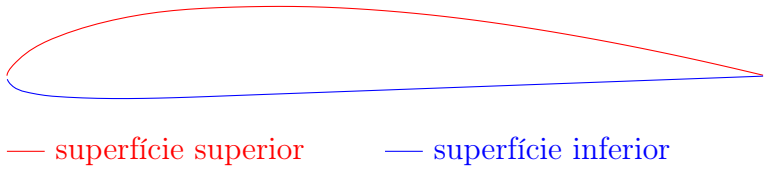

Figura 16: Aerofólio Clark Y.

Para compreendermos o cálculo da função objetivo primeiramente precisamos recapitular o funcionamento de uma curva de bézier, como elas podem ser utilizadas para construir splines e por fim quais as condições necessárias para que essas splines sejam suaves.

Curvas de Bézier

Uma curva de bézier é uma curva paramétrica polinomial que pode ser definida a partir de um conjunto de pontos chamados **pontos de controle** (ver a Figura 17).

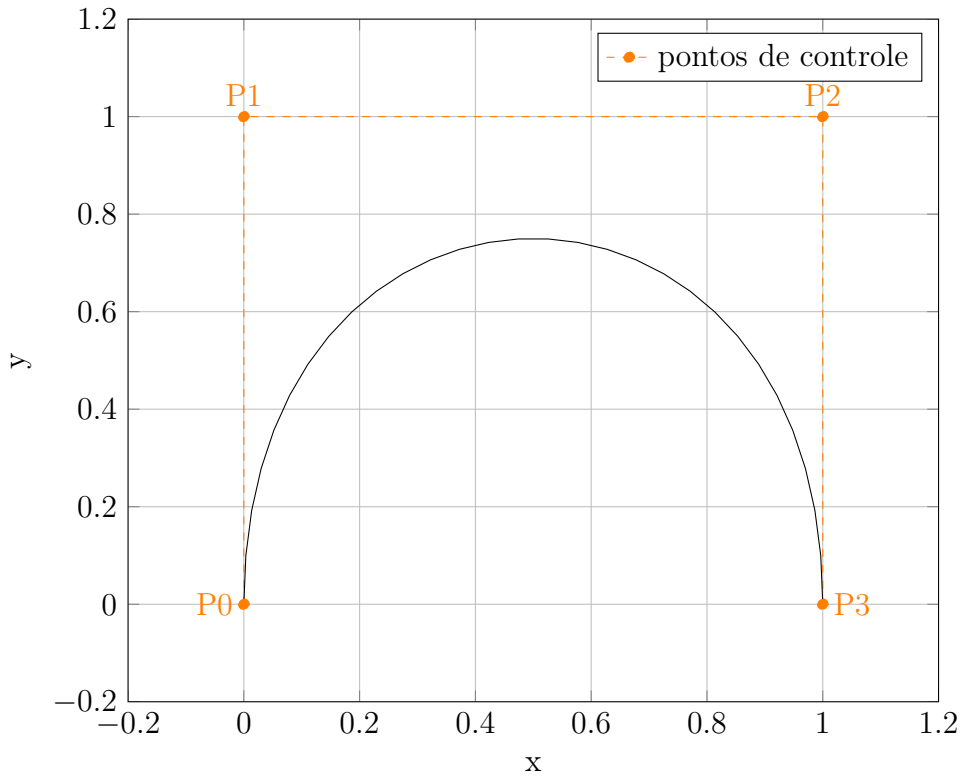

Figura 17: Exemplo curva de bezier cúbica.

Suas coordenadas x_{bezier} e y_{bezier} são funções dos n pontos de controle $\boldsymbol{P}_i = (P_{x,i}, P_{y,i})$ e de uma terceira coordenada paramétrica t que varia de 0 a 1 de acordo com a Equação 7.

$$\begin{aligned} x_{bezier}(t) &= \sum_{i=0}^{n} \binom{n}{i} P_{x,i}\ t^{(n-i)}(1-t)^i \\ y_{bezier}(t) &= \sum_{i=0}^{n} \binom{n}{i} P_{y,i}\ t^{(n-i)}(1-t)^i \end{aligned} \tag{7}$$

Um ponto importante a ser observado é que o primeiro e último ponto de controle sempre pertencem a curva final gerada. A partir da Equação 7 também é possível observar que a ordem do polinômio resultante depende do número de pontos de controle. Uma curva de bézier com 3 pontos é chamada quadrática, enquanto uma com 4 pontos é do tipo cúbica.

Spline de Bézier

Uma spline de bézier nada mais é que a junção de duas ou mais curvas de bézier, de forma que o final de uma curva se ligue ao começo da próxima (Figura 18).

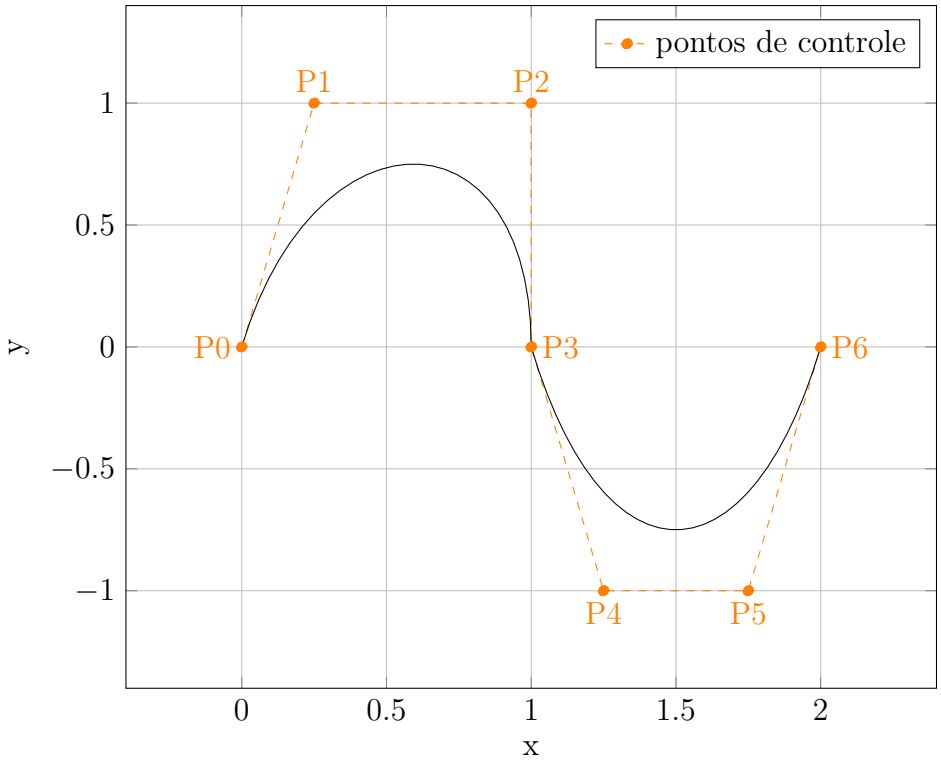

Figura 18: Exemplo spline de bezier cúbica.

Como cada curva de bézier é definida apenas para valores de t entre 0 e 1, para que a spline de bézier também fique definida no mesmo intervalo é feito um mapeamento entre uma coordenada paramétrica da spline u e a coordenada local de cada curva de bézier t.

Esse mapeamento pode ser feito de forma arbitrária. Na implementação utilizada nesse trabalho o mapeamento foi feito de forma que cada curva de bézier que compõe a spline possui um intervalo de mesmo tamanho dentro do intervalo entre 0 e 1s da coordenada paramétrica global u, como descrito na Equação 8.

$$t_i = nu - i \qquad \frac{i}{n} \leq u < \frac{i+1}{n} \qquad \begin{matrix} i = 0, \ldots, n \\ u \in [0, 1] \end{matrix} \tag{8}$$

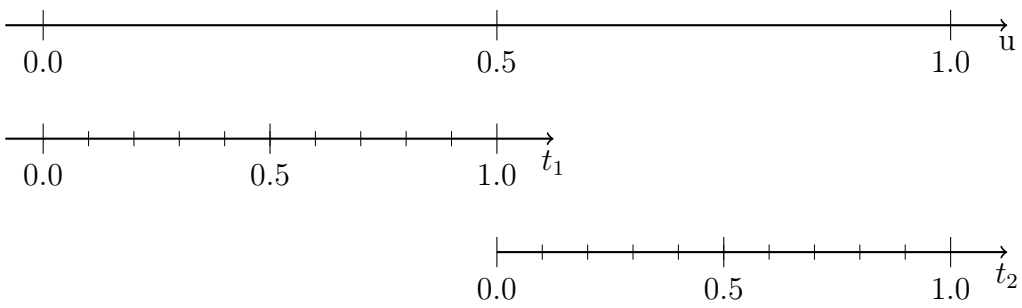

Figura 19: Mapeamento da coordenada global u para as coordenadas locais t_i.

Condição de Suavidade

Para que uma spline de bézier seja suave, isso é, para que não existam mudanças abruptas na curva final, é necessário que os pontos de controle antes e depois de um ponto de junção sejam colineares com o ponto de junção em questão.

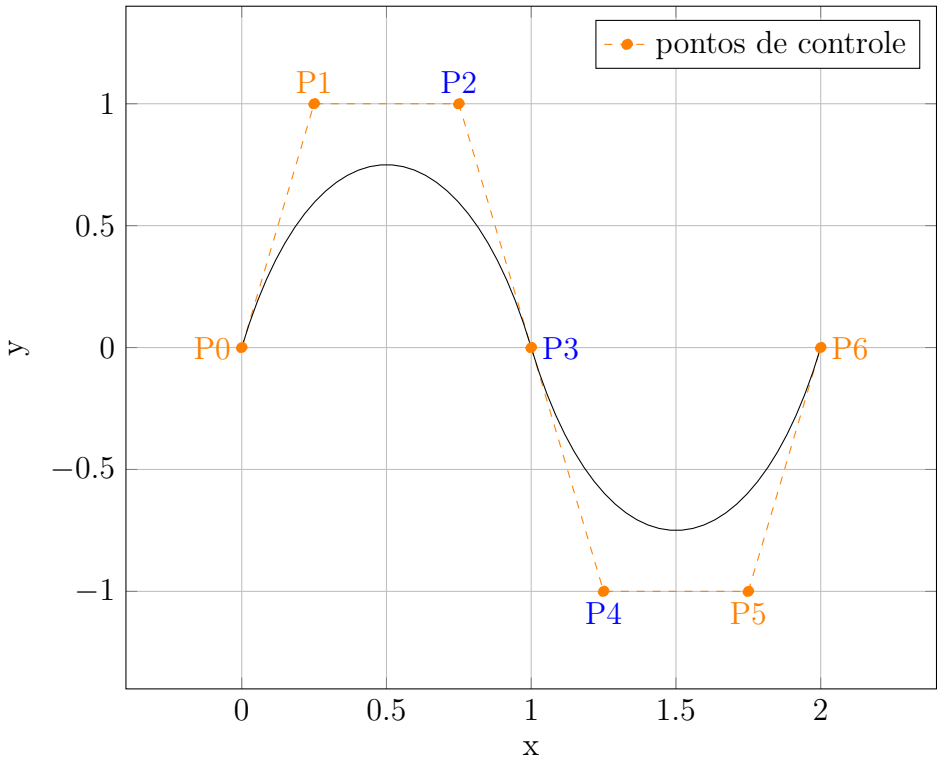

Figura 20: Exemplo spline de bezier cúbica suave.

Observe como os pontos P2 e P4 destacados são colineares ao ponto de junção P3 na Figura 20, formando uma curvatura suave na junção entre as curvas de bézier, ao contrário do que é observado na Figura 18, onde há uma mudança abrupta de direção no ponto de junção.

4.2.1 Função Objetivo

A função objetivo elaborada cria uma spline suave de bézier cúbica de forma que os pontos de controle na curva foram escolhidos de forma a coincidirem com a curva do aerofólio na mesma coordenada x fornecida. Os demais pontos, anteriores e posteriores aos pontos na curva, são colineares para garantir a suavidade da curva, e portanto são definidos por um ângulo com o eixo x e suas distâncias ao ponto pertencente a curva ao qual estão relacionados.

A spline de bézier construída contém 19 pontos de controle no total, sendo que 7 pontos são pertencentes a curva final e são pré-definidos e os outros 12 são definidos por 22 variáveis de projeto, que representam os ângulos e distâncias dos pontos de controle fora da curva com relação aos pontos pertencentes a curva ao qual estão relacionados.

Os pontos pré-definidos tem suas coordenadas x distribuídas conforme a Equação 9 para valores de t igualmente distribuídos entre 0 e 1. O efeito dessa equação na distribuição dos valores de x pode ser observado na Figura 21.

$$x = 1 + sen\left(\frac{t\pi}{2} + \frac{3\pi}{2}\right) \quad t \in [0, 1] \tag{9}$$

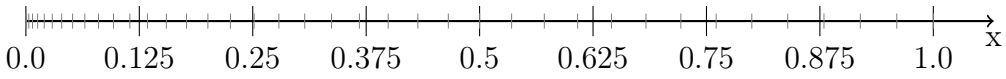

Figura 21: Efeito da Equação 9 na distribuição x.

Dessa forma os pontos de controle na curva da spline de bézier ficam mais concentrados em torno do bordo de ataque do aerofólio, onde é necessário que a curva fique bem definida, o que pode ser observado na Figura 22.

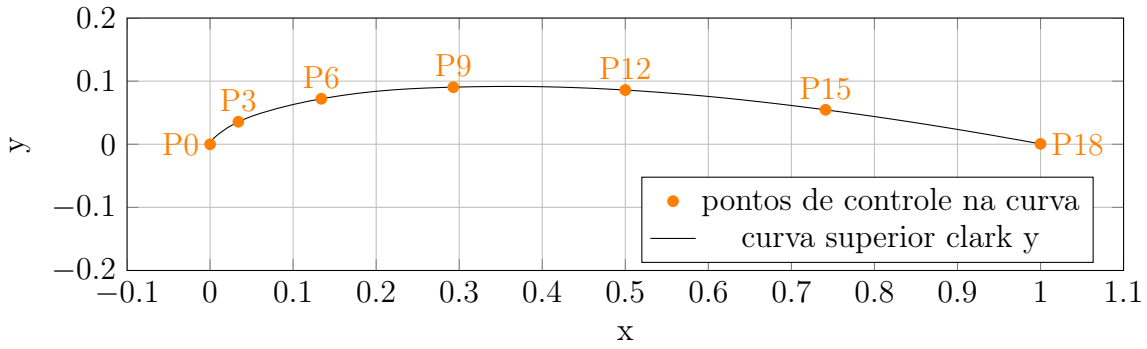

Figura 22: Efeito da Equação 9 na distribuição de pontos de controle.

A função objetivo busca estimar o erro entre os pontos de uma spline de bézier representada por um indivíduo $\boldsymbol{X}_{g,i}$ da rodada de otimização (ver Equação 1) e os pontos reais da superfície do aerofólio nas mesmas coordenadas x.

O cálculo do erro é feito ao somar os quadrados das diferenças entre os valores das coordenadas y do aerofólio nas coordenadas x dos pontos na spline de bézier e os valores das coordenadas y da spline de bézier utilizando 150 pontos (Equação 10 e Equação 11). Os dados da superfície superior do aerofólio são interpolados através de uma spline cúbica natural [6, p.516].

$$f_{obj}(\boldsymbol{X}_{g,i}) = \sum_{i=0}^{150} \left(y_{aerofolio}(x_{bezier,i}(\boldsymbol{X}_{g,i})) - y_{bezier,i}(\boldsymbol{X}_{g,i})\right)^2 \tag{10}$$

$$\begin{aligned} x_{bezier,i}(\boldsymbol{X}_{g,i}) &= x_{bezier}(\boldsymbol{X}_{g,i}, t_i) \\ y_{bezier,i}(\boldsymbol{X}_{g,i}) &= y_{bezier}(\boldsymbol{X}_{g,i}, t_i) \end{aligned} \qquad onde \qquad t_i = \frac{i}{150} \tag{11}$$

Após definida a função objetivo foi escrito um código em kotlin para atuar como AE e foram criadas diversas rodadas de otimização usando o corpo de requisição contido na Listagem 2. Os nomes, posições e limites superior e inferior das variáveis de otimização utilizados estão contidos na Tabela 11.

Listagem 2 Corpo da requisição para criação das rodadas de otimização de ajuste de spline

```
{
    "objectiveFunctionId": 2,
    "populationSize": 15,
    "strategy": "DE_RAND_1_BIN",
    "crossoverProbability": 0.8,
    "perturbationFactor": 0.5,
    "maxGenerations": 500,
    "objectiveFunctionEvaluationTimeoutSeconds": 1,
    "maxObjectiveFunctionReEvaluations": 0,
    "chromosomeElementDetails": [...]
}
```

Posição	Nome da variável	Limite inferior	Limite superior
0	ângulo no ponto de controle 0	40,0	90,0
1	distância do ponto após o ponto de controle 0	0,005	0,1
2	ângulo no ponto de controle 1	0,0	45,0
3	distância do ponto anterior ao ponto de controle 1	0,01	0,1
4	distância do ponto após o ponto de controle 1	0,005	0,1
5	ângulo no ponto de controle 2	-10,0	20,0
6	distância do ponto anterior ao ponto de controle 2	0,01	0,1
7	distância do ponto após o ponto de controle 2	0,01	0,1
8	ângulo no ponto de controle 3	-30,0	5,0
9	distância do ponto anterior ao ponto de controle 3	0,01	0,1
10	distância do ponto após o ponto de controle 3	0,01	0,1
11	ângulo no ponto de controle 4	-30,0	10,0
12	distância do ponto anterior ao ponto de controle 4	0,01	0,1
13	distância do ponto após o ponto de controle 4	0,01	0,1
14	ângulo no ponto de controle 5	-30,0	10,0
15	distância do ponto anterior ao ponto de controle 5	0,01	0,13
16	distância do ponto após o ponto de controle 5	0,01	0,1
17	ângulo no ponto de controle 6	-30,0	10,0
18	distância do ponto anterior ao ponto de controle 6	0,005	0,1
19	distância do ponto após o ponto de controle 6	0,01	0,1
20	ângulo no ponto de controle 7	-30,0	10,0
21	distância do ponto anterior ao ponto de controle 7	0,01	0,1

Tabela 11: Limites inferiores e superiores das variáveis de otimização.

4.2.2 Resultados

Após concluídas as rodadas de otimização os resultados foram armazenados. Alguns exemplos de variáveis de otimização de indivíduos otimizados e os valores da função objetivo associados estão presentes na Tabela 12 a seguir.

Posição	**Rodada 1**	**Rodada 2**	**Rodada 3**	**Rodada 4**	**Rodada 5**	**Rodada 6**
0	72,979	83,021	82,774	51,198	86,141	74,960
1	0,005	0,011	0,012	0,072	0,010	0,017
2	36,200	32,127	32,106	41,935	30,160	35,680
3	0,042	0,013	0,012	0,040	0,012	0,010
4	0,010	0,006	0,024	0,012	0,041	0,013
5	12,941	13,218	13,010	12,075	13,242	12,648
6	0,056	0,067	0,043	0,038	0,022	0,050
7	0,064	0,051	0,079	0,040	0,061	0,027
8	2,138	1,820	3,155	1,801	1,979	1,606
9	0,052	0,064	0,020	0,087	0,050	0,090
10	0,100	0,091	0,023	0,046	0,093	0,079
11	-4,249	-4,477	-4,473	-4,523	-4,567	-4,328
12	0,032	0,058	0,100	0,094	0,047	0,080
13	0,010	0,064	0,100	0,100	0,095	0,100
14	-9,948	-9,549	-10,022	-10,711	-9,488	-9,565
15	0,130	0,117	0,046	0,037	0,082	0,052
16	0,100	0,100	0,096	0,015	0,100	0,100
17	10,000	9,986	-29,511	9,876	2,678	-17,935
18	0,099	0,005	0,009	0,015	0,088	0,005
19	0,024	0,010	0,099	0,010	0,012	0,042
20	-19,889	-16,893	-13,410	-15,184	-15,482	-14,045
21	0,011	0,012	0,100	0,081	0,017	0,037
fitness	0,002059	0,000916	0,000980	0,007668	0,000862	0,001764

Tabela 12: Exemplos de indivíduos otimizados e o valor da função objetivo correspondente.

Assim como no caso do problema de otimização anterior, os valores do melhor, pior e média dos indivíduos de cada geração de uma rodada de exemplo pode ser encontrada na Figura 23 a seguir.

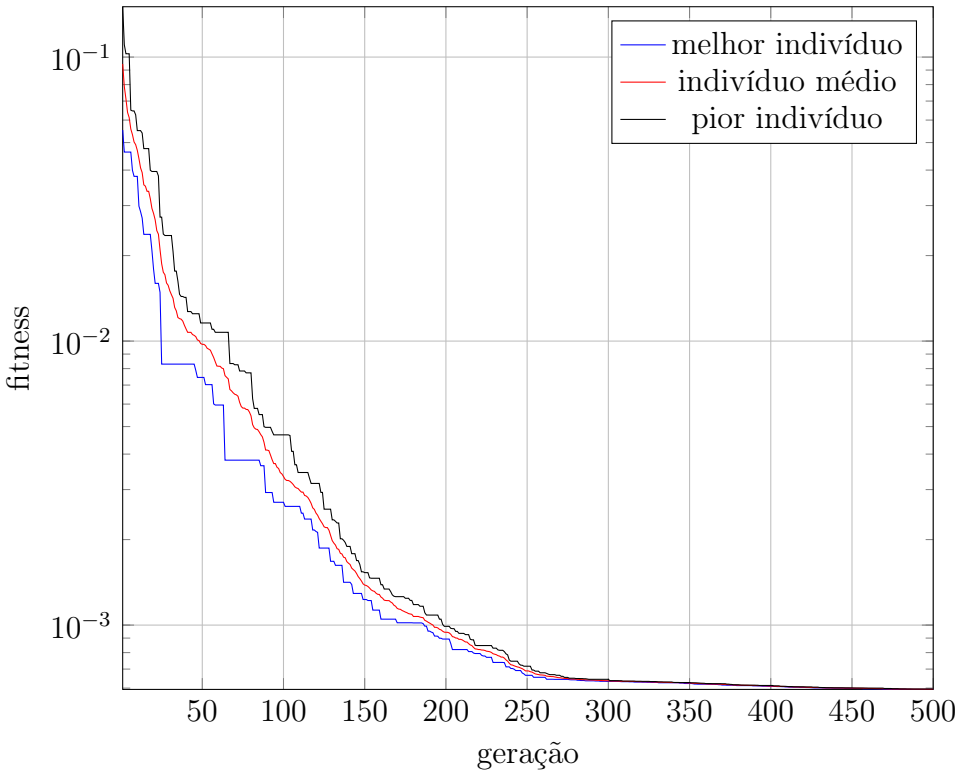

Figura 23: Resultado rodada de otimização de ajuste de spline.

Na Figura 24 é possível visualizar a spline de bézier formada por um indivíduo aleatório não otimizado, enquanto na Figura 25 é possível notar o resultado da curva formada por um indivíduo ao final de um processo de otimização, demonstrando o sucesso do processo na tarefa de ajustar a curva formada pela spline de bézier.

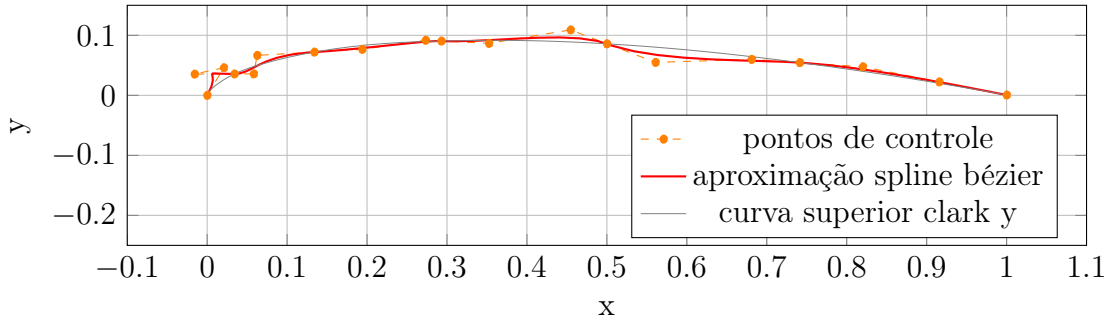

Figura 24: Aproximação da spline de bézier de um indivíduo aleatório.

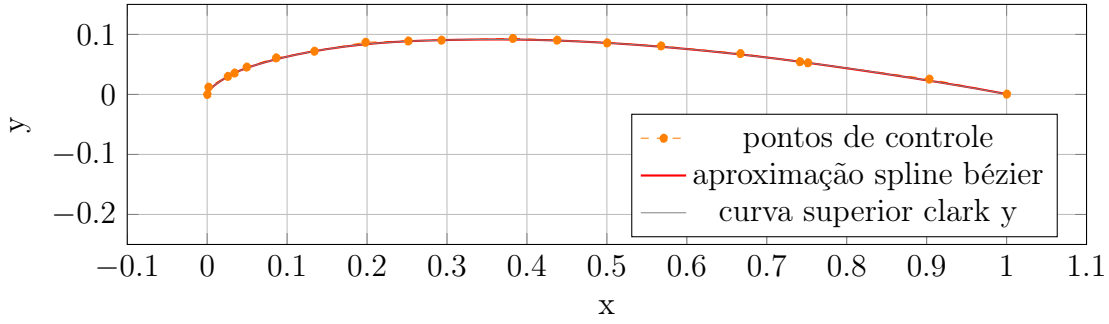

Figura 25: Aproximação da spline de bézier de um indivíduo otimizado.

4.2.3 Análise de Performance

Para avaliar o aumento de velocidade ao se utilizar várias instâncias da AE foi criada uma imagem de contêiner *docker* da aplicação. O *docker swarm* foi utilizado para instanciar replicas da aplicação e, para cada configuração de número de contêineres, foram feitas 5 rodadas de otimização. As rodadas de otimização foram executadas em um computador com 32Gb de memória DDR4 de 3600Mhz e processador AMD Ryzen 9 5950x de 16 núcleos e 32 threads e os tempos de execução e resultados de cada rodada foram armazenados.

Os resultados do tempo de duração das rodadas de otimização em relação ao número de contêineres utilizado está disposto na Tabela 13 e apresentado na Figura 26 a seguir.

Número de Contêineres	**Duração da Rodada de Otimização (s)**
1	$148,167 \pm 3,545$
2	$80,833 \pm 2,041$
3	$53,667 \pm 2,658$
4	$45,000 \pm 2,530$
5	$37,000 \pm 2,530$
6	$36,667 \pm 2,658$
7	$35,600 \pm 0,894$
8	$28,600 \pm 2,608$
9	$28,200 \pm 3,834$

Tabela 13: Número de contêineres vs Duração da Rodada de Otimização.

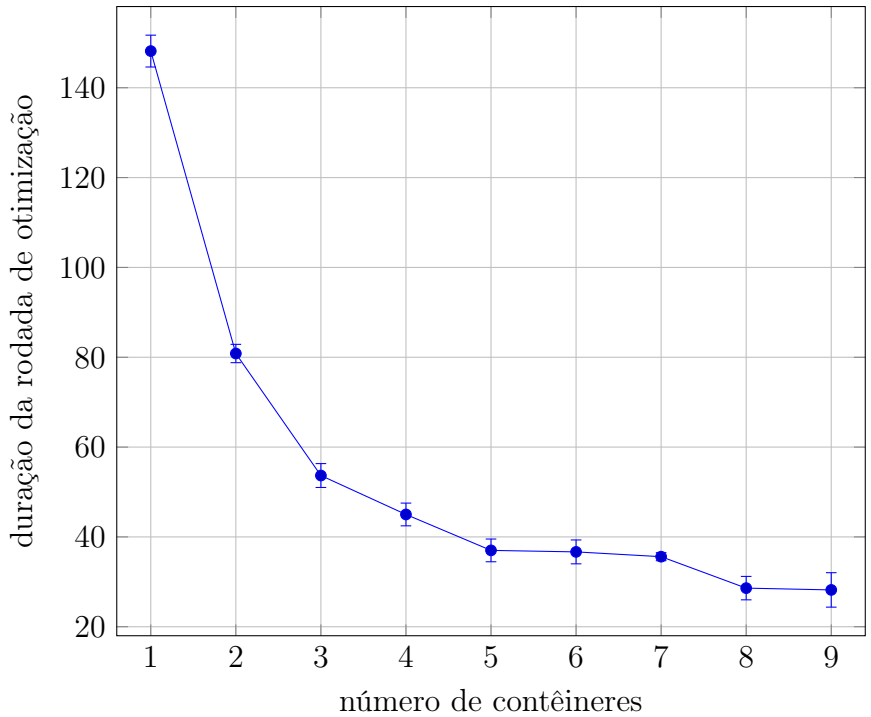

Figura 26: Número de Contêineres x Duração da Rodada de Otimização.

O aumento de velocidade Av de uma execução com N contêineres pode ser calculado pela Equação 12, onde T_N é o tempo de execução utilizando N contêineres e T_1 é o tempo de execução utilizando apenas 1 contêiner. Os resultados obtidos são comparados com o aumento teórico de velocidade com a mesma quantidade de contêineres na Figura 27.

$$Av = \frac{T_1}{T_N} \tag{12}$$

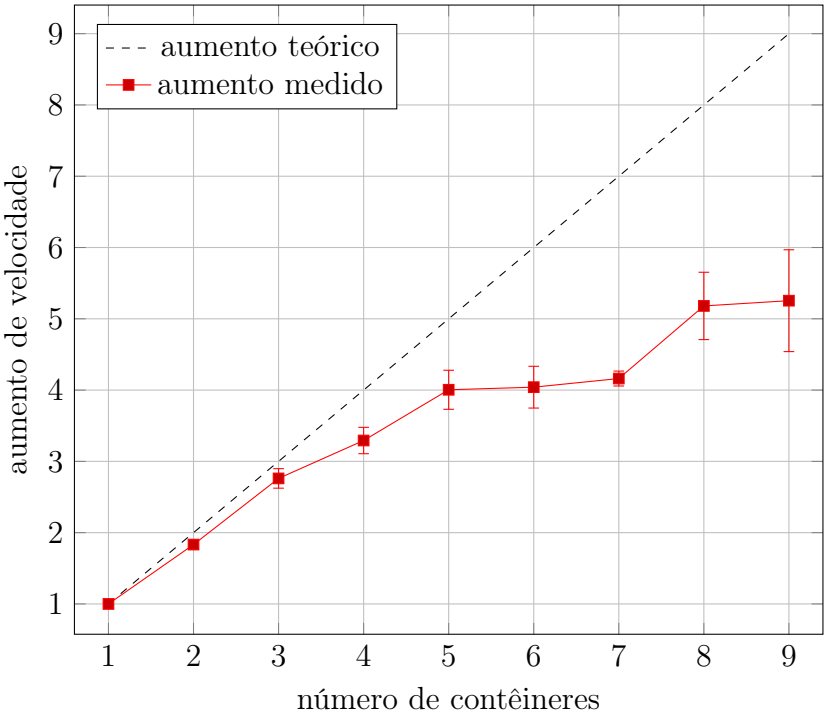

Figura 27: Aumento de velocidade x número de contêineres.

É possível notar uma diminuição no aumento de velocidade conforme o aumento de contêineres utilizados. Após o uso de uma determinada quantidade de contêineres o processo começa a desacelerar e deixa de apresentar ganhos, obtendo uma certa instabilidade no tempo de conclusão.

Um dos fatores que podem levar a esse comportamento é o fato de que os contêineres foram criados dentro de um mesmo computador e competem pelos mesmos recursos, aumentando o tempo médio da execução de cada indivíduo. Outros fatores que diminuem o ganho de velocidade é possibilidade dos indivíduos não serem distribuídos igualmente entre as AE como esperado e o aumento do tempo de espera que ocorre entre as gerações.

Devido a natureza do processo, se um dos contêineres demorar durante o cálculo da função objetivo ele pode manter as outras instâncias paradas aguardando a publicação do seu resultado para que haja o avanço da geração e novos indivíduos sejam gerados. Para evitar esse tipo de situação é importante configurar o parâmetro `objectiveFunctionEvaluationTimeoutSeconds` corretamente, assim se uma da instâncias demorar mais que o esperado outra pode assumir seu lugar e completar o processo no tempo correto.

Durante a execução da AE utilizando apenas um contêiner o tempo de duração das chamadas às rotas `/optimizationRun/{id}/chromosome/notEvaluated` e `/chromosome/{id}/evaluationResult` foram registrados e as distribuições de frequência resultantes

estão disponíveis nas figuras 28 e 29. O tempo de execução da função objetivo também foi armazenado e a sua distribuição de frequências está representada na Figura 30.

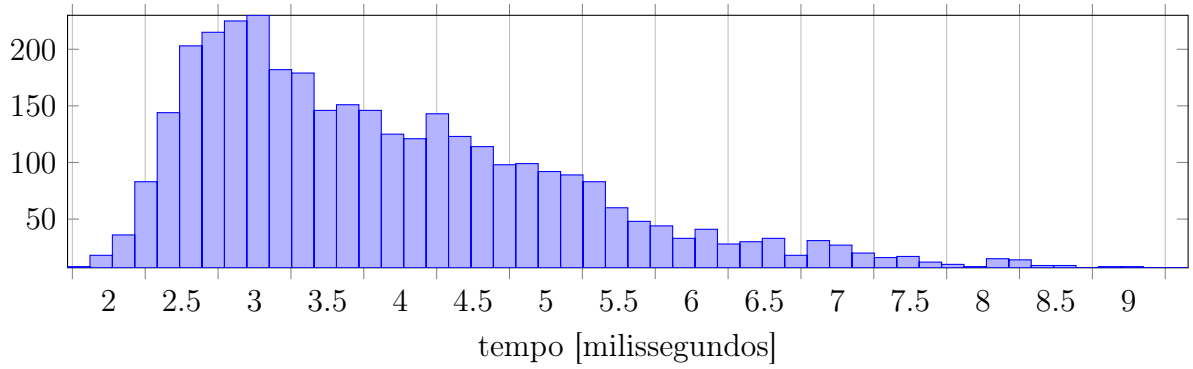

Figura 28: Distribuição de tempo para a chamada Get Chromosome (ms).

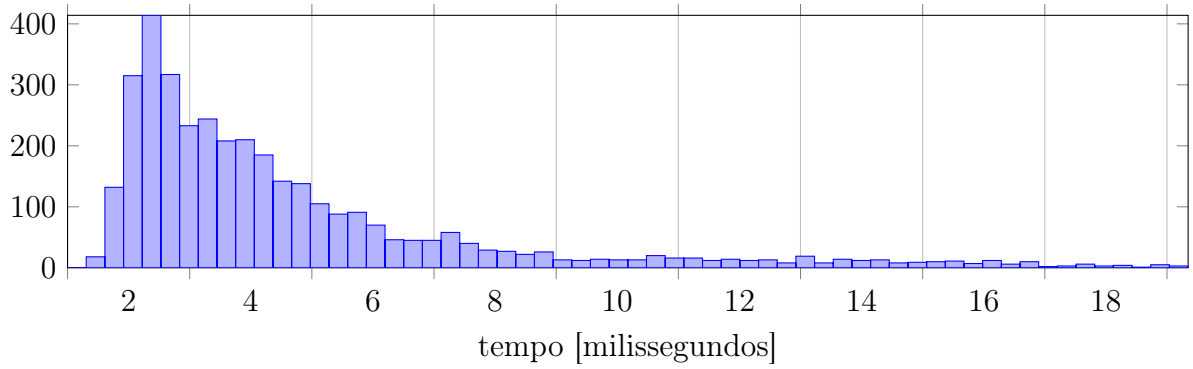

Figura 29: Distribuição de tempo para a chamada Publish Evaluation (ms).

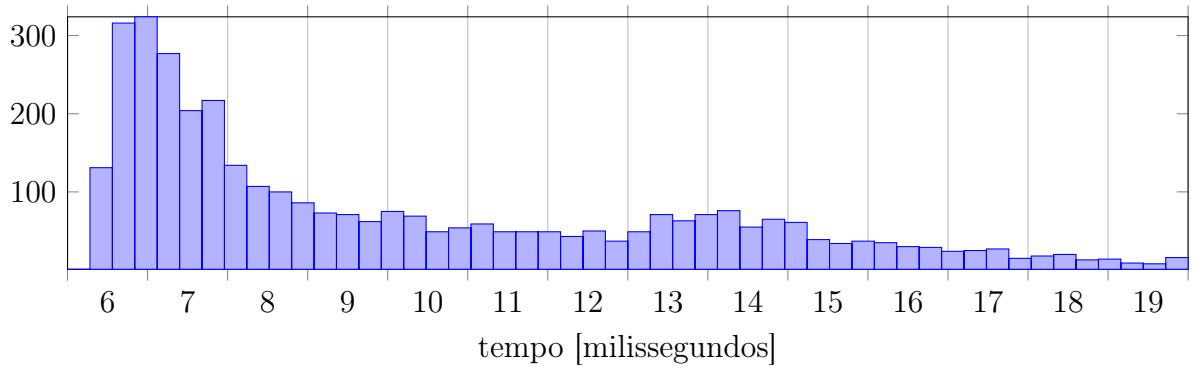

Figura 30: Distribuição de tempo para o cálculo da função objetivo (ms).

Em cada uma das rodadas de otimização utilizando N contêineres foi selecionado um contêiner participante para armazenar a duração de cada operação que compõe o ciclo de avaliação de um indivíduo, os tempos total e médio gasto em cada operação estão dispostos na Tabela 14 a seguir.

N°	**getChromosome (sucesso)**		**getChromosome (falha)**		**objective-Function**		**publish-Evaluation-Result**		**indivíduo**	
	total	média	total	média	total	média	total	média	total	média
1	20,21s	2,69ms	0,00s	0,00ms	82,69s	11,03ms	32,21s	4,30ms	135,12s	18,01ms
2	10,58s	2,82ms	4,09s	2,01ms	40,85s	10,99ms	15,24s	4,10ms	70,76s	19,91ms
3	7,48s	2,98ms	2,30s	2,04ms	27,51s	11,02ms	10,19s	4,08ms	47,47s	20,12ms
4	5,68s	3,07ms	4,01s	1,99ms	20,42s	11,12ms	7,65s	4,16ms	37,76s	20,35ms
5	4,89s	3,26ms	2,58s	2,02ms	16,80s	11,22ms	6,53s	4,36ms	30,80s	20,86ms
6	4,69s	3,77ms	5,41s	2,20ms	14,04s	11,33ms	5,56s	4,48ms	29,70s	21,79ms
7	4,13s	3,80ms	6,24s	2,01ms	12,31s	11,37ms	4,96s	4,58ms	27,63s	21,75ms
8	3,71s	3,91ms	3,48s	2,03ms	11,01s	11,62ms	4,38s	4,62ms	22,59s	22,19ms
9	3,39s	4,13ms	8,37s	1,94ms	9,51s	11,63ms	3,81s	4,66ms	25,09s	22,36ms

Tabela 14: Tempo total e médio de cada operação de um contêiner x número de contêineres utilizados.

É possível notar que conforme vão sendo utilizados mais contêineres nas rodadas de otimização o tempo total gasto por cada contêiner participante em cada operação diminui seguindo aproximadamente a proporção T_1/N, no entanto como o tempo médio de cada operação individual também aumenta com o uso de um número maior de contêineres há um desvio do valor esperado.

Também é interessante notar que conforme aumentamos a quantidade de contêineres utilizados o tempo total da operação **getChromosome (falha)**, que representa o tempo ocioso gasto pelos contêineres esperando que novos indivíduos estejam disponíveis para avaliação, tende a oscilar e diminuir em valores que são divisores do número de indivíduos por população.

5 Conclusão

5.1 Contribuições

Diante do que foi apresentado e dos resultados obtidos com a abordagem desenvolvida foi demonstrado que os objetivos propostos foram atingidos com sucesso. A utilização de um servidor HTTP REST com uma especificação openAPI garante que a AE possa ser escrita em qualquer linguagem e consiga interagir com a AM com facilidade. Também foi demonstrado a ganho de performance da abordagem distribuída e a resiliência do algoritmo de contornar eventuais erros durante a avaliação pelas AE, tanto quanto a capacidade de se recuperar de reinicializações da AM.

Desde que o nó executando a AM esteja disponível pela internet ou por uma rede privada e os nós trabalhadores estejam executando a mesma AE um processo de otimização pode ser realizado utilizando computação em grade, onde computadores heterogêneos interagem para realizar a mesma tarefa, ou através de computação em nuvem, onde os recursos computacionais podem ser alugados de um provedor conhecido.

Graças a persistência dos dados em disco e ao funcionamento a partir de requisições, o processo possui a capacidade de ser pausado e retomado a qualquer momento, mesmo que a aplicação sofra algum desligamento ou reinicialização.

5.2 Sugestões para Trabalhos Futuros

Apesar do uso da ED sem modificações levar a bons resultados, trabalhos futuros podem explorar melhorias comumente aplicadas para aumentar a velocidade de convergência do método, como a inicialização dos indivíduos da população inicial por hipercubo latino, atualização dinâmica de parâmetros como tamanho da população, probabilidade de crossover e fator de perturbação, e o uso de outras estratégias de mutação como a *current-to-pbest* como proposto por Zhang e Sanderson em [43].

Outras possibilidades são a adição de opções de configuração opcionais da AM para que seja possível a comunicação com os nós trabalhadores usando sistemas de fila intermediado por um *message broker* como rabbitMQ e Kakfa ou então o uso de outro protocolo de comunicação como o gRPC (google Remote Procedure Call) que trariam benefícios na velocidade de comunicação entre os processos.

A implementação de um sistema de *long pooling* ou a transmissão de mais de um indivíduo por requisição também poderiam gerar melhorias na comunicação entre os processos, já que reduziria a quantidade de requisições necessárias para obter os dados dos indivíduos a serem calculados e com isso reduziria a carga sobre a AM e, portanto, a ineficiência gerada pelo tempo de comunicação entre os processos.

Por fim também seria interessante o desenvolvimento de uma aplicação gráfica *front end* que facilite a interação do usuário com a AM, providenciando uma interface de usuário amigável para que o operador consiga criar e inspecionar os recursos REST com mais facilidade.

Referências

[1] W. Abu Abed. "Differential Evolution for Finite Element Model Updating: Algorithm and Application in Structural Analysis". Em: *Proceedings of the 10th European Workshop on Structural Health Monitoring*. 2024.

[2] Enrique Alba e José Ma Troya. "Cellular Evolutionary Algorithms: Evaluating the Influence of Ratio". Em: *Parallel Problem Solving from Nature PPSN VI*. Ed. por Marc Schoenauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 29–38. ISBN: 978-3-540-45356-7.

[3] Peter A. Bernstein e Eric Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann, 2002.

[4] Daniel P. Bovet e Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[5] Tadeusz Burczynski e Waclaw Kus. "Optimization of Structures Using Distributed and Parallel Evolutionary Algorithms". Em: *Parallel Processing and Applied Mathematics*. Ed. por Roman Wyrzykowski et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 572–579. ISBN: 978-3-540-24669-5.

[6] Raymond P Canale e Steven C Chapra. *Numerical methods for engineers*. Mcgraw-hill Education-Europe, 2014, pp. 515–516.

[7] X. Chen, F. Liu e H.-A. Jacobsen. "A Publish/Subscribe Model for Event-driven Systems". Em: *Journal of Distributed and Parallel Databases* 28.1 (2010), pp. 3–29.

[8] E Chong. *An Introduction to Optimization*. 2013.

[9] M. Dubreuil, C. Gagne e M. Parizeau. "Analysis of a master-slave architecture for distributed evolutionary computations". Em: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 36.1 (fev. de 2006), pp. 229–235. ISSN: 1941-0492. DOI: 10.1109/TSMCB.2005.856724.

[10] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[11] David B. Fogel. "Artificial Intelligence through Simulated Evolution". Em: *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1998, pp. 227–296. DOI: 10.1109/9780470544600.ch7.

[12] Gianluigi Folino e Giandomenico Spezzano. "P-CAGE: An Environment for Evolutionary Computation in Peer-to-Peer Systems". Em: *Genetic Programming*. Ed. por Pierre Collet et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 341–350. ISBN: 978-3-540-33144-5.

[13] Michel Gendreau, Jean-Yves Potvin et al. *Handbook of metaheuristics*. Vol. 2. Springer, 2010.

[14] Fred Glover. "Future paths for integer programming and links to artificial intelligence". Em: *Computers & Operations Research* 13.5 (1986). Applications of Integer Programming, pp. 533–549. ISSN: 0305-0548. DOI: https://doi.org/10.1016/0305-0548(86)90048-1. URL: https://www.sciencedirect.com/science/article/pii/0305054886900481.

[15] Guy Harrison e Steven Locke. *Message Brokers in Practice*. O'Reilly Media, 2013.

[16] Xiaolong He et al. "Robust aerodynamic shape optimization—from a circle to an airfoil". Em: *Aerospace Science and Technology* 87 (abr. de 2019), pp. 48–61. DOI: 10.1016/j.ast.2019.01.051.

[17] F. Herrera, M. Lozano e C. Moraga. "Hierarchical distributed genetic algorithms". Em: *International Journal of Intelligent Systems* 14.11 (1999), pp. 1099–1121. DOI: https://doi.org/10.1002/(SICI)1098-111X(199911)14:11<1099::AID-INT3>3.0.CO;2-0. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291098-111X%28199911%2914%3A11%3C1099%3A%3AAID-INT3%3E3.0.CO%3B2-0. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291098-111X%28199911%2914%3A11%3C1099%3A%3AAID-INT3%3E3.0.CO%3B2-0.

[18] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, abr. de 1992. ISBN: 9780262275552. DOI: 10.7551/mitpress/1090.001.0001. URL: https://doi.org/10.7551/mitpress/1090.001.0001.

[19] IBM. *IBM MQ: Technical Overview*. 2020. URL: https://www.ibm.com/docs/en/ibm-mq.

[20] J. Kennedy e R. Eberhart. "Particle swarm optimization". Em: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4. 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

[21] S. Kirkpatrick, C. D. Gelatt e M. P. Vecchi. "Optimization by Simulated Annealing". Em: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. eprint: https://www.science.org/doi/pdf/10.1126/science.220.4598.671. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[22] J. R. Koza. "Hierarchical genetic algorithms operating on populations of computer programs". Em: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*. Ed. por N. S. Sridharan. Vol. 1. Detroit, MI, USA: Morgan Kaufmann, 20-25 8 de 1989, pp. 768–774. URL: http://dl.acm.org/citation.cfm?id=1623755.1623877.

[23] Jay Kreps, Neha Narkhede e Jun Rao. "Kafka: A Distributed Messaging System for Log Processing". Em: *Proceedings of the ACM International Conference on Distributed Event-Based Systems (DEBS)* (2011), pp. 1–7.

[24] Dudy Lim et al. "Efficient Hierarchical Parallel Genetic Algorithms using Grid computing". Em: *Future Generation Computer Systems* 23.4 (2007), pp. 658–670. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2006.10.008. URL: https://www.sciencedirect.com/science/article/pii/S0167739X06001920.

[25] Daniel Molina, Francisco Moreno-García e Francisco Herrera. "Analysis among winners of different IEEE CEC competitions on real-parameters optimization: Is there always improvement?" Em: *2017 IEEE Congress on Evolutionary Computation (CEC)*. 2017, pp. 805–812. DOI: 10.1109/CEC.2017.7969392.

[26] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[27] H. Pierreval e J.-L. Paris. "Distributed evolutionary algorithms for simulation optimization". Em: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 30.1 (2000), pp. 15–24. DOI: 10.1109/3468.823477.

[28] Singiresu S. Rao. "Introduction to Optimization". Em: *Engineering Optimization*. John Wiley Sons, Ltd, 2009. Cap. 1, pp. 1–62. ISBN: 9780470549124. DOI: https://doi.org/10.1002/9780470549124.ch1. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470549124.ch1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470549124.ch1.

[29] I. Rechenberg. "Evolutionsstrategien". Em: *Simulationsmethoden in der Medizin und Biologie*. Ed. por Berthold Schneider e Ulrich Ranft. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 83–114. ISBN: 978-3-642-81283-5.

[30] Ingo Rechenberg. "Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution". Em: 1973. URL: https://api.semanticscholar.org/CorpusID:60975248.

[31] Gautam Roy et al. "A distributed pool architecture for genetic algorithms". Em: *2009 IEEE Congress on Evolutionary Computation*. 2009, pp. 1177–1184. DOI: 10.1109/CEC.2009.4983079.

[32] Birgitt Schönfisch e André de Roos. "Synchronous and asynchronous updating in cellular automata". Em: *Biosystems* 51.3 (1999), pp. 123–143. ISSN: 0303-2647. DOI: https://doi.org/10.1016/S0303-2647(99)00025-8. URL: https://www.sciencedirect.com/science/article/pii/S0303264799000258.

[33] Mourad Sefrioui e Jacques Périaux. "A Hierarchical Genetic Algorithm Using Multiple Models for Optimization". Em: *Parallel Problem Solving from Nature PPSN VI*. Ed. por Marc Schoenauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 879–888. ISBN: 978-3-540-45356-7.

[34] Rainer Storn e Kenneth Price. "Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces". Em: *Journal of Global Optimization* 23 (jan. de 1995).

[35] Dirk Sudholt. "Parallel Evolutionary Algorithms". Em: *Springer Handbook of Computational Intelligence*. Ed. por Janusz Kacprzyk e Witold Pedrycz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 929–959. ISBN: 978-3-662-43505-2. DOI: 10.1007/978-3-662-43505-2_46. URL: https://doi.org/10.1007/978-3-662-43505-2_46.

[36] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009. ISBN: 0470278587.

[37] Stefan Tilkov. "A Brief Introduction to REST". Em: *InfoQ* (2007).

[38] Maarten Van Steen e Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[39] Alvaro Videla e Jason J. Williams. *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning Publications, 2012.

[40] Bobby Woolf e Gregor Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[41] Chunyin Wu e Ko-Ying Tseng. "Topology Optimization of Structure Using Differential Evolution". Em: *Journal of Systemics, Cybernetics and Informatics* 2 (jan. de 2007).

[42] Chunmei Zhang, Jie Chen e Bin Xin. "Distributed memetic differential evolution with the synergy of Lamarckian and Baldwinian learning". Em: *Applied Soft Computing* 13.5 (2013), pp. 2947–2959. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2012.02.028. URL: https://www.sciencedirect.com/science/article/pii/S1568494612001226.

[43] Jingqiao Zhang e Arthur C. Sanderson. "JADE: Adaptive Differential Evolution With Optional External Archive". Em: *IEEE Transactions on Evolutionary Computation* 13.5 (2009), pp. 945–958. DOI: 10.1109/TEVC.2009.2014613.