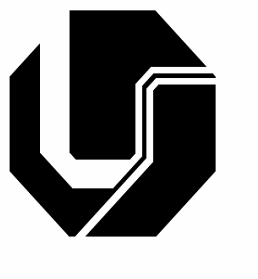
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELETRICA
POS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA



Aplicação de Parâmetros Autoadaptativos de Ajustes de Domínio na Otimização de Funções Contínuas Utilizando Colônia de Formigas

Jairo Gervásio de Freitas
Orientando

Keiji Yamanaka
Orientador

Uberlândia - MG

2021

Jairo Gervásio de Freitas

Aplicação de Parâmetros Autoadaptativos de Ajustes de Domínio na Otimização de Funções Contínuas Utilizando Colônia de Formigas

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, como requisito parcial para obtenção do título de Doutor em Ciências

Área de Concentração:
Inteligência Artificial

Orientador:
Keiji Yamanaka

Uberlândia - MG

2021

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

F866 2021

Freitas, Jairo Gervasio de, 1981-
Aplicação de parâmetros autoadaptativos de ajustes de domínio na otimização de funções contínuas utilizando colônia de formigas [recurso eletrônico] / Jairo Gervasio de Freitas. - 2021.

Orientador: Keiji Yamanaka.
Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-graduação em Engenharia Elétrica.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.te.2021.596
Inclui bibliografia.

1. Engenharia elétrica. I. Yamanaka, Keiji,1956-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Engenharia Elétrica. III. Título.

CDU: 621.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

Jairo Gervásio de Freitas

Aplicação de Parâmetros Autoadaptativos de Ajustes de Domínio na Otimização de Funções Contínuas Utilizando Colônia de Formigas

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, como requisito parcial para obtenção do título de Doutor em Ciências

Área de Concentração
Inteligência Artificial

Uberlândia, 21 de outubro de 2021

_____
Prof. PhD. Keiji Yamanaka – UFU

_____
Prof. PhD. Igor Santos Peretta – UFU

_____
Prof. PhD. Wesley Calixto Pacheco – IFG

_____
Prof. Dr. José Ricardo Gonçalves Manzan – IFTM

_____
Prof. Dr. Marcelo Rodrigues de Sousa – UFU

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Engenharia Elétrica
Av. João Naves de Ávila, 2121, Bloco 3N - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4707 - www.posgrad.feelt.ufu.br - copel@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Engenharia Elétrica				
Defesa de:	Tese de Doutorado, 293, PPGEELT				
Data:	Vinte e um de outubro de dois mil e vinte e um	Hora de início:	08:30	Hora de encerramento:	12:00
Matrícula do Discente:	11613EEL019				
Nome do Discente:	Jairo Gervásio de Freitas				
Título do Trabalho:	Aplicação de parâmetros autoadaptativos de ajustes de domínio na otimização de funções contínuas utilizando colônia de formigas				
Área de concentração:	Processamento da informação				
Linha de pesquisa:	Inteligência artificial				
Projeto de Pesquisa de vinculação:	Coordenador do projeto: Keiji Yamanaka. Título do projeto: Estudo e Aplicações de Técnicas de Inteligência Computacional. Agência financiadora: ___ Número do processo na agência financiadora: ___. Vigência do projeto: 2010 - atual.				

Reuniu-se por meio de videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Engenharia Elétrica, assim composta: Professores Doutores: Igor Santos Peretta - FEELT/UFU; Marcelo Rodrigues de Sousa - FEELT/UFU; Wesley Calixto Pacheco - IFG; José Ricardo Gonçalves Manzan - IFTM; Keiji Yamanaka - FEELT/UFU, orientador(a) do(a) candidato(a).

Iniciando os trabalhos o(a) presidente da mesa, Dr(a). Keiji Yamanaka, apresentou a Comissão Examinadora e o candidato(a), agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor(a) presidente concedeu a palavra, pela ordem sucessivamente, aos(às) examinadores(as), que passaram a arguir o(a) candidato(a). Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o(a) candidato(a):

Aprovado(a).

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **José Ricardo Gonçalves Manzan**, **Usuário Externo**, em 21/10/2021, às 12:08, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Igor Santos Peretta**, **Professor(a) do Magistério Superior**, em 21/10/2021, às 12:09, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcelo Rodrigues de Sousa**, **Professor(a) do Magistério Superior**, em 21/10/2021, às 12:10, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Wesley Pacheco Calixto**, **Usuário Externo**, em 21/10/2021, às 12:12, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Keiji Yamanaka**, **Professor(a) do Magistério Superior**, em 21/10/2021, às 12:12, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3118194** e o código CRC **ACA07D49**.

Referência: Processo nº 23117.069578/2021-10 SEI nº 3118194

AGRADECIMENTOS:

Primeiramente, gostaria de agradecer a Deus, por me guiar durante todo este processo, tornando possível a realização de mais essa conquista.

À minha esposa pelo amor, carinho e compreensão nos momentos difíceis, e por se alegrar comigo nos momentos felizes. Essa conquista é nossa!

Às minhas filhas Isabela e Maria Luísa, que me ensinaram como é o maior amor que existe. Desde que vocês nasceram, tudo que faço é por vocês.

Ao meu primogênito Davi, por simplesmente me deixar ser seu pai, realizando meu sonho de ter meu filho homem.

Aos meus pais, por serem meus maiores exemplos de vida e me ensinarem as coisas mais importantes: as coisas que não estão nos livros.

Às minhas irmãs e sobrinhos, pela torcida silenciosa pelo sucesso deste trabalho.

Ao IFTM e a UFU, por esse projeto de Doutorado Interinstitucional, que possibilitou que tantas pessoas pudessem se qualificar.

Aos meus colegas de IFTM e de doutorado, pelo incentivo e colaboração durante esses anos de curso. Em especial, ao meu amigo Reginaldo, que infelizmente não pode concluir este projeto.

Gostaria de agradecer bastante ao meu orientador Keiji Yamanaka, por essa parceria que vem desde o tempo de mestrado, por compartilhar todo o seu conhecimento e experiência sobre a área, e pela paciência e compreensão com minhas dificuldades e falta de tempo.

“Você me faz capaz, quando já não posso mais.
Você me faz voar, quando não posso nem andar.
Teu amor me faz seguir, quando nem sei por onde ir.
Quando penso que estou só, ouço seus passos por aqui”
Ziza Fernandes

RESUMO:

Freitas, J. G. **Aplicação de parâmetros autoadaptativos de ajuste de domínio na otimização de funções contínuas utilizando colônia de formigas**, Faculdade de Engenharia Elétrica, Universidade Federal de Uberlândia, 2021, 75p.

Existe uma grande variedade de métodos computacionais com o objetivo de resolver problemas de otimização. Dentre eles, existem várias estratégias que são derivadas do conceito de otimização de colônia de formigas (ACO). No entanto, a grande maioria destes métodos são algoritmos de busca de alcance limitado, ou seja, encontram a solução ótima, desde que o domínio fornecido contenha essa solução. A pesquisa propõe a utilização de um algoritmo de busca de amplo alcance, ou seja, que busca a solução ótima, com sucesso na maioria das vezes, mesmo que o domínio inicial fornecido não contenha essa solução, pois o domínio inicial será ajustado até que seja localizado um domínio que contém a solução. Esse algoritmo denominado ARACO, derivado do algoritmo RACO, possibilita a obtenção de melhores resultados, através de estratégias para aceleração dos parâmetros responsáveis por ajustar o domínio fornecido, em momentos oportunos e de, caso haja estagnação do algoritmo, ampliação do domínio em torno da melhor solução encontrada, para impedir que o algoritmo fique preso em um mínimo local. Através dessas estratégias, ARACO obtém resultados melhores que seus antecessores, em relação ao número de avaliações de funções necessárias para encontrar a solução ótima, além de conseguir cem por cento de taxa de sucesso em praticamente todas as funções testadas, demonstrando assim ser um algoritmo de alto desempenho e confiabilidade. O algoritmo foi testado em algumas funções de *benchmark* clássicas e também nas funções de *benchmark* do IEEE *Congress of Evolutionary Computation Benchmark Test Functions* (CEC 2019 *100-Digit Challenge*).

Palavras-chave: Otimização por colônia de formigas, otimização de funções contínuas, problemas de otimização, computação evolutiva

ABSTRACT:

Freitas, J. G. **Application of self-adaptive domain adjustment parameters in the optimization of continuous functions using ant colony optimization**, Faculdade de Engenharia Elétrica, Universidade Federal de Uberlândia, 2021, 75p.

There is a wide variety of computational methods used for solving optimization problems. Among these, there are various strategies that are derived from the concept of ant colony optimization (ACO). However, the great majority of these methods are limited-range-search algorithms, that is, they find the optimal solution, as long as the domain provided contains this solution. This becomes a limitation, due to the fact that it does not allow these algorithms to be applied successfully to real-world problems, as in the real world, it is not always possible to determine with certainty the correct domain. The search proposes the use of a broad-range search algorithm, that is, that seeks the optimal solution, with success most of the time, even if the initial domain provided does not contain this solution, as the initial domain provided will be adjusted until it finds a domain that contains the solution. This algorithm called ARACO, derived from RACO, makes for the obtaining of better results possible, through strategies for accelerating the parameters responsible for adjusting the supplied domain at opportune moments and, in case there is a stagnation of the algorithm, expansion of the domain around the best solution found to prevent the algorithm becoming trapped in a local minimum. Through these strategies, ARACO obtains better results than its predecessors, in relation to the number of function evaluations necessary to find the optimal solution, in addition to its one hundred percent success rate in practically all the tested functions, thus demonstrating itself as being a high performance and reliable algorithm. The algorithm has been tested on some classic benchmark functions and also on the benchmark functions of the IEEE Congress of Evolutionary Computation Benchmark Test Functions (CEC 2019 100-Digit Challenge).

Keywords: Ant colony optimization, continuous functions optimization, optimization problems, evolutionary computation

SUMÁRIO:

INDICE DE FIGURAS:

INDICE DE TABELAS:

INDICE DE GRÁFICOS:

LISTA DE ABREVIAÇÕES:

ABC	*Artificial Bee Colony*
ACO	*Ant Colony Optimization*
ACO_R	*Extended ACO for Continuous Domains*
AM-ACO	*Adaptative Multimodal Continuous Ant Colony Optimization*
ANFE	*Average Number of Functions Evaluations*
API	*after pachycondyla APIcalis*
ARACO	*Accelerated and Robust Algorithm for Ant Colony Optimization in Continuous Functions*
CACO	*Continuous ACO*
CACO-DE	*ACO for Continuous Optimization Based on Discrete Encoding*
CEC	*Congress of Evolutionary Computation*
CIAC	*Continuous Interacting Ant Colony*
CGA	*Continuous Genetic Algorithm*
CMA-ES	*Evolutionary Strategy with Covariance Matrix Adaptiation*
CSA-ES	*Evolutionary Strategy with Cumulative Step Size Adaptation*
$DACO_R$	*Diversity ACO_R*
DA	*Dragonfly Algorithm*
DE	*Differential Evolution*
EA	*Evolutionary Algorithm*
ECTS	*Enhanced Continuous Tabu Search*
ESA	*Enhanced Simulated Annealing*
FDO	*Fitness Dependent Optimizer*
GA	*Genetic Algorithm*
GP	*Genetic Programming*
$IACO_R$-LS	*Incremental Ant Colony Algorithm with Local Search*
IDE	*Integrated Development Environment*
IDEA	*Iterated Density Estimation Algorithm*
IEEE	*Institute of Electrical and Electronics Engineers*
MBOA	*Mixed-Bayesian Optimization Algorithm*
MNFE	*Median Number of Functions Evaluations*
PDF	*Probability Density Function*
PSO	*Particle Swarm Optimization*
RACO	*Robust Ant Colony Optimization for continuous functions*
SOA	*Swarm-Based Optimization Algorithms*

SSA	*Salp Swarm Algorithm*
TSP	*Travel Salesman Problem*
UACOR	*Unified Ant Colony Optimization*
WOA	*Whale Optimization Algorithm*
(1+1)-ES	*(1+1)-ES with 1/5th-success-rule*

1 – INTRODUÇÃO

1.1 – Problema

Dentro do estudo da computação, na área de inteligência artificial, existem várias alternativas de algoritmos que visam resolver problemas de otimização, ou seja, problemas que visam encontrar os melhores valores possíveis para as variáveis, com o objetivo de atingir o valor mínimo ou máximo desejado em uma função objetivo, obedecendo algumas restrições impostas. Os problemas de otimização estão presentes nas mais variadas áreas.

Com o passar do tempo, vários algoritmos foram propostos com o objetivo de resolver esses problemas de otimização. Uma dessas alternativas de algoritmos é a otimização por colônia de formigas (ACO – *Ant Colony Optimization*). O ACO utiliza uma abordagem evolutiva inspirada na natureza, que consiste em, a cada iteração do algoritmo, tentar encontrar soluções que melhor atendam ao problema. Dessa forma, com o passar das iterações, pode-se chegar a uma solução que atenda todas as restrições impostas.

A abordagem tradicional do ACO consiste em simular o comportamento de formigas reais em busca de alimento no meio ambiente, através da utilização de formigas artificiais. O algoritmo foi desenvolvido inicialmente, com o objetivo de solucionar o problema do caixeiro viajante (TSP – *Travel Salesman Problem*). Para isso, formigas artificiais são distribuídas nas bordas de um grafo que simula o problema do caixeiro viajante. A cada iteração, as formigas utilizam o valor da distância entre duas arestas do grafo e o conceito de feromônio, que é uma substância que as formigas liberam enquanto percorrem determinada rota, para escolher qual caminho é mais atrativo para ser percorrido. Quanto menor a distância entre duas arestas, melhor o caminho. Quanto mais feromônio distribuído entre duas arestas, melhor o caminho. Dessa maneira, cada formiga artificial deve percorrer todas as arestas do grafo, sem repetição e determinar o seu caminho. Assim como acontece no mundo real, ao fim de cada iteração, parte do feromônio que os caminhos possuem evapora, possibilitando que caminhos que não sejam interessantes fiquem sem feromônio com o tempo, deixando de ser percorridos. Por outro lado, a cada iteração realizada, os caminhos mais percorridos, que são as soluções mais atrativas, passam a possuir uma maior quantidade de feromônio. Esse processo faz com que as formigas artificiais sejam direcionadas para os caminhos mais atrativos nas próximas iterações, fazendo com que elas se aproximem cada vez mais da solução ótima.

Com o passar do tempo, foram surgindo evoluções nos conceitos do ACO e o algoritmo passou a ser aplicado com sucesso na solução de outros problemas do mundo real, que podem ser abordados através de conceitos que envolvem análise combinatória, além do problema do caixeiro viajante, como problemas de roteamento de veículos, ordenamento sequencial de tarefas, escalonamento de horários, dentre outros.

Graças ao desempenho alcançado pelo ACO na solução de problemas que envolvem otimização discreta, também surgiram adaptações do algoritmo com o objetivo de também resolver problemas com variáveis contínuas.

Surgiram então algoritmos, com o objetivo de otimizar problemas que utilizam variáveis contínuas, seguindo a estrutura e as operações do ACO. Porém, a grande maioria desses algoritmos só é capaz de encontrar a solução ótima ou uma solução próxima da ótima para um problema, se essa solução estiver dentro do domínio inicial fornecido ao algoritmo. Esses algoritmos são chamados de algoritmos de busca de alcance limitado. Embora os resultados obtidos por esses algoritmos sejam satisfatórios, existe uma dependência de que o valor ótimo a ser buscado esteja dentro do domínio fornecido.

1.2 – Justificativa

A alternativa para esse problema é trabalhar com um algoritmo de busca de amplo alcance, ou seja, algoritmos que consigam localizar a solução ótima, mesmo que o domínio inicial fornecido não contenha essa solução. Desta maneira, esse algoritmo tem potencial para ser aplicado a problemas do mundo real.

O algoritmo RACO, disponível na literatura, é um algoritmo de busca de amplo alcance e, portanto, atende a esse propósito. Ele funciona muito bem em cenários onde o domínio fornecido possui a solução ótima e também apresenta resultados satisfatórios quando o domínio fornecido não possui a solução ótima. Porém, esse algoritmo possui uma limitação pois, em alguns dos cenários testados, ele não consegue encontrar o valor ótimo para algumas das funções testadas, prejudicando assim a sua confiabilidade.

Pensando nisso, a proposta deste trabalho é contribuir com a melhoria dos resultados alcançados pela aplicação dos conceitos de ACO em problemas de otimização contínua, através da construção de um novo algoritmo de busca de amplo alcance, que permita encontrar a solução ótima ou uma solução próxima da ótima, em todas as funções de *benchmark* testadas, independentemente do domínio inicial fornecido. Esse algoritmo deve possuir alto desempenho e alta confiabilidade, quando comparado aos outros algoritmos já existentes que também trabalhem com otimização de funções contínuas, segundo os critérios de comparação adotados por eles.

Quando o algoritmo for aplicado em cenários onde o domínio inicial fornecido não contenha a solução ideal, ele deve ser capaz de obter com sucesso a solução ótima ou uma solução próxima da ótima, em praticamente todas as execuções, em todas as funções de *benchmark* testadas, garantindo assim potencial para que ele possa ser testado e, posteriormente aplicado em problemas do mundo real. O algoritmo também deve possuir resultados competitivos no quesito quantidade de avaliações de função necessárias até encontrar a solução ótima, quando comparado aos demais algoritmos já existentes.

Quando o algoritmo for aplicado em cenários mais simples, onde o domínio inicial fornecido contém a solução ótima, ele também deve apresentar resultados competitivos, quando comparado a outros algoritmos, em relação a quantidade de avaliações de função necessárias até encontrar a solução ótima, garantindo assim que seja considerado um algoritmo de alto desempenho. Além disso, o algoritmo deve priorizar manter a sua alta confiabilidade, conseguindo encontrar a solução ótima ou uma solução próxima da ótima em todas as funções testadas, em praticamente todas as execuções realizadas.

1.3 – Objetivos do Trabalho

1.3.1 – Objetivo Geral

O objetivo geral do presente trabalho é contribuir com a melhoria dos resultados alcançados pela aplicação dos conceitos de otimização por colônia de formigas na solução de problemas de otimização de funções contínuas, através do desenvolvimento de um novo algoritmo, que seja capaz de encontrar a solução ótima ou uma solução próxima da ótima em todos os cenários testados, em todas as funções de *benchmark* utilizadas, em praticamente todas as execuções realizadas, fornecendo resultados competitivos quando comparados a outros algoritmos existentes, independentemente do domínio inicial fornecido possuir ou não possuir a solução ótima.

1.3.2 – Objetivos Específicos

Os objetivos específicos do trabalho são:

- O algoritmo deve permitir que seja realizada uma busca de amplo alcance, ou seja, o algoritmo deve encontrar a solução ótima ou uma solução próxima da ótima, tanto em cenários onde o domínio inicial fornecido contenha essa solução, quanto em cenários onde não contenha, permitindo assim que o algoritmo possa ser testado e aplicado posteriormente em problemas do mundo real, sem a preocupação de se determinar corretamente um domínio onde a solução ótima se encontre.
- O algoritmo deve possuir melhor desempenho do que os seus antecessores segundo o critério de comparação utilizado por eles, aplicando estratégias que possibilitem a realização da otimização das funções contínuas testadas, utilizando uma menor quantidade de avaliações de função até encontrar a solução ótima ou uma solução próxima da ótima.
- O algoritmo deve possuir maior confiabilidade do que os seus antecessores segundo o critério de comparação utilizado por eles, alcançando cem por cento de taxa de sucesso em praticamente todas as funções de *benchmark* testadas, para todos os domínios iniciais fornecidos.

1.4 – Estrutura do Trabalho

Este trabalho está organizado da seguinte forma.

Após essa breve introdução, o capítulo 2 apresenta os principais conceitos relacionados a otimização por colônia de formigas, além de mostrar outros algoritmos desenvolvidos utilizando o conceito de ACO, com o objetivo de otimizar funções contínuas.

O capítulo 3 apresenta a metodologia utilizada durante a realização deste trabalho, detalhando as principais operações realizadas pelo ARACO, a sequência de etapas do algoritmo e o fluxograma que mostra a sua execução. São apresentadas também as diferenças entre o algoritmo ARACO e o seu predecessor, o algoritmo RACO.

O capítulo 4 mostra os resultados experimentais do ARACO, em cenários em que o domínio fornecido contém a solução ótima e em cenários em que não contém. Estes resultados são comparados com os resultados encontrados por outros métodos baseados em otimização de colônia de formigas em funções contínuas e com métodos que utilizem outros princípios para otimização. O capítulo também mostra os resultados experimentais alcançados com a utilização das funções de *benchmark* do *IEEE Congress of Evolutionary Computation Benchmark Test Functions* (CEC 2019 *100-Digit Challenge*).

Todos os resultados obtidos são discutidos no capítulo 5, onde também é apresentado um resumo de todos os resultados alcançados pelo algoritmo ARACO.

Por último, serão apresentadas as conclusões sobre a pesquisa desenvolvida, apresentando a contribuição do trabalho proposto e a sugestão de futuros trabalhos a serem realizados a partir deste estudo.

2 – FUNDAMENTAÇÃO TEÓRICA

Problemas de otimização que podem ser representados por modelos matemáticos são frequentemente estudados pela inteligência artificial. Esses problemas consistem em encontrar a melhor solução entre um conjunto exponencialmente grande de soluções possíveis [1]. Em outras palavras, trata-se de localizar a melhor solução para o problema, segundo uma métrica. Essa métrica pode ser um modelo matemático composto por uma função objetivo, um conjunto de variáveis de decisão e um conjunto de restrições [2]. Otimização é o ramo da matemática que engloba o estudo da qualidade das soluções ótimas e os métodos para encontrá-las [3]. Problemas de otimização ocorrem na maioria das disciplinas, como engenharia, física, matemática, economia, administração, comércio, ciências sociais e até mesmo política. Problemas de otimização são abundantes em diversos campos da engenharia, como engenharia elétrica, mecânica, civil e química.

O objetivo do processo de otimização é encontrar os valores para cada uma das variáveis de decisão, que resultem em um melhor desempenho do sistema, ou seja, resultem em um valor máximo ou mínimo de uma função denominada função objetivo, desde que os valores das variáveis atendam a um conjunto de restrições ou condições que, necessariamente, devem ser satisfeitas [4]. Esses problemas podem ser resolvidos por técnicas que envolvam Inteligência Artificial.

As variáveis de decisão numéricas, manipuladas pelos problemas de otimização, podem ser divididas em discretas e contínuas. As variáveis discretas são aquelas que só podem possuir um valor que esteja dentro de um conjunto de valores possíveis predeterminados, ou seja, possuem uma quantidade finita de possibilidades entre um valor mínimo e máximo. Por outro lado, as variáveis contínuas são aquelas que podem possuir uma quantidade infinita de valores entre um valor mínimo e um valor máximo, ou seja, elas podem assumir qualquer valor numérico dentro de um intervalo de valores [5].

Muitos algoritmos de otimização clássicos inspirados na natureza, baseados na utilização de população, têm sido propostos para tentar resolver problemas de otimização, para os quais soluções robustas são difíceis ou impossíveis de encontrar em tempo polinomial usando abordagens tradicionais [6].

O princípio fundamental de alguns desses algoritmos utiliza um método construtivo de obtenção da população inicial (soluções iniciais factíveis) e uma técnica de busca local com o objetivo de melhorar as soluções geradas a cada iteração, garantindo que os indivíduos (soluções) dessa população evoluam de acordo com regras especificadas, que consideram a troca de informações entre indivíduos. Esse processo conduz a população, gradativamente, em direção à obtenção de uma solução ótima ou próxima da ótima. Esses algoritmos são conhecidos como algoritmos de computação evolutiva [7].

Dentro da abordagem evolutiva podem ser destacados os algoritmos evolutivos (EA – *Evolutionary Algorithms*), como os Algoritmos Genéticos (GA – *Genetic Algorithms*) [8], a Evolução Diferencial (DE – *Differential Evolution*) [9], a Programação Genética (GP – *Genetic Programming*) [10], dentre outros, e os algoritmos de otimização baseados em enxames (SOA – *Swarm-Based Optimization Algorithms*), como a otimização por colônia de formigas (ACO – *Ant Colony Optimization*) [11], [12], [13], colônia artificial de abelhas (ABC – *Artificial Bee Colony*) [14], otimização por enxame de partículas (PSO – *Particle Swarm Optimization*) [15], dentre outros.

O ACO foi desenvolvido por Dorigo *et al.* [11], [12], [13]. Trata-se de uma estratégia computacional, baseada na simulação do comportamento de formigas reais, no processo de busca por alimento. No mundo real, as formigas buscam aleatoriamente alimento e, na medida em que caminham, espalham uma trilha de uma substância chamada feromônio. Quando uma formiga encontra alimento, ela retorna para o ninho utilizando a trilha de feromômios deixada. O rastro de feromônio deixado passa então a orientar as demais formigas no processo de busca à fonte de alimento, pois toda formiga tem maior probabilidade de escolher percorrer um caminho que tenha maior quantidade de feromônio espalhado. Por outro lado, com o passar do tempo, parte do feromônio deixado nas trilhas evapora, fazendo com que caminhos que não sejam vantajosos deixem de ser percorridos, enquanto os caminhos mais atrativos sempre se mantenham com maior quantidade de feromônio, passando a ser cada vez mais utilizados para direcionar as formigas até o seu objetivo. Em determinado momento, todas as formigas estarão seguindo pelo melhor caminho possível até a fonte de alimento. Esse processo é mostrado na Figura 1. No momento **a**, todas as formigas estão percorrendo o caminho entre o ninho **A** e a fonte de alimentação **E**. No instante de tempo **b**, é colocado um obstáculo no caminho, e as formigas irão contornar o obstáculo com igual probabilidade de escolher cada um dos caminhos. A partir do momento **c**, quando algumas formigas já percorreram os dois caminhos, passa a existir uma maior quantidade de feromônio depositada no caminho que contorna o obstáculo realizando o menor desvio. Esse feromônio faz com que as formigas passem a ter maior probabilidade de escolher percorrer o caminho com menor distância. Com o passar do tempo, todas as formigas tendem a seguir por esse caminho.

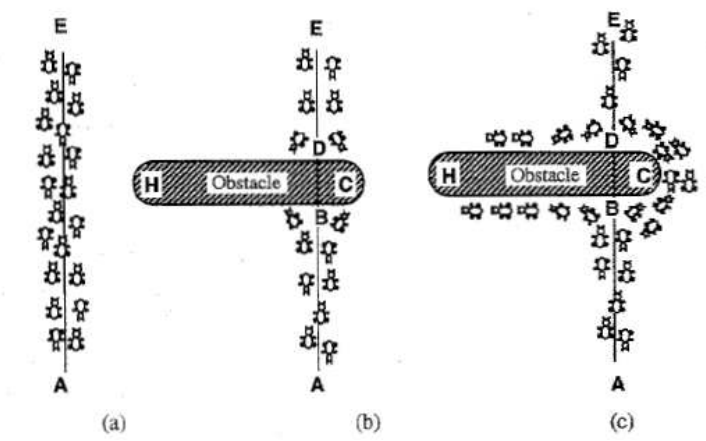


Figura 1 – Representação do comportamento de formigas reais [12]

O conceito de ACO foi desenvolvido inicialmente com o objetivo de solucionar o problema do caixeiro viajante (TSP – *Travel Salesman Problem*). O problema pode ser retratado através de um grafo, onde os vértices representam as cidades e as arestas representam o caminho entre elas.

Para solucionar o problema, as formigas artificiais são dispostas aleatoriamente nas arestas do grafo. Cada formiga escolhe a próxima cidade da sua rota através de um cálculo probabilístico, que considera a quantidade de feromônio artificial distribuído neste caminho e a distância entre as duas cidades. Não é possível visitar duas vezes a mesma cidade em uma solução. A possibilidade de uma formiga escolher determinada cidade como rota é representada por (1).

$$p_{ij}^{k} = \frac{\tau_{ij}^{\alpha} * n_{ij}^{\beta}}{\sum_{l \in N_i^k} \tau_{il}^{\alpha} * n_{il}^{\beta}} \quad (1)$$

Onde p_{ij}^{k} representa a possibilidade de uma formiga *k*, que está em uma aresta *i* escolher como seu próximo destino a aresta *j*; τ_{ij} representa o feromônio depositado entre as arestas *i* e *j*; α representa um parâmetro que deve possuir valor maior ou igual a 0, que é responsável por regular a informação referente ao feromônio; n_{ij} representa a informação heurística entre as arestas *i* e *j*; β representa um parâmetro que deve possuir valor maior ou igual a 0, que é responsável por regular a informação heurística; e N_i^k representa o conjunto de cidades ainda não percorridas por uma formiga *k* situada em uma cidade *i*. O cálculo da probabilidade é válido desde que a cidade *j* ainda não tenha sido percorrida pela formiga *k*, caso contrário, a probabilidade é 0.

Durante o percurso, cada formiga libera determinada quantidade de feromônio em cada parte do caminho percorrido, para influenciar na decisão do caminho das formigas que vierem em seguida. A quantidade de feromônio liberado por uma formiga é inversamente proporcional ao comprimento do caminho percorrido, de acordo com (2). Quanto mais feromônio existe em um caminho, mais atrativa essa rota se torna.

$$\tau_{ij}^{k} = \frac{Q}{L_{ij}} \quad (2)$$

Onde τ_{ij}^{k} representa o feromônio deixado pela formiga *k* entre as arestas *i* e *j*; *Q* é uma constante; e L_{ij} é a distância entre as arestas *i* e *j*.

Por outro lado, quanto menor é a distância entre as cidades em um caminho, mais atrativa essa rota se torna, de acordo com (3).

$$n_{ij} = \frac{1}{d_{ij}} \quad (3)$$

Onde, n_{ij} é a informação heurística entre as cidades *i* e *j*; e d_{ij} é a distância entre as cidades *i* e *j*.

Este processo é repetido até que todas as formigas realizem o percurso completo do caixeiro viajante. Neste momento, os caminhos que foram mais percorridos pelas formigas terão maior quantidade de feromônio, quando comparados aos caminhos menos percorridos.

A estratégia será repetida diversas vezes, e a cada iteração novas formigas artificiais serão geradas nas arestas do grafo. O feromônio em cada caminho será mantido, mas assim como acontece na natureza, ele irá evaporar gradativamente a cada execução, fazendo com que caminhos não vantajosos fiquem sem feromônio após várias execuções do algoritmo e deixem de ser percorridos pelas formigas, enquanto caminhos mais vantajosos permaneçam com feromônio suficiente para que se mantenham atrativos e sejam cada vez mais percorridos pelas formigas. O processo de evaporação do feromônio das formigas é realizado por (4).

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij} \tag{4}$$

Onde τ_{ij} é a quantidade de feromônio entre as arestas *i* e *j*; *t* é a iteração atual do algoritmo; ρ é a taxa de evaporação que é um parâmetro que deve possuir valor maior ou igual a 0 e menor que 1, que é responsável por regular a quantidade de feromônio que irá evaporar a cada iteração; $\sum_{k=1}^{m} \Delta\tau_{ij}$ representa o feromônio depositado por todas as formigas entre as arestas *i* e *j* na iteração atual.

A medida em que novas iterações são realizadas, é possível encontrar melhores soluções. Os feromônios dessas melhores soluções são reforçados, até que o caminho ótimo ou um caminho próximo ao ótimo seja encontrado.

Os principais parâmetros utilizados nos conceitos do algoritmo ACO possuem valores recomendados para utilização, definidos em [8], que são $\alpha = 1$, $\beta = 5$ e $\rho = 0.5$. Para utilização em uma aplicação prática, esses parâmetros podem ser ajustados para possibilitar encontrar melhores soluções para o problema proposto.

A Figura 2 mostra como ocorre o processo em que as formigas artificiais percorrem o caminho entre o ninho **A** e a fonte de alimentação **E**. No momento **a**, é representado o grafo inicial com seis arestas, que podem representar seis cidades, e a distância entre elas. As formigas artificiais irão percorrer o caminho entre o seu ninho **A** e a fonte de alimentação **E**. No instante de tempo **b**, considera-se que são colocadas 30 formigas no ninho e 30 formigas na fonte de alimentação. Embora o caminho entre as arestas **B** e **D** seja menor quando se passa pela aresta **C**, as formigas irão escolher com igual probabilidade o caminho que passa pela aresta **C** ou o que passa pela aresta **H**. No momento **c**, o caminho com menor distância já possui uma trilha de feromônios mais intensa, portanto, a maior parte das formigas passam a percorrer este caminho. Posteriormente, todas as formigas irão percorrer o caminho com menor distância.

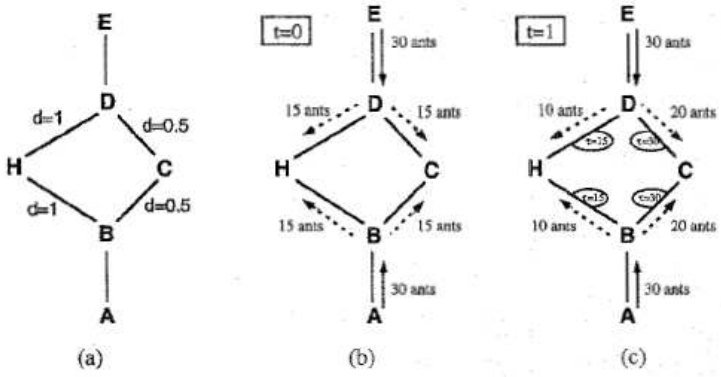


Figura 2 – Representação do comportamento de formigas artificiais [12]

A estratégia de otimização por colônia de formigas foi inicialmente desenvolvida para solucionar problemas de otimização discreta, como problemas de otimização combinatória, ou seja, problemas que podem ser mapeados através de grafos, como por exemplo, o problema do caixeiro viajante [13], problema do caminho mínimo [16], roteamento de veículos [17], escalonamento [18], entre outros.

Com o sucesso do ACO e dos algoritmos derivados dele na solução de problemas com variáveis discretas, surgiram iniciativas com o objetivo de aplicar o conceito na resolução de problemas com variáveis contínuas.

A abordagem mais simples para aplicar o ACO a problemas com variáveis contínuas seria discretizar o domínio de valores reais das variáveis, ou seja, converter os valores reais em um intervalo de valores finito [19]. Discretizar variáveis contínuas é uma tarefa complexa, pois o intervalo onde a busca será realizada pode ser muito amplo, fazendo com que seja necessária grande quantidade de pontos discretos para amostrar corretamente o intervalo contínuo. O intervalo também pode ser muito curto, fazendo com que, caso haja grande quantidade de pontos discretos, a diferença entre os valores do intervalo seja muito pequena dificultando o processo. Em ambos os casos, a discretização seria inviável [20]. Outro problema possível no processo de discretização de variáveis é que a solução ótima pode requerer um grau de precisão maior do que o contemplado pelos valores que foram discretizados. Neste caso, caso a solução ótima esteja em um espaço não abrangido pela discretização, ela não será encontrada.

Embora a abordagem tradicional de ACO seja comprovadamente eficaz para se trabalhar com problemas que envolvem variáveis discretas, ela demonstrou limitações na tentativa de resolver problemas que utilizem variáveis contínuas. Com o passar do tempo, foram propostos algoritmos, com diferentes alternativas para abordar essa dificuldade e alguns deles serão apresentados nos tópicos a seguir.

2.1 – CACO

No método *Continuous ACO* (CACO) [21], as formigas iniciam o processo de busca a partir de um ponto base, chamado de ninho. Em cada iteração, as formigas armazenam suas melhores soluções em um conjunto de vetores. Essas melhores soluções são utilizadas probabilisticamente para orientar o processo de busca na próxima iteração, quando as formigas irão selecionar um dos vetores para continuar o processo de busca. É utilizada a estratégia de seleção aplicada nos GA para a realização de uma busca global para adequar o espaço de busca. Também são usados conceitos de Colônia de Formigas para a realização de uma busca local, depositando feromônios em regiões que representem melhorias nos resultados encontrados, para conseguir atrair cada vez mais outras formigas para estes locais, otimizando as soluções encontradas a cada iteração.

CACO possui variações como CACO-DE [22], que realiza uma codificação discreta das variáveis contínuas.

2.2 – CIAC

O *Continuous Interacting Ant Colony* (CIAC) [23] apresenta uma diferença significativa em relação aos demais algoritmos, pois além de utilizar o processo de distribuição dos feromônios para orientar a busca, ele também utiliza um mecanismo de comunicação entre as formigas. Ou seja, as formigas se comunicam não só através da trilha de feromônios depositada, mas também através de comunicação direta.

O algoritmo funciona através de uma estratégia heterárquica, privilegiando a comunicação entre as formigas, que pode ocorrer entre quaisquer formigas e a qualquer momento, através de canais de comunicação, fazendo com que cada formiga possa influenciar as demais através de fluxos de informação.

2.3 – API

Outra abordagem que realiza otimização de problemas com variáveis contínuas é o *after Pachycondyla APIcalis* (API) [24]. Neste método, as formigas realizam sua pesquisa paralelamente ao redor de um ponto inicial, chamado de ninho. O ninho é deslocado periodicamente, baseado nas buscas que foram melhores sucedidas.

Dessa maneira, ocorre uma busca global, já que as formigas dividem a região de busca em pequenas regiões individuais, procurando a região mais atrativa. Periodicamente, o ninho é migrado para o melhor ponto encontrado, fazendo com que seja possível direcionar o processo de busca para uma melhor região. Também ocorre uma busca local, pois cada formiga realiza a exploração da região próxima ao ninho.

2.4 – ACO_R

Os algoritmos citados anteriormente se inspiraram no ACO, mas não seguem estritamente a estrutura do ACO. Portanto, são considerados algoritmos inspirados em formigas, e não extensões reais do ACO para funções contínuas [25]. Os algoritmos citados a partir deste tópico podem ser classificados como extensões do ACO para funções contínuas.

Socha e Dorigo [20] estenderam o ACO tradicional para que ele também pudesse resolver problemas que trabalhassem com variáveis contínuas, e deram o nome de *Extended ACO for continuous domains* (ACO_R). Nessa abordagem, cada variável de uma formiga obtém seu novo valor, através de um cálculo que utiliza a amostragem probabilística de uma função de densidade de probabilidade (PDF – *Probability Density Function*), ao invés de realizar o cálculo de probabilidade discreta utilizado na abordagem tradicional. A Figura 3 ilustra o processo de representação das duas possibilidades de funções de distribuição de probabilidade. Na alternativa **a**, é mostrado o processo com a distribuição discreta, onde é representada a probabilidade de cada um dos valores finitos c_{i1}, c_{i2}, ..., c_{i10} ser selecionado. Na alternativa **b**, é mostrado o processo com a distribuição contínua, onde qualquer valor entre x_{min} e x_{max} pode ser selecionado. O valor da probabilidade, na distribuição contínua, é determinado pela função PDF. A PDF mais utilizada para este processo é a Gaussiana.

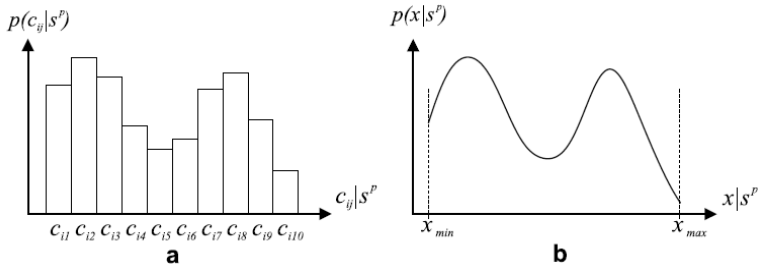


Figura 3 – Representação das funções de distribuição de probabilidade [20]

A função Gaussiana representa um modelo de distribuição do feromônio no ambiente, com base no arquivo população. O arquivo população é inicializado com *m* soluções possíveis aleatórias, onde *m* é um parâmetro que representa o número de soluções que serão armazenadas. Essas soluções são ordenadas de acordo com o valor da função objetivo para cada uma delas. Com a representação adotada com a utilização da função Gaussiana, os valores representados com maior probabilidade de amostragem irão se referir as melhores soluções encontradas. O valor de cada variável de uma nova solução é calculado realizando a amostragem que utiliza dois valores: a média e o desvio padrão. O valor utilizado como média é escolhido dentro do arquivo população, sendo que valores que produziram melhores resultados tem maior probabilidade de serem selecionados. O desvio padrão é calculado baseado na distância entre o valor utilizado como média e as demais soluções existentes no arquivo população.

O processo de atualização do feromônio ocorre no momento em que todas as formigas já tiverem calculado os novos valores para todas as suas variáveis. A cada iteração realizada, as novas soluções encontradas pelas formigas são adicionadas ao arquivo população, que passará a possuir *2*m* soluções. O arquivo será então reordenado, para que sejam descartadas as *m* piores soluções. Dessa maneira, o arquivo população sempre irá se manter com as *m* melhores soluções, e sempre irá representar a amostragem da função Gaussiana para a próxima iteração, fazendo com que os resultados encontrados continuem em evolução e as melhores soluções possam ser encontradas.

2.5 – Outras propostas derivadas de ACO_R

Existem diversos algoritmos que são derivados do ACO_R para otimização de problemas com funções contínuas, alguns deles serão apresentados nessa seção.

O primeiro destes algoritmos é o *Diversity* ACO_R ($DACO_R$) [26]. O $DACO_R$ é mais apropriado para problemas de otimização contínua com grande quantidade de dimensões, pois ele tenta evitar a perda da diversidade nas primeiras iterações. O objetivo é preservar a diversidade pelo maior tempo possível, para explorar mais regiões do espaço de busca, antes que o algoritmo possa convergir.

Outra variação é o *Incremental Ant Colony Algorithm with Local Search* ($IACO_R$-LS) [27]. O $IACO_R$-LS é uma variação do ACO_R, que possui uma estratégia de diversificação de pesquisa para acarretar um aumento no arquivo de soluções. Além disso, são acrescentados três procedimentos de pesquisa local para acelerar o processo de busca da solução.

O *Unified Ant Colony Optimization* (UACOR) [28] inclui componentes dos algoritmos ACO_R, $DACO_R$ e $IACO_R$-LS, e pode instanciar cada um deles, escolhendo componentes específicos do algoritmo e permitindo ajustar automaticamente diversos parâmetros.

O *Adaptative Multimodal Continuous Ant Colony Optimization* (AM-ACO) é uma extensão de ACO_R para otimização multimodal [29]. O AM-ACO utiliza de estratégias de *niching*, dividindo toda a população em parcelas menores ao invés de trabalhar com a população total, além de executar um ajuste autoadaptativo de alguns parâmetros na primeira etapa. Posteriormente, é utilizado um operador de mutação de evolução diferencial para acelerar a velocidade de convergência. Por último, é realizado um processo de busca local para refinar a qualidade da solução encontrada, baseado na distribuição Gaussiana.

O *Ant Colony Optimization Algorithm for Continuous Domains Based on Position Distribution Model of Ant Colony Foraging* [30], baseia-se no princípio que a fonte de alimento das formigas está em todo lugar no espaço contínuo, apenas a qualidade da fonte de alimento é diferente. Cada formiga verifica a qualidade de sua posição, a concentração de feromônio no restante do espaço, através de uma função de densidade de feromônios do grupo, e migra para áreas de maior concentração, podendo explorar regiões desconhecidas durante essa movimentação.

2.6 – RACO

Todos os métodos de otimização de funções contínuas citados nos tópicos anteriores pertencem a uma categoria chamada de algoritmos de busca de alcance limitado, ou seja, que encontram a solução ideal, mas apenas dentro dos domínios pré-determinados [25]. O problema desses algoritmos é que eles são dependentes do domínio inicial. Caso o domínio inicial não seja estimado corretamente e não possua a solução ótima, o algoritmo não conseguirá encontrar a solução correta, pois as formigas não conseguem sair do domínio estabelecido inicialmente.

Chen *et al.* [25] propuseram um algoritmo de busca de amplo alcance, ou seja, capaz de encontrar a solução ótima, mesmo que o domínio inicial fornecido não contenha os valores ótimos. Isso ocorre pois, nessas situações, o algoritmo é capaz de adaptar o domínio inicial fornecido, encontrando um novo domínio que contenha a solução ótima para o problema. O algoritmo proposto, chamado de *Robust Ant Colony Optimization for continuous functions* (RACO), utiliza o método da grade para discretizar as variáveis contínuas, além de aplicar estratégias autoadaptativas para o ajuste de domínio, incremento de feromônio, divisão de domínio e quantidade de formigas. Dessa maneira, RACO possibilita que a busca seja realizada com sucesso, independente do domínio inicial fornecido abranger ou não os valores ótimos para o problema.

O algoritmo RACO, conforme abordado posteriormente, apresenta bons resultados na realização de buscas em cenários onde o domínio fornecido possui a solução ótima, e também apresenta resultados satisfatórios quando o domínio fornecido não possui a solução ótima. Porém, foi detectado que, em alguns dos cenários testados, ele não consegue encontrar o valor ótimo para algumas das funções testadas, em nenhuma das execuções realizadas. Em outros cenários testados, com algumas das funções testadas, ele encontra o valor ótimo em algumas execuções, porém não encontra em outras.

3 – METODOLOGIA

A proposta deste trabalho é contribuir para o aprimoramento dos resultados da aplicação de ACO em problemas de otimização contínua, através da apresentação de um novo algoritmo de busca de amplo alcance, derivado do RACO, chamado de *Accelerated and Robust Algorithm for Ant Colony Optimization in Continuous Functions* (ARACO). O ARACO tem como diferencial a utilização de parâmetros autoadaptativos relacionados ao ajuste do domínio. A aceleração destes parâmetros, em momentos oportunos, faz com que o domínio que contém a solução ótima possa ser encontrado mais rapidamente, bem como possibilita que, dentro do domínio correto, as formigas possam convergir para a solução ótima ou próxima da ótima mais rapidamente. Dessa maneira, ARACO permite que os valores ótimos ou valores próximos dos ótimos para as variáveis sejam encontrados em um menor número de avaliações de função, quando comparado aos demais algoritmos.

Com a aceleração destes parâmetros, no algoritmo ARACO, é possível que o valor máximo e mínimo do domínio das variáveis fiquem estagnados em uma região, fazendo com que a melhor solução encontrada seja um mínimo local. Mesmo sem utilizar uma estratégia para acelerar os parâmetros autoadaptativos relacionados ao ajuste de domínio, RACO também enfrenta problemas de estagnação e fica preso em regiões de mínimo local. A diferença é que, para essas situações, ARACO utiliza uma estratégia de tratamento da estagnação do algoritmo, que permite que o domínio das variáveis seja expandido gradativamente em torno da melhor solução encontrada até o momento, permitindo que seja determinado um novo domínio, que possibilite que sejam alcançados valores que estejam fora da região em que se encontra o mínimo local. Dessa maneira, ARACO consegue atingir regiões do domínio que não são alcançadas pelo seu antecessor RACO, pois o algoritmo ARACO consegue escapar de regiões de mínimo local, definindo um novo domínio fora dessas regiões, quando o algoritmo está estagnado, permitindo assim que os valores ótimos ou valores próximos dos valores ótimos sejam alcançados com uma maior taxa de sucesso.

3.1 – Definição do problema

Um problema de otimização contínua pode ser definido formalmente com o seguinte modelo: $P = (X, \Omega, f)$ [25], onde X é um vetor de solução com n variáveis contínuas x_i ($i = 1, 2, ..., n$), Ω é um conjunto de restrições que devem ser atendidas pelas variáveis e f é a função objetivo a ser otimizada. No caso de um problema de minimização, o objetivo é encontrar o valor de X^*, que minimiza a função: $f(X^*) \leq f(X), \ \forall X \in S$. No caso de um problema de maximização, o objetivo é encontrar o valor de X^*, que maximiza a função: $f(X^*) \geq f(X), \ \forall X \in S$, com S representando o domínio inicial de onde será feita a busca pelo valor de X.

3.2 – Discretização das variáveis

O algoritmo ARACO realiza a discretização das variáveis, utilizando o método de grade. Neste método, o domínio contínuo definido é convertido em um domínio discreto. Este processo ocorre no início de cada iteração do algoritmo, pois em cada iteração realizada é definido um novo domínio contínuo para *X*. Por definição, *X* é um vetor de soluções com *n* dimensões com variáveis contínuas x_i (i = 1, 2, ..., n).

No início de cada iteração do algoritmo, será determinado, para cada variável, um domínio contínuo inicial (x_i^{min} , x_i^{max}), onde x_i^{min} representa o menor valor para a variável *x*, na dimensão *i*, enquanto x_i^{max} representa o maior valor para a variável *x*, na dimensão *i*. O algoritmo irá realizar a busca pela solução do problema dentro deste domínio. O algoritmo utiliza a variável *k* para realizar a discretização e, neste momento, o domínio inicial será dividido em *k + 1* partes para cada uma das *n* variáveis.

Para realizar o processo de discretização, em cada variável, o domínio é dividido em *k + 1* valores discretos. Os valores discretos do domínio podem ser representados por uma matriz de *n* * (*k + 1*) dimensões, conforme a Tabela 1, onde *n* representa a quantidade de variáveis do problema.

Tabela 1 – Matriz de valores discretos do domínio

x_1^{min}	$x_1^{min}+h_1$	$x_1^{min}+2*h_1$	...	$x_1^{min}+k*h_1$
x_2^{min}	$x_2^{min}+h_2$	$x_2^{min}+2*h_2$	...	$x_2^{min}+k*h_2$
...	...	...	...	...
x_n^{min}	$x_n^{min}+h_n$	$x_n^{min}+2*h_n$	...	$x_n^{min}+k*h_n$

Para calcular o valor de cada elemento da matriz, é necessário primeiro calcular o valor de h_i, de acordo com (5):

$$h_i = \left(x_i^{max} - x_i^{min}\right) / k \quad (5)$$

Onde *i* representa cada variável do problema, variando entre 1 e *n*.

3.3 – Construção da nova solução

O principal papel das formigas no ARACO é a construção das novas soluções. Cada formiga armazena uma solução possível para o problema que está sendo otimizado, ou seja, um valor para cada variável. Para construir uma nova solução, as formigas inicialmente deverão percorrer várias rotas aleatórias para distribuir o feromônio inicial, que será utilizado como base para conduzir as formigas no processo de construção das próximas soluções, juntamente com o valor heurístico da função contínua que está sendo otimizada.

A criação de rotas aleatórias no início de cada iteração permite que exista feromônio inicial antes que as formigas construam suas rotas. Este feromônio inicial fará com que as rotas sejam determinadas pelas formigas de forma não aleatória, pois elas serão direcionadas pelo feromônio inicial, fazendo com que a convergência do algoritmo aumente.

Para a criação das rotas iniciais para distribuição de feromônio, cada formiga seleciona aleatoriamente, para cada variável, um valor dentre os valores discretos possíveis, que foram obtidos pelo método de grade. As soluções aleatórias criadas terão seus valores heurísticos calculados, de acordo com a função que será minimizada ou maximizada. Todas as rotas aleatórias criadas são ordenadas, de acordo com o seu valor heurístico, e apenas as melhores soluções geradas terão seu feromônio distribuído.

Para armazenamento dos feromônios, será utilizada uma matriz, denominada τ. Essa matriz possui $n * (k + 1)$ dimensões e está representada na Tabela 2. Note que as dimensões da matriz τ são as mesmas da matriz de valores discretos do domínio, apresentada na Tabela 1, pois os valores armazenados em uma posição da matriz τ equivalem a quantidade de feromônio armazenado no valor discreto que ocupa a mesma posição na matriz de valores discretos.

Tabela 2 – Matriz de feromônios τ

τ_{11}	τ_{12}	τ_{13}	...	$\tau_{1(k+1)}$
τ_{21}	τ_{22}	τ_{23}	...	$\tau_{2(k+1)}$
...	...	...	...	...
τ_{n1}	τ_{n2}	τ_{n3}	...	$\tau_{n(k+1)}$

Os valores dos feromônios das melhores soluções são depositados na matriz de feromônios τ, segundo (6):

$$\tau_{ij} = \tau_{ij} + Q / f \tag{6}$$

Onde τ_{ij} representa o valor do feromônio na variável i, no ponto j, mapeado de acordo com a discretização das variáveis; f representa o valor heurístico da solução; e Q representa o valor do incremento autoadaptativo do feromônio, que será descrito posteriormente, mas que é calculado através de (7):

$$Q = 10^{OM_{min}+1} \tag{7}$$

Onde OM_{min} representa a ordem de magnitude, ou seja, o valor do expoente da melhor solução encontrada.

A partir do momento em que o feromônio das melhores soluções geradas aleatoriamente estiver depositado na matriz, ARACO passará para uma nova etapa. Novas formigas serão geradas e elas passarão a determinar suas rotas, utilizando um cálculo de probabilidade dentre as rotas disponíveis, representado por (8):

$$p_{ij} = \tau_{ij} / \sum_{i=1}^{k+1} \tau_{ij} \tag{8}$$

Onde p_{ij} representa a probabilidade de, na variável i, o ponto j ser selecionado como rota para a nova formiga.

Baseado nas equações anteriores, é importante destacar que a probabilidade de um valor ser selecionado como solução para uma nova formiga é calculada utilizando informações relacionadas ao valor heurístico e a quantidade de feromônio que a possível rota possui, ou seja, o algoritmo utiliza os mesmos parâmetros adotados nos conceitos básicos de ACO. Esse é um dos objetivos do algoritmo ARACO, seguir estritamente a estrutura do ACO, para ser considerado uma extensão real do ACO para funções contínuas, e não mais um algoritmo inspirado em formigas. Seguindo os conceitos adotados, os valores das soluções com menor valor heurístico e maior quantidade de feromônio possuem maior probabilidade de serem selecionados como rota para as novas formigas.

É importante destacar que todo este trabalho foi realizado considerando a solução de um problema de minimização, por isso a afirmação de que um menor valor heurístico contribui para que haja maior probabilidade de seleção de um valor de uma solução. Caso o problema a ser analisado fosse de maximização, o raciocínio empregado em relação a informação heurística seria o contrário, um maior valor heurístico contribuiria para uma maior probabilidade de seleção. Em ambos os casos, o raciocínio referente a quantidade de feromônio seria o mesmo, uma maior quantidade de feromônio acarreta maior possibilidade de seleção.

Outro fator importante é que o método de seleção utilizado no algoritmo é a roleta, que é muito utilizada em GA.

Quando todas as formigas tiverem criado as suas novas rotas, irá ocorrer o processo de evaporação do feromônio. Este processo impede que o feromônio fique acumulado apenas em alguns pontos, dificultando o processo de diversificação das soluções. Além disso, com a evaporação do feromônio, pontos que não representam alternativas interessantes para a construção de novas soluções vão ficando sem feromônios aos poucos e, consequentemente, deixam de ser percorridos. Por outro lado, pontos que representam alternativas interessantes sempre se mantém com feromônio suficiente para guiar as próximas formigas. Após o processo de evaporação, o novo valor do feromônio em cada ponto será dado por (9):

$$\tau_{ij}^{new} = (1 - \rho) * \tau_{ij}^{old} \quad (9)$$

Onde ρ representa a taxa de evaporação e seu valor deve ser determinado por um número entre 0 e 1. No algoritmo ARACO, o valor recomendado para ρ é 0.5. Este valor foi determinado empiricamente, ou seja, foram realizados testes exaustivos para determinar o valor que produz os melhores resultados.

Após o processo de evaporação, a melhor rota dentre as rotas criadas terá o seu feromônio reforçado na matriz de feromônios.

O processo de construção de novas soluções irá se repetir, e todas as formigas deverão criar novamente suas rotas. A quantidade de formigas é determinada pela variável *m*. Todas as soluções criadas serão novamente avaliadas, o feromônio da melhor rota será novamente reforçado e o processo de evaporação irá novamente ocorrer. A quantidade de vezes que este processo será repetido é determinado pela variável *nc_max*.

3.4 – Ajuste autoadaptativo do domínio

A estratégia de ajuste autoadaptativo do domínio permite que a solução ótima ou uma solução próxima da ótima seja encontrada, mesmo que domínio inicial atribuído não contenha os valores ótimos. Para que isso ocorra, o domínio é ajustado automaticamente a cada iteração. O processo de ajuste do domínio é realizado através da análise dos feromônios depositados pelas formigas na matriz τ. Cada linha i da matriz τ representa os feromônios distribuídos em uma variável i, que possui os valores mapeados na matriz mostrada na Tabela 2.

Pode-se afirmar que, caso o feromônio esteja concentrado próximo ao centro do domínio, ou seja, os percursos realizados pelas formigas foram concentrados na região central do domínio, existe maior possibilidade de que a solução ideal esteja localizada dentro do domínio. Por outro lado, caso o feromônio esteja concentrado próximo as extremidades do domínio, ou seja, os percursos realizados pelas formigas foram concentrados na região próxima as bordas do domínio, existe maior possibilidade de que a solução ideal esteja localizada fora do domínio.

A análise e o ajuste do domínio devem ser realizados variável por variável, visto que é possível que, para uma variável a solução esteja fora do domínio e, para outra variável a solução esteja dentro do domínio.

O algoritmo utiliza a variável θ para decidir qual ajuste será realizado. O valor de θ representa a porcentagem das $(k+1)$ posições, que irá determinar se a melhor solução está dentro ou fora do domínio atual, de acordo com os conceitos dos parágrafos anteriores. A escolha do ajuste a ser realizado também utiliza a variável r_i, que representa qual posição, entre 0 e $(k+1)$, possui maior concentração do feromônio depositado pelas formigas, para cada uma das i variáveis. Utilizando essas duas variáveis, em um exemplo onde a variável θ tem valor 0.2, quer dizer que, caso r_i esteja localizado nas primeiras ou nas últimas 20% de $(k + 1)$ posições, existe maior possibilidade de que a solução ideal esteja localizada fora do domínio. O valor recomendado para a variável θ é entre 0.1 e 0.3. Este intervalo de valores foi determinado dessa maneira, porque é o intervalo utilizado no algoritmo RACO. Os valores dos parâmetros existentes nos dois algoritmos são iguais, para que seja possível avaliar melhor as novas funcionalidades implementadas no ARACO.

Analisando as variáveis r_i e θ, caso $r_i \leq \theta * (k+1)$ ou $r_i \geq (1-\theta) * (k+1)$, significa que o feromônio depositado pelas formigas está concentrado próximo a extremidade do domínio, ou seja, existe maior possibilidade de que a solução ideal esteja localizada fora do domínio atual. Nessa situação, o ajuste autoadaptativo do domínio deve ser realizado de maneira a deslocar e ampliar o domínio atual, para localizar um domínio que abranja a solução ideal.

Neste caso, os novos valores de x_i^{min} e x_i^{max} devem ser determinados considerando o valor armazenado na posição r_i como o centro do novo domínio, visto que r_i representa o ponto com maior quantidade de feromônios localizado, segundo (10) e (11).

$$x_i^{min} = r_i - \left(k/2 + \Delta_1\right) * h_i \quad (10)$$

$$x_i^{max} = r_i + \left(k/2 + \Delta_1\right) * h_i \quad (11)$$

Onde o parâmetro Δ_1 determina o quanto o novo domínio deverá ser maior que o antigo para a variável *i*. O valor de Δ_1 deve ser pequeno, para que o novo domínio criado não seja muito maior que o antigo domínio. No algoritmo ARACO, o valor inicial recomendado é 1.25, mas é importante destacar que o valor é alterado dinamicamente para acelerar o processo de conversão da solução, conforme mostrado no item 3.8. Este valor inicial foi determinado dessa maneira, porque é o valor utilizado no algoritmo RACO.

A situação inversa, em outras palavras, quando a solução ideal tende a estar localizada dentro do domínio atual, é representada quando o feromônio está concentrado próximo ao centro do atual domínio, ou seja, $\theta * (k+1) \leq ri \leq (1-\theta) * (k+1)$. Neste cenário, o domínio atual deve ser reduzido, para que seja executada uma busca mais detalhada.

Neste caso, os novos valores de x_i^{min} e x_i^{max} devem ser determinados segundo (12) e (13). Note que é realizada uma redução nas duas extremidades do domínio.

$$x_i^{min} = x_i^{min} + \left(x_i^{max} - x_i^{min}\right) * \Delta_2 \quad (12)$$

$$x_i^{max} = x_i^{max} - \left(x_i^{max} - x_i^{min}\right) * \Delta_2 \quad (13)$$

Onde o parâmetro Δ_2 determina o quanto o domínio será reduzido. Portanto o seu valor não deve ser muito grande, para que o domínio não seja minimizado de forma que a solução ideal não esteja mais dentro dele. A cada iteração que utilizar este ajuste, o novo domínio será $2 * \Delta_2$ menor que o domínio antigo. No algoritmo ARACO, o valor inicial recomendado é 0.05, mas é importante destacar que o valor é alterado dinamicamente para acelerar o processo de conversão da solução, conforme mostrado no item 3.8. Este valor inicial foi determinado dessa maneira, porque é o valor utilizado no algoritmo RACO.

Existe outro mecanismo para acelerar o processo de convergência em situações em que a solução ideal se encontra fora do domínio atual. Nessa situação, caso $r_i \leq \theta * (k+1)$, ou seja, se o feromônio está concentrado próximo a borda inferior e, consequentemente, o deslocamento do domínio foi efetuado no sentido do valor de x_i^{min}, o valor de x_i^{min} será mantido e o valor de x_i^{max} será alterado conforme (14).

$$x_i^{max} = x_i^{max} - \left(x_i^{max} - x_i^{min}\right) * \Delta_3 \quad (14)$$

Por outro lado, caso $r_i \geq (1-\theta) * (k+1)$, ou seja, se o feromônio está concentrado próximo a borda superior e, consequentemente, o deslocamento do domínio foi efetuado no sentido do valor de x_i^{max}, o valor de x_i^{max} será mantido e o valor de x_i^{min} será alterado conforme (15).

$$x_i^{min} = x_i^{min} + \left(x_i^{max} - x_i^{min}\right) * \Delta_3 \quad (15)$$

Em (14) e (15), o parâmetro Δ_3 determina o quanto o domínio será reduzido, na sua borda inferior ou na sua borda superior, dependendo do valor r_i. No algoritmo ARACO, o valor inicial recomendado é $2 * \Delta_2$, ou seja, 0.1, mas é importante destacar que o valor é alterado dinamicamente para acelerar o processo de conversão da solução, conforme mostrado no item 3.8. Este valor inicial foi determinado dessa maneira, porque é o valor utilizado no algoritmo RACO.

3.5 – Incremento autoadaptativo do feromônio

Conforme explicado brevemente na seção 3.3, para calcular os valores da matriz de feromônios τ, são utilizados o valor heurístico da solução (f) e o valor da variável Q. O valor da variável Q não pode se manter constante durante toda a execução do algoritmo, pois a medida em que as iterações do algoritmo são executadas, o valor de f é alterado, já que melhores soluções são encontradas. Como o valor de f é alterado a cada iteração e, como $\tau_{ij} = Q / f$ (6), caso o valor de Q permaneça constante, os valores de Q e f se tornarão desproporcionais a partir de alguma iteração do algoritmo. Isso fará com que a matriz de feromônios τ tenha os seus valores similares, impossibilitando a convergência do algoritmo. É importante destacar que o problema da matriz de feromônios τ ter apenas valores semelhantes não poderia ser evitado, definindo um valor muito baixo ou muito alto para Q, pois de qualquer maneira, a variação ocorrida no valor de f iria fazer com que, em algum momento do algoritmo, os valores de Q e f fiquem desproporcionais.

A solução encontrada foi definir o incremento de feromônio da matriz τ_{ij} através de uma estratégia autoadaptativa, fazendo com que em cada iteração, como o domínio é alterado, a variável Q também possa ter um valor diferente. Este valor deve ser sempre proporcional ao valor heurístico da melhor solução gerada aleatoriamente no início de cada iteração, por isso, $Q = 10^{OM_{min}+1}$ (7). Como OM_{min} representa o valor do expoente da melhor solução encontrada nessa iteração, o valor de Q nunca ficará desproporcional ao valor de f.

3.6 – Divisão autoadaptativa do domínio

Uma das principais características do ARACO é conseguir encontrar a melhor solução, mesmo se ela não estiver contida no domínio inicial fornecido. Isso só é possível porque é realizado um ajuste no domínio a cada iteração (item 3.3) utilizando, dentre outros parâmetros, a variável k, que também influencia diretamente o processo de discretização do domínio.

Em cada iteração, o domínio é alterado e discretizado, porém caso o valor de k se mantenha constante em todo o algoritmo, pode não ser possível encontrar a solução ideal. Por exemplo, caso o valor de k seja baixo durante todo o algoritmo, o domínio será convertido em poucos pontos discretos, que podem não ser suficientes para amostrar corretamente o domínio e conduzir as formigas em direção a solução ideal do problema, já que é possível que nenhum dos pontos esteja próximo de uma região ótima. Por outro lado, caso o valor de k seja alto durante todo o algoritmo, o domínio será convertido

em muitos pontos discretos, que podem possuir valores heurísticos similares, fazendo com que a quantidade de feromônio nestes pontos também possa ser similar e que, consequentemente, o algoritmo demore ou mesmo não seja capaz de encontrar uma região mais promissora dentre as regiões disponíveis.

O fato é que o grau de precisão da discretização do domínio depende da variável *k*, e necessita ter o seu valor definido através de método autoadaptativo, para que *k* possa ter um valor mais baixo, no início da execução, fazendo com que o algoritmo possa convergir rapidamente para uma área promissora. Porém, em algum momento, o valor de *k* irá limitar o algoritmo, fazendo com que ele não encontre mais soluções melhores a cada iteração. Essa limitação é detectada, quando o algoritmo percorre um certo número de iterações sem que ocorra nenhuma melhoria no valor da melhor solução encontrada. Quando isso ocorrer, o valor de *k* deve ser incrementado em uma unidade, para que a precisão da busca dentro do domínio seja aumentada, e o algoritmo possa encontrar soluções que não eram possíveis de serem alcançadas com o valor anterior de *k*. O valor inicial recomendado para a variável *k* é 11. Este valor inicial foi determinado dessa maneira, porque é o valor utilizado no algoritmo RACO.

3.7 – Quantidade de formigas autoadaptativa

Outra condição que influencia o sucesso do algoritmo é a quantidade de formigas que irão realizar a busca em cada iteração. A quantidade de formigas depende diretamente do processo de discretização do domínio e não pode ter valor constante. Isso ocorre porque, em momentos iniciais, quando o domínio contínuo for convertido em poucos pontos, poucas formigas já podem ser capazes de encontrar a região mais promissora para realizar a busca. Porém, em momentos posteriores, quando o domínio for convertido em muitos pontos, podem ser necessárias muitas formigas para encontrar as melhores soluções.

Dessa maneira, a quantidade de formigas utilizadas também precisa ser um parâmetro autoadaptativo, pois no instante em que o domínio for convertido em poucos pontos, ou seja, quando *k* possuir um valor baixo, a quantidade de formigas *m* também necessita ser baixa. Porém, a medida em que o valor de *k* aumenta, devido a necessidade de maior precisão na busca por melhores soluções, a quantidade de formigas também necessita aumentar. A quantidade de formigas é determinada através de (16).

$$m = k + \Delta_m \tag{16}$$

Onde o parâmetro Δ_m representa o quanto *m* deve ser maior do que *k*, para possibilitar que as formigas encontrem as melhores soluções sem gastar muitas iterações para isso. O valor recomendado para Δ_m é 2. Este valor inicial foi determinado dessa maneira, porque é o valor utilizado no algoritmo RACO.

3.8 – Aceleração dos parâmetros autoadaptativos de ajuste

Com o objetivo de acelerar o processo de convergência para a solução ótima ou para uma solução próxima da ótima, ARACO utiliza valores autoadaptativos para os parâmetros Δ_1, Δ_2 e Δ_3, responsáveis pelo ajuste do domínio a cada iteração. Os parâmetros são ajustados em determinadas situações com o objetivo de acelerar a convergência do algoritmo, mas voltam aos seus valores iniciais quando este ajuste não for mais recomendado.

Conforme dito anteriormente, na seção 3.4, quando o feromônio depositado pelas formigas está concentrado próximo ao centro do domínio de uma variável, pode-se dizer que a solução ótima tende a estar localizada dentro do domínio atual para essa variável. Nestes casos, é necessário que o domínio seja reduzido, para que seja realizada uma busca com maior precisão. A redução é realizada utilizando (12) e (13), que removem uma pequena parte das duas extremidades do domínio atual. O parâmetro responsável por determinar o quanto o domínio será diminuído é Δ_2 e o seu valor inicial é 0.05. A estratégia proposta em ARACO verifica se o ponto com maior concentração de feromônios está localizado na posição central do domínio atual, na posição anterior ou posterior a posição central, em outras palavras, se $r_i = (k+1) / 2$ ou $r_i = ((k+1) / 2) - 1$ ou $r_i = ((k+1) / 2) + 1$. Caso isso ocorra, o parâmetro Δ_2 será incrementado em 0.05 porque, como a solução ideal tende a estar localizada praticamente no centro do domínio, pode-se remover uma maior quantidade de valores localizados nas bordas do domínio, realizando o estreitamento do domínio em uma velocidade mais acelerada, para que as formigas possam localizar a melhor solução mais rapidamente. Nos outros casos, onde a solução ideal tende a estar dentro do domínio, mas não próximo ao centro dele, o parâmetro Δ_2 será incrementado em 0.005. A aceleração do parâmetro Δ_2 possui limite e só pode ser realizada até que ele atinja o valor máximo de 0.15, para evitar que o domínio não seja tão reduzido de maneira que ele deixe de conter a solução ótima. É importante destacar que, no momento em que ocorrer uma iteração em que o ponto com maior concentração de feromônios não estiver localizado dentro do domínio, Δ_2 retorna ao seu valor inicial.

Por outro lado, quando o feromônio depositado pelas formigas está concentrado próximo a borda do domínio de uma das variáveis, pode-se dizer que a solução ideal tende a estar localizada fora do domínio atual para essa variável. Nestes casos, é necessário que o domínio seja deslocado e ampliado, utilizando (10) e (11), para que um domínio que contenha a solução ótima possa ser encontrado. O parâmetro responsável por determinar o quanto o domínio será deslocado e ampliado é Δ_1 e seu valor inicial é 1.25. A estratégia proposta em ARACO verifica se o ponto com maior concentração de feromônios está localizado exatamente na borda inferior ou superior do domínio, em outras palavras, se $r_i = 1$ ou $r_i = (k + 1)$. Caso isso ocorra, o parâmetro Δ_1 será incrementado em 0.25, porque como a solução ideal tende a estar distante da borda do domínio, pode-se realizar um deslocamento e um aumento mais significativo, para possibilitar que um domínio onde o valor ótimo esteja presente seja localizado mais rapidamente. Nas demais situações, onde a solução ideal tende a estar fora do domínio, mas não tão distante da borda, o parâmetro Δ_1 será incrementado em 0.025. A aceleração do parâmetro

Δ_1 também possui limite e só pode ser realizada até que ele atinja o valor máximo de 1.75, para impedir que o novo domínio se torne tão grande que inviabilize a realização da busca com desempenho satisfatório. É importante destacar que, no momento em que ocorrer uma iteração em que o ponto com maior concentração de feromônios não estiver mais localizado fora do domínio, Δ_1 retorna ao seu valor inicial.

Estes valores de incremento foram determinados empiricamente, ou seja, foram realizados testes exaustivos para determinar os valores que produzem os melhores resultados.

O parâmetro Δ_3, utilizado em (14) e (15), também é ajustado dinamicamente, porém a sua variação depende diretamente do ajuste de Δ_2, pois conforme dito anteriormente, na seção 3.4, $\Delta_3 = 2 * \Delta_2$.

3.9 – Ampliação do domínio em torno da melhor solução

Em algumas situações, a aceleração dos parâmetros Δ_1, Δ_2 e Δ_3 pode fazer que o algoritmo convirja para uma região que possua um valor mínimo local e fique preso lá. ARACO possui uma estratégia para tratamento dessas situações de estagnação do algoritmo, através de um mecanismo de ajuste de domínio. Este ajuste faz com que, caso seja detectada a estagnação do algoritmo, um novo domínio seja definido em torno da melhor solução encontrada até o momento, permitindo que o algoritmo possa escapar da região de mínimo local.

O processo de tratamento de estagnação do algoritmo é executado quando forem percorridas 30 iterações sem que ocorra uma melhoria, de pelo menos 10%, no valor heurístico da melhor solução encontrada até o momento. Nestes casos, é considerado que o algoritmo está estagnado e um novo domínio será determinado em torno da melhor solução já encontrada. Este novo domínio será ampliado para apenas uma das variáveis, para que este aumento possa ocorrer de maneira gradativa, permitindo que o algoritmo deixe a região de mínimo local. A escolha da variável que terá o seu domínio ampliado é realizada através de uma verificação, que localiza a variável que proporcionalmente teria menor variação com a ampliação do domínio. Essa variável tende a estar mais estagnada que as demais. É importante destacar que, caso sejam percorridas mais 30 iterações sem que ocorra otimização de pelo menos 10% no valor da melhor solução encontrada, o processo de tratamento de estagnação do domínio será novamente executado, porém outra variável pode ser selecionada. O valor do novo domínio será determinado por (17) e (18):

$$x_i^{min} = best^i - {}^{k}/_{2} * \Delta_1 * H_i * e^i \quad (17)$$

$$x_i^{max} = best^i + {}^{k}/_{2} * \Delta_1 * H_i * e^i \quad (18)$$

Onde, $best^i$ representa o valor da melhor solução encontrada até o momento para a variável i; H_i representa o valor de h_i quando a melhor solução até o momento foi encontrada; e o parâmetro e^i representa o quanto o novo domínio será ampliado para sair da região de mínimo local.

O parâmetro e^i, para cada variável i, é inicializado com 1. Sempre que forem executadas 30 iterações e a melhor solução permanecer estagnada, ele terá o valor dobrado apenas para a variável que foi selecionada para ter o domínio ampliado. Quando a melhor solução sair da região de mínimo local, ou seja, quando for encontrada uma nova solução que tenha o valor pelo menos 10% menor do que a melhor solução encontrada, considerando-se um problema de minimização, o parâmetro e^i volta para 1, para todas as variáveis i. Dessa maneira, é garantido que o novo domínio gerado irá aumentar gradativamente em torno da melhor solução encontrada, até que sejam localizados novos valores fora de regiões de mínimo local.

Os valores dos parâmetros responsáveis por detectar e tratar a estagnação do algoritmo foram determinados empiricamente, ou seja, foram realizados testes exaustivos para determinar os valores que produzem os melhores resultados.

Conforme dito anteriormente, o parâmetro H_i representa o valor de h_i quando a melhor solução até o momento foi encontrada. Mas, para permitir que o novo domínio não seja tão pequeno a ponto de ser necessário executar essa ampliação várias vezes prejudicando assim o desempenho do algoritmo, ou seja tão grande a ponto de não ser possível localizar uma melhor solução em apenas 30 iterações, o parâmetro H_i tem o seu valor obrigatoriamente dentro do intervalo $0.1 \leq H_i \leq 1$.

É importante destacar que é necessário redefinir os valores dos parâmetros Δ_1, Δ_2 e Δ_3 para os valores iniciais, após o processo de ampliação do domínio em torno da melhor solução encontrada.

3.10 – Estrutura do algoritmo ARACO

Após detalhar as principais operações realizadas pelo algoritmo ARACO, este tópico detalha a estrutura do algoritmo. Primeiramente, serão mostradas as estapas percorridas pelo algoritmo, descrevendo em detalhes todas as ações realizadas, desde a atribuição dos parâmetros iniciais até a finalização do algoritmo, quando a condição de parada é atingida. Em seguida, é apresentado um fluxograma que representa a sequência e a interação entre as ações do algoritmo.

3.10.1 – Principais etapas do algoritmo

Nesta seção, as principais etapas seguidas pelo algoritmo ARACO na busca por uma solução ótima são apresentados e detalhados.

Uma das principais melhorias geradas pelo algoritmo ARACO está relacionada à obtenção de um melhor desempenho, com a execução da etapa 17. Através desta etapa, o algoritmo é capaz de encontrar o valor ótimo ou um valor próximo do ótimo, utilizando menor quantidade de avaliações de função quando comparado com outros algoritmos, como será mostrado na seção 4, graças à aceleração dos parâmetros responsáveis pelo ajuste do domínio em momentos apropriados. Este processo ocorre independentemente do domínio inicial fornecido conter ou não a solução ótima.

Outra melhoria está relacionada à obtenção de maior confiabilidade, com a execução da etapa 18. Através desta etapa, o algoritmo é capaz de encontrar o valor ótimo ou um valor próximo do ótimo em praticamente todas as execuções realizadas, graças ao processo que trata a estagnação do algoritmo. Este processo expande o domínio em torno da melhor solução encontrada até o momento, evitando que o algoritmo fique preso em uma região de mínimo local. Este processo também ocorre independentemente do domínio inicial fornecido conter ou não a solução ótima.

O Algoritmo 1 mostra todas as etapas percorridas.

Algoritmo 1 Algoritmo ARACO

Entrada:	Variáveis k, θ, ρ, nc_max, Δ_m, Δ_1, Δ_2, Δ_3, F_{MIN}, t e o número de iterações sem melhoria
Etapa 1:	Inicializar os valores de todas as variáveis. Inicializar os valores de x^i_{min} e x^i_{max}, que representam o domínio fornecido para cada variável, ou seja, o valor mínimo e máximo de cada uma das dimensões do domínio. Definir a condição de parada do algoritmo.
Etapa 2:	Dividir o domínio inicial em k pontos iguais para cada variável, segundo (5). Neste momento, ocorre o processo de discretização das variáveis, através do método da grade. Com a execução deste processo, o domínio contínuo é convertido em um domínio discreto. Este processo é importante porque o todo o restante do algoritmo irá trabalhar com o domínio discretizado. O resultado é mostrado na Tabela 1.
Etapa 3:	Inicializar a matriz de feromônios τ, que representa a quantidade de feromônios armazenada em cada ponto da matriz de valores discretos do domínio, e que está mostrada na Tabela 2. Inicializar a quantidade de formigas autoadaptativa m, que possui um valor dinâmico devido ao fato de que, quando um domínio é dividido em poucos pontos discretos, uma pequena quantidade de formigas é suficiente para a realização da busca. Porém, quando o domínio for dividido em vários pontos discretos, podem ser necessárias muitas formigas para realizar a busca. Inicializar o valor da variável f_{min}, que armazena o valor heurístico da melhor solução encontrada na iteração atual, e o valor da variável nc, que controla o número de vezes que as formigas irão gerar novas rotas a cada iteração.
Etapa 4:	Gerar algumas soluções aleatórias.
Etapa 5:	Calcular o valor heurístico das soluções aleatórias criadas, de acordo com a função que será minimizada ou maximizada pelo algoritmo. Classificar as soluções aleatórias criadas de acordo com o valor heurístico.
Etapa 6:	Calcular o valor do incremento adaptativo Q do feromônio nesta iteração, conforme (7), de forma que o valor de Q permaneça proporcional ao valor heurístico das soluções encontradas na solução atual, mantendo assim a distribuição do feromônio proporcional na matriz τ. Distribuir o feromônio inicial das melhores soluções criadas, para fornecer algumas informações iniciais à matriz do feromônio τ, e assim conduzir as primeiras formigas em direção a soluções mais atrativas, acelerando o processo de conversão do algoritmo.
Etapa 7:	Criar novas rotas para formigas, através do método da roleta, utilizando (8). Cada formiga cria sua nova rota, variável a variável, realizando cálculo probabilístico, que considera o feromônio existente em cada ponto da matriz de feromônios. Como o cálculo da quantidade de feromônio utiliza o valor heurístico das soluções, pode-se dizer que são utilizados os mesmos critérios definidos no conceito inicial de ACO.
Etapa 8:	Avaliar as soluções geradas nesta iteração nc, de acordo com a função que será minimizada ou maximizada pelo algoritmo. O objetivo desta etapa é encontrar e armazenar a melhor solução, dentre as soluções criadas nesta iteração.
Etapa 9:	Realizar o processo de evaporação do feromônio, segundo (9). Este processo garante que os caminhos que possuam menor concentração de feromônios se tornem cada vez menos atraentes, até que não sejam mais percorridos pelas formigas. Por outro lado, os caminhos que apresentam maior concentração de feromônios se mantém atraentes e capazes de orientar as formigas na busca por melhores soluções.

Etapa 10:	Reforçar o feromônio em todos os pontos da melhor solução encontrada nesta iteração, utilizando (6), de forma que a matriz do feromônio sempre tenha quantidade maior de feromônio nos caminhos que são mais atrativos.
Etapa 11:	Atualizar o valor de f_{min} para o valor da melhor solução encontrada na atual iteração *nc*, caso esse valor seja menor que f_{min}. Neste momento, f_{min} deve armazenar o valor heurístico da melhor solução encontrada na iteração atual.
Etapa 12:	Incrementar o valor de *nc*. Caso *nc* <= *nc_max*, volte à etapa, pois não foi gerado o número de rotas necessárias para o ajuste do domínio. Caso contrário, ou seja, *nc* > *nc_max*, prosseguir para a etapa 13.
Etapa 13:	Verificar se a melhor solução encontrada nessa iteração (f_{min}) é a melhor solução global encontrada (F_{MIN}). Em caso afirmativo, ir para a etapa 14. Caso contrário, ir para a etapa 15.
Etapa 14:	Armazenar o valor da nova melhor solução global na variável F_{MIN} e verificar se a nova solução encontrada é 10% melhor que a anterior, para controlar se o algoritmo está estagnado e redefinir o valor de H_i, se for necessário. O conhecimento de quando o algoritmo está estagnado é importante, porque permite a execução subsequente de uma estratégia para tratamento desse estado de estagnação, permitindo que o algoritmo saia da região de mínimo local e continue buscando melhores soluções. Para tanto, é necessário armazenar o valor de H_i, no momento em que um novo valor de F_{MIN} for encontrado e o algoritmo não estiver estagnado.
Etapa 15:	Incrementar a variável que controla a quantidade de iterações sem melhoria para detectar a estagnação do algoritmo, e também a variável *t*. A variável *t* irá incrementar *k*, responsável pela divisão autoadaptativa do domínio, quando o valor de *t* for 15. A variável *k* é um parâmetro adaptativo, que precisa ter um pequeno valor inicial para que o algoritmo possa localizar uma região promissora, mas seu valor é aumentado gradativamente para permitir que novas buscas sejam realizadas com maior grau de precisão, quando melhores resultados não são alcançados.
Etapa 16:	Realizar o processo de ajuste de domínio autoadaptativo, com base na posição da matriz de feromônios τ que possui concentração maior de feromônios. Para isso, se o feromônio depositado pelas formigas estiver concentrado próximo à borda do domínio, a melhor solução encontrada nesta iteração tende a estar localizada fora do domínio. Neste caso, para localizar um domínio que contenha a solução ótima, é necessário expandir e ajustar o domínio por meio de (10), (11), (14) e (15). Por outro lado, se o feromônio depositado pelas formigas estiver concentrado próximo do centro do domínio, a melhor solução encontrada nesta iteração tende a estar dentro do domínio. Neste caso, será necessário reduzir o domínio utilizando (12) e (13) para realizar a busca na próxima iteração com maior grau de precisão.
Etapa 17:	Aumentar a velocidade de convergência do algoritmo, acelerando os parâmetros de ajuste de domínio autoadaptativos, se for o caso. Em outras palavras, acelerar o valor de Δ_2 quando o feromônio estiver concentrado em torno das posições centrais do domínio e, acelerar o valor de Δ_1 quando o feromônio estiver concentrado na borda superior ou inferior do domínio. Quando essas situações oportunas deixarem de existir, os parâmetros Δ_1 e Δ_2 voltam aos seus valores iniciais.
Etapa 18:	Aumentar a taxa de sucesso do algoritmo, ou seja, o número de execuções em que o algoritmo consegue encontrar o valor ótimo, por meio do processo de tratamento da estagnação do algoritmo. Em outras palavras, se o algoritmo estiver estagnado, redefinir e ampliar o domínio gradativamente em torno da melhor solução encontrada até o momento, usando (17) e (18), permitindo assim que o algoritmo escape de uma região de mínimo local.
Etapa 19:	Verificar se a condição de parada foi alcançada. Em caso afirmativo, ir para a etapa 20. Caso contrário, voltar para a etapa 2.
Etapa 20:	Finalizar o algoritmo.
Saída:	Valores ótimos ou próximos do ótimo para a função de *benchmark* otimizada

3.10.2 – Fluxograma

A Figura 4 descreve graficamente todo o processo executado durante o algoritmo ARACO. É importante destacar que cada ação realizada na Figura 4 é identificada por um número, que é o número da ação correspondente na etapa da seção 3.10.1.

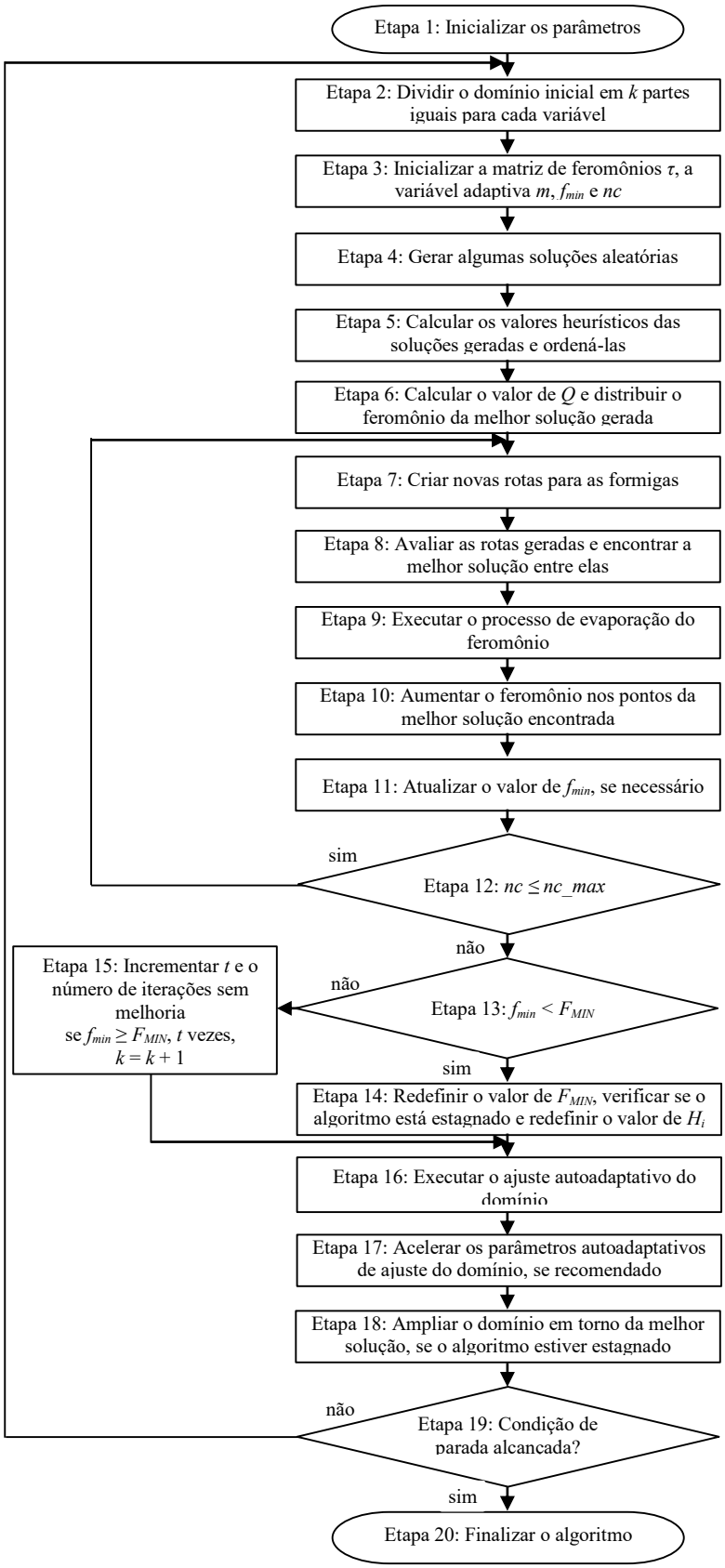


Figura 4 – Fluxograma do ARACO

3.11 – Comparação entre ARACO e RACO

Após a apresentação das principais operações realizadas por ARACO e da sequência das ações realizadas por ele, este tópico apresenta uma breve comparação entre o algoritmo ARACO e o seu predecessor, o algoritmo RACO. A Tabela 3 apresenta essa comparação.

Tabela 3 – Comparação entre os algoritmos ARACO e RACO

Critério de comparação	ARACO	RACO
É um algoritmo de busca de amplo alcance	Sim	Sim
Aplica estratégias autoadaptativas para o ajuste de domínio	Sim	Sim
Aplica estratégias autoadaptativas para o incremento de feromônio	Sim	Sim
Aplica estratégias autoadaptativas para divisão do domínio	Sim	Sim
Define a quantidade de formigas por maneira autoadaptativa	Sim	Sim
Realiza a aceleração dos parâmetros autoadaptativos de ajuste do domínio	Sim	Não
Realiza o tratamento de estagnação do algoritmo	Sim	Não

3.12 – Teste e validação

Para comparação dos resultados obtidos por ARACO com os resultados encontrados pelos outros algoritmos citados no trabalho, foram utilizados os valores disponíveis nos trabalhos de Dorigo *et al.* [20], Chen *et al.* [25] e Abdullah e Ahmed [31].

Todas as comparações entre os algoritmos são baseadas no número de avaliações de função realizadas até que a condição de parada seja alcançada, ao invés do tempo de execução ou outra medida que possa estar relacionada ao desempenho do equipamento utilizado ou da linguagem de programação utilizada. Ressalta-se aqui que este critério foi escolhido, porque os demais algoritmos citados na pesquisa também utilizam este critério. Dessa maneira, torna-se possível comparar o ARACO com eles, pois utilizam o mesmo critério de avaliação, além da mesma condição de parada. O ARACO considera uma avaliação de função quando é realizado o processo de ajuste adaptativo do domínio, que se refere à etapa 16 do fluxograma apresentado na seção 3.10.1. Este critério é utilizado para verificar o desempenho do algoritmo.

Outro critério de comparação utilizado é a taxa de sucesso, que representa qual é a porcentagem das execuções do algoritmo que conseguiram encontrar o valor ótimo ou um valor próximo ao ótimo. Este critério é utilizado para verificar a confiabilidade do algoritmo.

Em todos os cenários utilizados para teste neste trabalho, os valores dos principais parâmetros do ARACO são $k = 11$, $\theta = 0.2$, $\rho = 0.5$, $nc\_max = 50$, $\Delta_m = 2$, $\Delta_1 = 1.25$, $\Delta_2 = 0.05$ e $\Delta_3 = 0.1$. Os valores dos parâmetros foram definidos dessa forma, pois são os valores utilizados no algoritmo RACO e, como o algoritmo ARACO foi comparado em todos os cenários com o algoritmo RACO, é necessário que os parâmetros iniciais dos dois tenham os mesmos valores, para que a comparação seja mais justa. Vale a pena destacar que os valores de k, Δ_1, Δ_2, e Δ_3 são alterados dinamicamente durante a execução do algoritmo, então os valores atribuídos se referem apenas a valores iniciais. Outro parâmetro interessante a ser destacado é que são geradas 100 rotas aleatórias em cada início de iteração do algoritmo, e que as

30 rotas que gerarem o melhor valor heurístico para a função que será otimizada irão depositar seu feromônio na matriz τ, para fornecer ao algoritmo a informação inicial necessária para conduzir a construção das próximas soluções. Outra informação importante é que o algoritmo será considerado estagnado, e será executada a estratégia de tratamento de estagnação, através do ajuste do domínio em torno da melhor solução encontrada, quando forem executadas 30 iterações sem que seja encontrada uma solução que tenha valor heurístico pelo menos 10% melhor do que o valor heurístico da melhor solução encontrada até o momento. Os valores dos parâmetros responsáveis pela detecção e tratamento da estagnação foram determinados empiricamente, ou seja, foram realizados testes exaustivos para determinar os valores que produziram os melhores resultados.

4 – RESULTADOS

O objetivo dessa seção é comprovar a eficiência do ARACO. Para isso, foram realizados testes nos mesmos cenários dos testes realizados pelo algoritmo RACO, já que ARACO é proposto como um algoritmo derivado e que propõe melhorias a RACO. Por isso, ARACO será comparado com RACO durante boa parte dessa seção.

Para realizar os testes e comprovar a eficiência do algoritmo ARACO, foi desenvolvida uma aplicação utilizando como ambiente de desenvolvimento integrado (IDE – *Integrated Development Environment*), o MATLAB, na versão R2016A.

A Figura 5 mostra a tela para teste e validação dos resultados alcançados pelo algoritmo ARACO em um dos grupos de funções testado. Pode-se notar que, além de selecionar qual é a função que será otimizada, também é possível realizar a alteração do valor inicial de vários parâmetros do algoritmo do lado esquerdo da tela, como k, θ, Δ_m, Δ_1, Δ_2, Δ_3, nc_max e ρ. Pode-se também visualizar, no canto superior direito, qual é o valor estimado encontrado na literatura como solução para o problema de otimização e quais são os valores de cada uma das variáveis da função, quando esse valor estimado é encontrado. Após a execução do algoritmo, é possível visualizar, no canto inferior direito, qual é o valor encontrado como solução para o problema de otimização, quais são os valores obtidos para cada uma das variáveis do problema e quantas avaliações de função haviam sido realizadas no momento em que a melhor solução foi encontrada. Também fica disponível durante a execução do algoritmo, um gráfico onde é possível visualizar a evolução dos resultados encontrados na medida em que são realizadas novas avaliações de função no algoritmo.

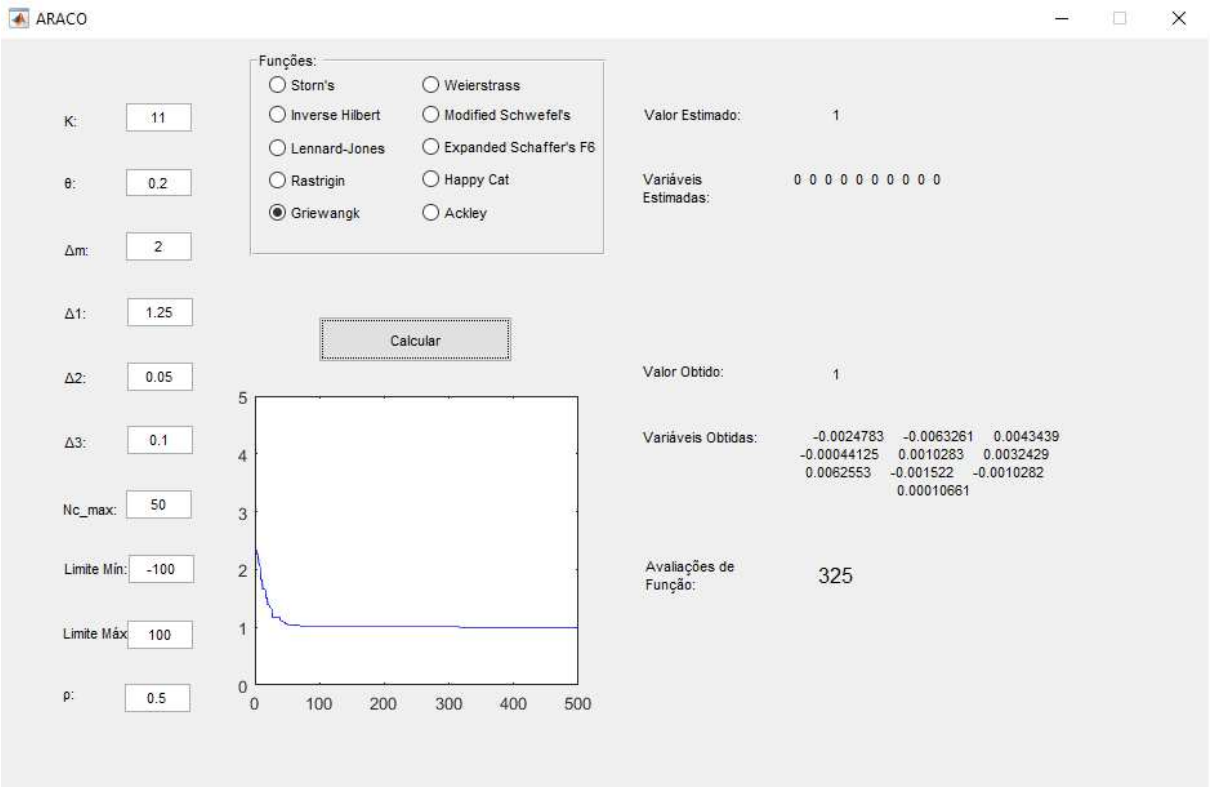


Figura 5 – Tela para simulação do funcionamento do ARACO

Inicialmente, serão apresentados os resultados de testes realizados em um cenário em que o problema é solucionado por um algoritmo de busca de alcance limitado, ou seja, quando é fornecido um domínio inicial que contém a solução ideal para a função. Também serão apresentados os resultados de testes realizados em um cenário em que o problema é solucionado apenas por um algoritmo de busca de amplo alcance, ou seja, quando é fornecido um domínio inicial que não contém a solução ideal. Após o resultado destes testes iniciais, serão apresentados os resultados de testes realizados utilizando as funções de *benchmark* usadas no *IEEE Congress of Evolutionary Computation Benchmark Test Functions* (CEC 2019 *100-Digit Challenge*).

4.1 – Cenário em que o domínio inicial contém a solução ótima

Nessa seção, os resultados obtidos pelo algoritmo ARACO foram comparados principalmente com os resultados encontrados pelo algoritmo ACO_R [20], que é o algoritmo desenvolvido pelo criador do conceito de ACO, e com os métodos citados por ACO_R, além dos resultados encontrados pelo algoritmo RACO [25], que foi a referência utilizada para a criação do ARACO.

Como realizado em ACO_R [20], as comparações foram divididas em três grupos: métodos de aprendizado de probabilidade que modelam e amostram distribuições de probabilidade, metaheurísticas desenvolvidas para otimização combinatória e adaptadas a domínios contínuos e, métodos inspirados no comportamento das formigas. Nos três grupos de comparações, são acrescentadas a comparação com o ACO_R e com o RACO.

4.1.1 – Métodos de aprendizado de probabilidade que modelam e amostram distribuições de probabilidade

Os métodos para comparação nessa seção são três versões de estratégias evolutivas: estratégia evolutiva com 1/5 da regra de sucesso (*(1+1)-ES with 1/5th-success-rule*), estratégia de evolutiva com adaptação cumulativa de tamanho de passo (CSA-ES – *Evolutionary Strategy with Cumulative Step Size Adaptation*), estratégia evolutiva com adaptação da matriz de covariância (CMA-ES – *Evolutionary Strategy with Covariance Matrix Adaptation*), além do algoritmo evolutivo de estimação de densidade iterada (IDEA – *Iterated Density Estimation Algorithm*) e do algoritmo de otimização bayesiana mista (MBOA – *Mixed-Bayesian Optimization Algorithm*). O tamanho da população usada nos algoritmos citados anteriormente é escolhido para cada par algoritmo-problema [32]. A menor população é selecionada do conjunto p ∈ [10, 20, 50, 100, 200, 400, 800, 1600, 3200]. Os parâmetros utilizados por ACO_R são *m* (número de formigas) = 2, *n* (velocidade de convergência) = 0.85, *q* (localidade do processo de busca) = 10^{-4} e *k* (tamanho do arquivo) = 50. Os parâmetros utilizados por RACO são exatamente os mesmos que os utilizados por ARACO.

Para a realização dos testes, cada função de *benchmark* foi executada 20 vezes. A quantidade de execuções realizadas foi determinada desta maneira, por ser a mesma quantidade de execuções realizadas por RACO.

O critério de comparação entre os algoritmos utilizado nessa seção é a mediana da quantidade de avaliações de função (MNFE – *Median Number of Functions Evaluations*) realizadas até que a condição de parada seja alcançada. Este critério de comparação foi escolhido por ser o mesmo critério utilizado pelos outros métodos utilizados para comparação nessa seção. A condição de parada é mostrada por (19).

$$|f - f^*| < \in \tag{19}$$

Onde *f* é o melhor valor heurístico encontrado pelo ARACO; *f** é o valor ótimo encontrado na literatura para a função de *benchmark*; e ∈ equivale a 10^{-10}

Pode-se notar que essa condição de parada não é a ideal, visto que ela utiliza o valor ótimo encontrado na literatura para a função, mas ela foi mantida porque é a mesma condição de parada utilizada pelos demais algoritmos. Sendo assim, para que seja realizada uma comparação mais justa entre os algoritmos, a mesma condição de parada é utilizada.

A Tabela 4 mostra as funções de *benchmark* utilizadas neste cenário. Todas elas possuem 10 dimensões. A coluna Função mostra o nome da função de *benchmark*. A coluna Fórmula mostra a fórmula usada para calcular o valor da função que será minimizada. A coluna Valor ótimo mostra o valor ideal para uma das variáveis da função. A coluna Mínimo mostra o valor mínimo da função, quando os valores ótimos para cada uma das variáveis foi encontrado.

Tabela 4 – Funções de *benchmark* utilizadas no cenário 5.1.1

Função	Fórmula	Valor ótimo x*	Mínimo f(x*)
Sphere	$f(\vec{x}) = \sum_{i=1}^{n} {x_i}^2$	$\overrightarrow{x^*} = (0, \dots, 0)$	$f_{min} = 0$
Ellipsoid	$f(\vec{x}) = \sum_{i=1}^{n} (100^{\frac{i-1}{n-1}} x_i)^2$	$\overrightarrow{x^*} = (0, \dots, 0)$	$f_{min} = 0$
Cigar	$f(\vec{x}) = {x_1}^2 + 10^4 \sum_{i=2}^{n} {x_i}^2$	$\overrightarrow{x^*} = (0, \dots, 0)$	$f_{min} = 0$
Tablet	$f(\vec{x}) = 10^4 {x_1}^2 + \sum_{i=2}^{n} {x_i}^2$	$\overrightarrow{x^*} = (0, \dots, 0)$	$f_{min} = 0$
Rosenbrock	$f(\vec{x}) = \sum_{i=1}^{n-1} [100({x_i}^2 - x_{i+1})^2 + (x_i - 1)^2]$	$\overrightarrow{x^*} = (1, \dots, 1)$	$f_{min} = 0$

Os resultados utilizados para comparação de ACO_R, (1+1) ES, CSA-ES, CMA-ES, IDEA e MBOA foram obtidos de Dorigo *et al.* [20]. Enquanto, os resultados utilizados para comparação de RACO foram obtidos de Chen *et al.* [25].

A Tabela 5 mostra a mediana da quantidade de avaliações de função necessárias para que cada método possa atender a condição de parada. A mediana é mostrada entre parênteses, após o 1.0, apenas para o algoritmo que possui o melhor resultado para uma função. As medianas dos outros algoritmos podem ser calculadas proporcionalmente com base no valor mostrado na Tabela 5. Por exemplo, se um algoritmo tem a sua mediana mostrada como 1.0 (86), significa que o algoritmo tem a melhor mediana e o valor dessa mediana é 86. Por outro lado, se um algoritmo que não possui o melhor resultado está com sua mediana mostrada na Tabela 5 como 2.23, quer dizer que o valor de sua mediana é igual a 2.23 multiplicado pela mediana do algoritmo que encontrou o melhor resultado para aquela função. Este método foi adotado para auxiliar no processo de verificar o quanto, proporcionalmente, um método apresenta desempenho superior a outro método. Alguns métodos não conseguiram encontrar o valor ótimo em todas as execuções para a função de Rosenbrock. Estes casos foram representados na Tabela 5 com *.

Tabela 5 – Comparação em relação a MNFE entre ARACO e os algoritmos do cenário 5.1.1

Funcão	ARACO	RACO	ACO_R	(1+1) ES	CSA-ES	CMA-ES	IDEA	MBOA
Sphere $\vec{x}$:[-3,7]^n, n=10	**1.0 (86)**	2.23	17.52	15.93	25.48	20.70	79.65	764.65
Ellipsoid $\vec{x}$:[-3,7]^n, n=10	**1.0 (103)**	2.11	112.33	2851.45	4752.42	43.20	69.12	604.85
Cigar $\vec{x}$:[-3,7]^n, n=10	**1.0 (114.5)**	2.05	46.95	20457.64	26829.69	33.53	154.27	402.44
Tablet $\vec{x}$:[-3,7]^n, n=10	**1.0 (89)**	2.28	28.84	1326.76	1874.77	49.03	83.64	692.22
Rosenbrock $\vec{x}$:[-5,5]^n, n=10	***1.0 (997)**	1.25	*7.93	*367.79	1298.09	7.21	*1514.44	*7932.79

Em relação aos resultados obtidos, pode-se notar que ARACO tem desempenho superior aos demais algoritmos em todas as funções testadas. Mais do que isso, em quatro das cinco funções, ARACO consegue otimizar as funções gastando menos da metade das avaliações de função do que o algoritmo que alcançou o segundo melhor resultado. Se o desempenho for comparado aos demais algoritmos, a

diferença se torna ainda mais impressionante, pois a quantidade de avaliações de função realizadas por ARACO chega a ser 90% menor.

A superioridade do ARACO neste cenário foi obtida graças a aceleração dos parâmetros autoadaptativos de ajuste de domínio Δ_1, Δ_2 e Δ_3, em momentos oportunos. É possível realizar essa afirmação porque foram realizados testes para verificar se a estratégia de tratamento de estagnação do algoritmo foi utilizada em algum momento e, estes testes mostraram que essa estratégia não foi utilizada e, portanto, em nenhum momento houve estagnação do algoritmo, em nenhuma das funções de *benchmark* testadas.

O Gráfico 1 mostra uma comparação entre os dois algoritmos que alcançaram os melhores resultados, em relação a MNFE, para as funções de *benchmark* da Tabela 5.

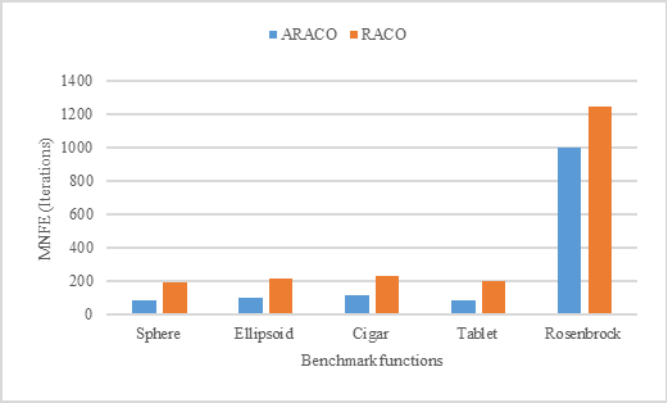


Gráfico 1 – Comparação dos resultados de ARACO e RACO no cenário 5.1.1

4.1.2 – Metaheurísticas desenvolvidas para otimização combinatória e adaptadas a domínios contínuos

As metaheurísticas utilizadas para comparação neste cenário são o algoritmo genético contínuo (CGA – *Continuous Genetic Algorithm*), a busca tabu contínua aprimorada (ECTS – *Enhanced Continuous Tabu Search*), o método de recozimento simulado aprimorado (ESA – *Enhanced Simulated Annealing*) e a evolução diferencial (DE – *Differential Evolution*). Os resultados utilizados para comparação de ACO_R, CGA, ECTS, ESA e DE foram obtidos de Dorigo *et al.* [20]. Assim como, os resultados utilizados para comparação de RACO foram obtidos de Chen *et al.*[25].

As funções de *benchmark* utilizadas neste cenário estão apresentadas na Tabela 6, exceto a função Rosenbrock, que já foi apresentada na Tabela 4. A coluna Função mostra o nome da função de *benchmark*. A coluna Fórmula mostra a fórmula usada para calcular o valor da função que será minimizada. A coluna Ótimo mostra o valor ideal para uma das variáveis da função. A coluna Mínimo mostra o valor mínimo da função, quando os valores ótimos para cada uma das variáveis foi encontrado.

Para realização dos testes, o algoritmo ARACO foi executado 100 vezes para cada função de *benchmark*. A quantidade de execuções realizadas foi determinada desta maneira, por ser a mesma quantidade de execuções realizadas por RACO.

Os critérios de comparação utilizados nessa seção são a média da quantidade de avaliações de função (ANFE – *Average Number of Functions Evaluations*) realizadas até que a condição de parada seja alcançada, além da taxa de sucesso. Estes critérios de comparação foram escolhidos por serem os mesmos critérios utilizados pelos outros métodos utilizados para comparação nessa seção. A taxa de sucesso, que é um critério utilizado pela primeira vez nessa seção, representa a porcentagem das execuções realizadas que conseguiram alcançar o valor ótimo ou um valor próximo do ótimo. A condição de parada é mostrada por (20):

$$|f - f^*| < \epsilon_1 f^* + \epsilon_2 \quad (20)$$

Onde f é o melhor valor heurístico encontrado pelo ARACO; f^* é o valor ótimo encontrado na literatura para a função de *benchmark*; e $\epsilon_1 = \epsilon_2 = 10^{-4}$.

Pode-se notar que essa condição de parada não é a ideal, visto que ela utiliza o valor ótimo encontrado na literatura para a função, mas ela foi mantida porque é a mesma condição de parada utilizada pelos demais algoritmos. Sendo assim, para que seja realizada uma comparação mais justa entre os algoritmos, a mesma condição de parada é utilizada.

Tabela 6 – Funções de benchmark utilizadas no cenário 5.1.2

Funcão	Fórmula	Ótimo x*	Mínimo f(x*)
Branin RCOS	$f(\vec{x}) = (x_2 - \frac{5{x_1}^2}{4\pi^2} + \frac{5x_1}{\pi} - 6)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10$	3 $\acute{o}timos$	$f_{min} = 0.397887$
B2	$f(\vec{x}) = {x_1}^2 + 2{x_2}^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$	$\overrightarrow{x^*} = (0,0)$	$f_{min} = 0$
Easom	$f(\vec{x}) = -\cos(x_1)\cos(x_2)\exp(-((x_1 - \pi)^2 + (x_2 - \pi)^2))$	$\overrightarrow{x^*} = (\pi, \pi)$	$f_{min} = -1$
Goldstein and Price	$f(\vec{x}) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3{x_1}^2 - 14x_2 + 6x_1x_2)][30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12{x_1}^2 + 48x_2 - 36x_1x_2 + 27{x_2}^2)]$	$\overrightarrow{x^*} = (0,-1)$	$f_{min} = 3$
Zakharov	$f(\vec{x}) = \sum_{i=1}^{n} {x_i}^2 + (\sum_{i=1}^{n} 0.5ix_i)^2 + (\sum_{i=1}^{n} 0.5ix_i)^4$	$\overrightarrow{x^*} = (0,\ldots,0)$	$f_{min} = 0$
De Jong	$f(\vec{x}) = {x_1}^2 + {x_2}^2 + {x_3}^2$	$\overrightarrow{x^*} = (0,0,0)$	$f_{min} = 0$
Hartmann ($H_{3,4}$)	$f(\vec{x}) = -\sum_{i=1}^{4} c_i \exp(-\sum_{j=1}^{3} a_{ij}(x_j - p_{ij})^2)$ $a_{ij} = \begin{vmatrix} 3.0 & 10.0 & 30.0 \\ 0.1 & 10.0 & 35.0 \\ 3.0 & 10.0 & 30.0 \\ 0.1 & 10.0 & 35.0 \end{vmatrix}$ $c_i = \begin{vmatrix} 1.0 \\ 1.2 \\ 3.0 \\ 3.2 \end{vmatrix}$ $p_{ij} = \begin{vmatrix} 0.3689 & 0.1170 & 0.2673 \\ 0.4699 & 0.4387 & 0.7470 \\ 0.1091 & 0.8732 & 0.5547 \\ 0.0382 & 0.5743 & 0.8828 \end{vmatrix}$	${x_1}^* = 0.114$ ${x_2}^* = 0.555$ ${x_3}^* = 0.855$	$f_{min} = -3.8628$

Shekel ($S_{4,k}$, k = 5,7,10)	$f(\vec{x}) = -\sum_{i=1}^{k}(\sum_{j=1}^{4}(x_j - a_{ji})^2 + c_i)^{-1}$ $a_{ij} = \begin{vmatrix} 4.0 & 4.0 & 4.0 & 4.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \\ 8.0 & 8.0 & 8.0 & 8.0 \\ 6.0 & 6.0 & 6.0 & 6.0 \\ 3.0 & 7.0 & 3.0 & 7.0 \\ 2.0 & 9.0 & 2.0 & 9.0 \\ 5.0 & 3.0 & 5.0 & 3.0 \\ 8.0 & 1.0 & 8.0 & 1.0 \\ 6.0 & 2.0 & 6.0 & 2.0 \\ 7.0 & 3.6 & 7.0 & 3.6 \end{vmatrix}$ $c_i = \begin{vmatrix} 0.1 \\ 0.2 \\ 0.2 \\ 0.4 \\ 0.4 \\ 0.6 \\ 0.3 \\ 0.7 \\ 0.5 \\ 0.5 \end{vmatrix}$	$x_1^* = 4$ $x_2^* = 4$ $x_3^* = 4$ $x_4^* = 4$	$f_{min}^{k=5} = -10.1532$ $f_{min}^{k=7} = -10.4029$ $f_{min}^{k=10} = -10.5364$
Hartmann ($H_{6,4}$)	$f(\vec{x}) = -\sum_{i=1}^{4} c_i \exp(-\sum_{j=1}^{3} a_{ij}(x_j - p_{ij})^2)$ $a_{ij} = \begin{vmatrix} 10.0 & 3.00 & 17.0 & 3.50 & 1.70 & 8.00 \\ 0.05 & 10.0 & 17.0 & 0.10 & 8.00 & 14.0 \\ 3.00 & 3.50 & 1.70 & 10.0 & 17.0 & 8.00 \\ 17.0 & 8.00 & 0.05 & 10.0 & 0.10 & 14.0 \end{vmatrix}$ $c_i = \begin{vmatrix} 1.0 \\ 1.2 \\ 3.0 \\ 3.2 \end{vmatrix}$ $p_{ij} = \begin{vmatrix} 0.1312 & 0.1696 & 0.5569 & 0.0124 & 0.8283 & 0.5886 \\ 0.2329 & 0.4135 & 0.8307 & 0.3736 & 0.1004 & 0.9991 \\ 0.2348 & 0.1451 & 0.3522 & 0.2883 & 0.3047 & 0.6650 \\ 0.4047 & 0.8828 & 0.8732 & 0.5743 & 0.1091 & 0.0381 \end{vmatrix}$	$x_1^* = 0.201$ $x_2^* = 0.150$ $x_3^* = 0.477$ $x_4^* = 0.275$ $x_5^* = 0.311$ $x_6^* = 0.657$	$f_{min} = -3.3223$
Griewangk	$f(\vec{x}) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$\overrightarrow{x^*} = (0, \dots, 0)$	$f_{min} = 0$
Martin & Gaddy	$f(\vec{x}) = (x_1 - x_2)^2 + (\frac{x_1 + x_2 - 10}{3})^2$	$\overrightarrow{x^*} = (5, 5)$	$f_{min} = 0$

A Tabela 7 mostra uma comparação entre a média da quantidade de avaliações de função necessárias para que cada método consiga alcançar a condição de parada. Assim como na seção anterior, a média é mostrada entre parênteses, após o 1.0, apenas para o algoritmo que possui o melhor resultado. As médias dos outros algoritmos podem ser calculadas proporcionalmente com base no valor mostrado na Tabela 7. Quando não existe valor mostrado na Tabela 7 para determinado algoritmo, significa que o resultado para aquela função de *benchmark* não estava disponível na literatura.

Tabela 7 – Comparação em relação a ANFE entre ARACO e os algoritmos do cenário 5.1.2

Funcão	ARACO	RACO	ACO_R	CGA	ECTS	ESA	DE
Branin RCOS $\vec{x}$:[-5,15]^n, n=2	**1.0 (17.8)**	4.48	48.17	34.41	13.76	-	-
B_2 $\vec{x}$:[-100,100]^n, n=2	**1.0 (19.8)**	4.09	28.23	21.71	-	-	-
Easom $\vec{x}$:[-100,100]^n, n=2	**1.0 [96%] (49.7)**	1.11 [70%]	15.53 [98%]	29.51	-	-	-
Goldstein and Price $\vec{x}$:[-2,2]^n, n=2	**1.0 (17)**	2.88	23.1	24.45	13.58	46.2	-
Rosenbrock (R_2) $\vec{x}$:[-5,10]^n, n=2	**1.0 (74.8)**	1.05	10.9	12.83	6.41	10.9	8.34
Zakharov (Z_2) $\vec{x}$:[-5,10]^n, n=2	**1.0 (15.5)**	1.29	18.87	40.25	12.58	1019	-
De Jong $\vec{x}$:[-5.12,5.12]^n, n=3	**1.0 (11)**	3.81	35.63	67.70	-	-	35.63
Hartmann ($H_{3,4}$) $\vec{x}$:[0,1]^n, n=3	**1.0 (10.8)**	2.48	31.66	53.83	50.66	63.33	-
Shekel ($S_{4,5}$) $\vec{x}$:[0,10]^n, n=4	6.06 **[87%]**	**1.0** [56%] **(47.9)**	16.55 [57%]	12.73 [76%]	17.82 [75%]	24.19 [54%]	-
Shekel ($S_{4,7}$) $\vec{x}$:[0,10]^n, n=4	3.32	**1.0** [92%] **(48.9)**	15.29 [79%]	13.9 [83%]	18.07 [80%]	25.03 [54%]	-
Shekel ($S_{4,10}$) $\vec{x}$:[0,10]^n, n=4	2.97	**1.0** [97%] **(49.6)**	14.41 [81%]	13.1 [83%]	18.34 [80%]	23.58 [50%]	-
Rosenbrock (R_5) $\vec{x}$:[-5,10]^n, n=5	2.01	**1.0 (184.3)**	13.94 [97%]	22.08	11.62	29.05	-
Zakharov (Z_5) $\vec{x}$:[-5,10]^n, n=5	**1.0 (36.3)**	1.7	20.02	38.05	62.08	1922.64	-
Hartmann ($H_{6,4}$) $\vec{x}$:[0,1]^n, n=6	3.31 **[97%]**	**1.0** [50%] **(28.6)**	25.24	32.81	53.01	93.40	-
Griewangk $\vec{x}$:[-5.12,5,12]^n, n=10	16.78	**1.0** [97%] **(29.7)**	46.8 [61%]	-	-	-	430.57

Analisando os resultados obtidos, pode-se notar que ARACO consegue melhor desempenho em nove das quinze funções testadas, quando a comparação é realizada utilizando o valor de ANFE como critério. Mais do que isso, em cinco das nove funções em que ARACO consegue os melhores resultados dentre todos os métodos, ele consegue otimizar as funções gastando menos da metade das avaliações de função do segundo melhor algoritmo. Se o desempenho for comparado com os demais algoritmos, a diferença se torna ainda mais impressionante, pois a quantidade de iterações realizados por ARACO chega a ser 90% menor.

É interessante destacar que, ARACO fica em segundo lugar no quesito desempenho, nas seis funções em que não consegue os melhores resultados, tendo seus resultados superados apenas por RACO. É possível melhorar os resultados obtidos por ARACO alterando o procedimento adotado quando o algoritmo entra em estado de estagnação, fazendo que o domínio seja ampliado ao redor do menor valor encontrado em todas as variáveis, ao invés de apenas na variável que está mais estagnada. Embora os testes realizados mostraram que essa ação resulta em menor número de avaliações de função, ela não é recomendada e não foi implementada, pois apresenta como efeito colateral diminuição considerável nos valores de outro importante parâmetro mostrado na Tabela 7, a taxa de sucesso, o que compromete o objetivo do ARACO.

A taxa de sucesso é importante porque nem todos algoritmos encontram a solução ótima em todas as execuções realizadas. Através dela, pode-se medir qual é a porcentagem de execuções do algoritmo que encontra a solução ótima ou uma solução próxima da ótima, para cada função testada. Quanto mais alta for a taxa de sucesso, mais confiável é o algoritmo. A porcentagem mostrada entre colchetes na Tabela 7 representa a taxa de sucesso de cada algoritmo. Quando não existe uma porcentagem mostrada entre colchetes, significa que a taxa de sucesso é de 100%, ou seja, todas as execuções foram realizadas com sucesso. Analisando os resultados da Tabela 7, em todas as funções de *benchmark* propostas, ARACO consegue encontrar uma solução que atenda ao problema com uma taxa de sucesso igual ou superior ao seu principal concorrente, RACO. Por exemplo, a taxa de sucesso de RACO para a função Hartmann ($H_{6,4}$) é de 50%, e para a função de Shekel ($S_{4,5}$) é de 56%, enquanto para ARACO estes valores são de 97% e 87%, respectivamente.

Essa superioridade no quesito taxa de sucesso é explicada pela estratégia utilizada para tratamento da estagnação do algoritmo, através da ampliação do domínio em torno da melhor solução encontrada. Dessa maneira, existe maior probabilidade de ocorrer uma melhoria contínua dos valores obtidos, pois quando o algoritmo fica preso em uma região de mínimo local, fato que poderia fazer o algoritmo não encontrar a solução ideal, a estratégia utilizada por ARACO possibilita que o domínio seja ampliado ao redor da região onde se localiza o melhor valor encontrado até o momento, possibilitando ao algoritmo buscar gradativamente soluções que estejam fora dessa região de mínimo local, já que a ampliação pode ser efetuada várias vezes. Este processo garante que a taxa de sucesso do ARACO seja igual ou próxima a 100% para todas as quinze funções de *benchmark* testadas, fato que não acontece com os outros algoritmos.

Desta forma, pode-se dizer que o desempenho do ARACO é superior aos demais algoritmos, pois consegue encontrar o valor ótimo ou um valor próximo do ótimo, utilizando quantidade menor de avaliações de funções para isso. E o algoritmo também possui maior confiabilidade, porque é capaz de atingir uma taxa de sucesso maior que os demais algoritmos, ou seja, encontra a solução ótima ou uma solução próxima da ótima em percentual maior das execuções realizadas.

O Gráfico 2 mostra uma comparação entre os dois algoritmos que alcançaram os melhores resultados, em relação a ANFE, para as funções de *benchmark* da Tabela 7.

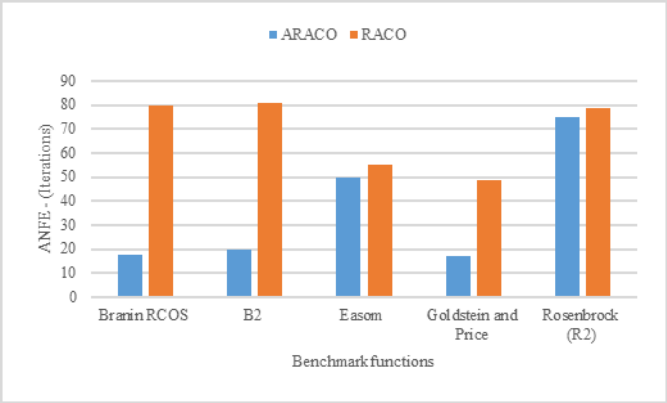


Gráfico 2 – Comparação dos resultados de ARACO e RACO no cenário 5.1.2

4.1.3 – Métodos inspirados no comportamento das formigas

Os métodos utilizados para comparação neste cenário são: CACO, API e CIAC. Os parâmetros utilizados em CACO, API e CIAC foram retirados de Dorigo *et al.* [20] e são: para o algoritmo CACO, quantidade de formigas m = 10, número de regiões r = 200, probabilidade de mutação p_1 = 0.5, probabilidade de crossover p_2 = 1; para o algoritmo API, quantidade de formigas m = 20, número de explorações para cada formiga t = 50, intervalos de pesquisa com falhas P_{lacal} = 50; e para o algoritmo CIAC, quantidade de formigas m = 100, intervalo de taxas de distribuição σ = 0.5, persistência do feromônio ρ = 0.1, número de mensagens iniciais μ = 10. Os parâmetros utilizados em ACO_R são: quantidade de formigas m = 2, velocidade de convergência n = 0.85, localidade do processo de busca q = 10^{-1} e tamanho do arquivo k = 50. Por último, os parâmetros utilizados por RACO são exatamente os mesmos utilizados por ARACO.

Os resultados utilizados para comparação de ACO_R, CACO, API e CIAC foram obtidos de Dorigo *et al.* [20]. Assim como os resultados utilizados para comparação de RACO foram obtidos de Chen *et al.* [25].

As funções de *benchmark* utilizadas neste cenário já foram apresentadas anteriormente, nas Tabela 4 e na Tabela 6. Para a realização dos testes, o algoritmo ARACO foi executado 100 vezes para cada função de *benchmark*. A quantidade de execuções realizadas foi determinada desta maneira, por ser a mesma quantidade de execuções realizadas por RACO.

Os critérios de comparação utilizados nessa seção são a média da quantidade de avaliações de função (ANFE – *Average Number of Functions Evaluations*) realizadas até que a condição de parada seja alcançada, além da taxa de sucesso. Estes critérios de comparação foram escolhidos por serem os mesmos critérios utilizados pelos outros métodos utilizados para comparação nessa seção A condição de parada utilizada é apresentada em (21):

$$|f - f^*| < \epsilon_1 f^* + \epsilon_2 \quad (21)$$

Onde f é o melhor valor heurístico encontrado pelo ARACO; f^* é o valor ótimo encontrado na literatura para a função de *benchmark*; e $\epsilon_1 = \epsilon_2 = 10^{-4}$.

Pode-se notar que essa condição de parada não é a ideal, visto que ela utiliza o valor ótimo encontrado na literatura para a função, mas ela foi mantida porque é a mesma condição de parada utilizada pelos demais algoritmos. Sendo assim, para que seja realizada uma comparação mais justa, a mesma condição de parada é utilizada.

A Tabela 8 mostra uma comparação entre a média da quantidade de avaliações de função necessárias para que cada método possa alcançar a condição de parada. Assim como na seção anterior, a média é mostrada entre parênteses, após o 1.0, apenas para o algoritmo que possui o melhor resultado, as outras médias podem ser calculadas proporcionalmente com base no valor mostrado na Tabela 8. Quando não existe valor mostrado na Tabela 8 para determinado algoritmo, significa que o resultado para aquela função de *benchmark* não estava disponível na literatura. A porcentagem mostrada entre colchetes na Tabela 8 representa a taxa de sucesso dos algoritmos. Quando não existe porcentagem entre colchetes, significa que a taxa de sucesso foi de 100%, ou seja, todas as execuções foram realizadas com sucesso.

Tabela 8 – Comparação em relação a ANFE entre ARACO e os algoritmos do cenário 5.1.3

Funcão	ARACO	RACO	ACO_R	CACO	API	CIAC
Rosenbrock (R_2) $\vec{x}$:[-5,10]^n, n=2	**1.0 (74.8)**	1.05	10.96	90.98	131.55	153.47
Sphere $\vec{x}$:[-5.12,5.12]^n, n=6	**1.0 (16.4)**	2.86	47.62	1333.41	619.08	3047.80
Griewangk $\vec{x}$:[-5.12,5,12]^n, n=10	16.61	**1.0 (30)** [97%]	46.33 [61%]	1668	-	1668 [52%]
Goldstein and Price $\vec{x}$:[-2,2]^n, n=2	**1.0 (17)**	2.88	22.58	316.23	-	1377.88 [56%]
Martin and Gaddy $\vec{x}$:[-20,20]^n, n=2	**1.0 (16)**	1.0	21.56	107.81	-	733.12 [20%]
B_2 $\vec{x}$:[-100,100]^n, n=2	**1.0 (19.8)**	4.07	27.37	-	-	602.31
Rosenbrock (R_5) $\vec{x}$:[-5,10]^n, n=5	2.01	**1.0 (184.3)**	13.49 [97%]	-	-	215.90 [90%]
Shekel ($S_{4,5}$) $\vec{x}$:[0,10]^n, n=4	6.06 **[87%]**	**1.0 (47.9)** [56%]	16.43 [57%]	-	-	821.5 [5%]

Analisando os resultados obtidos, pode-se observar que ARACO possui resultados superiores ou iguais em cinco entre as oito funções de *benchmark* testadas, quando a comparação é realizada utilizando o valor da ANFE como critério. O algoritmo fica em segundo lugar nas outras três funções, sendo superado apenas por RACO.

Porém, nos casos onde ARACO fica em segundo lugar, e em todas as funções avaliadas, a taxa de sucesso alcançada por ele é igual ou superior a todos os outros algoritmos, não chegando a 100% apenas na função Shekel ($S_{4,5}$). Enquanto os outros algoritmos possuem resultados bem mais modestos,

como por exemplo, utilizando para comparação a função Shekel ($S_{4,5}$), que é a única em que ARACO não possui 100% de taxa de sucesso, ARACO alcança 87%, enquanto RACO possui 56%, ACO_R possui 57% e CIAC apenas 5%.

Pode-se concluir então, que ARACO apresenta desempenho superior aos demais algoritmos em relação à média da quantidade de avaliações de função necessárias para alcançar a condição de parada na maioria das funções testadas. Nos casos em que o desempenho é inferior, o algoritmo consegue taxa de sucesso maior que o algoritmo com melhor desempenho, alcançando assim com sucesso o objetivo do algoritmo, que é minimizar a quantidade de avaliações de função necessárias para localizar o valor ótimo ou um valor próximo do ótimo, privilegiando obter a maior taxa de sucesso possível.

O Gráfico 3 mostra uma comparação entre os dois algoritmos que alcançaram os melhores resultados, em relação a ANFE, para as funções de *benchmark* da Tabela 8.

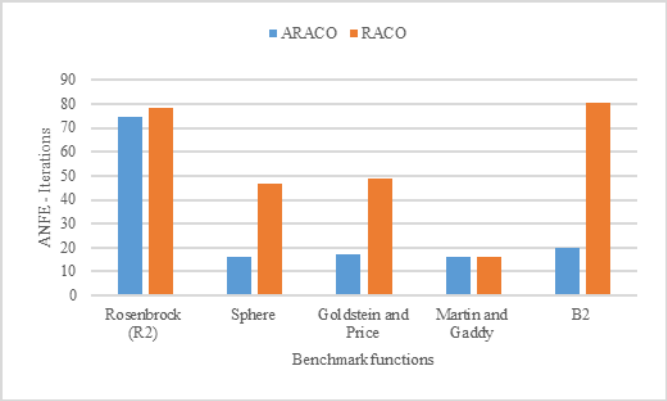


Gráfico 3 – Comparação dos resultados de ARACO e RACO no cenário 5.1.3

4.2 – Cenário em que o domínio inicial não contém a solução ótima

No cenário contemplado por essa seção, apenas algoritmos de busca de amplo alcance são capazes de encontrar a solução ótima ou uma solução próxima da ótima. Por isso, todas as comparações realizadas neste cenário utilizam apenas ARACO e RACO, pois eles são os únicos algoritmos citados na seção anterior, que possuem capacidade para encontrar a solução ideal, quando é fornecido um domínio inicial que não contém essa solução.

Os parâmetros utilizados nos testes dos algoritmos ARACO e RACO neste cenário são idênticos. Eles correspondem aos parâmetros iniciais ideais dos algoritmos, já citados anteriormente neste trabalho, que são $k = 11$, $\theta = 0.2$, $\rho = 0.5$, $nc\_max = 50$, $\Delta_m = 2$, $\Delta_1 = 1.25$, $\Delta_2 = 0.05$, e $\Delta_3 = 0.1$. Vale a pena lembrar que vários destes parâmetros são alterados dinamicamente, em ambos ou apenas em ARACO.

Foram utilizadas 14 funções de *benchmark* para comparação entre os dois algoritmos neste cenário. A Tabela 9 mostra 8 dessas funções, enquanto as outras 6 já foram apresentadas anteriormente neste trabalho. A coluna Função mostra o nome da função de *benchmark*. A coluna Fórmula mostra a

fórmula usada para calcular o valor da função que será minimizada. A coluna Valor ótimo mostra o valor ideal para uma das variáveis da função. A coluna Mínimo mostra o valor mínimo da função, quando os valores ótimos para cada uma das variáveis foi encontrado.

Para realização dos testes, o algoritmo ARACO foi executado 20 vezes para cada função de *benchmark*, em cada um dos domínios fornecidos. A quantidade de execuções realizadas foi determinada desta maneira, por ser a mesma quantidade de execuções realizadas por RACO.

Os critérios utilizados para comparação entre os algoritmos são a média da quantidade de avaliações de função (ANFE – *Average Number of Functions Evaluations*) realizadas até que a condição de parada seja alcançada, além da taxa de sucesso. A condição de parada adotada é apresentada em (22):

$$\max(h_1, h_2, \dots, h_n) < \in \qquad (22)$$

Onde h_i (i =1, 2, ..., n) é o valor de cada parte na divisão da grade do domínio, calculado em (5); e $\in$ é 10^{-5}.

A condição de parada mostrada em (22) poderia ser utilizada durante todos os cenários anteriores no trabalho, para evitar que a condição de parada ficasse condicionada ao valor determinado na literatura para a função de *benchmark*. Essa ação não foi realizada, porque era necessário que fossem utilizadas as mesmas condições de parada dos algoritmos citados, para que pudesse ser realizada uma comparação justa entre os resultados obtidos por eles.

Tabela 9 – Funções de *benchmark* utilizadas na comparação no cenário 5.2

Funcão	Fórmula	Valor Ótimo x*	Mínimo f(x*)
Rastrigin	$f(\vec{x}) = ({x_1}^2 + {x_2}^2 - \cos(18x_1) - \cos 18x_2$	$\overrightarrow{x^*} = (0,0)$	$f_{min} = 0.397887$
Shubert	$f(\vec{x}) = (\sum_{i=1}^{5} i\cos(i+(i+1)x_1))(\sum_{i=1}^{5} i\cos(i+(i+1)x_2))$	18 *ótimos*	$f_{min} = -186.7309$
Ackley	$f(\vec{x}) = -20\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} {x_i}^2}) - \exp(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i))$	$\overrightarrow{x^*} = (0,\dots,0)$	$f_{min} = 0$
Levy	$f(\vec{x}) = \sin^2(\pi y_1) + \sum_{i=1}^{n-1}[(y_i-1)^2(1 + 10\sin^2(\pi y_1+1))] + (y_n-1)^2(1 + 10\sin^2 2\pi y_n))$ $y_i = 1 + \frac{1}{4}(x_i - 1)$	$\overrightarrow{x^*} = (1,\dots,1)$	$f_{min} = 0$
Beale	$f(\vec{x}) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1{x_2}^2)^2 + (2.625 - x_1 + x_1{x_2}^3)^2$	$\overrightarrow{x^*} = (3, 0.5)$	$f_{min} = 0$
Six-Hump Camel-Back	$f(\vec{x}) = 4{x_1}^2 - 2.1{x_1}^4 + {x_1}^6/_3 + x_1x_2 - 4{x_2}^2 + 4{x_2}^4$	$\overrightarrow{x^*} = (0.08983, -0.7126), (-0.08983, 0.7126)$	$f_{min} = 0$
Bohachevsky	$f(\vec{x}) = {x_1}^2 + {x_2}^2 - 0.3\cos(3\pi x_1) + 0.3\cos(4\pi x_2) + 0.3$	$\overrightarrow{x^*} = (0, 0.24), (0, -0.24)$	$f_{min} = -0.24$
Hansen	$f(\vec{x}) = (\cos(1) + 2\cos(x_1+2) + 3\cos(2x_1+3) + 4\cos(3x_1+4) + 5\cos(4x_1+5))(\cos 2x_2 + 1) + 2\cos(3x_2+2) + 3\cos(4x_2+3) + 4\cos 5x_2 + 4) + 5\cos(6x_2+5))$	$\overrightarrow{x^*} = (-1.30671, 4.85806)$	$f_{min} = -176.5418$

Para a realização dos testes, assim como realizado na implementação do algoritmo RACO, foram utilizados cinco cenários equivalentes a cinco domínios que não possuem a solução ideal e que possuem características bem distintas. São eles:

- Dois domínios positivos e com valores grandes: $x_{1min} = 100$, $x_{1max} = 200$, $x_{2min} = 50$ e $x_{2max} = 80$;
- Dois domínios negativos e com valores grandes: $x_{1min} = -300$, $x_{1max} = -180$, $x_{2min} = -600$ e $x_{2max} = -50$;
- Um domínio positivo e um negativo, com valores longe da solução ideal: $x_{1min} = 1800$, $x_{1max} = 1900$, $x_{2min} = -230$ e $x_{2max} = -110$;
- Um domínio positivo e um negativo, com intervalos de valores extremamente estreitos: $x_{1min} = 1$, $x_{1max} = 2$, $x_{2min} = -3$ e $x_{2max} = -1$;
- Um domínio positivo com valor estreito, e um domínio negativo: $x_{1min} = 100$, $x_{1max} = 110$, $x_{2min} = -300$ e $x_{2max} = -190$;

A Tabela 10 mostra a comparação realizada entre os resultados encontrados por ARACO e RACO, utilizando como critérios, a média da quantidade de avaliações de função realizadas até que a condição de parada seja alcançada e a taxa de sucesso dos algoritmos. Estes critérios de comparação foram escolhidos por serem os mesmos critérios utilizados pelo algoritmo RACO, que foi utilizado para comparação nessa seção. É importante destacar que, para alguns dos domínios fornecidos, RACO não consegue encontrar a solução ótima ou uma solução próxima da ótima para algumas funções de *benchmark* propostas. Estes casos estão marcados com – na Tabela 10, simbolizando que RACO não conseguiu encontrar a solução ótima ou próxima da ótima em nenhuma das vinte execuções realizadas para aquela função, com aquele domínio fornecido. Essa situação ocorre em um domínio proposto para a função Shubert, três domínios propostos para a função Ackley, dois domínios propostos para a função Levy e um domínio proposto para a função Six-Hump Camel-Back. Esta situação não ocorre com ARACO em nenhuma função de *benchmark*, em nenhum domínio proposto. Este é um fato que representa uma das grandes vantagens observadas no algoritmo, ele consegue encontrar a solução ótima ou uma solução próxima da ótima em todos os cenários propostos para todas as funções de *benchmark* testadas.

Além disso, a Tabela 10 também mostra que ARACO possui taxa de sucesso igual ou superior a RACO em todas as funções de *benchmark* testadas, para todos os domínios fornecidos. A taxa de sucesso alcançada por ARACO é de 100% para todas as funções, exceto em um cenário proposto para apenas uma função. Enquanto RACO, além de não conseguir encontrar a solução ótima ou próxima da ótima em alguns dos cenários propostos, possui valores de taxa de sucesso bem mais modestos em diversos cenários, que serão discutidos detalhadamente posteriormente. Resumidamente, os dados mostrados na Tabela 10 provam que ARACO tem potencial para ser testado e posteriormente aplicado com sucesso em problemas do mundo real.

Tabela 10 – Comparação entre ARACO e RACO no cenário 5.2

	x_1 = (100, 200) x_2 = (50, 80)		x_1 = (-300, 180) x_2 = (-600, -50)		x_1 = (1800, 1900) x_2 = (-230, 110)		x_1 = (1, 2) x_2 = (-3, -1)		x_1 = (100, 110) x_2 = (-300, -190)		Funções
	ARACO	RACO	ARACO	RACO	ARACO	RACO	ARACO	RACO	ARACO	RACO	
ANFE	**380**	383	**161**	370	2057	**1559**	**35**	111	415	**292**	Goldstein and
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	Price
ANFE	**54**	155	**53**	171	**72**	187	**38**	108	**57**	163	Zakharov
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	
ANFE	**59**	155	**64**	169	**72**	184	**44**	125	**60**	164	Martin and
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	Gaddy
ANFE	239	**188**	368	**220**	**214**	217	**37**	108	300	**249**	Griewangk
T. Suces.	**100%**	70%	**100%**	80%	**100%**	40%	100%	100%	**100%**	25%	
ANFE	**94**	156	**79**	189	**79**	190	**63**	111	**82**	159	Rastrigin
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	
ANFE	**123**	151	**133**	282	**124**	284	**54**	-	**96**	266	Shubert
T. Suces.	100%	100%	100%	100%	100%	100%	**100%**	0%	100%	100%	
ANFE	**121**	171	**119**	-	**392**	-	**59**	108	**166**	-	Ackley
T. Suces.	100%	100%	**100%**	0%	**85%**	0%	100%	100%	**100%**	0%	
ANFE	**48**	157	**52**	170	**61**	182	**37**	108	**52**	156	B2
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	
ANFE	**52**	157	**52**	170	385	**261**	**37**	-	**196**	-	Levy
T. Suces.	100%	100%	100%	100%	100%	100%	**100%**	0%	**100%**	0%	
ANFE	**68**	178	**67**	178	**104**	190	**46**	118	**77**	167	Beale
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	
ANFE	352	**253**	257	**242**	**794**	802	**124**	163	709	**423**	Rosenbrock
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	
ANFE	**39**	153	**35**	169	**58**	182	**25**	119	**34**	168	Bohachevesky
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	
ANFE	**183**	205	**225**	261	**197**	260	**57**	113	**158**	188	Hansen
T. Suces.	100%	100%	100%	100%	100%	100%	100%	100%	**100%**	95%	
ANFE	**30**	166	**37**	174	**43**	177	**45**	-	**36**	176	Six-Hump
T. Suces.	100%	100%	100%	100%	100%	100%	**100%**	0%	100%	100%	Camel-Back

Analisando detalhadamente o primeiro cenário, se a comparação for realizada de acordo com ANFE, ARACO supera RACO em 12 das 14 funções propostas, chegando a apresentar resultado 82% melhor do que o fornecido por RACO, na função Six-Hump Camel-Back. Para a função Griewangk, que é uma das duas únicas em que RACO possui desempenho superior, é importante destacar que, a taxa de sucesso de RACO é de 70%, enquanto ARACO possui 100% de taxa de sucesso nos testes realizados, o que evidencia a vantagem de ARACO em encontrar a solução ótima ou uma solução próxima da ótima, em todas as funções, para todos os domínios fornecidos, em praticamente todas as execuções realizadas, fato que conforme dito anteriormente, representa grande vantagem do algoritmo.

O Gráfico 4 mostra uma comparação entre os dois algoritmos, em relação a ANFE, para as funções de *benchmark* da Tabela 9.

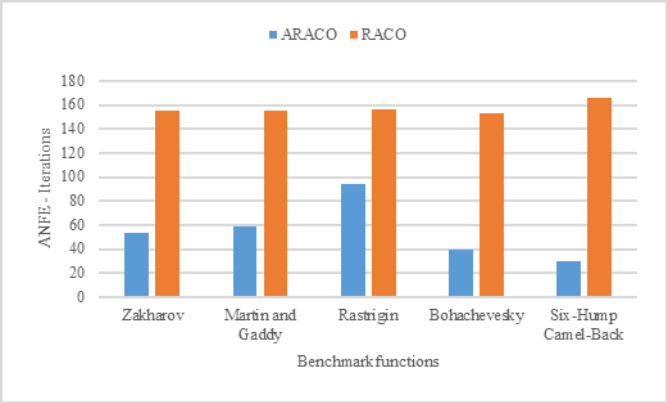


Gráfico 4 – Comparação dos resultados de ARACO e RACO no cenário 5.2

O segundo cenário proposto apresenta resultados similares em relação a ANFE, sendo que ARACO supera RACO em 12 das 14 funções propostas, chegando a apresentar resultado 79% melhor do que o fornecido por RACO, na função Bohachevsky. Para a função Griewangk, em que RACO possui desempenho superior, a taxa de sucesso de RACO é de 80%, enquanto ARACO possui 100% de taxa de sucesso nos testes realizados. Para a função Ackley, RACO não consegue encontrar a solução ótima ou próxima da ótima em nenhuma execução realizada, enquanto ARACO consegue encontrar em todas as execuções.

No terceiro cenário, ARACO volta a superar RACO em 12 das 14 funções propostas em relação a ANFE, chegando a apresentar resultado 75% melhor do que o fornecido por RACO, na função Six-Hump Camel-Back. Mais uma vez, é importante destacar que RACO consegue apenas 40% de taxa de sucesso na função Griewangk, que se mostrou uma função de difícil otimização em todo este trabalho, enquanto ARACO possui desempenho superior em relação a ANFE, além de possuir 100% de taxa de sucesso nos testes realizados. RACO não consegue encontrar a solução ótima ou próxima da ótima para função Ackley, enquanto ARACO consegue encontrar em 85% das execuções.

O quarto cenário é onde a diferença se torna mais significativa, pois ARACO consegue desempenho superior em todas as 14 funções propostas em relação a ANFE, chegando a apresentar resultado 79% melhor do que o fornecido por RACO, na função Bohachevesky. Além disso, RACO não consegue encontrar a solução ótima ou próxima da ótima em nenhuma das execuções em três funções: Six-Hump Camel-Back, Shubert e Levy. Enquanto ARACO consegue 100% de taxa de sucesso, em todas as funções propostas, nos testes realizados.

Por último, no quinto cenário, ARACO supera RACO em 11 das 14 funções propostas em relação a ANFE, chegando a apresentar resultado 81% melhor do que o fornecido por RACO, na função Six-Hump Camel-Back. Para a função Griewangk, em que RACO possui desempenho superior, pode-se destacar que a taxa de sucesso de RACO é de apenas 25%, enquanto ARACO possui 100% de taxa de sucesso nos testes realizados. Nas funções Ackley e Levy, RACO não consegue encontrar a solução ótima ou próxima da ótima em nenhuma das execuções, enquanto ARACO consegue 100% de taxa de sucesso nos testes realizados.

Se a comparação entre os algoritmos se resumir ao critério ANFE, os testes mostram que ARACO tem desempenho superior, ou seja, menor valor de ANFE, em 87% dos testes realizados em cenários onde o domínio inicial fornecido não contém a solução ideal.

Em alguns dos testes em que ARACO não tem desempenho superior a RACO, RACO não consegue encontrar a solução ótima ou próxima da ótima em todas as execuções, enquanto ARACO consegue encontrar em mais execuções nestes cenários, obtendo quase sempre 100% de taxa de sucesso nos testes realizados. Além disso, RACO não consegue encontrar a solução ótima ou próxima da ótima em nenhuma execução em sete cenários propostos, enquanto ARACO consegue encontrar em todos estes cenários.

Todas as comparações realizadas evidenciam a superioridade de ARACO, tanto em relação a encontrar a solução ótima ou uma solução próxima da ótima em uma menor média de quantidade de avaliações de função, quanto para conseguir alcançar maior taxa de sucesso e para encontrar a solução ótima ou uma solução próxima da ótima em todos os cenários propostos.

Melhores valores de ANFE são obtidos graças a aceleração dos parâmetros de ajuste de domínio em momentos oportunos, permitindo que um domínio onde a solução ótima esteja presente seja encontrado mais rapidamente e, quando este domínio for encontrado, o processo de aceleração dos parâmetros de ajuste permita que o valor ótimo ou próximo do ótimo seja encontrado mais rapidamente. Enquanto, a maior taxa de sucesso e a possibilidade de encontrar a solução ótima em todos os cenários, são vantagens obtidas graças a estratégia de tratamento da estagnação do domínio, que permite ampliar o domínio próximo a melhor solução encontrada, permitindo que o algoritmo possa buscar o valor ótimo, fora de uma região de mínimo local.

4.3 – Testes com as funções de *benchmark* CEC 2019

As funções de teste de *benchmark* CEC 2019 *100-Digit Challenge* [33] são um grupo de funções difíceis de otimizar, conhecidas como "O desafio de 100 dígitos" e que foram usadas em uma competição anual de otimização em 2019. As funções são apresentadas na Tabela 11. A coluna Função mostra o nome da função do *benchmark*. A coluna Fórmula mostra a fórmula usada para calcular o valor da função que será minimizada. A coluna Mínimo mostra o valor mínimo da função, quando os valores ótimos para cada uma das variáveis são encontrados.

As metaheurísticas utilizadas para comparação neste cenário são algoritmos baseados em PSO, sendo eles: Otimizador Dependente de Fitness (FDO – *Fitness Dependent Optimizer*) [33], Algoritmo da Libélula (DA – *Dragonfly Algorithm*) [34], Algoritmo de Otimização de Baleia (WOA – *Whale Optimization Algorithm*) [35] e Algoritmo de Enxame de Salpas (SSA – *Salp Swarm Algorithm*) [36].

Tabela 11 – Funções de *benchmark* CEC 2019 – *100-Digit Challenge*

Funcão	Fórmula	Mínimo f(x*)
Storn's Chebyshev Polynomial Fitting Problem	$f(\vec{x}) = p_1 + p_2 + p_3$ $p_1 = \begin{cases} (u-d)^2 & ,if\ u < d \\ 0, & ,otherwise \end{cases}$ $u = \sum_{j=1}^{D} x_j (1.2)^{D-j}$ $p_2 = \begin{cases} (v-d)^2 & ,if\ v < d \\ 0, & ,otherwise \end{cases}$ $v = \sum_{j=1}^{D} x_j (-1.2)^{D-j}$ $pk = \begin{cases} (w_k - 1)^2 & ,if\ w_k > 1 \\ (w_k + 1)^2 & ,if\ w_k < 1 \\ 0, & ,otherwise \end{cases}$ $w_k = \sum_{j=1}^{D} x_j \left(\frac{2k}{m} - 1\right)^{D-j}$ $p_3 = \sum_{m=0}^{m} p_k, \quad k = 0, 1, \dots, m, \quad m = 32D$ $d = 72.661\ for\ D = 9$	$f_{min} = 1$
Inverse Hilbert Matrix Problem	$f(\vec{x}) = \sum_{i=1}^{n} \sum_{k=1}^{n} \lvert w_{i,k} \rvert$ $(w_{i,k}) = W = HZ - I, \quad I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$ $H = (h_{i,k}), \quad h_{i,k} = \frac{1}{i+k-1}, \quad i,k = 1, 2, \dots, n, \quad n = \sqrt{D}$ $Z = (z_{i,k}), \quad z_{i,k} = x_{i+n(k-1)}$	$f_{min} = -186.7309$
Lennard-Jones Minimum Energy Cluster	$f(\vec{x}) = 12.7120622568 + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{1}{d_{i,j}^2} - \frac{2}{d_{i,j}}\right)$ $d_{i,j} = \left(\sum_{k=0}^{2} \left(x_{3i+k-2} - x_{3j+k-2}\right)^2\right)^3 \quad , n = \frac{D}{3}$	$f_{min} = 1$
Rastrigin's Function	$f(\vec{x}) = \sum_{i=1}^{D} (x_i^2 - 10\cos(2\pi x_i) + 10)$	$f_{min} = 1$
Griewangk's Function	$f(\vec{x}) = \sum_{i=1}^{D} \frac{{x_i}^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$f_{min} = 1$
Weierstrass Function	$f(\vec{x}) = \sum_{i=1}^{D} \left(\sum_{k=0}^{k_{max}} \left[a^k \cos\left(2\pi b^k (x_i + 0.5)\right)\right] \right) - D \sum_{k=0}^{k_{max}} a^k \cos(\pi b^k)$ $a = 0.5, \quad b = 3, \quad k_{max} = 20$	$f_{min} = 1$
Modified Schwefel's Function	$f(\vec{x}) = 418.9829D - \sum_{i=1}^{D} g(z_i)$ $z_i = x_i + 420.9687462275036$	$f_{min} = 1$

	$g(z_i) = \begin{cases} z_i \sin\left(\lvert z_i \rvert^{\frac{1}{2}}\right) & if\ \lvert z_i \rvert \le 500 \\ \left(500 - mod(z_i, 500)\right) \sin\left(\sqrt{\lvert 500 - mod(z_i, 500) \rvert}\right) - \frac{(z_i - 500)^2}{10000D} & if\ z_i > 500 \\ \left(mod(z_i, 500) - 500\right) \sin\left(\sqrt{\lvert mod(z_i, 500) - 500 \rvert}\right) - \frac{(z_i + 500)^2}{10000D} & if\ z_i < -500 \end{cases}$	
Expanded Schaffer's F6 Function	$g(x, y) = 0.5 + \frac{\sin^2\left(\sqrt{x^2 + y^2}\right) - 0.5}{(1 + 0.001(x^2 + y^2))^2}$ $f(\vec{x}) = g(x_1, x_2) + g(x_2, x_3) \ldots + g(x_{D-1}, x_D) + g(x_D, x_1)$	$f_{min} = 1$
Happy Cat Function	$f(\vec{x}) = \left\lvert \sum_{i=1}^{D} x_i^2 - D \right\rvert^{1/4} + \left(0.5 \sum_{i=1}^{D} x_i^2 + \sum_{i=1}^{D} x_i\right)/D + 0.5$	$f_{min} = 1$
Ackley Function	$f(\vec{x}) = -20\exp(-0.2 \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i{}^2}) - \exp(\frac{1}{n} \sum_{i=1}^{n} \cos(2\pi x_i))$	$f_{min} = 1$

Para facilitar a avaliação, todas as implementações das funções de *benchmark* CEC 2019 *100-Digit Challenge*, utilizadas na competição em 2019 e também no ARACO e nos demais algoritmos, foram adaptadas para que os valores ótimos buscados fossem 1. Todos as funções de *benchmark* têm 10 dimensões e o domínio fornecido aos algoritmos é x^i_{min} = -100 and x^i_{max} = 100, para cada dimensão i, exceto para as funções Storn's Chebyshev Polynomial Fitting Problem, Inverse Hilbert Matrix Problem e Lennard-Jones Minimum Energy Cluster, que possuem diferentes dimensões e diferentes domínios iniciais, conforme mostrado na Tabela 11.

Os resultados usados para comparar FDO, DA, WOA e SSA foram obtidos de Abdullah e Ahmed [33]. Todos os algoritmos foram executados 30 vezes para cada função de *benchmark* e o critério de comparação usado nesta seção é a média do valor mínimo encontrado pelos algoritmos, após 500 avaliações de função terem sido executadas. Este critério foi escolhido em ARACO por ser o critério utilizado pelos outros algoritmos utilizados para comparação nessa seção.

A Tabela 12 mostra a comparação entre as médias dos valores mínimos encontrados pelos algoritmos após 30 execuções, com 500 avaliações de função a cada execução. É importante destacar que os resultados obtidos pelo ARACO são superiores aos encontrados pelos outros algoritmos em seis das dez funções de *benchmark* testadas. O algoritmo está em segundo lugar em três outras funções, e em terceiro lugar apenas na função de Weierstrass.

Tabela 12 – Comparação entre ARACO e outras metaheurísticas no cenário 5.3

Função	ARACO	FDO	DA	WOA	SSA
Storn's Chebyshev Polynomial Fitting Problem $\vec{x}$:[-8192, 8192]^n, n=9	5.60E9	**4585.27**	5.43E10	4.11E10	6.05E9
Inverse Hilbert Matrix Problem $\vec{x}$:[-16384, 16384]^n, n=16	**1.9243**	4	78.0368	17.3495	18.3434
Lennard-Jones Minimum Energy Cluster $\vec{x}$:[-4, 4]^n, n=18	**12.6736**	13.7024	13.7026	13.7024	13.7025
Rastrigin's Function $\vec{x}$:[-100, 100]^n, n=10	**5.6495**	34.0837	344.3561	394.6754	41.6936
Griewangk's Function $\vec{x}$:[-100, 100]^n, n=10	**1.0635**	2.1392	2.5572	2.7342	2.2084
Weierstrass Function $\vec{x}$:[-100, 100]^n, n=10	10.5288	12.1332	9.8955	10.7085	**6.0798**
Modified Schwefel's Function $\vec{x}$:[-100, 100]^n, n=10	**1.0005**	120.4858	578.9531	490.6843	410.3964
Expanded Schaffer's F6 Function $\vec{x}$:[-100, 100]^n, n=10	**1.8266**	6.1021	6.8734	6.909	6.37
Happy Cat Function $\vec{x}$:[-100, 100]^n, n=10	3.0513	**2**	6.0467	5.9371	3.6723
Ackley Function $\vec{x}$:[-100, 100]^n, n=10	12.9822	**2.7182**	21.2604	21.2761	21.04

Analisando os resultados encontrados, observa-se que os resultados de ARACO são competitivos. Além disso, ARACO é o único algoritmo que apresenta resultados que se aproximam do valor ótimo com apenas 500 avaliações de função percorridas, conseguindo encontrar o número inteiro do valor ótimo em 4 das funções testadas.

O Gráfico 5 mostra uma comparação entre os resultados encontrados pelos cinco algoritmos contemplados neste cenário, em relação à média do valor mínimo encontrado para algumas das funções de *benchmark* na Tabela 12, após a execução de 500 avaliações de função.

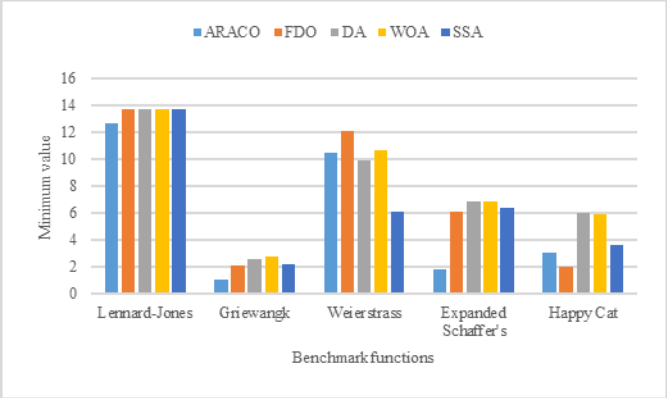


Gráfico 5 – Comparação dos resultados de ARACO e das outras metaheurísticas no cenário 5.3

No entanto, é importante destacar que o ARACO atinge resultados ainda melhores para as funções de *benchmark* propostas, após a realização de 500 avaliações de função. No entanto, para que uma comparação justa seja feita, os mesmos critérios e condição de parada definidos pelos outros algoritmos foram mantidos. Apesar disso, para orientar comparações em estudos futuros, os resultados alcançados pelo ARACO após a execução de 5000 avaliações de função, são apresentados a seguir. Nota-se que com essa condição de parada, o algoritmo atinge resultados superiores aos obtidos com 500 avaliações de função. Isso prova que, à medida que as iterações passam, as formigas são capazes de encontrar soluções ainda melhores. A Tabela 13 mostra a média do valor mínimo alcançado pelo ARACO após 30 execuções, com 5000 avaliações de função em cada execução.

Tabela 13 – Resultados de ARACO após 5000 avaliações de função

Funcão	ARACO
Storn's Chebyshev Polynomial Fitting Problem $\vec{x}$:[-8192, 8192]^n, n=9	5.23E6
Inverse Hilbert Matrix Problem $\vec{x}$:[-16384, 16384]^n, n=16	1.3992
Lennard-Jones Minimum Energy Cluster $\vec{x}$:[-4, 4]^n, n=18	8.3911
Rastrigin's Function $\vec{x}$:[-100, 100]^n, n=10	3.3252
Griewangk's Function $\vec{x}$:[-100, 100]^n, n=10	1.0085
Weierstrass Function $\vec{x}$:[-100, 100]^n, n=10	2.2961
Modified Schwefel's Function $\vec{x}$:[-100, 100]^n, n=10	1.0001
Expanded Schaffer's F6 Function $\vec{x}$:[-100, 100]^n, n=10	1.2062
Happy Cat Function $\vec{x}$:[-100, 100]^n, n=10	2.4945
Ackley Function $\vec{x}$:[-100, 100]^n, n=10	10.8957

5 – DISCUSSÃO

Após a apresentação dos resultados realizada no tópico anterior, essa seção tem como objetivo discutir e resumir os resultados encontrados em todas as funções de *benchmark* testadas em todos os cenários testados, mostrando quais foram os ganhos obtidos através da utilização do algoritmo ARACO em comparação com outros algoritmos, em relação ao desempenho (quantidade de avaliações de função realizadas) e confiabilidade (taxa de sucesso alcançada).

5.1 – Desempenho

A Tabela 14 mostra que nos cinco cenários testados durante este trabalho, o algoritmo ARACO apresenta resultado superior aos demais algoritmos em relação ao desempenho. É importante relembrar que o critério adotado para essa comparação é a quantidade de avaliações de função realizadas até o algoritmo encontrar a solução ótima ou uma solução próxima da ótima. Este critério foi adotado por ser o mesmo critério utilizado em todos os algoritmos citados no trabalho e, por isso, também foi utilizado no ARACO.

Tabela 14 – Resumo dos resultados obtidos pelos algoritmos em relação ao desempenho

	ARACO	RACO	Outro Algoritmo
Cenário onde o domínio inicial contém a solução ótima – Comparação com métodos de aprendizado de probabilidade que modelam e amostram distribuições de probabilidade	5	0	0
Cenário onde o domínio inicial contém a solução ótima – Comparação com metaheurísticas desenvolvidas para otimização combinatória e adaptadas a domínios contínuos	9	6	0
Cenário onde o domínio inicial contém a solução ótima – Comparação com métodos inspirados no comportamento das formigas	5	3	0
Cenário onde o domínio inicial não contém a solução ótima	61	9	0
Funções de *benchmark* CEC-2019	6	-	4
Total:	86	18	4

O trabalho desenvolvido dividiu as funções de *benchmark* testadas em três cenários: quando o domínio inicial contém a solução ótima, quando o domínio inicial não contém a solução ótima e os testes realizados com as funções de *benchmark* CEC-2019.

O primeiro cenário, referente aos itens 4.1.1, 4.1.2 e 4.1.3 deste trabalho está representado na segunda, na terceira e na quarta linha da Tabela 14, mostra que ARACO apresentou resultados superiores em 19 das 28 funções testadas, ou seja, em 67,8% dos testes realizados.

O segundo cenário, referente ao item 4.2 deste trabalho está representado na quinta linha da Tabela 14, mostra que ARACO apresentou resultados superiores em 61 das 70 funções testadas, valor que representa 81,7% dos testes realizados.

Por último, o terceiro cenário, referente ao item 4.3 deste trabalho está representado na sexta linha da Tabela 14, mostra que ARACO apresentou resultados superiores em 6 das 10 funções testadas, ou seja, em 60% dos testes realizados.

A Tabela 14 considera um resultado superior, quando o algoritmo encontra o valor ótimo ou próximo do ótimo em menos avaliações de função do que os demais algoritmos.

Se todos os dados forem agrupados, pode-se observar que foram executados 108 diferentes testes, com diversas funções de *benchmark*, com variados domínios iniciais fornecidos. Deste total, ARACO conseguiu resultados superiores em 86, ou seja, em 79,6% dos testes realizados, em relação a quantidade de avaliações de função necessárias até que o algoritmo encontre a solução ótima ou uma solução próxima da ótima. Estes números comprovam que o algoritmo desenvolvido possui resultados competitivos em relação ao desempenho.

5.2 – Confiabilidade

Apenas três cenários descritos neste trabalho apresentam informações referentes a taxa de sucesso dos algoritmos. É importante relembrar que a taxa de sucesso é utilizada para representar o percentual das execuções de um algoritmo que consegue encontrar a solução ótima ou uma solução próxima da ótima, sendo assim o critério utilizado para mensurar a confiabilidade dos algoritmos. A Tabela 15 mostra que nos três cenários possíveis de comparação, ARACO apresenta resultado superior aos demais algoritmos em relação a confiabilidade.

Tabela 15 – Resumo dos resultados obtidos pelos algoritmos em relação a confiabilidade

	Superior	Igual	Inferior
Cenário onde o domínio inicial contém a solução ótima – Comparação com metaheurísticas desenvolvidas para otimização combinatória e adaptadas a domínios contínuos	6	9	0
Cenário onde o domínio inicial contém a solução ótima – Comparação com métodos inspirados no comportamento das formigas	2	6	0
Cenário onde o domínio inicial não contém a solução ótima	12	58	0
Total:	20	73	0

O primeiro cenário, referente ao item 4.1.2 deste trabalho, está representado na segunda linha da Tabela 15. Nele, ARACO consegue taxa de sucesso superior ao algoritmo RACO, em 6 das 15 funções testadas. Nas outras 9 funções, a taxa de sucesso foi igual, com valor de 100%. É importante destacar que em 5 das 6 funções que ARACO obteve desempenho inferior em relação a RACO neste cenário, conforme mostrado no item 5.1, o algoritmo consegue taxa de sucesso superior a RACO, comprovando assim o objetivo do algoritmo de conseguir alto desempenho, privilegiando manter alta confiabilidade.

O segundo cenário, referente ao item 4.1.3 deste trabalho, está representado na terceira linha da Tabela 15. Nele, ARACO consegue taxa de sucesso superior ao algoritmo RACO, em 2 das 8 funções testadas. Nas outras 6 funções, a taxa de sucesso foi igual, com valor de 100%. É importante destacar

que em 2 das 3 funções que ARACO obteve desempenho inferior em relação a RACO neste cenário, conforme mostrado no item 5.1, o algoritmo consegue taxa de sucesso superior a RACO

O terceiro cenário, referente ao item 4.2 deste trabalho, está representado na quarta linha da Tabela 15. Nele, ARACO consegue taxa de sucesso superior ao algoritmo RACO, em 12 das 70 funções testadas. Nas outras 58 funções, a taxa de sucesso foi igual, com valor de 100%. É importante destacar que em 3 das 9 funções que ARACO obteve desempenho inferior em relação a RACO neste cenário, conforme mostrado no item 5.1, o algoritmo consegue taxa de sucesso superior a RACO. Além disso, em 7 funções propostas RACO consegue taxa de sucesso de 0%, ou seja, não consegue alcançar o valor ótimo ou próximo do ótimo em nenhuma das execuções tentadas, fato que não acontece com nenhuma função de *benchmark* em ARACO. Outra observação é que ARACO só não consegue chegar a 100% de taxa de sucesso em 1 das 70 funções testadas neste cenário, enquanto RACO não consegue chegar a 100% em 12 funções.

Se todos os dados forem agrupados, pode-se observar que foram executados 93 diferentes testes, com diversas funções de *benchmark*, com variados domínios iniciais fornecidos. Deste total, ARACO conseguiu resultados superiores aos alcançados por RACO em 20, em relação ao percentual das execuções de um algoritmo que consegue encontrar a solução ótima ou uma solução próxima da ótima. Nos outros 73 testes realizados, a taxa de sucesso dos dois algoritmos foi igual, no caso, 100%. Em nenhum dos cenários testados, com nenhum domínio fornecido, em nenhuma das funções de *benchmark* fornecidas aos algoritmos, ARACO teve taxa de sucesso inferior a RACO. Estes números comprovam que o algoritmo desenvolvido possui ótimos resultados em relação a confiabilidade.

6 – CONCLUSÃO

O ARACO foi proposto com o objetivo de contribuir com a melhoria dos resultados alcançados pelo ACO em otimização de funções contínuas, com o potencial de permitir resolver problemas de otimização do mundo real, possibilitando encontrar a solução ótima ou uma solução próxima da ótima, independentemente do domínio inicial conter a solução ótima ou não. Para isso, foram propostas algumas melhorias no algoritmo RACO. A primeira delas permite, em momentos oportunos, realizar a aceleração dos parâmetros responsáveis por localizar o domínio correto e, quando encontrado, realizar a sua redução. O objetivo dessa ação é otimizar o desempenho do algoritmo para que ele possa encontrar a solução ótima ou uma solução próxima da ótima mais rapidamente. O objetivo da segunda estratégia é permitir ao algoritmo sair de regiões de mínimo local, quando os resultados do algoritmo se mostrarem estagnados, proporcionando ao ARACO realizar o tratamento dessa estagnação e encontrar a solução ótima ou próxima da ótima em praticamente todas as execuções, independente do domínio fornecido.

Foram realizados testes para comprovar a eficácia das estratégias propostas, em relação a operação em domínios fornecidos que contém a solução ótima. Neste caso, ARACO fornece resultados promissores, conseguindo encontrar o valor ótimo ou um valor próximo do ótimo utilizando uma menor quantidade de avaliações de função, na maioria das funções testadas, nos três grupos de comparação propostos: métodos de aprendizado de probabilidade que modelam e amostram distribuições de probabilidade; metaheurísticas desenvolvidas para otimização combinatória e adaptadas a domínios contínuos; e métodos inspirados no comportamento das formigas. Além disso, a taxa de sucesso alcançada por ARACO foi igual ou superior a encontrada por seus concorrentes em todos os cenários testados.

A obtenção de bons resultados na operação em domínios fornecidos que não contém a solução ótima também é muito desejada, porque elimina a necessidade de se determinar corretamente o domínio inicial fornecido para o algoritmo. Nestes casos, ARACO demonstra desempenho muito eficaz, pois além de conseguir encontrar o valor ótimo ou um valor próximo do ótimo em menos avaliações de função do que o seu antecessor RACO em 87% dos cenários testados, ARACO consegue encontrar o valor ótimo ou próximo do ótimo em todos os cenários propostos, fato que não acontece com RACO, sendo também capaz de alcançar taxa de sucesso de 100% em praticamente todas as funções testadas. Essas vantagens são obtidas graças a aceleração dos parâmetros autoadaptativos de ajuste de domínio em momentos oportunos, permitindo maior velocidade de convergência do algoritmo, e graças a estratégia de tratamento da estagnação do algoritmo, que realiza a ampliação do domínio em torno da melhor solução encontrada, quando o algoritmo entra em estagnação, permitindo que seja encontrada uma nova solução que esteja fora da região de mínimo local, fazendo com que o valor ótimo ou próximo do ótimo seja encontrado em praticamente todas as execuções.

No que se refere aos testes realizados com as funções de *benchmark* CEC 2019 – *100-Digit Challenge*, ARACO obteve excelentes resultados, apresentando desempenho superior aos demais algoritmos na maioria das funções de *benchmark* testadas, além de ser o único algoritmo a atingir os resultados mais próximos aos valores ideais em 500 avaliações de função. Além disso, os resultados podem ser melhorados ainda mais se mais avaliações de função forem executadas.

6.1 – Publicação

A contribuição bibliográfica produzida durante a realização deste trabalho está disponível através da seguinte publicação.

J. G. Freitas, K. Yamanaka. "**An Accelerated and Robust Algorithm for Ant Colony Optimization in Continuous Functions**," *Journal of the Brazilian Computer Society*. *Received*: 12 *February* 2021. *Accepted*: 31 *August* 2021.

6.2 – Continuação do Trabalho

Como futuros trabalhos, o algoritmo pode ser expandido para permitir trabalhar com funções que possuem mais variáveis, conseguindo obter bom desempenho, sem o comprometimento da precisão e da taxa de sucesso já alcançados. Estes resultados já foram alcançados com sucesso em algumas das funções implementadas no trabalho, como as funções com 10 variáveis Sphere, Ellipsoid, Cigar e Tablet. Porém, em outras funções, como Griewangk (10 variáveis) e Hartmann (6 variáveis), o algoritmo atinge altas taxas de sucesso, mas perde desempenho, sendo necessário grande número de avaliações de função até encontrar o valor ideal. As funções de *benchmark* CEC 2019 *100-Digit Challenge* implementadas também demonstram a necessidade de determinar melhores estratégias para obter resultados superiores em funções que possuam mais dimensões.

Além disso, podem ser realizados estudos que permitam determinar melhores estratégias de detecção de estagnação do domínio e melhores taxas de ampliação do domínio em torno da melhor solução encontrada, de acordo com a natureza das funções, para melhorar ainda mais os resultados obtidos.

Como o algoritmo ARACO apresentou grande potencial em ser aplicado na resolução de problemas do mundo real, outro trabalho futuro sugerido é a aplicação prática do algoritmo nessas situações, que são muito mais complexas comparadas as funções de *benchmark* testadas durante este trabalho.

REFERÊNCIAS:

[1] A. Antoniou, W. S. Lu, "The optimization problem," In: A. Antoniou, W. S. Lu (eds.) *Practical Optimization*. pp. 1-26. Springer, Boston, 2007. https://doi.org/10.1007/978-0-387-71107-2_1

[2] P. S. Prampero, R. D. Attux, "Novas abordagens para otimização multimodal baseadas em enxames de partículas e clusterização," Ph.D. dissertation, Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, SP, Brazil, 2014.

[3] J. Edmonds, "How to Think About Algorithms," Cambridge University Press, New York, 2008. https://doi.org/10.1017/CBO9780511808241

[4] M. P. Saka, E. Dogan, I. Aydogdu, "Analysis of swarm intelligence-based algorithms for constrained optimization," In: X. S. Yang, Z. Cui, R. Xiao, A. H. Gandomi, M. Karamanoglu (eds) *Swarm Intelligence and Bio-inspired Compuation*. Elsevier, Oxford, UK, 2013. https://doi.org/10.1016/B978-0-12-405163-8.00002-8

[5] S. P. Kaur, "Variables in research," *Indian Journal of Research and Reports in Medical Sciences*, vol. 3, no. 4, pp, 36-38, 2013.

[6] Z. Wu, R. Xue, "A Cyclical Non-Linear Inertia-Weighted Teaching-Learning-Based Optimization Algorithm," *Algorithms,* vol. 12, no. 5, pp. 94, May, 2019. https://doi.org/10.3390/a12050094

[7] A. B. S. Serapião, "Fundamentos de otimização por inteligência de enxames: uma visão geral," *Sba Controle & Automação*, vol. 20, no. 3, pp. 271-304, Sep, 2009. https://doi.org/10.1590/S0103-17592009000300002

[8] D. E. Goldberg, "Generic Algorithm in search, optimization and machine learning," 1st ed. Reading, MA, USA: Addison-Wesley, 1989.

[9] R. Storn, K. Price, "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces," *Jornal of Global Optimization*, vol. 11, no. 4, pp. 341-359, Dec, 1997. https://doi.org/10.1023/A:1008202821328

[10] J. R. Koza, "Genetic programming: ont the programming of computers by means of natural selection," 1st ed. Cambridge, MA, USA: MIT Press, 1992.

[11] M. Dorigo, V. Maniezzo, A. Colorni, "Positive feedback as a search strategy," Dipartimento di Elettronica, Politecnico di Milano, Italy, Technical Report No. 91-016, 1991.

[12] M. Dorigo, V. Maniezzo, A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics),* vol. 26, no. 1, pp. 29-41, Feb, 1996. https://doi.org/10.1109/3477.484436

[13] M. Dorigo, L. M. Gambardella, "Ant Colony System: A cooperative learning approach to the traveling salesman problem," *IEEE Trasactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53-66, Apr, 1997. https://doi.org/10.1109/4235.585892

[14] D. Karaboga, “An idea based on honey bee swarm for numerical optimization, ” Computer Engineering Department, Engineering Faculty, Erciyes University, Kayseri, Turkey, Technical Report – TR06, 2005.

[15] J. Kennedy, R. Eberhart, “Particle swarm optimization,” *Proceedings of ICNN’95 – International Conference on Neural Networks*, Perth, WA, Australia, 1995, pp. 1942-1948, vol. 4. https://doi.org/10.1109/ICNN.1995.488968

[16] D. Sudholt, C. Thyssen, “Running time analysis of Ant Colony Optimization for shortest path problems,” *Journal of Discrete Algorithms*, vol. 10, pp. 165-180, 2012. https://doi.org/10.1016/j.jda.2011.06.002

[17] Q. L. Ding, X. P. Hu, L. J. Sun, Y. Z. Wang, “An improved ant colony optimization and its application to vehicle routing problem with time windows,” *Neurocomputing* vol. 98, pp. 101–107, 2012. https://doi.org/10.1016/j.neucom.2011.09.040

[18] C. Blum, M. Sampels, “An ant colony optimization algorithm for shop scheduling problems,” *Journal of Mathematical Modelling and Algorithms* vol. 3, pp. 285-304, 2004. https://doi.org/10.1023/B:JMMA.0000038614.39977.6f

[19] M. Dorigo, T. Stutzle, “Ant Colony Optimization: Overview and Recent Advances,” Handbook of Metaheuristics, International Series in Operations Research & Management Science, vol. 272, pp 311-351. https://doi.org/10.1007/978-3-319-91086-4_10

[20] K. Socha, M. Dorigo, “Ant colony optimization for continuous domains,” *European Journal of Operational Research*, vol. 185, issue 3, pp. 1155-1173, Mar, 2008. https://doi.org/10.1016/j.ejor.2006.06.046

[21] G. Bilchev, I. Parmee, “The Ant Colony Metaphor for Searching Continuous Design Spaces,” *Evolutionary Computing, Selected Papers from AISB Workshop, Sheffield*, vol. 993, pp. 25-39, Jan, 2006. https://doi.org/10.1007/3-540-60469-3_22

[22] H. Huang, Z. Hao, “ACO for Continuous Optimization Based on Discrete Encoding,” in *ANTS 2006 – Ant Colony Optimization and Swarm Intelligence. ANTS 2006*. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Sep, 2006, vol. 4150, pp. 504-505. https://doi.org/10.1007/11839088_53

[23] J. Dréo, P. Siarry, “Continuous interacting ant colony algorithm based on dense heterarchy," *Future Generation Computer Systems - FGCS*, vol. 20, pp. 841-856, Jun, 2004. https://doi.org/10.1016/j.future.2003.07.015

[24] N. Monmarché, G. Venturini, M. Slimane, “On how Pachycondyla apicalis ants suggest a new search algorithm,” *Future Generation Computer Systems – FGCS*, vol. 16, pp. 937-946, Jun, 2000. https://doi.org/10.1016/S0167-739X(00)00047-9

[25] Z. Chen, Z. Zhou, J. Luo, “A robust ant colony optimization for continuous functions,” *Expert Systems with Applications: An International Journal*, vol. 81, pp. 309-320, Mar, 2017. https://doi.org/10.1016/j.eswa.2017.03.036

[26] G. Leguizamón, C. A. C. Coello, “An alternative ACOR algorithm for continuous optimization problems” in *Proceedings of the 7th international conference on Swarm Intelligence. ANTS’10.* Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2010, vol. 6234, pp. 48-59. https://doi.org/10.1007/978-3-642-15461-4_5

[27] T. J. Liao, M. A. Montes da Oca, D. Aydin, T. Stutlze, M. Dorigo, "An incremental ant colony algorithm with local search for continuous optimization," in *Proceedings of the genetic and evolutionary computation conference – GECCO'11*. Association for Computing Machinery, New York, NY, USA, 2011, pp. 125-132. https://doi.org/10.1145/2001576.2001594

[28] T. J. Liao, T. Stutzle, M. A. Montes da Oca, M. Dorigo, "A unified ant colony optimization algorithm for continuous optimization," *European Journal of Operational Research*, vol. 234, pp. 597-609, May, 2014. https://doi.org/10.1016/j.ejor.2013.10.024

[29] Q. Yang, W. Chen, Z. Yu, T. Gu, Y. Li, H. Zhang, J. Zhang, "Adaptive Multimodal Continuous Ant Colony Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 21, no. 2, pp. 191-205, Apr, 2017. https://doi.org/10.1109/TEVC.2016.2591064

[30] L. Liu, Y. Dai, J. Gao, "Ant Colony Optimization Algorithm for Continuous Domains Based on Position Distribution Model of Ant Colony Foraging," *The Scientific World Journal*, vol. 2014, May, 2014. https://doi.org/10.1155/2014/428539

[31] J. M. Abdullah, T. Ahmed, "Fitness Dependent Optimizer: Inspired by the Bee Swarming Reproductive Process," *IEEE Access* vol. 7, pp. 43473-43486, 2019. https://doi.org/10.1109/ACCESS.2019.2907012

[32] S. Kern, S. D. Muller, N. Hansen, D. Buche, J. Ocenasek, P. Koumoutsakos, "Learning probability distributions in continuous evolutionary algorithms – A comparative review," *Natural Computing*, vol. 3, pp. 77-112, Mar, 2004. https://doi.org/10.1023/B:NACO.0000023416.59689.4e

[33] K. V. Price, N. H. Awad, M. Z. Ali, P. N. Suganthan, "The 100-digit challenge: Problem definitions and evaluation criteria for the 100-digit challenge special session and competition on single objective numerical optimization," Nanyang Technological University, Singapore, Technical Report, 2018.

[34] S. Mirjalili, "Dragonfly algorithm: A new meta-heuristic optimization technique for solving single-objective discrete and multi-objective problems," *Neural Comput. Appl.*, vol. 27, no. 4, pp. 1053-1073, 2015. https://doi.org/10.1007/s00521-015-1920-1

[35] S. Mirjalili, A. Lewis, "The whale optimization algorithm," *Advances in Engineering Software,* vol. 95, pp. 51-67, 2016. https://doi.org/10.1016/j.advengsoft.2016.01.008

[36] S. Mirjalili, A. H. Gandomi, S. Z. Mirjalili, S. Saremi, H. Faris, S. M. Mirjalili, "Salp swarm algorithm: A bio-inspired optimizer for engineering design problems," *Advances Engineering Software* vol. 114, pp. 163-191, 2017. https://doi.org/10.1016/j.advengsoft.2017.07.002