UNIVERSIDADE FEDERAL DO PARANÁ

JEAN CARLO BAENA VICENTE

ESTUDOS ACERCA DO ACO NA SOLUÇÃO DO TSP E UMA NOVA ABORDAGEM

CURITIBA

2021

JEAN CARLO BAENA VICENTE

ESTUDOS ACERCA DO ACO NA SOLUÇÃO DO TSP E UMA NOVA ABORDAGEM

Trabalho apresentado como requisito parcial para a obtenção do título de Mestre em Matemática Aplicada e Computacional pelo Programa de Pós Graduação Métodos Numéricos em Engenharia, Setor de Ciências Exatas e Setor de Tecnologia, da Universidade Federal do Paraná.

Orientador: Prof Paulo Henrique Siqueira, DSc

CURITIBA

2021

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

V632e Vicente, Jean Carlo Baena
Estudos acerca do ACO na solução do TSP e uma nova abordagem [recurso eletrônico] / Jean Carlo Baena Vicente. – Curitiba, 2021.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Métodos Numéricos em Engenharia, 2021.

Orientador: Paulo Henrique Siqueira.

1. Otimização matemática. 2. Programação heurística. I. Universidade Federal do Paraná. II. Siqueira, Paulo Henrique. III. Título.

CDD: 519.6

Bibliotecária: Vanusa Maciel CRB- 9/1928

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em MÉTODOS NUMÉRICOS EM ENGENHARIA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **JEAN CARLO BAENA VICENTE** intitulada: **Estudos acerca do ACO na solução do TSP e uma nova abordagem**, sob orientação do Prof. Dr. PAULO HENRIQUE SIQUEIRA, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 28 de Julho de 2021.

Assinatura Eletrônica
28/07/2021 14:08:48.0
PAULO HENRIQUE SIQUEIRA
Presidente da Banca Examinadora

Assinatura Eletrônica
29/07/2021 01:16:51.0
LUIZ CARLOS MATIOLI
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
03/08/2021 18:56:44.0
LUIZ CARLOS DE ABREU RODRIGUES
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

Para autenticar este documento/assinatura, acesse https://www.prppg.ufpr.br/siga/visitante/autenticacaoassinaturas.jsp e insira o codigo 103260

AGRADECIMENTOS

Ao professor Dr. Paulo Henrique Siqueira, pela orientação, disposição e paciência para com este trabalho.

Aos meus familiares, por sempre estarem ao meu lado, pelo apoio e incentivo.

Aos amigos de longa e curta data, com quem sei que posso partilhar momentos bons e ruins.

Ao Programa de Pós-Graduação em Métodos Numéricos em Engenharia (PPGMNE) da Universidade Federal do Paraná.

Agradeço a todos que direta ou indiretamente contribuíram para a conclusão deste trabalho.

RESUMO

A otimização é uma das áreas da matemática aplicada que possui maior visibilidade e relevância atualmente, principalmente dentro de grandes empresas, onde modelos matemáticos podem ser o diferencial que irá manter a empresa a frente da concorrência. Os modelos tornam-se cada vez mais complexos, fazendo com que métodos determinísticos sejam inadequados diante da demanda computacional necessária para encontrar uma solução, tendo em vista a tecnologia atual. Surgem então métodos que não prometem soluções perfeitas, mas boas o suficiente e que podem ser executados em tempo hábil. Por muito tempo as meta-heurísticas foram destaque na área da otimização, e por mais que outros métodos promissores estejam surgindo, alguns dos melhores resultados presentes na literatura foram obtidos por meta-heurísticas e suas adaptações. Neste trabalho é sugerida uma abordagem diferente para a otimização por colônia de formigas (ACO), que é uma meta-heurística com grande destaque na solução do problema do caixeiro viajante (TSP). No algoritmo original do ACO, as formigas constroem, uma a uma, circuitos dentro de um grafo que com o tempo serão melhorados pra gerar uma possível solução do TSP. A adaptação proposta toma o algoritmo do ACO e faz as devidas alterações para simular uma movimentação simultânea de todas as formigas, com a expectativa de que caminhos curtos recebam naturalmente maiores quantidades de feromônios. Neste trabalho são mostrados os testes realizados para averiguar quais os parâmetros ideais a serem utilizados e então comparar resultados com o algoritmo original para concluir quais as vantagens e desvantagens nesta adaptação.

Palavras-chaves: Meta-heurística; Sistema de colônia de formigas; Problema do caixeiro viajante.

ABSTRACT

Optimization is one of the fields in applied mathematics with most visibility and relevance nowadays, mainly inside great companies, where mathematical models could be the reason to keep the company ahead of the competition. As the models complexity increases, deterministic methods becomes worse at solving them, in view of the huge computational demand required to find a solution. Methods that doesn't ensure the optimal answer comes up, the solutions are close enough to the optimum and can be found in much faster times. For a long time meta heuristics have been in spotlight on optimization, other promising methods exists, but some of the best results in literature came from meta heuristics and its variations. In this work is suggested a new approach to the ant colony optimization algorithm (ACO), which is a benchmark meta heuristic on solving the traveling salesman problem (TSP). On the original ACO, one by one, the ants construct their circuits in a graph, these circuits will be enhanced as the time passes in order to obtain a solution for the TSP. The suggested approach adapts the ACO, making necessary alterations to simulate a simultaneous movimentation between the ants, it is expected that edges in shorter solutions to naturally have greater pheromones amounts accumulated. Tests were made to estimate the best parameter configuration to be used and the suggested algorithm results were compared to the original ACO verifying the advantages and disadvantages presents in this adaptation.

Key-words: Meta heuristic. Ant colony optimization. Traveling salesman problem.

SUMÁRIO

1 INTRODUÇÃO

Não é exagero dizer que a apresentação do método simplex foi um marco na história da matemática aplicada, apesar disso, o método não é a melhor opção para qualquer caso, ele exige muitos recursos computacionais para ser executado, o que torna o tempo de processamento demasiadamente grande, então provavelmente não será o método usado quando se espera uma solução imediata para um problema.

Por esse motivo, métodos determinísticos tem sido deixados de lado e abrem espaço para alternativas que apresentem soluções boas o suficiente em tempo hábil. Métodos heurísticos e meta-heurísticos surgem com esse intuito. Por terem objetivos comuns, frequentemente são confundidos, por outro lado, o prefixo “meta” em meta-heurística significa “além de” e supõe que estes métodos devem ser mais robustos. K. Sörensen e Fred W. Glover sugerem a definição para meta-heurísticas

> Uma estrutura algorítmica de alto nível que seja problema-independente e proporciona um sequência de orientações ou estratégias para desenvolver algoritmos heurísticos de otimização. Sörensen; Glover (2013)

Meta-heurísticas não são métodos direcionados para solução de um modelo de problema específico. Ainda assim, é necessária uma certa adaptação para a execução do algoritmo e os resultados obtidos podem ser melhores para problemas específicos.

Outro método que tem conquistado muita atenção é o math-heurístico, Boschetti *et al.* (2009) o define como “... uma interpolação entre meta-heurísticas e técnicas de programação matemática”. Novamente as meta-heurísticas aparecem para suprir os pontos fracos de métodos exatos, de forma rápida a parte heurística encontra uma boa solução inicial para o problema e, a partir desta, técnicas matemáticas buscam por soluções locais que aperfeiçoem os resultados obtidos.

1.1 OBJETIVOS

O problema do caixeiro viajante (*traveling salesman problem* - TSP) é um problema de teoria de grafos, que começou a ser estudado a partir do século 18. É o mais popular na área de pesquisa operacional, talvez por este motivo ele comumente seja considerado como primeira opção para testar novos métodos de resolução. Outro motivo deste problema ser tão atraente é a vasta gama de aplicações as quais está relacionado, praticamente qualquer problema relacionado a logística estará ligado ao problema do caixeiro viajante, roteamento de veículos, ordenação de coleta em armazéns, perfuração de placas de circuitos elétricos são alguns exemplos (Matai *et al.*, 2010).

1.1.1 Objetivo Geral

No decorrer do trabalho serão abordados modelos e métodos que se tornaram referências na resolução do problema do caixeiro viajante e, por este motivo, são utilizados nas últimas décadas como inspiração para novos métodos, ou mesmo incorporados em outros algoritmos para aprimorar os resultados obtidos.

Dentre os métodos apresentados ressalta-se o de otimização por colônia de formigas (*ant colony optimization - ACO*). Neste trabalho será realizado um estudo mais detalhado acerca de variações do ACO, que foram apresentadas em publicações nos últimos 10 anos. Depois será implementada uma alteração do algoritmo de otimização por colônia de formigas, que será testada ao resolver o problema do caixeiro viajante.

1.1.2 Objetivos Específicos

O objetivo principal deste trabalho, será implementar uma abordagem diferente do algoritmo de otimização por colônia de formigas, que surgiu ao observar o funcionamento do mesmo. Tanto o algoritmo original quanto esta nova abordagem serão implementadas utilizando as mesmas linguagens, estruturas e recursos computacionais. Assim, a comparação dos algoritmos indicará se esta é uma ideia promissora para ser explorada e adaptada para outros problemas que não sejam o TSP.

1.1.3 Limitações do Trabalho

Acredita-se que a abordagem proposta neste trabalho terá maior afinidade ao resolver o problema do caixeiro viajante, por isso toda a revisão literária realizada, assim como os testes efetuados, giram em torno deste problema. Com as devidas adaptações, tanto o ACO quanto o algoritmo proposto ao fim deste trabalho, podem resolver outros tipos de problemas, mas este não é o objetivo principal.

Durante a seção de métodos, são apresentadas diversas heurísticas utilizadas na resolução do TSP, estas mesmas serão utilizadas por alguns trabalhos mencionados durante a revisão da literatura. Alguns algoritmos, que também são utilizados durante a revisão, não são descritos neste trabalho, pois estes métodos não estão diretamente relacionados a solução do TSP, ainda assim foram referenciados trabalhos que descrevem mais detalhadamente o funcionamento deles.

Durante os testes realizados, com objetivo de averiguar os melhores parâmetros, não foram realizadas todas as combinações possíveis entre os parâmetros, isto resultaria em uma quantidade demasiadamente grande de testes, o que levaria a centenas de horas apenas para a realização dos testes, e também seria inviável apresentar neste trabalho todos os resultados obtidos.

Por fim, ao comparar o ACO com a adaptação proposta, não foram realizadas grandes alterações ao algoritmo original, além das essenciais para o funcionamento da adaptação. Outros métodos de melhoria poderiam ser aplicados durante o algoritmo, que certamente trariam resultados melhores do que os obtidos, mas também iriam mascarar a eficácia da adaptação ao obter as soluções.

1.1.4 Estrutura do Trabalho

Inicialmente será abordado o problema do caixeiro viajante, como este é formulado e como pode ser modelado para ser solucionado através de métodos determinísticos. Em seguida serão descritos alguns dos métodos heurísticos e meta-heurísticos mais comuns de serem utilizados na resolução do TSP. Uma revisão literária sistemática será realizada, afim de ressaltar quais são as alterações mais comuns ao tentar melhorar o algoritmo de otimização por colônia de formigas, na resolução do TSP. Por fim, uma abordagem diferente será proposta, descrita e testada, com o intuito de obter os parâmetros que levam aos melhores resultados, para então ser comparada com o algoritmo original.

2 MÉTODOS

Neste capítulo será descrito detalhadamente o problema do caixeiro viajante e os modelos matemáticos mais conhecidos na sua resolução. Também serão abordados diversos métodos heurísticos relacionados ao PCV, que serão futuramente utilizados em melhorias citadas neste trabalho, na revisão sistemática da literatura.

2.1 PROBLEMA DO CAIXEIRO VIAJANTE

Dado um grafo $\mathcal{G}(V, E)$, onde V é o conjunto de vértices presentes no grafo e E o conjunto de arestas conectando os vértices, chama-se de hamiltoniano um circuito que passe por cada vértice de V apenas uma vez, voltando ao vértice inicial no fim do percurso. Pode-se então definir que o TSP tem por objetivo encontrar o menor circuito hamiltoniano de um grafo $\mathcal{G}(V, E)$.

O problema ainda possui duas versões, a versão simétrica e a assimétrica, sendo a primeira um caso particular da segunda. O TSP simétrico considera que o custo aplicado a uma aresta que leva o vértice i até um vértice j deve ser o mesmo quando a aresta é tomada no sentido contrário, ou seja, do vértice j até o i, no caso assimétrico isto não é garantido. Uma prática comum para se referir ao problema assimétrico é utilizar um grafo definido por $\mathcal{G}(V, A)$, onde A é o conjunto de arcos direcionados do grafo, deixando claro que a orientação do percurso é relevante.

Com alguns conceitos de análise combinatória pode-se calcular a quantidade de elementos presentes na região factível do problema. Imaginando que, uma solução do problema pode ser descrita simplesmente pela ordem na qual os vértices são visitados e que uma mesma solução pode ser escrita de várias formas, simplesmente alterando o vértice inicial e mantendo a orientação original, este descreve um problema de permutação circular. Logo, considerando V um conjunto com n vértices, este terá $(n-1)!$ soluções no caso assimétrico, e $(n-1)!/2$ no caso simétrico, pois a inversão da orientação no percurso não é mais considerada uma solução distinta.

Existem diversas formulações matemáticas para o TSP, a mais popular é dada por Dantzig *et al.* (1954). Supondo um grafo $\mathcal{G}(V, E)$, de tal forma que c_{ij} represente o custo da aresta que liga o vértice i até o vértice j e que x_{ij} represente uma variável de decisão binária, tal que

$$x_{ij} = \begin{cases} 1, \text{ se a aresta que liga os vérices } i \text{ e } j \text{ está no circuito solução} \\ 0, \text{ caso contrário} \end{cases},$$

então o objetivo do problema é minimizar a expressão

$$\sum_{\forall i,j \in V} c_{ij} \cdot x_{ij},$$

sujeito as seguintes restrições

$$\sum_{\forall j \in V} x_{ij} = 1, \quad \forall i \in V, \tag{2.1}$$

$$\sum_{\forall i \in V} x_{ij} = 1, \quad \forall j \in V, \tag{2.2}$$

$$\sum_{\forall i,j \in S} x_{ij} \leq |S| - 1, \tag{2.3}$$

onde S é um subconjunto não vazio de V e $|S|$ a sua cardinalidade.

As restrições (2.1) e (2.2) garantem que cada vértice presente no grafo $\mathcal{G}$ será conectado por uma aresta que chega até ele e outra que sai deste mesmo vértice. O conjunto de restrições (2.3) é necessário para evitar subciclos, ou seja, uma solução com múltiplos circuitos. Em exemplo é ilustrado pela Figura 1.

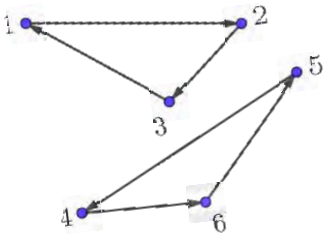


FIGURA 1 – Solução com subciclos. Fonte: Autor.

Na situação ilustrada tem-se que $x_{12} = x_{23} = x_{31} = x_{46} = x_{65} = x_{54} = 1$ e todas as demais variáveis de decisão nulas. Como todos os vértices estão conectados por exatamente duas arestas os conjuntos de restrições (2.1) e (2.2) são satisfeitos, porém, tomando $S = \{1, 2, 3\}$, tem-se

$$\sum_{\forall i,j \in S} x_{ij} = 3,$$

contradizendo uma das restrições (2.1), (2.2) ou (2.3).

O conjunto de restrições em (2.3) ainda é equivalente aos dois conjuntos

$$\sum_{\forall i,j \in \bar{S}} x_{ij} \leq |V| - |S| - 1,$$

$$\sum_{\forall i \in S, \forall j \in \bar{S}} x_{ij} \geq 2,$$

onde $\bar{S}$ é o conjunto complementar de S.

O problema neste modelo é a quantidade de restrições existentes. Caso $|V| = n$, tem-se exatamente n restrições para os conjuntos (2.1) e (2.2) e mais $2^n - 2$ no conjunto 2.3, pois haverá uma restrição para cada elemento no conjunto das partes de V, com exceção do conjunto vazio e o próprio V, ou seja, a quantidade de restrições no modelo cresce exponencialmente com o aumento de vértices no grafo.

Miller *et al.* (1960) propõem um modelo diferente para reduzir a quantidade de restrições. Para tal, se faz necessária a adição de variáveis auxiliares u_i, estas representam a ordem na qual cada vértice i é visitado e as restrições que evitam subciclos ficam da forma

$$u_i - u_j + nx_{ij} \leq n - 1 \quad i, j = 2, 3, \ldots, n. \tag{2.4}$$

Ainda é possível resolver o TSP como uma sucessão de problemas de designação, que é exatamente o que o modelo se torna ao excluir quaisquer restrições que evitam subciclos.

O processo é uma variação do método *branch and bound*, onde há uma relaxação inicial do modelo, para achar uma solução que pode ou não ser factível, caso não seja, restrições novas são adicionadas gerando novos subproblemas e estes são resolvidos até que se encontre uma solução factível. A vantagem deste método é que a partir do momento que se encontra uma solução factível com custo C, quaisquer subproblemas que alcancem um custo maior que C podem ser descartados, pois não há a possibilidade do custo baixar quando restrições estão sendo adicionadas.

Tome por exemplo uma situação onde um problema inicial de minimização é resolvido, porém a solução não é factível e gera três subproblemas novos, cada um destes também é resolvido e os resultados seguem como ilustrado na Figura 2.

A solução 2 é factível e por isso será referência para as próximas soluções encontradas, o seu custo será um limitante superior (*upper bound*). A solução 4 é infactível logo deve gerar novos subproblemas, porém estes não serão relevantes, pois esta solução está acima do limitante, então quaisquer subproblemas que derivem dela também estarão. A solução 3 também é infactível e gerará novos subproblemas, como esta solução está abaixo do limitante, é possível que haja uma solução de custo inferior a já encontrada, então estes novos subproblemas devem ser resolvidos. O processo prossegue até que não hajam mais ramos que possam gerar soluções inferiores ao limitante.

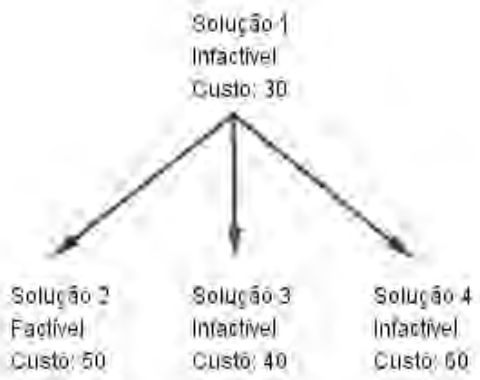


FIGURA 2 – Exemplo *branch and bound*. Fonte: Autor.

Para resolver o TSP especificamente, é necessário encontrar, na solução, o subciclo S com menor quantidade de vértices, a quantidade de subproblemas gerados será igual a quantidade de vértices em S, porém não se adicionam restrições ao problema, caso contrário não será mais um problema de designação, o que se faz é alterar os custos c_{ij} de forma que o subciclo gerado anteriormente já não seja uma opção atraente. Supondo um subciclo $S = (y_1, y_2, \ldots, y_k)$, um dos subproblemas gerados terá os custos $c_{y_1y_2} = c_{y_1y_3} = \cdots = c_{y_1y_k} = \infty$, desta forma, o vértice y_1 estará fora deste subciclo ao encontrar uma solução para este subproblema.

Tomando como exemplo um trajeto formado por dois subciclos $(1, 2, 3)$ e $(4, 6, 5)$, o primeiro gera três novos subproblemas, no primeiro caso, o vértice 1 é desligado do subciclo ao definir $c_{12} = c_{13} = \infty$. De forma análoga, o segundo subproblema utiliza $c_{21} = c_{23} = \infty$ e no terceiro $c_{31} = c_{32} = \infty$.

Outros modelos existem para descrever o TSP, alguns são variações dos modelos acima expostos, outros nem mesmo são baseados em restrições de designação. Independentemente do modelo, em grafos muito grandes é extremamente difícil utilizar métodos exatos para encontrar a solução do TSP, dados os recursos computacionais exigidos.

2.2 HEURÍSTICAS PARA O TSP

Nicholson (1971) define uma heurística como um procedimento "... para resolver problemas através de uma abordagem intuitiva na qual a estrutura do problema possa ser interpretada e explorada de forma inteligente a fim de se obter uma solução razoável."

Nesta seção serão abordadas algumas heurísticas bem simples e intuitivas, que são geralmente utilizadas para obter soluções iniciais para o PCV que não sejam totalmente aleatórias.

2.2.1 Métodos Gulosos

As heurísticas mais simples que existem para resolver problemas de otimização são as gulosas, para o caso do TSP existem duas, ambas são focadas em uma construção progressiva do circuito final. A primeira é comumente chamada de “vizinho mais próximo”, nela o circuito é construído nó a nó, mantendo a ordem na qual são escolhidos e sempre optando pelo nó mais próximo ao último escolhido para ser o seguinte no percurso.

Na Figura 3 as arestas tracejadas representam as arestas a serem avaliadas a cada iteração, a menor é selecionada e novas arestas que conectem o último vértice são avaliadas. O algoritmo do vizinho mais próximo deve ser como segue.

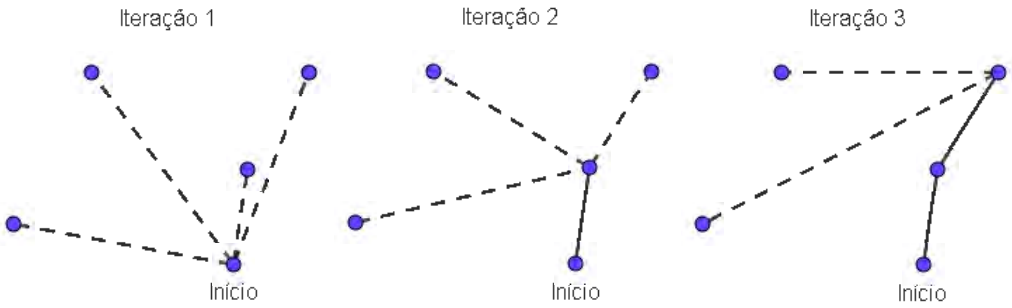


FIGURA 3 – Exemplo de construção pelo vizinho mais próximo. Fonte: Autor.

- Algoritmo 1: Vizinho mais próximo

Passo 1: Selecionar um vértice para dar início ao circuito;

Passo 2: Dentre os vértices que não foram visitados definir qual está mais próximo do último vértice visitado e coloca-lo ao fim do circuito;

Passo 3: Se exitem vértices não visitados, voltar ao passo 2;

Passo 4: Retornar circuito.

Note que cada vértice inicial utilizado no algoritmo possivelmente retornará um circuito diferente, por isso uma prática comum ao utilizar esta heurística é executar o programa diversas vezes, cada uma iniciando em um dos vértices do grafo armazenando o circuito que obtiver o menor custo.

A segunda heurística gulosa constrói o circuito com foco nas arestas, a cada iteração busca-se a menor aresta possível para colocar no trajeto. Nesta heurística é necessário um cuidado para que a próxima aresta a ser inserida não gere ciclos, o que irá ocorrer quando esta aresta possui uma extremidade com um vértice que já esteja ligado a outras duas arestas, ou quando esta fechar um circuito que possui uma quantidade de vértices inferior a quantidade de vértices no grafo.

Na Figura 4 as arestas tracejadas estão em avaliação e a que está destacada em vermelho será a próxima a entrar no circuito. Note que na iteração 3 algumas arestas podem gerar circuitos não hamiltonianos, ou subciclos, estas estão pontilhadas em cinza e não serão avaliadas para entrar na solução.

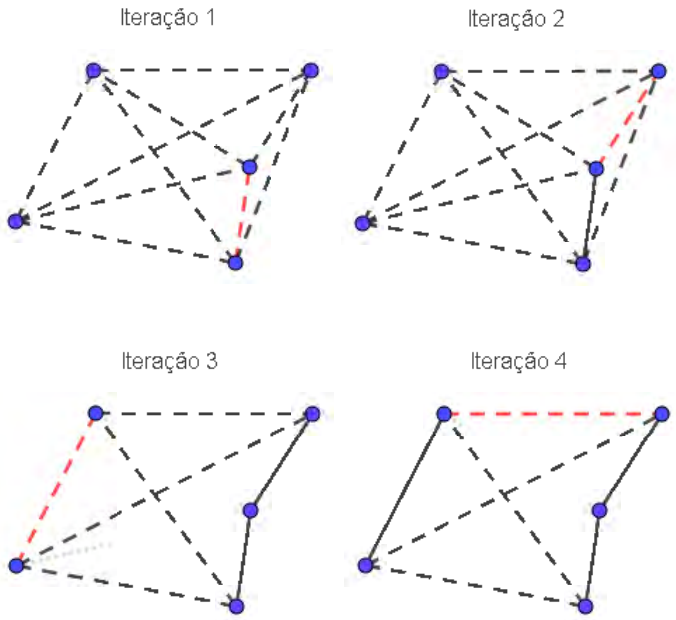


FIGURA 4 – Exemplo de construção pela heurística gulosa. Fonte: Autor.

- Algoritmo 2: guloso

Passo 1: Buscar a menor aresta no grafo que não gere subciclos e adiciona-la a solução;

Passo 2: Se existem vértices do grafo que não estão na solução retornar ao passo 1;

Passo 3: Retornar circuito.

Algumas heurísticas bem conhecidas para a resolução do TSP que são um pouco mais elaboradas, mas ainda possuem caráter guloso, estão baseadas em inserção. A ideia geral é encontrar uma solução de um subproblema do grafo original e a cada iteração um novo vértice é inserido nesta solução, até que todos os vértices sejam contemplados.

2.2.2 Métodos de Inserção

Algoritmos de inserção se diferenciam pela forma em que decidem qual o próximo nó e o local no qual este será alocado. Geralmente o critério de decisão utilizado para determinar o posicionamento do novo vértice a entrar na solução é o custo de inserção, este simboliza o valor que será incorporado pelo custo do circuito

após a inserção do vértice. Ao inserir um vértice k entre dois vértices i e j de um circuito já formado, a maior parte das arestas utilizadas no circuito anterior estarão também presentes no novo, com exceção da aresta que conectava i e j, esta deve ser excluída, e as arestas que irão conectar estes vértices a k serão criadas. Então, o custo de inserção C_{ikj} para que o vértice k esteja entre i e j é dado por

$$C_{ikj} = c_{ik} + c_{kj} - c_{ij}. \tag{2.5}$$

A inserção do mais próximo irá definir, dentre os vértices que não estão no subgrafo, qual está mais próximo a qualquer vértice presente no subgrafo, para ser o próximo inserido na solução atual, como subgrafo inicial são utilizados os vértices que estão nas extremidades da menor aresta no grafo. Um exemplo é ilustrado pela Figura 5.

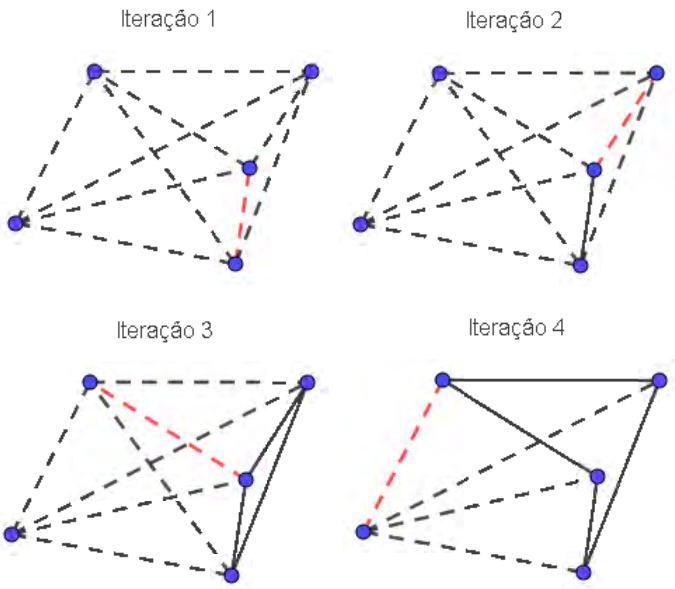


FIGURA 5 – Exemplo de inserção do mais próximo. Fonte: Autor.

- Algoritmo 3: inserção do mais próximo

Passo 1: Encontrar a menor aresta do grafo e adicionar os vértices nas suas extremidades a solução;

Passo 2: Dentre os vértices que não estão na solução, encontrar o que mais se aproxima de algum vértice da solução;

Passo 3: Inserir o vértice encontrado no passo anterior onde o custo de inserção 2.5 for menor;

Passo 4: Se houver algum ponto fora da solução, voltar para passo 2;

Passo 5: Retornar circuito.

Tomando esta heurística e alterando a seleção de vértices para o mais distante tem-se outro método de inserção. O circuito inicial também é alterado para os vértices que estão nos extremos da maior aresta no grafo.

Note na Figura 6 que a partir da segunda iteração são avaliadas apenas as menores arestas que ligam os vértices de fora do subciclo aos de dentro, então o processo não é simplesmente escolher as maiores arestas do grafo.

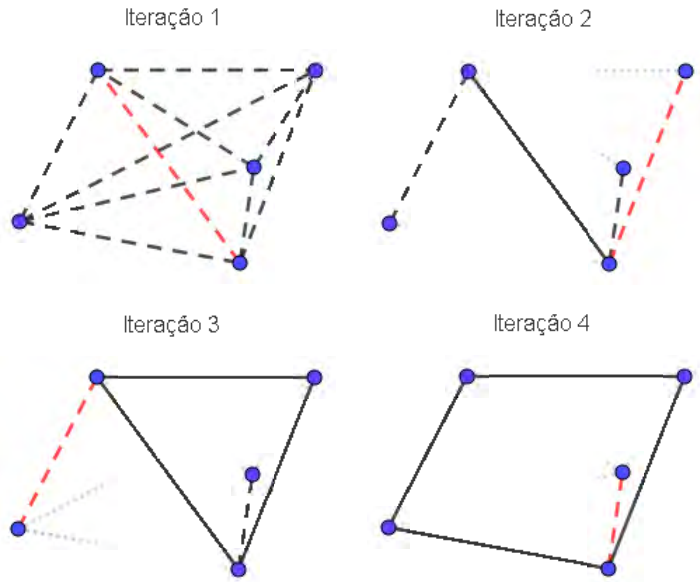


FIGURA 6 – Exemplo de inserção do mais distante. Fonte: Autor.

- Algoritmo 4: inserção do mais distante

Passo 1: Encontrar a maior aresta do grafo e adicionar os vértices nas suas extremidades a solução;

Passo 2: Dentre os vértices que não estão na solução, encontrar o que mais se distancia de todos os vértices da solução;

Passo 3: Inserir o vértice encontrado no passo anterior onde o custo de inserção (2.5) for menor;

Passo 4: Se houver algum vértice fora da solução, voltar para passo 2;

Passo 5: Retornar circuito.

O algoritmo de inserção do mais distante costuma obter soluções melhores por ter uma tendência a escolher os pontos mais exteriores do grafo primeiro, isso ajuda a evitar que arestas que se cruzam estejam no circuito solução. É baseado nesse pensamento que surge a heurística do envoltório convexo, um circuito inicial é criado de forma que não existam vértices exteriores ao circuito, feito isso o processo segue como o algoritmo de inserção do mais próximo. Este processo é ilustrado pela Figura 7.

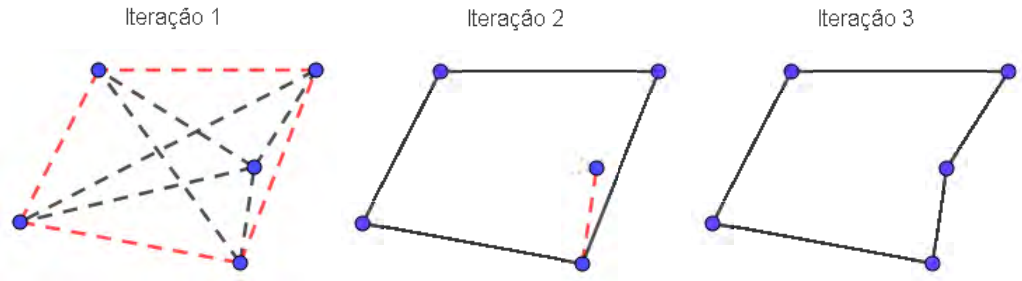


FIGURA 7 – Exemplo do envoltório convexo. Fonte: Autor.

- Algoritmo 5: envoltório convexo

Passo 1: Encontrar o envoltório convexo do grafo e tornar este circuito a solução atual;

Passo 2: Dentre os vértices que não estão na solução, encontrar o que mais se aproxima de algum vértice da solução;

Passo 3: Inserir o vértice encontrado no passo anterior onde o custo de inserção (2.5) for menor;

Passo 4: Se houver algum ponto fora da solução, voltar para passo 2;

Passo 5: Retornar circuito.

Estas heurísticas são extremamente rápidas, graças aos critérios simples de decisão ao construir o circuito, mas claramente não chegarão a soluções tão boas quanto outros métodos mais elaborados, elas serão utilizadas geralmente para obter soluções iniciais que não sejam completamente aleatórias e em seguida aplicar rotinas de melhoria para o caminho encontrado.

2.2.3 Método de Christofides

Outra heurística específica para a solução do TSP é descrita por Christofides (1975). A primeira etapa do processo consiste em resolver o problema de mínima arborescência do grafo, o que é muito simples de fazer, os algoritmos de Greenberg (1998) são os mais conhecidos para resolver este problema.

A segunda etapa é mais complexa, deve-se encontrar na árvore mínima um circuito euleriano. Diz-se que um circuito é euleriano quando todas as arestas do grafo são utilizadas exatamente uma vez no percurso, porém, para que um grafo possua um circuito euleriano, necessariamente os vértices deste devem ser de ordem par (Souza, 2013). Também pode-se demonstrar que qualquer árvore mínima gerada pode ser adaptada para existir circuitos eulerianos com a criação de algumas arestas artificiais Teixeira (2006).

A solução encontrada pelo algoritmo de Christofides será melhor de acordo com as arestas artificiais que forem criadas, por isso o melhor a se fazer é tomar o

subgrafo formado apenas pelos vértices de ordem ímpar e encontrar o acoplamento mínimo perfeito entre os vértices, isto é, encontrar as arestas que formem caminhos entre estes vértices, par a par, de forma que a soma dos custos destas arestas seja a menor possível. Este processo pode ser realizado utilizando o algoritmo de Floyd-Warshall, para definir a menor distância entre os vértices do subgrafo, combinado com o algoritmo de *blossom* (Kolmogorov, 2009), para encontrar os acoplamentos.

É possível também criar uma segunda aresta artificial para cada aresta existente na árvore mínima, desta forma é garantida a existência de um circuito euleriano, mas como dito, há uma perda em solução, porém também há ganho em velocidade de execução. Na Figura 8, o circuito euleriano gerado é dado por $(1, 2, 3, 5, 3, 4, 3, 2, 1)$, removendo nós repetidos temos o circuito hamiltoniano $(1, 2, 3, 5, 4)$, ilustrado na Figura 9.

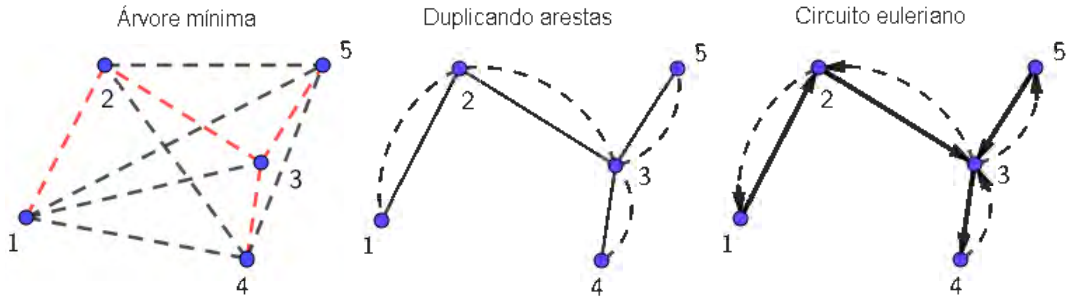


FIGURA 8 – Exemplo do algoritmo de Christofides. Fonte: Autor.

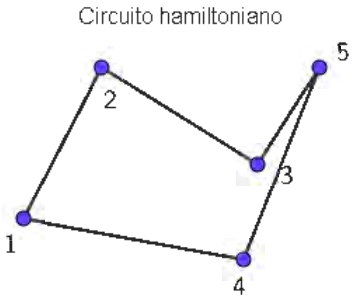


FIGURA 9 – Resultado do algoritmo de Christofides. Fonte: Autor.

- Algoritmo 6: Christofides

Passo 1: Encontrar a árvore mínima geradora do grafo;

Passo 2: Encontrar, dentre os vértices de grau ímpar, o acoplamento mínimo perfeito (ou duplicar cada uma das arestas na árvore);

Passo 3: Encontrar um circuito euleriano do grafo;

Passo 4: Remover do circuito encontrado vértices duplicados.

2.3 MÉTODOS DE MELHORIA

Em problemas complexos como o do caixeiro viajante, é comum utilizar heurísticas rápidas, as quais não se espera uma solução ótima, para na sequência melhorar esta solução utilizando algum método de busca local. Estes métodos alteram o circuito, poucas arestas por vez, repetindo o processo sempre que uma solução melhor for encontrada, até alcançar um mínimo local.

2.3.1 O método 2-OPT

Para o TSP em particular os métodos mais conhecidos são os 2-OPT, 3-OPT e k-OPT. O método 2-OPT foi proposto por Croes (1958), mas o movimento já havia sido sugerido anteriormente por Flood (1956). A lógica deste método é excluir uma certa quantidade de arestas, não adjacentes, do circuito, duas para o 2-OPT, três para o 3-OPT e assim em diante, quando estas arestas são excluídas o circuito é separado em caminhos e avalia-se qual deve ser a melhor forma de reconectar seus extremos.

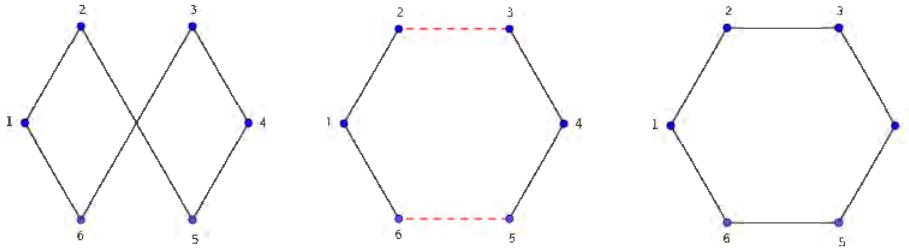


FIGURA 10 – Exemplo de um movimento 2-OPT. Fonte: Autor.

Um dos pontos fracos no uso do 2-OPT e suas variações é a notação do circuito. É comum utilizar a notação sequencial dos vértices visitados, isto tornaria a representação no primeiro circuito da Figura 10 (125436), mas a representação após a conclusão de um passo 2-OPT fica (123456), ou seja, toda a sequência de vértices precisou ser invertida entre os vértices 3 e 5. Em instâncias muito grandes, a quantidade de micro operações necessárias para fazer esta inversão cresce rapidamente, principalmente por que cada alteração no circuito reinicia o processo.

No 2-OPT, sempre que duas arestas forem quebradas quatro vértices estarão disponíveis para serem ligados a partir de três combinações de arestas, porém, uma das opções levará ao circuito antigo, outra irá gerar um subciclo, deixando apenas um único circuito para ser avaliado. Assim como na heurística de inserção, a maior parte do circuito não será alterada, então, quebrando duas arestas x_{ij} e x_{rs} a única comparação a ser feita é

$$c_{is} + c_{rj} < c_{ij} + c_{rs}, \tag{2.6}$$

sempre que a relação 2.6 for satisfeita, as arestas x_{ij} e x_{rs} são substituídas por x_{is} e x_{rj}.

Claramente quanto maior a quantidade de arestas excluídas, maior a quantidade de possíveis arestas a serem avaliadas. A Figura 11 mostra que para cada três arestas excluídas em um passo 3-OPT será necessário avaliar 8 conexões possíveis para os caminhos separados.

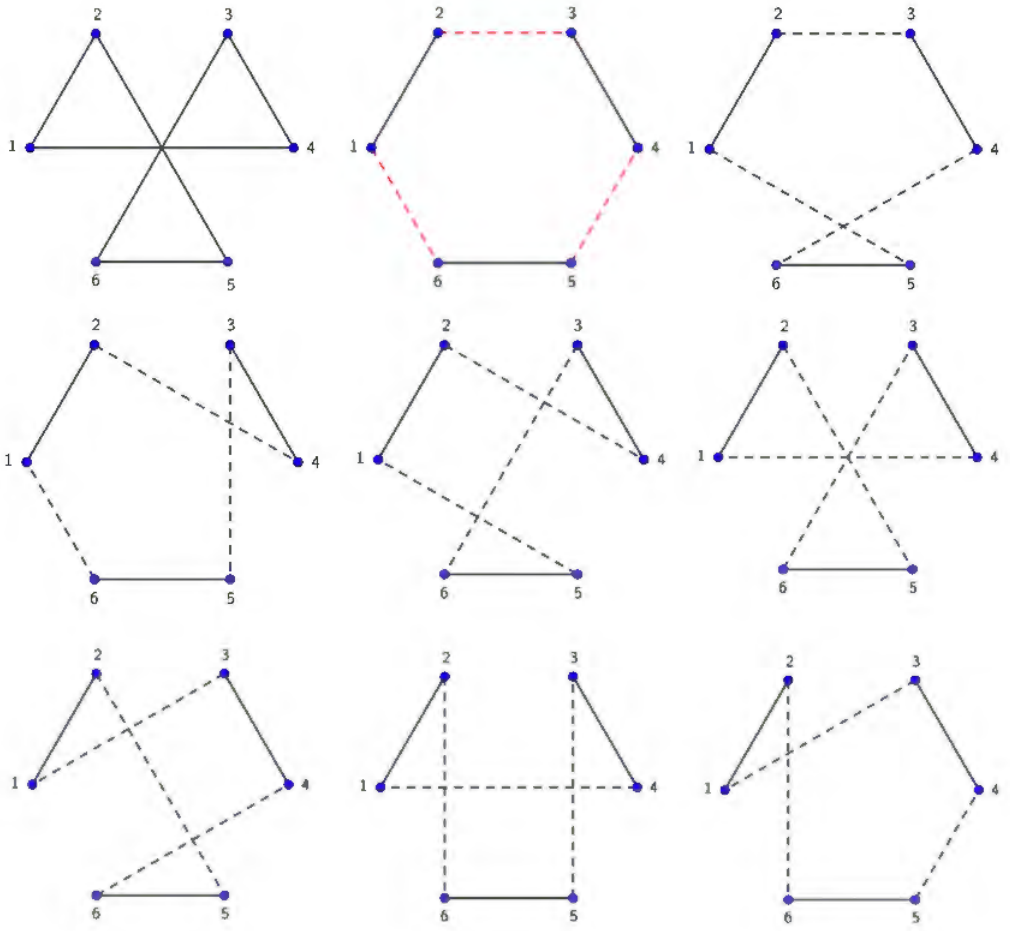


FIGURA 11 – Possíveis circuitos a serem avaliados em um passo 3-OPT. Fonte: Autor

A quantidade de conexões possíveis para um caso k-OPT pode ser calculada de forma recursiva. Imaginando uma situação onde k arestas serão excluídas e que $Q(k)$ representa a quantidade de possíveis ligações a serem feitas, haverão $2k$ vértices expostos, fixando um destes para estar na primeira aresta a ser gerada o segundo vértice não poderá ser o outro extremo de seu caminho, para não gerar um subciclo, nem o próprio vértice, existindo então $2k-2$ opções para criar esta primeira aresta e restando outras $k-1$ arestas a serem criadas, levando a recursão.

$$Q(k) = (2k-2)Q(k-1) = 2(k-1)Q(k-1). \tag{2.7}$$

Levando em consideração que o caso 2-OPT possui 2 possíveis configurações,

a original e a que as arestas excluídas se invertem, pode-se desenvolver a Equação (2.7).

$$\begin{aligned} Q(k) &= 2(k-1)Q(k-1) \\ &= 2(k-1)[2(k-2)Q(k-2)] \\ &= 2^2(k-1)(k-2)[2(k-3)Q(k-3)] \\ &\vdots \\ &= 2^{k-1}(k-1)!. \end{aligned}$$

Isto torna métodos k-OPT muito difíceis de serem implementados, Lin; Kernighan (1973) percebem que há uma forma de selecionar melhor quais os casos que devem ser analisados, ainda assim o processo irá exigir muito processamento, dependendo do valor de k. Pela característica de fazer trocas de arestas apenas quando existir um ganho de solução, valores de k baixos, como 2-OPT ou 3-OPT, serão levados a mínimos locais mais facilmente, para valores mais altos de k alguns destes mínimos podem ser evitados, tornando a busca local mais abrangente.

2.3.2 Busca Tabu

Outra técnica de melhoria bastante comum é a busca tabu, ela foi apresentada em 1989 por Glover (1990) e seus conceitos aprofundados posteriormente por Glover; Laguna (1997). Este método tem características de busca local mais fortes, por permitir movimentos que piorem a solução, mínimos locais serão evitados mais facilmente.

O processo se inicia gerando uma solução inicial através de uma heurística qualquer, a partir desta, um conjunto de soluções vizinhas é construído, realizando pequenas alterações na solução inicial, geralmente estas alterações são trocas de arestas como no 2-OPT. Deste conjunto de soluções vizinhas é escolhido aquele que otimiza a função objetivo para prosseguir com a busca, criando um novo conjunto de soluções vizinhas. A Figura 12 ilustra esse processo, onde S^0 é a solução inicial que gera o conjunto de soluções S^1, destas, S_i^{1*} é a melhor, e o processo segue.

Note que, a partir do momento que um segundo conjunto de soluções vizinhas é criado, espera-se que alguma destas seja aquela que deu origem ao primeiro conjunto de soluções, pois o mesmo critério de alteração de circuito está sendo utilizado, e como o processo admite que a busca seja feita em circuitos que piorem soluções encontradas anteriormente, possivelmente estas soluções serão escolhidas em etapas futuras do processo, fazendo com que o algoritmo ande em círculos. Para resolver este problema utiliza-se uma lista tabu, que contém movimentos já utilizados anteriormente, estes não serão cogitados futuramente, evitando o retrocesso do processo.

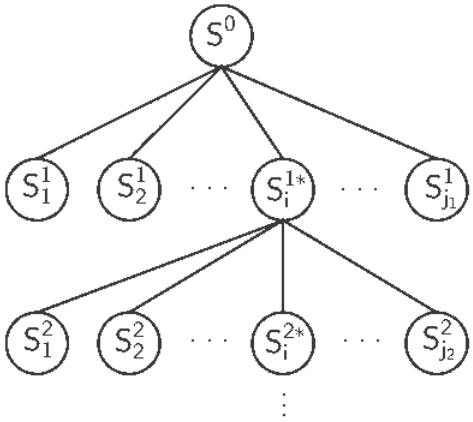


FIGURA 12 – Evolução do algoritmo de busca tabu. Fonte: Autor

O critério para seleção de movimentos que entraram na busca tabu e a forma de movimentar cada circuito para obter seus vizinhos ficam a critério do usuário, por isso este método pode ser muito forte para melhoria de caminhos. Uma forma simples de se utilizar a busca tabu é utilizar movimentos 2-OPT e a cada iteração colocar as duas arestas substituídas, ou apenas uma delas na lista tabu.

2.3.3 Recozimento Simulado

O método do recozimento simulado (*simulated annealing*) foi proposto por Metropolis *et al.* (1953) baseado no processo físico de recozimento sólido. Uma estrutura cristalina após a liquefação gasta o mínimo de energia possível durante a redução gradual da temperatura até atingir novamente o estado sólido.

O algoritmo inicia estabelecendo uma solução inicial como sendo a atual e define-se uma temperatura T, a cada iteração uma solução vizinha a atual é gerada e calcula-se o valor $\Delta = S* - S$, onde $S*$ representa o valor da função objetivo obtido pela solução atual e S o valor proveniente do vizinho gerado. Caso o valor Δ seja negativo, isto significa que uma solução melhor foi encontrada e imediatamente esta é definida como a solução atual, caso contrário, a solução pode ser aceita como a atual por um critério de probabilidade dado por

$$P = (exp)^{-\frac{\Delta}{T}}, \tag{2.8}$$

um valor aleatório é gerado e se este for maior que P a solução S será aceita, a temperatura é alterada para $T_{n+1} = T_n \cdot \alpha$ com $0 < \alpha < 1$ e o processo reinicia.

Como a aceitação de soluções é dada de forma probabilística se faz possível que soluções piores sejam admitidas e segundo a Equação (2.8) será mais fácil que isto ocorra no início do processo, quando T assume valores altos, evitando assim cair em mínimos locais rapidamente.

O recozimento simulado, originalmente, seleciona apenas um vizinho por iteração como próximo candidato a se tornar a solução atual, isto é bom em termos de desempenho computacional, porém ruim para a solução geral do problema já que é possível que soluções melhores não sejam nem avaliadas. Apesar disto, o recozimento simulado consegue alcançar resultados muito bons quando utiliza métodos mais elaborados para a geração de vizinhos combinados de uma listagem das soluções obtidas anteriormente, assim os vizinhos avaliados podem ser melhor direcionados.

2.4 META-HEURÍSTICAS

Boussaïd *et al.* (2013) mencionam similaridades que quase todas as meta-heurísticas possuem em comum: elas não utilizam o gradiente ou a matriz hessiana da função objetivo; elas possuem diversos parâmetros que devem ser selecionados manualmente; fazem uso de componentes estocásticos; são inspiradas por fenômenos presentes na natureza.

As características acima tornam um pouco mais clara a distinção entre heurísticas e meta-heurísticas, os elementos estocásticos presentes nas meta-heurísticas possibilitam buscas locais nas proximidades das soluções encontradas, o que não ocorre em heurísticas de caráter “guloso” por exemplo, onde a solução certamente não será a melhor possível e provavelmente nem mesmo uma boa solução, frequentemente caindo em um ótimo local. De forma geral, meta-heurísticas sinalizam de alguma forma quais variáveis parecem mais atraentes, o que tem um caráter guloso, porém, também contam com uma seleção probabilística destas variáveis, isso faz com que o algoritmo tome decisões localmente ruins, mas que possivelmente levam a soluções melhores no aspecto global.

Outro ponto a se ressaltar é a inspiração na natureza, isso também explica porque não se usa o gradiente ou a matriz hessiana. O raciocínio darwinista nos leva a crer que a fauna e flora no nosso planeta encontraram seus próprios meios de sobrevivência, e portanto possuem seus métodos para solucionar problemas, métodos que não estão diretamente relacionados a cálculos. Algoritmos genéticos simulam a troca de informação genética na reprodução de seres vivos, redes neurais simulam as sinapses que ocorrem no cérebro, o algoritmo de nuvem de partículas reproduz o movimento em manada de grupos de seres vivos (como um cardume ou um bando de pássaros), muitas outras meta-heurísticas, que apresentam ótimos resultados, inspiradas na natureza existem e outras novas continuam a surgir, o que reforça a relevância destas para a pesquisa operacional.

2.4.1 Otimização por Colônia de Formigas

Quando estão buscando fontes de sacarose, as formigas andam por caminhos aleatórios até que alguma encontre uma fonte, no seu retorno ao formigueiro esta deixa uma trilha de feromônio para as outras formigas. Enquanto a fonte fornecer alimento, as formigas que retornam continuam reforçando a trilha, a partir do momento que a fonte extinguir a trilha não será reforçada e com o tempo evaporará.

Em sua dissertação de doutorado, Dorigo propõe um algoritmo de otimização baseado no comportamento de formigas descrito acima (Dorigo *et al.*, 1999). Nesta dissertação, são descritas 3 variações desta meta-heurística, a que será utilizada como base durante o trabalho é denominada *ant cycle*, esta obtém melhores resultados tanto no algoritmo original quanto na adaptação aqui proposta.

Para resolver o problema do caixeiro viajante em particular, geralmente é colocada uma formiga em cada um dos n nós presentes no grafo e define-se uma quantidade pequena de feromônio em cada uma das possíveis arestas. As formigas escolhem, nó a nó, o caminho a ser tomado segundo a probabilidade

$$p_{ij}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{\forall j \in P}[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}, & \text{se } j \in P \\ 0, \text{ caso contrário} \end{cases}, \tag{2.9}$$

onde P é o conjunto dos nós que ainda não foram escolhidos pela formiga. O valor $\tau_{ij}(t)$ é a quantidade de feromônio depositada na aresta que vai do nó i (atual) até o nó j e $\eta_{ij} = Q/d_{ij}$ é o fator que leva em consideração a distância entre os nós (d_{ij}). Os valores Q, α e β são parâmetros fornecidos pelo usuário no início da execução do algoritmo.

Note que as formigas nunca escolhem caminhos de forma totalmente aleatória, no início da execução a regra de probabilidade dará maior relevância a caminhos mais curtos e ao fim a probabilidade penderá para caminhos com maiores acúmulos de feromônios, levando a uma estagnação, que é geralmente o critério de parada do algorítimo, ou uma quantidade limite de iterações.

Para evitar repetição de cálculos, é comum criar uma matriz A que represente a visibilidade de cada aresta, de forma que $a_{ij} = [\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}$, e a Equação (2.9) fica

$$p_{ij}(t) = \begin{cases} \dfrac{a_{ij}}{\sum_{\forall j \in P}(a_{ij})}, & \text{se } j \in P \\ 0, \text{ caso contrário} \end{cases}.$$

No *ant cycle* o depósito de feromônio ocorre após todas as formigas terminarem seus trajetos, se múltiplas formigas andarem pela mesma aresta, então mais feromônio será depositado nela. A quantidade de feromônio depositado leva em consideração

a distância total percorrida pela formiga, de forma que arestas utilizadas para obter caminhos menores sejam mais atraentes. O cálculo é dado por

$$\tau_{ij} = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t, t+n), \tag{2.10}$$

o valor $\Delta\tau_{ij}(t, t+n)$ será a soma dos feromônios depositados por cada uma das formigas

$$\Delta\tau_{ij}(t, t+n) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t, t+n),$$

com

$$\Delta\tau_{ij}^{k}(t, t+n) = Q/L^{k}. \tag{2.11}$$

Nas equações acima estão sendo utilizadas as mesmas notações presentes no trabalho original (Dorigo *et al.*, 1999), m representa a quantidade de formigas utilizadas no programa e k toma cada uma destas formigas em particular. Os valores t e n descrevem o momento no qual o feromônio está sendo depositado, então os momentos andam de n em n, justamente porque as formigas só depositam seus feromônios ao fim do trajeto, ou seja, após escolher o trajeto entre os n nós.

Algoritmo 7: *Ant Cycle*

```
Iniciar variáveis
Enquanto Condição de parada não satisfeita fazer
    Para cada formiga iniciada fazer
        Enquanto lista tabu não estiver cheia fazer
            Selecionar próximo nó de acordo com 2.9
            Colocar nó selecionado na lista tabu_k
        fim
    fim
    Para cada formiga k iniciada fazer
        Computar a distância L^k percorrida pela formiga k
        Para s = 1 até n fazer
            Definir (h, l) = (tabu_k(s), tabu_k(s+1))
            Δτ_hl(t+n) := Δτ_hl(t+n) + Q/L^k
        fim
    fim
    Atualizar feromônios em cada aresta
    Atualizar visibilidade em cada aresta
    Memorizar melhor caminho encontrado até o momento
    Esvaziar listas tabu
    Para i = 1 até n fazer
        Para cada formiga k fazer
            tabu_k(1) = i
        fim
    fim
fim
Imprimir melhor caminho
```

Note que em (2.10) existe um fator ρ que representa a evaporação de feromônios. A evaporação irá tornar arestas que frequentemente levam a trajetos ruins cada vez mais improváveis de serem escolhidas.

O cálculo de probabilidades e depósito de feromônios são as características principais do ACO para resolver problemas, além disso, como o objetivo é resolver o problema do caixeiro viajante, será feito uso de uma lista tabu, apenas para garantir que cada cidade será visitada exatamente uma única vez por cada formiga. Simplificadamente, o algoritmo da versão *ant cycle* do ACO deve ser parecido com o Algoritmo 7.

Vale ressaltar que o ACO costuma obter ótimos resultados em problemas relacionados a grafos, e o algoritmo acima é adaptado para solucionar especificamente o problema do caixeiro viajante, apesar disto, ele pode ser utilizado para solucionar outros problemas de otimização fazendo as devidas adaptações.

2.4.2 Otimização por Colônia de Abelhas Artificiais

Karaboga (2005), publicou uma ideia de meta-heurística inspirada na forma com que abelhas buscam por fontes alimentícias, mas em 2001 os mesmos princípios já haviam sido utilizados por Lucic; Teodorović (2001) para resolver o TSP. Uma vez que as abelhas podem se afastar por alguns quilômetros da colmeia até encontrar uma fonte promissora, estas fazem uso de um método de comunicação para indicar a outras abelhas os melhores caminhos.

O modelo é baseado principalmente em quatro componentes, a fonte alimentícia, as abelhas trabalhadoras, as oportunistas e as exploradoras. A fonte alimentícia pode ser representada por um simples valor numérico, este deverá levar em consideração a concentração de alimento encontrada e também a proximidade da fonte em relação a colmeia. As abelhas trabalhadoras estão associadas a fontes alimentícias específicas e possuem as informações de localização e valor potencial referentes a esta fonte, estas informações podem ser transmitidas para outras abelhas com uma certa probabilidade. As abelhas não trabalhadoras estarão buscando por novas fontes a serem exploradas, estas podem ser exploradoras ou oportunistas. As abelhas oportunistas seguem outras abelhas trabalhadoras, levando em consideração o valor de alimento arrecadado por cada abelha trabalhadora. As abelhas exploradoras irão buscar por novas fontes alimentícias de forma totalmente aleatória.

O mais comum a se fazer ao utilizar colônia de abelhas para resolver o TSP, é definir uma quantidade de abelhas trabalhadoras igual a quantidade de abelhas oportunistas que por sua vez é igual a quantidade de vértices no grafo. Inicialmente, um circuito é gerado, para cada abelha trabalhadora, através do algoritmo do vizinho mais próximo, sempre iniciando por um vértice diferente, as abelhas memorizam seus

Algoritmo 8: *Colônia de abelhas artificiais*

```
Iniciar uma abelha (bee) para cada um dos n vértices no grafo;
Iniciar contador M_i = 0 para cada abelha;
Iniciar melhor solução encontrada gBest;
gBestFit = ∞;
Para i = 1 até n fazer
    S = VizinhoMaisProximo(i);
    bee_i = S;
fim
Enquanto Condição de parada não satisfeita fazer
    Para i = 1 até n fazer
        S' = Permuta(bee_i);
        Se fit(S') < fit(bee_i) então
            M_i = 0;
            Se fit(S') < gBestFit então
                gBest = S';
                gBestFit = fit(S');
            fim
        fim
    fim
    Para i = 1 até n fazer
        Seleciona i pela probabilidade 2.12;
        S' = Permuta(bee_i);
        Se fit(S') < fit(bee_i) então
            M_i = 0;
            Se fit(S') < gBestFit então
                gBest = S';
                gBestFit = fit(S');
            fim
        fim
    fim
    Para i = 1 até n fazer
        Se M_i > limit então
            bee_i = CircuitoAleatorio();
            M_i = 0;
            break;
        fim
    fim
fim
Retornar gBest;
```

circuitos e seus respectivos valores *fit* , que representa o valor da função objetivo agregado a circuito.

A cada iteração do algoritmo, as abelhas trabalhadoras tentam encontrar um circuito que esteja na vizinhança do memorizado e seja melhor que este. Li *et al.* (2012) definem os circuitos vizinhos por simples trocas na ordem em que os vértices são visitados, a quantidade de trocas e os índices a serem permutados serão dados de forma aleatória. Calcula-se o *fit* do novo circuito gerado, se este for menor que o anterior ele irá sobrepor o memorizado anteriormente pela abelha.

Em seguida, é a vez das abelhas oportunistas atuarem, estas ainda buscam melhorias para os circuitos atuais, da mesma forma como as trabalhadoras fazem, porém, escolhem o caminho a ser seguido de forma probabilística segundo (2.12). Desta forma, caminhos que forem mais promissores serão os mais explorados com o decorrer do programa.

$$P_i = \frac{fit_i}{\sum fit_n}. \tag{2.12}$$

Sempre que uma abelha oportunista conseguir melhorar um circuito, este é

memorizado pela abelha a qual ela estava seguindo.

Em seguida é utilizado um parâmetro definido pelo usuário, $limit$ será um valor inteiro para determinar quando um ciclo será descartado, um contador M é definido para cada circuito existente na execução atual do programa, e será incrementado sempre que as abelhas trabalhadoras e oportunistas falharem e encontrar uma melhoria para o circuito. Se $M > limit$, aquele circuito é substituído por um novo gerado por uma abelha exploradora. De forma geral, o algoritmo não conta com muitas abelhas exploradoras, elas compõe em torno de 5% a 10% da colônia, para resolver o TSP em particular, é comum utilizar apenas uma abelha exploradora, o que significa que não mais que um circuito será substituído por cada ciclo do algoritmo.

Por fim, o critério de parada é verificado, geralmente uma quantidade limite de iterações do algoritmo, caso este seja satisfeito o melhor circuito encontrado durante o processo é retornado.

O forte caráter estocástico faz com que o ABC (*Artificial Bee Colony*) possua grande capacidade de busca de soluções, porém também faz com que a convergência das soluções seja geralmente lenta. Entretanto, para problemas não discretos a convergência é um pouco melhor, pois a seleção de soluções vizinhas não é tão aleatória, como descrito na equação a seguir:

$$s'_{ij} = x_{ij} + \phi(x_{ij} - x_{kj}). \tag{2.13}$$

Então a alteração na variável j da solução X_i memorizada pela abelha i, ocorre por uma aproximação à solução X_j com uma taxa ϕ. Os valores j, k e ϕ ainda são selecionados de forma aleatória.

2.4.3 Algoritmo Genético

Os conceitos que fundamentam algoritmos genéticos para solução de problemas em pesquisa operacional foram propostos por John Holland em 1960, estes se baseiam na teoria de Darwin da evolução e seleção natural, posteriormente, estes conceitos foram aprofundados por Goldberg (1989), aluno de Holland. Este método é um dos mais populares na área de otimização, por ser aplicável a uma vasta diversidade de problemas e as soluções obtidas costumam a ser satisfatórias.

Inicialmente é gerada uma população (conjunto de soluções) aleatória, o algoritmo consiste na execução de 3 passos, seleção, reprodução e mutação. Durante a seleção serão escolhidos os pais responsáveis por gerar a população utilizada na próxima iteração, isto pode ser feito de algumas formas, as mais comuns são os métodos da roleta, torneio e ranqueamento. O método da roleta escolhe indivíduos da população

Algoritmo 9: *Algoritmo genético*

```
Iniciar vetor população Pop com n indivíduos;
Iniciar vetor Pais;
Iniciar vetor Filhos;
Iniciar melhor solução encontrada gBest;
gBestFit = ∞;
Para i = 1 até n fazer
    Pop(i) = SoluçãoAleatória();
fim
Enquanto condição de parada não satisfeita fazer
    Pais = SelecionaPais(Pop);
    Filhos = Reproduzir(Pais);
    para cada filho em Filhos hacer
        Se fit(filho) < gBestFit então
            gBestFit = fit(filho);
            gBest = filho;
        fim
    fin
    Pop = NovaPopulação(Pais, Filhos);
    Se condição de mutação satisfeita então
        Pop = Mutação(Pop);
    fim
fim
Retornar gBest;
```

de forma proporcional ao valor *fitness* atribuído a cada indivíduo pela fórmula:

$$P(i) = \frac{fit_i}{\sum_k fit_k}, \tag{2.14}$$

com $P(i)$ sendo a probabilidade de escolha do i-ésimo indivíduo e k variando entre cada indivíduo da população. A Equação (2.14) deixa evidente que as maiores probabilidades de escolha serão atribuídas aos indivíduos com maiores valores *fitness*, então este método supõe que o problema a ser solucionado é de maximização. Uma adaptação simples para a fórmula (2.14), afim de se trabalhar com problemas de minimização, é utilizar $fit'_i = C - fit_i$ onde C é uma constante maior que os valores *fitness* que serão utilizados.

Note que no exemplo apresentado pela Figura 13, o indivíduo I_5 possui uma probabilidade muito baixa de ser escolhido, isto por estar muito próximo ao valor de C selecionado no exemplo, e esta probabilidade diminui com o aumento da população, portanto, apesar desta adaptação atender um problema de minimização, ela provavelmente não trará os melhores resultados.

A seleção por torneio, ilustrada na Figura 14, é mais simples e atende problemas de maximização e minimização, e por isso talvez seja a melhor opção para lidar

Indivíduo	Fitness	$fit'(C = 16)$
I_1	4	12
I_2	6	10
I_3	10	6
I_4	11	5
I_5	15	1

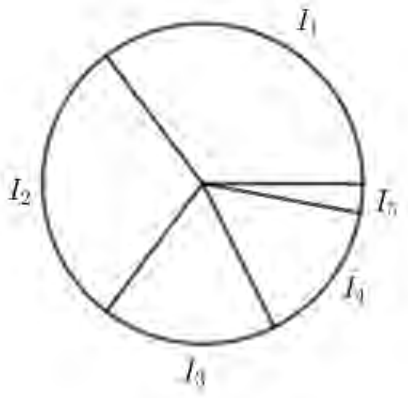


FIGURA 13 – Exemplo do método de seleção por roleta. Fonte: Autor.

com o TSP. A ideia é separar a população em diversos subconjuntos, a quantidade de subconjuntos a ser utilizada é variável, mas, para o algoritmo genético, é usual que a quantidade de pais selecionados possa gerar tantos filhos quanto forem os indivíduos descartados, para que o tamanho da população não se altere com o progresso do algoritmo. Os indivíduos que irão para cada subconjunto são escolhidos de forma aleatória e o que possuir o melhor *fitness* é selecionado.

Indivíduo	Fitness
I_1	4
I_2	6
I_3	10
I_4	11
I_5	15

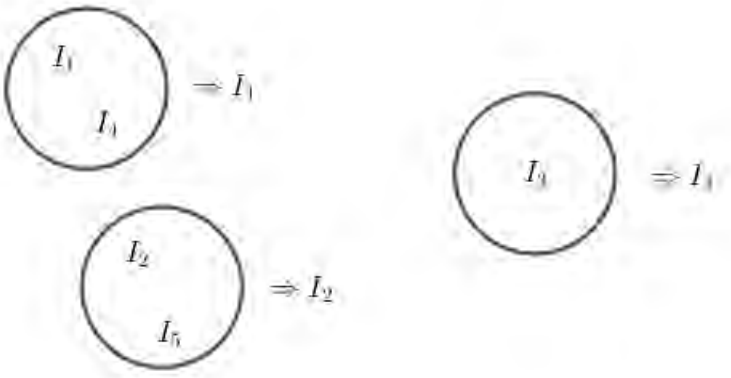


FIGURA 14 – Exemplo do método de seleção por torneio. Fonte: Autor.

O objetivo da seleção por ranqueamento, ilustrada na Figura 15, é fazer com que o valor *fitness* de cada indivíduo não seja tão decisivo para a escolha dos pais, mas ainda assim tenha a sua importância. Inicialmente toda a população é ordenada, a partir do *fitness*, e a posição de cada indivíduo nesta ordenação será utilizada para definir a probabilidade de seleção do mesmo, segundo a Equação (2.15).

$$P(i) = 2 - SP + \left(2 \cdot (SP - 1)\frac{pos(i) - 1}{n - 1}\right) \tag{2.15}$$

Na Equação (2.15), o valor $P(i)$ representa a probabilidade do indivíduo i ser escolhido, também $1 \leq SP \leq 2$ irá definir as probabilidades esperadas do melhor e do

pior indivíduo classificado na seleção, assim como a diferença entre as probabilidades de seleção de ranques subsequentes, $pos(i)$ representa a posição do i-ésimo indivíduo no ranque e n a quantidade de indivíduos na população.

Indivíduo	Fitness	$Pos(i)$	$P(i)\|SP = 1.5\|$
I_1	4	1	1,5
I_2	6	2	1,25
I_3	10	3	1,0
I_4	11	4	0,75
I_5	15	5	0,5

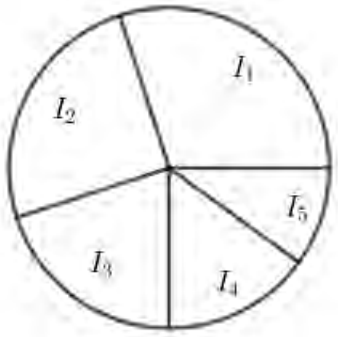


FIGURA 15 – Exemplo do método de seleção por ranqueamento. Fonte: Autor.

A Figura 15 mostra como as probabilidades variam de forma linear, um comparativo com a Figura 13 mostra que a probabilidade de escolha de indivíduos com valores *fitness* baixos tem maiores chances de serem explorados, o que ajuda também a evitar mínimos locais, porém atrapalha a convergência da solução. Além disto, como a população está sempre mudando a cada iteração se faz necessário reordenar os indivíduos, isto agrega uma carga operacional maior ao método.

Um comparativo destes métodos de seleção em função das soluções obtidas e o tempo de execução do algoritmo pode ser encontrada na publicação de Razali; Geraghty (2011).

Após a seleção dos pais, dois a dois irão cruzar suas informações genéticas para repor os indivíduos deixados de lado na etapa anterior. Geralmente o processo se faz por uma simples combinação de uma parte da solução armazenada pelo primeiro pai e o restante vindo da solução armazena pelo segundo, porém o TSP possui as suas próprias restrições e portanto combinações tão diretas provavelmente não gerarão soluções factíveis, alguns dos métodos mais populares de se fazer estes cruzamentos de informação, utilizando a notação sequencial de vértices visitados, serão descritos a seguir.

O operador de ordem foi proposto por Davis (1985), nele são sorteados dois índices, a sequência de vértices visitados entre os dois índices é repassada para os filhos, então, o restante da solução do filho que recebeu a sequencia do pai 1 será completa com os vértices ainda não visitados na ordem em que aparecem para o pai 2, e vice-versa. Este processo é ilustrado na Figura 16.

O operador parcialmente mapeado foi proposto por Goldberg; Lingle (1985), da mesma forma que o operador anterior, serão sorteados dois índices para repassar

$$P1 - \{1,5,6,|8,4,2,9|,3,7\}$$
$$P2 - \{2,5,9,|4,3,8,7|,1,6\}$$
$$F1 - \{\_,\_,\_,|8,4,2,9|,\_,\_\} \rightarrow \{5,3,7,|8,4,2,9|,1,6\}$$
$$F2 - \{\_,\_,\_,|4,3,8,7|,\_,\_\} \rightarrow \{1,5,6,|4,3,8,7|,2,9\}$$

FIGURA 16 – Exemplo operador de ordem. Fonte: Autor.

a sequência de vértices visitados entre eles aos filhos, isto também deixa definido um mapa, para cada índice k entre os sorteados, o k-ésimo vértice visitado do pai 1 será associado ao k-ésimo vértice visitado pelo pai 2. Feito isso, o filho que recebeu o trecho inicial do pai 1, completa suas informações com a solução que provém do pai 2, alterando apenas os vértices que foram mapeados.

$$\left.\begin{array}{l} P1 - \{1,5,6,|8,4,2,9|,3,7\} \\ P2 - \{2,5,9,|4,3,8,7|,1,6\} \end{array}\right\} Mapa \rightarrow 8 \leftrightarrow 4, 4 \leftrightarrow 3, 2 \leftrightarrow 8, 9 \leftrightarrow 7$$

$$F1 - \{\_,5,\_,|8,4,2,9|,1,6\} \rightarrow \{3,5,7,|8,4,2,9|,1,6\}$$

$$F2 - \{1,5,6,|4,3,8,7|,\_,\_\} \rightarrow \{1,5,6,|4,3,8,7|,2,9\}$$

FIGURA 17 – Exemplo operador parcialmente mapeado. Fonte: Autor.

Vale ressaltar que no exemplo apresentado na Figura 17 é necessária uma transitividade entre os vértices mapeados, de forma que $2 \leftrightarrow 8 \leftrightarrow 4 \leftrightarrow 3$, é o mesmo que $2 \leftrightarrow 3$.

O operador de cruzamento cíclico foi proposto por Oliver *et al.* (1987), o objetivo dele é fazer com que o momento em que cada vértice é visitado seja o mesmo de algum dos pais. O processo prossegue da seguinte forma: fixa-se a posição inicial $i = 1$, faz-se $F1(i) = P1(i)$, busca-se em $P1$ o índice k do vértice $P2(i)$, define-se $i = k$ e repete-se o processo até que $k = 1$. Os vértices do filho 1 que não foram preenchidos serão repassados pelo pai 2 na mesma ordem em que são visitados.

$$P1 = \{1,5,6,8,4,2,9,3,7\}$$
$$P2 = \{2,5,9,4,3,8,7,1,6\}$$

$$F1 = \{1,\_,\_,8,4,2,\_,3,\_\} \Rightarrow \{1,5,9,8,4,2,7,3,6\}$$

FIGURA 18 – Exemplo operador de cruzamento cíclico. Fonte: Autor.

Outro filho pode ser gerado invertendo a função de ambos os pais no processo.

Por fim, a mutação geralmente é realizada depois de uma determinada quantidade de iterações, pois mutações muito frequentes podem fazer com que soluções boas sejam perdidas antes de serem exploradas. Para o TSP uma mutação pode ser realizada por uma simples troca na ordenação de dois vértices visitados ou uma inversão da ordenação de um determinado trecho da solução.

2.4.4 Otimização por Nuvem de Partículas

A otimização por nuvem de partículas (*particle swarm optimization*), usa como inspiração o comportamento social de insetos ao se movimentar em enxames. A ideia de aplicar este comportamento para resolver problemas combinatoriais foi proposta por Goldberg; Lingle (1985).

Neste algoritmo, cada partícula será representada por uma posição (X) e uma velocidade (v), as posições são possíveis soluções para o problema, as velocidades são a tendência de alteração da posição atual. Inicialmente, tanto as posições quanto as velocidades são definidas para cada partícula de forma aleatória, a cada iteração a velocidade de cada partícula se altera, levando em consideração a melhor solução que aquela partícula já encontrou ($pBest$) e a melhor solução global encontrada ($gBest$), então a nova posição desta partícula será definida pela soma desta velocidade com a posição anterior, segundo as equações em (2.16).

$$\begin{cases} v_{t+1} = c_1 v_t + c_2(pBest - X_t) + c_3(gBest - X_t) \\ X_{t+1} = X_t + v_{t+1} \end{cases} \tag{2.16}$$

Os valores c_1, c_2 e c_3 são entradas do algoritmo, que representam o momento e as relevâncias do mínimo local e global respectivamente. Ainda é possível multiplicar cada uma destas constantes por um valor aleatório, melhorando a busca local do algoritmo.

As equações (2.16) deixam claro que o método de nuvem de partículas foi elaborado para trabalhar com problemas contínuos, então são necessárias adaptações para resolver o TSP, assim como outros problemas discretos. Uma forma de utilizar nuvem de partículas para resolução do TSP é descrita por Clerc (2004), a base desta adaptação é a reinterpretação da velocidade, como esta deve aproximar a solução atual dos ótimos local e global serão utilizadas trocas de vértices, de forma que a troca force a solução antiga a ter maior similaridade com os ótimos locais. A velocidade então fica definida por uma sequência de trocas e será denotada por $v = ((i_k, j_k))$ representando a troca de posições do vértice i_k com o vértice j_k e vice-versa, com $1 \leq i, j \leq \|v\|$, onde $i, j \in \mathbb{N}$ e $\|v\|$ representa a quantidade de trocas a serem realizadas por v. A Figura 19 exemplifica a aplicação da velocidade sobre um circuito.

$$
\begin{aligned}
S_1 &= (1,5,3,4,2,6) \\
v &= ((2,5),(3,5),(4,6)) \\
\\
(2,5) &\rightarrow (1,2,3,4,5,6) \\
(3,5) &\rightarrow (1,2,5,4,3,6) \\
(4,6) &\rightarrow (1,2,5,6,3,4) \\
\\
S_1 \oplus v &= (1,2,5,6,3,4)
\end{aligned}
$$

FIGURA 19 – Exemplo de aplicação de velocidade a uma posição. Fonte: Autor.

No exemplo apresentado na Figura 20, a alteração da posição se dá por 3 trocas em sequência dos vértices em S_1. Note que a ordem na qual serão realizadas as trocas tem influência direta na posição obtida ao fim do processo.

Algoritmo 10: *Algoritmo de nuvem de partículas*

Iniciar vetor com n partículas $Part()$;
Iniciar vetor com n velocidades $Vel()$;
Iniciar vetor com melhor solução pessoal encontrada $pBest$;
Iniciar melhor solução encontrada $gBest$;
$gBestFit = \infty$;
Para $i = 1$ **até** n **fazer**
  $Part(i) =$ SoluçãoAleatória();
  $pBest(i) = Part(i)$;
  $Vel(i) =$ VelocidadeAleatória();
  Se $fit(Part(i)) < gBestFit$ **então**
    $gBest = Part(i)$;
    $gBestFit = fit(Part(i))$;
  fim
fim
Enquanto *condição de parada não satisfeita* **fazer**
  Para $i = 1$ **até** n **fazer**
    Atualizar velocidade e posição da partícula $Part(i)$ segundo as equações em 2.17;
    Se $fit(Part(i)) < fit(pBest(i))$ **então**
      $pBest(i) = Part(i)$;
      Se $fit(Part(i)) < gBestFit$ **então**
        $gBest = Part(i)$;
        $gBestFit = fit(Part(i))$;
      fim
    fim
  fim
fim
Retornar $gBest$;

Como o objetivo é, aos poucos, aproximar as soluções dos mínimos, define-se a diferença entre duas posições $S_1 - S_2$ como a velocidade a qual $S_1 = S_2 \oplus v$, ou seja,

a sequência de trocas necessárias para transformar a solução S_2 em S_1.

$$S_1 = (1, 5, 3, 4, 2, 6)$$
$$S_2 = (1, 3, 4, 6, 5, 2)$$
$$S_1 - S_2 = ((3, 5), (4, 3), (6, 4), (6, 2))$$

FIGURA 20 – Exemplo diferença de posições. Fonte: Autor.

Também será definida operação produto entre velocidade (v) e um número real (c), exemplificado pela Figura 21. Para tal, é necessário avaliar três casos possíveis. No primeiro caso $0 < c < 1$, e nesta situação faz-se $cv = ((i_k, j_k))$ com $1 \leq k \leq c\|v\|$, o valor $c\|v\|$ é truncado para o maior inteiro menor que ele mesmo, desta forma apenas uma parte de v é aplicada. Caso $c > 1$, faz-se $c = k + c'$ com $k \in \mathbb{N}$ e $c' \in [0, 1[$, e desta forma

$$cv = \underbrace{v \oplus v \oplus \cdots \oplus v}_{k \text{ vezes}} \oplus c'v.$$

Por fim, caso $c < 0$, faz-se $cv = c(\neg v)$ com $\neg v$ representando as trocas definidas por v sendo realizadas com sua ordenação invertida.

$$\begin{aligned} v &= ((2,5),(3,5),(4,6)) \\ 0.7v &= ((2,5),(3,5)) \\ 2.5v &= ((2,5),(3,5),(4,6)) \oplus ((2,5),(3,5),(4,6)) \oplus ((2,5)) \\ -1.5v &= ((4,6),(3,5),(2,5)) \oplus ((4,6),(3,5)) \end{aligned}$$

FIGURA 21 – Exemplos de produto entre velocidade e número real. Fonte: Autor.

Com os operadores acima definidos, é possível utilizar as mesmas equações em (2.16) para resolver o TSP. Ainda neste trabalho, é apresentada uma proposta de alteração para a primeira equação, fazendo com que as posições se movam em direção a um ponto (P) entre o mínimo local e o global, este ponto é calculado por $P = pBest + (gBest - pBest) * 0.5$, reescrevendo as equações como descritas em (2.17).

$$\begin{cases} v_{t+1} = c_1 v_t + c_2'(P - X_t) \\ X_{t+1} = X_t + v_{t+1} \end{cases} \tag{2.17}$$

As equações em (2.17) são propostas baseadas na ausência de estudos indicando que a escolha de $c_2 \neq c_3$ em (2.16) possa trazer melhoras significativas para a solução obtida ao fim do processo.

2.4.5 Busca Harmônica

Os conceitos do algoritmo de busca harmônica foram apresentados por Geem *et al.* (2001), inspirados pelo processo de improvisação de músicos de jazz. Quando mais de um instrumento está sendo tocando simultaneamente a sobreposição de ondas sonoras pode gerar um som que não seja agradável, com a prática os músicos decidem alterar ou ajustar as notas tocadas até que os sons produzidos por cada instrumento encontrem uma harmonia.

Assim como o algoritmo genético, a busca harmônica usa uma população de soluções para procurar por soluções melhores a cada iteração, a diferença é que, para o primeiro, novas soluções são geradas com duas soluções, enquanto no segundo todas as soluções são utilizadas simultaneamente.

No início do processo as soluções são produzidas aleatoriamente, este conjunto de soluções é chamado de memória harmônica (*harmony memory* - HM), e a cada iteração será atribuída a cada variável do problema um valor igual ao valor selecionado por alguma solução na HM, uma vez construída uma nova solução, esta será avaliada, e se o seu valor *fitness* for melhor que a pior solução na HM, então a nova solução é adicionada na HM e a pior é esquecida. Note que, somente o processo acima não é capaz de fazer buscas locais, para tal inclui-se uma taxa de consideração da memória harmônica (*harmony memory considering rate - HMCR*), este valor é geralmente alto, aproximadamente 95%, e indica em que proporção as variáveis das novas soluções serão selecionadas a partir da HM, caso a variável a ser selecionada fuja das opções na HM, duas situações podem ocorrer, ou novo valor da variável será atribuído de forma aleatória, ou este valor será próximo a algum selecionado na HM, a primeira opção terá uma chance maior de ocorrer, aproximadamente 90%, pois a intenção do HMCR é escapar de máximos ou mínimos locais, a probabilidade de ocorrência de cada um é dada pela taxa de ajuste de tonalidade (*pitch adjusting rate - PAR*).

A adaptação do algoritmo para o TSP é feita de forma simples e direta (Boryczka; Szwarc, 2018), em suma, o vértice de partida é comum a todas as soluções na HM, em seguida, para cada índice da nova solução será atribuído um vértice, no mesmo índice indicado, selecionado por alguma solução na HM, caso ocorra um ajuste no vértice basta selecionar o vértice mais próximo a este, ou um dentre os mais próximos.

A busca harmônica possui uma dificuldade maior para encontrar soluções boas para o TSP em instâncias de problemas muito grandes, aproximadamente 500 vértices ou mais, e geralmente não é utilizada nessas situações, mas, Tseng (2016) propõe uma busca harmônica elitista para suprir esta deficiência. A ideia é separar a HM em dois grupos, um grupo pequeno com as melhores soluções (grupo elite) na HM e outro

Algoritmo 11: *Busca Harmônica*

```
Iniciar vetor com n soluções HM();
Iniciar melhor solução encontrada gBest;
gBestFit = ∞;
Para i = 1 até n fazer
  | HM(i) = SoluçãoAleatória();
fim
OrdenarFit(HM);
gBest = HM(1);
gBestFit =Fit(HM(1));
Iniciar valor flutuante alfa;
Enquanto condição de parada não satisfeita fazer
  | Iniciar nova solução new;
  | Para i = 1 até m fazer
  |   | alfa = ValorAleatório(0,1);
  |   | Se alfa<HMCR então
  |   |   | new(i) =SelecionaVértice(new, HM, i);
  |   | caso contrário
  |   |   | alfa = ValorAleatório(0,1);
  |   |   | Se alfa<PAR então
  |   |   |   | new(i) =VérticeAleatório(new, i);
  |   |   | caso contrário
  |   |   |   | new(i) = VérticePróximo(new, HM, i);
  | fim
  | Se Fit(new)<Fit(HM(n)) então
  |   | Inserir(new, HM);
  |   | gBest = HM(1);
  |   | gBestFit =Fit(HM(1));
  | fim
fim
Retornar gBest;
```

com as demais soluções, ao atribuir um valor a uma variável, será mais provável que este venha de uma solução no grupo elite, o resto do processo permanece igual ao original.

Como a cada iteração é necessário saber qual a pior solução presente na HM, e quando esta for substituída, saber a nova pior solução, é útil realizar uma ordenação das soluções a partir de seus valores *fitness*, principalmente se a versão elitista do método for utilizada. Apesar da ordenação só precisar ser realizada ao construir a primeira HM, para mantê-la organizada com o decorrer do processo, a menos do uso de uma estrutura especializada, uma realocação de memória é necessária, o que pode aumentar o tempo de processamento.

As funções do algoritmo que selecionam vértices para as novas soluções devem tomar cuidado de que estes vértices já não estejam presentes nos índices

anteriores da solução, a função inserir também deve tomar o cuidado de manter a HM ordenada e excluir a pior solução.

Isoladamente nenhuma das heurísticas anteriores será excepcional ao solucionar o problema do caixeiro viajante, ou qualquer outro problema de otimização, mas estes algoritmos podem ser combinados para obterem resultados melhores. Este trabalho é focado no ACO, então, no próximo capítulo é feito um estudo sobre as principais alterações realizadas neste algoritmo, que melhoram os resultados obtidos por ele.

3 REVISÃO BIBLIOGRÁFICA

O algoritmo de otimização por colônia de formigas é relativamente novo em comparação a muitos algoritmos que são utilizados na área de pesquisa operacional. Por este motivo é muito comum encontrar estudos que façam alterações no algoritmo original do ACO, afim de melhorar seu desempenho, seja de forma geral ou para problemas particulares.

Blum (2005), relata que as melhorias, baseadas no ACO, que obtiveram maior sucesso até 2005 foram: *Elitist Ant System (EAS), Rank-based Ant System (RAS), MAX-MIN Ant System (MMAS), Ant Colony System (ACS)* e *Hyper-Cube Framework (HCF)*. No EAS, é realizada uma alteração durante a atualização de feromônios, para cada iteração, a memória da melhor solução encontrada até o momento é utilizada no depósito de feromônios, assim como as soluções encontradas durante a mesma iteração, além disso, a quantidade de feromônios depositada nas arestas presentes na melhor solução encontrada podem ter um peso elevado. No RAS, a melhor solução encontrada também é incorporada ao conjunto de soluções de cada iteração, porém, este conjunto será ordenado, afim de atribuir um *rank* para cada solução, este *rank* tem influência no peso de cada solução ao depositar feromônios nas arestas. O MMAS é, dentre estes, o que possui resultados mais significativos, o algoritmo usa duas regras de atualização de feromônios, uma de forma mais generalizada no início, permitindo uma busca local mais eficaz, e outra mais focada na melhor solução encontrada ao fim da execução, provocando uma aceleração na convergência da solução. Também é definido um limite inferior e superior para a quantidade de feromônios presentes em cada aresta, o valor máximo é alterado de acordo com a melhor solução encontrada pelo algoritmo. O HCF é a melhoria mais recente dentre as citadas anteriormente, e é caracterizada pelo peso aplicado às atualizações realizadas na matriz de feromônios, este peso leva em consideração o valor *fit* obtido das soluções que são utilizadas na atualização, desta forma os feromônios se adaptam ao progresso do algoritmo automaticamente, além disto, a melhoria pode ser utilizada em conjunto com a maioria das variações do ACO, possivelmente obtendo resultados ainda melhores.

O objetivo deste capítulo é avaliar que tipo de abordagem tem sido utilizada recentemente, para aprimorar o algoritmo ACO, para tal, foram utilizados dois sites de busca acadêmica, *Science Direct* e *Google Scholar*. Centenas de publicações científicas podem ser encontradas a respeito do ACO, as pesquisas neste trabalho serão concentradas nos materiais que relatam melhorias utilizando como base de testes o TSP, então as palavras-chave utilizadas foram: “ant”, “colony”, “optimization”, “traveling”, “salesman” e “problem”. Aproximadamente 30 publicações foram encontradas no

Science Direct, nos últimos 10 anos, que mencionam simultaneamente ACO e TSP, enquanto aproximadamente 70 foram encontradas no *Google Scholar*. Deixando de lado as publicações que não propõe quaisquer tipos de melhorias e também as que utilizam variações do problema do caixeiro viajante, restam 12 publicações no *Science Direct* e 29 no *Google Scholar*, após uma leitura mais atenta, outras 16 publicações serão deixadas de lado, restando então 25.

Nas publicações obtidas pela revisão sistemática da literatura descrita acima, é possível notar que as melhorias estão focadas em quatro pontos principais, não necessariamente o algoritmo gira em torno de apenas uma alteração, na verdade, o mais comum é que duas ou três alterações sejam utilizadas. As alterações mais comuns observadas foram: combinação do ACO com outras heurísticas, modificações na atualização de feromônios, paralelismo e modificações na tomada de decisão das formigas. As próximas seções detalham melhor cada um destes pontos, citando os trabalhos que utilizaram estas alterações e relatando de forma simples como o fizeram.

3.1 HEURÍSTICAS ADICIONAIS

A técnica mais comum de melhoria que há é a combinação de heurísticas. É possível utilizar uma heurística que possua uma forte busca local seguida de outra que possa aprimorar ainda mais a solução encontrada anteriormente. Para o caso do TSP é muito comum o uso do 2-opt como técnica de aprimoramento. Há também algoritmos que misturam os conceitos de heurísticas diferentes, acoplando passos de uma na outra a cada iteração, estes são mais interessantes e o foco desta seção.

Yun *et al.* (2013) fazem uso do ACO combinado a busca harmônica, apresentada na Seção 2.4.5 deste trabalho. O algoritmo inicia exatamente como o algoritmo de busca harmônica, gera-se um conjunto de soluções aleatórias para ser a memória harmônica inicial, em seguida uma nova harmonia é criada e a memória harmônica é atualizada, excluindo as piores soluções, neste momento a HM será utilizada para o depósito de feromônios, as formigas buscam por novas soluções, se boas soluções forem encontradas, elas atualizam a memória harmônica novamente e o processo se repete gerando uma nova harmonia.

Então o algoritmo descrito anteriormente basicamente realiza uma iteração da busca harmônica seguida de uma iteração do ACO, mas é interessante também notar como estas iterações não estão isoladas, uma vez que memória harmônica influencia diretamente no depósito de feromônios que é a principal característica do ACO, e, por outro lado, as soluções encontradas pelas formigas também são avaliadas a entrar na memória harmônica. Durante o processo, também é relatada uma tendência de a solução ótima encontrada pela busca harmônica ser uma solução presente na HM, o que pode ocasionar uma convergência muito rápida, então o autor também faz uso de

uma técnica de mutação, natural do algoritmo genético, para ampliar a busca local do algoritmo. Por fim, o algoritmo obteve grande sucesso, foi testado em 20 instâncias do TSP Lib (Discrete and Combinatorial Optimization, 2020) obtendo a melhor solução já encontrada na maior parte deles, e melhorando as soluções anteriores de 2 casos.

Uma combinação simples do ACO com algoritmo genético pode ser efetuada, pela simples execução do ACO para gerar as populações que serão cruzadas no GA. Sahana; Jain (2011) utilizam esta ideia de algoritmo para propor um aprimoramento utilizando dois algoritmos simples de melhoria atualizando os feromônios a cada melhoria implementada.

O processo se inicia com a geração de uma população inicial, como sugerido anteriormente, seguindo os passos tradicionais do ACO, então o GA realiza o cruzamento entre as soluções, boas soluções substituem os pais com piores *fit*, a matriz de feromônios será atualizada neste momento. Em seguida utiliza-se a primeira técnica de melhoria em cada uma das soluções obtidas, esta remove um a um os vértices da solução para então inseri-los novamente, isto ajusta vértices mal posicionados na solução, outra atualização da matriz de feromônios é realizada. A segunda técnica utilizada é uma busca local, na publicação é utilizado um opt, após a segunda melhoria, a matriz de feromônios é atualizada uma última vez, antes de se iniciar uma nova iteração.

O conceito do algoritmo anterior é combinado por ainda mais técnicas no trabalho de Chen; Chien (2011), que propõe o algoritmo intitulado *genetic simulated annealing ant colony system with particle swarm optimization techniques*. A combinação de algoritmo GA com recozimento simulado já havia sido implementada para solução do problema de roteamento de veículos anteriormente, o autor então coloca este conceito junto ao ACO e o algoritmo de otimização por nuvem de partículas.

O ACO novamente será responsável por povoar as gerações que são utilizadas a cada iteração, porém estes indivíduos serão separados em G conjuntos distintos. A atualização dos feromônios é realizada após gerar as soluções e os melhores indivíduos são selecionados para o cruzamento no GA. A operação de mutação dos indivíduos será realizada para cada cromossomo, probabilisticamente, segundo o conceito do recozimento simulado (2.8). Este processo de cruzamento de indivíduos é realizado para uma quantidade pré-definida de gerações, em seguida, uma comunicação entre as matrizes de feromônios é realizada, utilizando o conceito de nuvem de partículas. Cada uma das G colônias de formigas é tratada como uma partícula e uma atualização de feromônios global é realizada por meio das seguintes equações:

$$\tau_i(r,s) \leftarrow \tau_i(r,s) + v$$

$$v = 2R_1(\tau_{in}(r,s) - \tau_i(r,s)) + 2R_2(\tau_{gb}(r,s) - \tau_i(r,s)),$$

onde $R_1, R_2 \in [0,1]$ e τ_{gb} representa os feromônios da colônia que apresentou a melhor solução até então, i e in variam entre os índices das colônias de forma que $i \neq in \neq gb$. Após a comunicação entre colônias, uma nova população é gerada pelo ACO iniciando a próxima iteração.

O autor relata resultados muito bons, com uma solução média encontrada no máximo 3% acima da melhor solução conhecida em instâncias com menos de 600 vértices. Porém nada é relatado a respeito do tempo de execução, e o algoritmo parece exigir muitos recursos computacionais.

Mohsen (2016) faz uso do *elitist ant system* EAS combinado com cozimento simulado e um operador de mutação. Como o EAS utiliza apenas uma parte das formigas para o depósito de feromônios é comum a ocorrência de rápida convergência para um mínimo local, para prevenir isso, o algoritmo avalia a cada iteração o conjunto de soluções encontradas. Caso elas estejam muito concentradas em torno de uma solução, o operador de mutação será utilizado, caso as soluções estejam muito dispersas o cozimento simulado será aplicado. A mutação auxilia a exploração de novos caminhos, porém, muitas mutações afastam as formigas de quaisquer soluções que sejam consideradas boas, então, o cozimento simulado força novamente uma convergência. Após a aplicação da mutação ou do SA, uma busca local é realizada antes do depósito de feromônios e o início da próxima iteração.

A ideia do algoritmo de Mohsen então é manter as soluções próximas de um mínimo, mas não tanto que caminhos melhores que não estejam na vizinhança não possam ser levados em consideração. O autor relata que algumas das soluções obtidas são melhores que alguns algoritmos renomados da literatura.

Liao; Liu (2018) apostam na solução particionada da instância, fazendo uso do algoritmo *density peaks clustering (DPC)* combinado com ACO e k-opt.

O DPC é utilizado para separar conjuntos de vértices do problema, trata-se de uma técnica de clusterização assim como o k-médias, com a diferença que o DPC o faz com apenas uma iteração. Uma vez que os vértices são clusterizados o ACO é utilizado, para resolver o TSP em cada um dos conjuntos definidos anteriormente, duas as duas as soluções obtidas serão mescladas, o critério de seleção destas soluções será por proximidade dos *clusters*. Por fim, depois que todas as soluções se tornarem uma única, os algoritmos 2-opt e 3-opt são usados para verificar se há alguma melhoria para a solução.

Os resultados para esse algoritmo são muito bons, obtendo soluções com uma taxa média de erro de 1,85% comparado às melhores soluções conhecidas.

Yang *et al.* (2015) propõem um algoritmo que combina os conceitos do EAS com SA, além disso é realizada uma pequena alteração na matriz de feromônios ao fim do processo, com o objetivo de criar uma balanço entre exploração e convergência das soluções.

Após inicialização das variáveis, cada formiga constrói sua solução, realizando o depósito de feromônios após a seleção de cada vértice até que todas as formigas concluam seus caminhos. As melhores soluções passam por melhorias de SA e 2-opt, enquanto as demais soluções passam apenas pelo 2-opt, concluindo a fase de melhoria das soluções uma atualização global de feromônios será realizada. Por fim, são realizadas duas verificações, ambas serão avaliadas pela quantidade de iterações desde que a melhor solução foi encontrada, em um primeiro momento uma variação dos feromônios é realizada segundo a fórmula

$$\tau_{i,j} = \tau_{i,j} + r \cdot \mathsf{v}ar(t, \tau_{me}),$$

com $\tau_{i,j}$ sendo a quantidade de feromônios na aresta $(i.j)$, $r \in U(-1, 1)$ um valor aleatório e var a função de variação calculada por

$$\mathsf{v}ar(t, \tau_{me}) = \tau_{me} \cdot \frac{t - t_{stay}}{t_{max} - t_{stay}},$$

onde τ_{me} representa a quantidade média de feromônios na melhor solução encontrada, t é a iteração atual em execução, t_{stay} a iteração na qual foi encontrada a melhor solução e t_{max} a quantidade máxima de iterações a serem executadas pelo algoritmo. Num segundo momento, caso as variações de feromônios não consigam melhorar a melhor solução encontrada por muitas iterações, então a matriz de feromônios é reiniciada.

Gambardella *et al.* (2012) relatam que a combinação entre ACO e algoritmos robustos de pesquisa local pode não ser tão eficiente, considerando que a busca local cobre uma vasta vizinhança em torno de cada solução gerada o ACO acaba realizando um trabalho redundante no momento de construir as soluções.

A proposta então é construir cada solução de forma probabilística, na maior parte das vezes as formigas escolhem o próximo vértice a ser visitado copiando a melhor solução encontrada, quando isto não ocorre então o processo de decisão será realizado como comumente é feito no ACO, acelerando a etapa de construção de soluções. Antes de aplicar a busca local, é realizada uma lista tabu, nela estarão contidos os vértices que não serão considerados durante a etapa de busca local, estes serão os vértices da nova solução, cujo antecessor e o sucessor são comuns também na melhor solução encontrada, assim a busca será aplicada apenas aos trechos novos que forem construídos.

Na publicação, Gambardela utiliza a mesma estrutura de algoritmo para diversos problemas, e algumas adaptações são realizadas de acordo com o problema a ser trabalhado, para uma variação do TSP, a busca local utilizada é o 2.5-opt, que utiliza as mesmas operações do 2-opt e além disso avalia também se uma melhoria pode ser obtida ao reinserir um vértice em uma posição diferente da solução.

3.2 DEPÓSITO DE FEROMÔNIOS

O cerne das meta-heurísticas baseadas em colônias é a troca de informação entre as soluções encontradas, em geral são utilizados dois mecanismos, um deles incentiva novas soluções a serem parecidas com as melhores soluções já encontradas pela colônia, o outro mecanismo faz o contrário, desencoraja a construção de caminhos onde foram encontradas soluções ruins. No caso do ACO, apenas o primeiro mecanismo é utilizado originalmente, que é feito a partir do depósito de feromônios, este é o processo mais característico do ACO, logo, é natural também que muitas melhorias surjam ao alterar a forma com que as formigas utilizam os feromônios no algoritmo.

Uma alteração bem simples ao método de depósito de feromônios é proposta por Yang; Wang (2016). O algoritmo original do ACO conta apenas com o coeficiente de evaporação e uma quantidade menor no deposito de feromônios para que arestas ruins deixem de ser selecionadas com tanta frequência, durante a construção da solução. Yang então adiciona uma remoção de feromônios, a cada iteração, seguindo o modelo elitista, a melhor formiga fica responsável por adicionar feromônios nas arestas de sua solução, enquanto a pior formiga retira feromônios, como descrito pela Equação (3.1).

$$\Delta\tau_{ij} = \begin{cases} Q/L_{best} & \text{, se } (i,j) \in S_{best} \\ -Q/L_{worst} & \text{, se } (i,j) \in S_{worst} \\ 0 & \text{, caso contrário} \end{cases} . \tag{3.1}$$

S_{best} e S_{worst} representam as soluções da melhor e da pior formiga e L_{best} e L_{worst} seus respectivos valores *fit*. Note que o decréscimo de feromônios, realizado pela pior formiga, é relativo ao próprio valor *fit*, isto significa que a quantidade retirada de feromônios no início do processo não será tão grande quanto o acréscimo, preservando a busca local, mas ao fim do processo estas quantidades tendem a se igualar, uma vez que as formigas escolhem caminhos muito semelhantes, e o algoritmo será levado rapidamente a uma convergência.

Jun-man; Yi (2012) propõem duas alterações que obtiveram bons resultados. A primeira é a utilização de apenas uma formiga para atualização da matriz de feromônios, muitos algoritmos utilizam essa mesma estratégia, mas geralmente a melhor formiga é encontrada após comparar todas as soluções obtidas, Jun-man faz com que, durante a construção das soluções, a próxima formiga a escolher o próximo vértice a ser visitado,

seja sempre aquela que percorreu o menor caminho até então, desta forma, a primeira formiga a visitar todos os vértices é a que obteve a melhor solução naquela iteração. Esta alteração irá poupar tempo na etapa de construção de soluções.

A segunda alteração realizada muda o comportamento das formigas individualmente através da alteração dos parâmetros de relevância dos feromônios (α) e relevância heurística (β), utilizados no cálculo de probabilidades 2.9. Na proposta apresentada pelo autor, sempre que uma formiga é a primeira a completar o circuito na iteração, esta tem um incremento no valor de α, enquanto β sofrerá um declínio, assim, partindo dos parâmetros $\alpha = 1$ e $\beta = 5$, as formigas terão estes valores alterados até que fiquem $\alpha = 5$ e $\beta = 1$ e assim permanecerão até o fim do algoritmo. O objetivo desta alteração é tornar os feromônios depositados mais relevantes com o passar do tempo, forçando uma convergência das soluções, e diminuindo o tempo de execução.

O algoritmo também utiliza o método 2-opt em cada solução construída pelas formigas. Em testes com instâncias de até 200 vértices, esta melhoria superou o ACO em tempo de execução e qualidade de solução.

Ning *et al.* (2018) também falam sobre o custo computacional elevado e a tendência de uma estagnação prematura do algoritmo, então propõe duas alterações ao modelo original para amenizar estes problemas.

A primeira alteração está diretamente ligada ao depósito de feromônios, a fórmula de atualização de feromônios utilizada agora depende da melhor solução encontrada na iteração atual ($\Delta\tau_{ij}^{ibest}$) e da melhor solução encontrada até o momento ($\Delta\tau_{ij}^{best}$) e o cálculo é realizado pela Equação (3.2)

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \Delta\tau_{ij}^{best} + \Delta_{ij}^{ibest}, \tag{3.2}$$

onde o valor de $\Delta\tau_{ij}^{best}$ é descrito pela Equação (3.3)

$$\Delta\tau_{ij}^{best} = \begin{cases} \frac{1}{f(S_{best})}, \text{ se } < i,j > \in S_{best} \\ 0 \qquad\quad , \text{ caso contrário} \end{cases} \tag{3.3}$$

e $f(S_{best})$ é o valor *fit* do melhor caminho encontrado S_{best}. De modo análogo é calculado o valor de $\Delta\tau_{ij}^{ibest}$.

Esta atualização de feromônios contribui ainda mais para a atratividade de soluções boas, acelerando a convergência do algoritmo e também o processo de estagnação em um mínimo local. Pensando nisso é implementado um mecanismo que reconhece quando o algoritmo está se aproximando da estagnação, e então a matriz de feromônios é alterada pela Equação (3.4), amenizando o acúmulo de feromônios em um único caminho, e assim permitindo que novas soluções sejam exploradas

$$\tau_2 = \rho(\tau_{max} - \tau_1). \tag{3.4}$$

Os valores ρ, τ_1, τ_2 e τ_{max} representam respectivamente o fator de evaporação, quantidade de feromônios no caminho em estagnação, quantidade de feromônios após a amenização e quantidade máxima de feromônios permitida em uma aresta.

Outro algoritmo que faz uso da adaptação dos parâmetros com o decorrer do tempo é proposto por Yan *et al.* (2017), além dos parâmetros α e β os autores também sugerem métodos para dinamizar o coeficiente de evaporação (ρ) e a variação no depósito de feromônios Q.

Para os valores α e β, a cada iteração são tomadas as formigas responsáveis pela melhor e a pior solução obtidas na iteração, enquanto a melhor formiga da iteração recebe um incremento no valor de α, a pior formiga terá seus parâmetros substituídos pelos que foram utilizados pela formiga vitoriosa durante esta iteração, como mostra a Equação (3.5)

$$\begin{array}{l} \Phi_{winner}(t+1) = [\alpha_{winner}(t) + \Delta, \beta_{winner}(t)] \\ \Phi_{loser}(t+1) = [\alpha_{winner}(t), \beta_{winner}(t)] \end{array} . \tag{3.5}$$

O valor de Δ é definido no início da execução pelo usuário, e $\Phi_k(t)$ representa os valores de α e β armazenados pela formiga k na iteração t.

O coeficiente de evaporação é calculado de forma que este seja mais forte em arestas com maior acúmulo de feromônios, permitindo que mais caminhos distintos sejam explorados antes da ocorrência de uma estagnação

$$\rho = \frac{\tau_{ij}(t)}{\tau_{max} + \tau_{min}}. \tag{3.6}$$

Os valores τ_{max} e τ_{min} representam o maior e o menor acúmulo de feromônios permitido em uma aresta, nesta publicação os autores estipularam valores como sugerido por Sttzle; Hoos (1996).

A variação dos feromônios é calculada quase sempre da mesma forma, como descrito na expressão (3.7), onde Q é uma constante definida pelo usuário e L_k o valor *fit* da solução obtida pela formiga k

$$\Delta\tau_{ij}(t) = \begin{cases} Q/L_k, \text{ caso a aresta } (i,j) \text{ tenha sido escolhida na solução} \\ 0 \quad\quad , \text{ caso contrário} \end{cases} . \tag{3.7}$$

Neste algoritmo, o valor desta constante também será dinâmico e será calculado pela Equação (3.8), levando em consideração a melhor solução encontrada até o momento (L_{gb}), a melhor solução encontrada na iteração t ($L_k(t)$), e um limite superior para as soluções (UB)

$$Q = \frac{UB - L_k(t)}{UB - L_{gb}}. \tag{3.8}$$

O valor de UB é definido pelo algoritmo do vizinho mais próximo.

O valor dinâmico de Q interfere diretamente no depósito de feromônios, fazendo com que novos caminhos encontrados recebam imediatamente uma grande quantidade de feromônios em comparação às soluções que não são melhores do que as que já foram encontradas.

Uma das técnicas mais conhecidas para a resolução de problemas discretos em pesquisa operacional é o *backtracking*. Neste método as variáveis do problema são selecionadas uma a uma, de forma ordenada, até o momento em que todas as variáveis sejam escolhidas, se a solução obtida for um ótimo local ou não atender às expectativas de alguma forma o algoritmo refaz suas últimas escolhas, até obter o resultado desejado ou sanar todas as possibilidades de solução. Claramente é uma heurística de força bruta, portanto exige muito tempo para sua execução e não costuma ser aplicado desta forma em problemas com vasta quantidade de possíveis soluções, porém, o seu conceito é muito útil quando combinado a outras métodos de otimização.

Liu *et al.* (2011) tentam combinar o conceito de *backtracking* com o ACO, e faz isso através da matriz de feromônios. Ao definir os parâmetros, são adicionados dois valores N e M, de forma que $M > N$, também são contadas quantas iterações foram executadas desde que o algoritmo encontrou a melhor solução já obtida (S_{best}), a partir do momento que esta quantidade de iterações supere N será considerado que o algoritmo está preso em um mínimo local, e uma atualização global de feromônios será realizada de acordo com a Equação (3.9)

$$\tau_{ij}(t+1) = \tau_{ij}(t) - N \cdot \frac{Q}{L_{best}}, \tag{3.9}$$

onde L_{best} representa a distância percorrida pela solução S_{best}. Esta é a primeira etapa do *backtracking*, o objetivo é fazer com que o acúmulo de feromônios fique parecido com o que era quando S_{best} foi encontrado. Esta atualização de feromônios é aplicada novamente a cada N iterações sem melhorias em S_{best}. A segunda etapa do *backtracking* é utilizada quanto a quantidade de iterações ultrapassa M. Aplicando a regra de atualização (3.10) os feromônios retornam aos seus valores iniciais (τ_0), e quantidades menores serão depositadas nas arestas presentes em S_{best}, assim, o algoritmo é induzido a selecionar outras arestas ao construir suas soluções

$$\tau_{ij}(0) = \begin{cases} \tau_0/N, \text{ se } (i,j) \in S_{best} \\ \tau_0 \quad \text{, caso contrário} \end{cases}. \tag{3.10}$$

Esta reinicialização de feromônios também será repetida quantas vezes forem necessárias, até que o número máximo de iterações, que será o critério de parada do algoritmo, seja alcançado. Os testes efetuados mostram que o algoritmo chega a solução ótima em apenas 50 iterações, utilizando uma instância de 30 vértices.

Ao observar um circuito hamiltoniano, de um conjunto de vértices quaisquer, pode-se garantir que este não será o menor possível se houver duas arestas se

cruzando, também, é de se esperar, que arestas muito grandes não estarão presentes nele. Utilizando as duas características mencionadas anteriormente, Tuba *et al.* (2013) implementam um método de correção de feromônios, sempre observando quais são as arestas da melhor solução atual, que seriam as mais prováveis de não estarem na melhor solução global.

Todas as arestas que geram cruzamentos são consideradas altamente suspeitas de não estarem na solução global e recebem uma correção imediata na quantidade de feromônios depositados. As arestas (rs) consideradas suspeitas por serem grandes, são ranqueadas de acordo com o seu comprimento e podem receber uma correção nos seus feromônios de acordo com a probabilidade $P(rs)$ em (3.11)

$$P(rs) = \frac{RK - RankSusp(rs)}{2 \cdot RK}. \tag{3.11}$$

O valor RK representa o número máximo de arestas que serão consideradas para correção, $RankSusp(rs)$ é a ranque atribuído a aresta rs. O ranqueamento baseado apenas pelo comprimento não se mostra tão efetivo, pois acaba selecionando repetidamente as mesmas arestas, o autor propões então um método de ranqueamento baseando-se na quantidade de vezes que as arestas foram selecionadas como suspeitas anteriormente, de acordo com a Equação (3.12).

$$RankSusp(rs) = Rank(rs) \cdot ExSusp(rs), \tag{3.12}$$

com $Rank(rs)$ sendo o ranque da aresta rs baseado apenas em seu comprimento e $ExSusp(rs)$ o fator de correção do ranque baseado nas arestas selecionadas como suspeitas anteriormente, que é calculado pela Equação (3.13)

$$ExSusp(rs) = ExSusp(rs) \cdot \lambda \cdot \frac{len(rs)}{len(MaxEdge)}. \tag{3.13}$$

Inicialmente o valor de $ExSusp(rs)$ é definido como 1 para todas as arestas, o valor $\lambda \in (0, 1)$ será definido como parâmetro e $len(rs)$ e $len(MaxEdge)$ serão os comprimentos das arestas rs, considera suspeita, e $MaxEdge$, a maior presente na melhor solução atual.

A correção realizada será simplesmente o produto entre o valor $\delta \in (0, 1)$ pela quantidade de feromônios τ_{rs} presente na aresta rs. Então, a todas as arestas selecionadas por 3.11 será aplicada a Equação (3.14)

$$\tau_{rs} = \delta \cdot \tau_{rs}. \tag{3.14}$$

Sempre que uma nova solução superar a melhor até então encontrada, os valores de $ExSusp$ serão reiniciados com valores iguais a 1. Nos testes apresentados

na publicação, esta melhoria apresenta um erro médio, relativo a melhor solução global, menor em todas as instâncias testadas, quando comparado a colônia de formigas e nuvem de partículas.

Como já foi dito anteriormente, o depósito de feromônios representa a forma das formigas comunicarem entre si quais são as melhores arestas para escolher no circuito. Este é um método indireto de comunicação, uma vez que as formigas tomam suas decisões baseadas nas informações que provém de todas as outras formigas. Na natureza esta troca de informações também ocorre de forma direta quando a presença de uma formiga pode influenciar na decisão da outra. A melhoria proposta por Mavrovouniotis; Yang (2010) tem justamente o propósito de implementar esta comunicação direta entre as formigas no algoritmo MMAS. Inicialmente é definida uma forma de calcular a proximidade entre as formigas, esta será calculada pela Equação (3.15) e utiliza a quantidade de arestas comuns selecionadas pelas formigas para construir a vizinhança de cada uma delas

$$R_i = \left\{ ant_j \in P | 1 - \frac{CE_{ij}}{n} \leq T_r \right\}. \tag{3.15}$$

O alcance da vizinhança de cada formiga i (R_i) será definido a partir de um parâmetro $T_r < 1$, CE_{ij} representa a quantidade de arestas comuns nas soluções obtidas pelas formigas i e j ao percorrer os n vértices do grafo.

Após todas as formigas construírem suas soluções, a vizinhança de cada uma é calculada e uma troca direta de informação ocorre da seguinte forma: para cada formiga i, será selecionada uma formiga aleatória j em sua vizinhança e um vértice aleatório k em sua solução, as formigas i e j trocam entre si os vértices sucessores e antecessores do vértice k. Se o processo anterior leva a uma melhoria da solução anterior um depósito extra será realizado nas arestas alteradas segundo a Equação (3.16), caso haja uma piora, as alterações serão descartadas

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{(\tau_{max} - \tau_{min})(1 - w)}{n}. \tag{3.16}$$

O valor de w representa a relevância que as comunicações diretas terão no algoritmo ao atualizar os feromônios τ_{ij} das arestas (i, j) que forem admitidas nas trocas. Experimentalmente, o autor chegou a conclusão que $w = 0.5$ obtém melhores resultados em comparação a outros valores de w, também os melhores valores para T_r variam de 0.6 até 0.8. Os experimentos mostram também que esta melhoria não é tão rápida quanto o MMAS para convergir a uma solução definitiva, porém, obteve uma significativa melhora na consistência das soluções encontradas.

Apesar de muitos algoritmos alterarem de alguma forma as fórmulas de depósito de feromônio para obterem resultados, a maioria deles utiliza o comprimento

de cada caminho para calcular a variação de feromônio, apresentada na Seção 2.4.1. Desta forma é possível controlar os feromônios presentes em cada aresta, sempre sendo mais fortes em arestas presentes em caminhos melhores. Zhang; Feng (2012) fazem uma pequena alteração nesta variação de feromônios, utilizando o algoritmo elitista do ACO.

Primeiramente, a atualização de feromônios pode ocorrer em dois estágios, em um deles as r melhores formigas da iteração realiza depósito de feromônios, em outro estágio, somente a melhor formiga da iteração, ou a melhor formiga global, faz o depósito. O autor sugere dois métodos para definir qual estágio de atualização será utilizado, o primeiro utiliza apenas a quantidade de iterações realizadas até o momento, então, a partir do momento que este valor ultrapassa o definido pelo usuário, passa-se a utilizar o segundo estágio no lugar do primeiro, este se dá pelo cálculo do fator de convergência (cf)

$$cf = 2\left(\frac{\sum_{j=1}^{m} max\{\tau_{max} - \tau_{ij}, \tau_{ij} - \tau_{min}\}}{m \cdot (\tau_{max} - \tau_{min})} - 0.5\right). \tag{3.17}$$

O valor m representa a quantidade de formigas utilizadas, τ_{max} e τ_{min} os limites superior e inferior da quantidade de feromônios que podem ser acumulados em uma aresta, e o valor cf deve satisfazer $0 < cf < 1$. Portanto, a alteração de estágio tem a função de acelerar a convergência do algoritmo ao fim do processo.

A variação de feromônios utilizada é relativa ao ranque (ω) de cada solução e uma função de qualidade

$$\Delta\tau_{ij}^{\omega}(t) = \begin{cases} F(\omega)\rho\tau_{max}, \text{ se } (ij) \in S^{\omega} \\ 0 \qquad\qquad\quad , \text{ caso contrário} \end{cases}. \tag{3.18}$$

O ranque de cada solução é definido apenas pelo valor *fit* de cada solução e a função de qualidade F utiliza este ranque de alguma forma para ainda ter um certo controle sobre o depósito de feromônios. Um exemplo de função de qualidade é $F = 1/\omega$, garantindo que quantidades maiores de feromônios são depositadas em arestas mais promissoras.

Zhou; Wang (2012) apresentam um aprimoramento ao EAS com o uso de memória da última solução construída por cada formiga, desta forma as formigas fazem uma busca local mais forte antes de convergir para uma solução definitiva. A cada atualização global de feromônios, sempre será considerada a melhor solução encontrada até o momento, e, durante o cálculo de atualização, haverá um peso atribuído de acordo com o ranque de cada formiga para aquela iteração

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \epsilon \cdot \tau_{ij}^{G}(t+1). \tag{3.19}$$

Na Equação (3.19), $\tau_{ij}(t+1)$ representa o acúmulo de feromônios utilizado globalmente em cada aresta ij na próxima iteração, este é calculado pelo acúmulo da iteração anterior após aplicar a taxa de evaporação ρ somado com a variação de feromônios global $\tau_{ij}^{G}(t+1)$, que é dada pelas equações (3.20), (3.21) e (3.22). Também há um parâmetro $\epsilon \in (0,1)$ que pondera o quão relevante será o acúmulo de feromônios em relação a memória da formiga, durante a construção da solução

$$\tau_{ij}^{G}(t+1) = \sum_{r=1}^{w-1}(w-r)\cdot\Delta\tau_{ij}^{r}(t) + w\cdot\Delta\tau_{ij}^{gb}(t), \tag{3.20}$$

$$\Delta\tau_{ij}^{r}(t) = \begin{cases} \frac{Q}{L^{r}(t)}, \text{ se } (i,j)\in S^{r}(t) \\ 0 \quad \text{, caso contrário} \end{cases}, \tag{3.21}$$

$$\Delta\tau_{ij}^{gb}(t) = \begin{cases} \frac{Q}{L^{gb}(t)}, \text{ se } (i,j)\in S^{gb}(t) \\ 0 \quad \text{, caso contrário} \end{cases}. \tag{3.22}$$

A variação global será dada pela soma das variações das formigas de ranque 1 até r, com a melhor solução encontrada até o momento gb, um peso w irá intensificar o acúmulo de feromônios nas arestas presentes em soluções melhores. Por fim, a quantidade de feromônios que será utilizada pela formiga k para construção da solução da próxima iteração é feita pela Equação (3.23)

$$\tau_{ij}^{k}(t+1) = \tau_{ij}(t+1) + (1-\epsilon)\cdot\tau_{ij}^{I_k}(t+1). \tag{3.23}$$

O valor $\tau_{ij}^{I_k}(t+1)$ representa a memória da formiga, esta sofre evaporação completa ao fim de cada iteração, o que significa que seu valor dependerá apenas da solução anterior construída pela formiga e nada mais

$$\Delta\tau_{ij}^{I_k}(t) = \begin{cases} \frac{Q}{L^{k}(t)}, \text{ se } (i,j)\in S^{k}(t) \\ 0 \quad \text{, caso contrário} \end{cases}. \tag{3.24}$$

Após executar o algoritmo 100 vezes seguidas para 4 instâncias distintas, com valores de ϵ variando de 0.0 até 1.0, os melhores resultados foram alcançados quando $\epsilon = 0.125$. O algoritmo demonstrou melhoras nas soluções encontradas quando comparado ao EAS, algoritmo genético e nuvem de partículas, em instâncias de até 100 vértices. O algoritmo também costuma encontrar as soluções com menor número de iterações quando comparado ao EAS.

Como visto na seção anterior, é comum o uso de técnicas de mutações combinadas ao ACO pra aprimorar o algoritmo, mas Ratanavilisagul (2017) relata que os resultados obtidos destes algoritmos em particular podem ocorrer por uma exaustiva comparação das soluções obtidas pelas mutações com as construídas pelas próprias

formigas, isto é, como ocorrem mais mutações do que construções de soluções, é esperado que melhoras de soluções surjam eventualmente, além disto, o autor também fala que as mutações estarão sempre na vizinhança das soluções originais, e, por tanto, não lidarão tão bem com a convergência para um mínimo local.

Para lidar com as fraquezas mencionadas acima, Ratanavilisagul (2017) sugere uma mutação aplicada a matriz de feromônios. A ideia do algoritmo é um tanto quanto simples, sempre que as formigas encontrarem uma solução que seja melhor que a global até o momento, a matriz de feromônios desta iteração será gravada e um contador iniciado, se a quantidade de iterações executadas desde que a última melhor solução foi encontrada extrapolar o máximo permitido, que será inicialmente definido pelo usuário, então a matriz de feromônios sofrerá uma mutação. A mutação é uma simples substituição dos feromônios presentes nas arestas atuais pela quantidade presente na matriz de feromônios que foi salva, quando a melhor solução foi encontrada.

Esta mutação é aplicada na matriz de feromônios para tentar melhorar a busca por novos caminhos do algoritmo, pois como o algoritmo não encontra novas soluções, isto provavelmente significa que está preso em um mínimo local, então a mutação recupera a configuração dos feromônios quando o algoritmo não estava preso. A mutação ocorre apenas para uma parte da matriz de feromônios, na publicação (Ratanavilisagul, 2017) a proporção da matriz que sofre mutação tem um terço da quantidade de vértices presentes no problema. Experimentos mostram que o custo médio das soluções encontradas por este algoritmo é menor que os obtidos pelo ACO original e suas versões que usam mutações.

Todas as heurísticas apresentadas até o momento constroem soluções baseadas em fórmulas que levam em consideração a distância total percorrida pela solução, as distâncias individuais de cada aresta escolhida e as trocas de informações entre as soluções, assim ocorre com a maioria das heurísticas e técnicas de inteligências artificiais, porém existem técnicas de aprendizado de máquina onde o usuário interfere diretamente no processo (*interactive machine learning*). A ideia é baseada no fato de que uma inteligência artificial muitas vezes não é capaz de tomar decisões corretas em problemas muito complexos, problemas nos quais o cérebro humano pode facilmente perceber padrões e detalhes que levarão rapidamente a uma decisão quase sempre assertiva da situação. Holzinger *et al.* (2016) realizam experimentos utilizando o ACO com interferência humana, a função do usuário no algoritmo é, de tempos em tempos, avaliar a matriz de feromônios e identificar caminhos ruins que possuem quantidades altas de feromônio acumulado, assim como caminhos promissores que possuem poucos feromônios, a matriz será alterada imediatamente e as formigas se encarregarão de fazer uma busca local mais intensa.

Os resultados obtidos são bons, mas apesar disto também existem dúvidas

sobre quão benéfica é a interação humana na pesquisa operacional, pois o algoritmo pode ser influenciado a obter um mínimo local ou desviado de uma boa solução durante o processo.

3.3 PARALELISMO

O conceito de paralelismo pode ser descrito, de forma simples, como múltiplos processos sendo trabalhados simultaneamente para a execução de uma mesma tarefa. Esta ideia não é recente, mas ganhou muita força com o progresso da tecnologia, o surgimento de processadores multi-nucleares e o uso crescente de *clusters* para solução de problemas complexos.

Esta técnica é muito poderosa para diminuir o tempo de processamento, porém, não é tão simples dividir um algoritmo em partes menores, isto irá depender muito do algoritmo, pois, na maior parte das vezes, cada etapa de um algoritmo precisa de valores obtidos de outras etapas, então, quanto menos independentes forem estas, menos será proveitoso o uso de técnicas de paralelismo.

O ACO em si possui naturalmente processos independentes, uma formiga irá construir e refinar cada uma de suas soluções sem a interação com outras formigas, com exceção de um único processo, a atualização de feromônios, este deve ser realizado de forma sincronizada por todas as formigas. Pedemonte *et al.* (2011) fazem um estudo focado no uso de técnicas de paralelismo no ACO, revisando textos publicados entre 2001 e 2010, sugere categorizar estes algoritmos a partir do uso ou não, dentre outras características, de múltiplas colônias de formigas e a troca de informação, ou não, entre as formigas e colônias.

Gülcü *et al.* (2018) propõem uma aplicação de paralelismo ao ACO que utiliza o modelo de múltiplas colônias com cooperação entre estas, logo, cada computador será responsável por processar uma colônia, e um destes, o mestre, será responsável por coordenar todo o processo. Cada colônia utiliza o mesmo modelo de ACO com mesmos parâmetros, a única diferença é a comunicação. A cada I iterações o mestre recebe de cada colônia a melhor solução encontrada até o momento, após comparar todas as soluções o mestre possui a melhor solução global. Esta solução será repassada para cada uma das colônias com o intuito de evitar que colônias fiquem presas a um mínimo local.

É muito comum que algoritmos tenham seus pontos fortes e fracos, o mesmo acontece entre as variações do ACO, a quantidade e distribuição dos vértices ou o problema ser simétrico ou não, são características decisivas para determinar a eficiência de um algoritmo ao buscar soluções, mas é difícil saber qual o melhor método a se utilizar. Com este raciocínio, Lizárraga *et al.* (2013) propõem um algoritmo de múltiplas

colônias, onde cada colônia utiliza um dos três métodos propostos (AS, EAS e MMAS). Após cada iteração, as colônias compartilham as melhores soluções encontradas e a pior colônia será marcada, se durante as próximas iterações esta colônia ainda obtiver as piores soluções ela abandona o método escolhido. Com o decorrer do algoritmo os piores métodos são deixados de lado, restando apenas um ao final. Vale ressaltar que como a comunicação entre colônias é apenas a comparação de soluções, muitos outros métodos podem ser utilizados simultaneamente, tornando o algoritmo mais forte.

Uchida *et al.* (2012) utilizam a técnica de paralelismo, através de placas gráficas (GPUs), para acelerar os processos de construção de solução e depósito de feromônios. Partindo do processo de construção, o autor apresenta três métodos para se calcular as somas de probabilidades mostradas na Equação (2.9).

O primeiro método, que é o mais comum, faz o uso de uma lista tabu, implementada a partir de um vetor. A técnica para gerir esta lista tabu é simples, ao incluir um vértice na solução parcial, busca-se na lista o elemento que possui valor igual ao índice do vértice incluso, este elemento troca de lugar com o último elemento da lista que será removido desta logo em seguida.

O mesmo pode ser realizado de outras formas utilizando outras estruturas como listas encadeadas, cada uma terá suas vantagens e desvantagens, mas estas estruturas geralmente trazem também um peso computacional. Utilizar um vetor como lista tabu provavelmente será a forma mais rápida de implementação, o ônus é que a lista ficará embaralhada com o decorrer do processo.

Como o maior peso da construção de uma solução no ACO vem do cálculo da equação 2.9, a lista tabu auxilia a fazer apenas os cálculos necessários, porém a lista não será eficiente no início do processo, onde poucos vértices são excluídos.

O segundo método utiliza um vetor no lugar da lista tabu, em cada índice do vetor haverá um valor binário para controlar se o vértice que corresponde aquele índice já está incluso na solução ou não

$$u_j = \begin{cases} 0, \text{ caso o vértice } j \text{ já tenha sido visitado} \\ 1, \text{ caso contrário} \end{cases} . \tag{3.25}$$

A partir deste vetor são calculadas as somas das probabilidades de uma formiga se mover do vértice i para o j, utilizando as informações heurísticas e acúmulo de feromônios, e as informações armazenadas no vetor q, como mostra a Equação (3.26)

$$q_j = \sum_{s=0}^{j} [\tau(i,s)]^{\alpha} \cdot [\eta(i,s)]^{\beta} \cdot u_j, \tag{3.26}$$

onde $0 \leq j \leq n-1$, com n sendo a quantidade de vértices da instância. Após o cálculo deste vetor de probabilidades um número aleatório $r \in (0, q_{n-1})$ é gerado, e o vértice

k será o próximo a ser visitado na solução se satisfaz $q_{k-1} < r \leq q_k$. Efetivamente falando, nada está sendo feito de diferente, quando comparado ao ACO original, mas a forma com que é feito é o mais relevante, cada parcela de (3.26) pode ser calculada individualmente antes de realizar a soma que irá gerar o vetor q, e q é um vetor ordenado, então o espaço de busca pode ser particionado múltiplas vezes para realizar uma busca em paralelo.

Apesar deste método incluir vértices já visitados em seus cálculos, o uso de técnicas de paralelismo compensa a criação e gestão de uma lista tabu, tornando este método muito mais efetivo com quantidades maiores de vértices não visitados.

O último método utilizado leva em consideração o fato de a atualização de feromônios ocorrer apenas após a construção das soluções, então os cálculos de probabilidades podem ser realizados antes mesmo do início da construção de cada solução. Uma matriz q' é gerada de acordo com (3.27)

$$q'(i,j) = \sum_{s=0}^{j} [\tau(i,s)]^{\alpha} \cdot [\eta(i,s)]^{\beta}. \tag{3.27}$$

O cálculo é realizado da mesma forma que em (3.26), mas considerando que nenhum vértice foi visitado. Feito isso, considerando que uma formiga se encontra no vértice i e decidirá qual o próximo a visitar, basta observar a linha i da matriz q'. Gerando um valor aleatório $r \in (0, q'(i, n-1))$, busca-se o valor k, que satisfaz $q(i, k-1) < r \leq q(i,k)$, este será o próximo vértice a ser visitado se ainda não estiver na solução.

Então este terceiro método não utiliza qualquer forma de controle dos vértices visitados, isto só será verificado ao fim do processo, o que significa que o algoritmo pode ficar preso durante muito tempo caso a maioria dos vértices já tenham sido visitados.

Os três métodos apresentados têm eficiências dependentes apenas da quantidade de vértices que foram visitados, além disto cada um deles atua de forma mais ágil em etapas diferentes do processo, então Uchida faz uma mescla destes três métodos, com uma pequena verificação é possível saber qual dos métodos será o mais rápido para decidir o próximo passo de cada formiga.

Para realizar o depósito de feromônios em paralelo, o autor opta por denotar cada solução no formato de vetor, onde o valor j do índice i significa que a aresta (i,j) foi selecionada na solução. Neste formato, o primeiro valor de cada solução gerada será o vértice sucessor de 0 no circuito, então eles serão os responsáveis pelas alterações referentes a linha 0 da matriz de feromônios. Os múltiplos núcleos verificam em paralelo quais os valores presentes no índice i de cada solução e realizam os cálculos de acordo com (2.11), os resultados serão armazenados em um vetor compartilhado. Após todas

as soluções serem analisadas para o índice i, ocorre a evaporação dos feromônios e a soma com o vetor compartilhado, para a mesma linha. O processo se repete para o índice $i + 1$, como ilustra a Figura 22. Se o problema for simétrico, este processo não está considerando a aresta (i, j) como igual a (j, i), para contornar esta situação, após a evaporação, divide-se pela metade a quantidade remanescente de feromônios, e ao fim do processo, somam-se os valores τ_{ij} e τ_{ji} presentes na matriz de feromônios. Este processo restabelece o acúmulo de feromônios pós evaporação, unindo os novos depósitos em arestas simétricas.

Com estas alterações Uchida consegue consegue acelerar o tempo de processamento do ACO em até 43 vezes, testando em uma instância com mais de 1000 vértices.

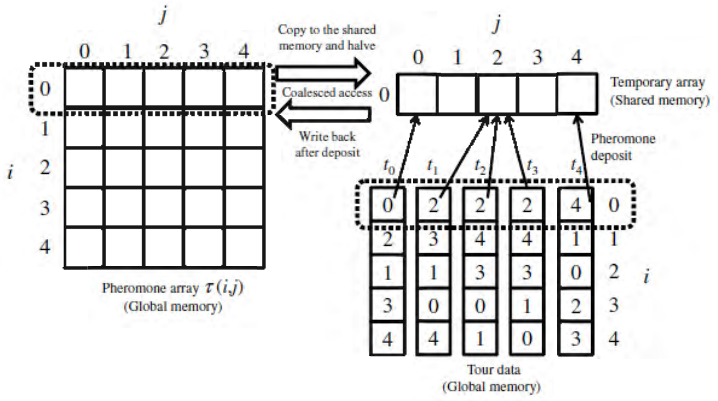


FIGURA 22 – Exemplo depósito de feromônios em paralelo. Fonte: Uchida (2012, p.100).

3.4 TOMADA DE DECISÃO

No ACO original e a maioria de suas melhorias, o processo de construção das soluções é probabilístico utilizando o método da roleta. Este método, na realidade é tão forte que é utilizado em muitas outras meta-heurísticas, porém, também há melhorias que fazem uso de outras técnicas ou alterações deste método, para obter resultados melhores ou adaptar o algoritmo à melhoria proposta. O objetivo desta seção é descrever alguns dos métodos propostos, que alteram a forma com que as formigas tomam decisões ao construir suas soluções.

Wang *et al.* (2012) propõem o uso de memória diretamente no processo de decisão, durante a construção de cada solução, juntamente as regras probabilísticas originais do ACO.

Durante o processo de construção, são utilizadas outras duas soluções, a memória da última solução obtida pela formiga ($S_{ms} = o_1, o_2, \ldots, o_i, o_{i+1} \ldots, o_j, \ldots, o_n$) e uma solução temporária (S_{ts}) que será alterada de acordo com o processo de construção. O vértice inicial será tomado de forma aleatória gerando uma solução parcial

$S_{ps} = (p_1)$, o próximo vértice a ser selecionado seguirá a regra de probabilidade do ACO e há um incremento na solução parcial, $S_{ps} = (p_1, p_2)$. Neste momento uma comparação é realizada com S_{ms}, se a memória possui o novo arco construído na solução parcial, o processo segue e um novo arco será adicionado pela regra de probabilidade, caso a memória não possua este arco, suponha $p_1 = o_i$ e $p_2 = o_j$, então uma solução temporária será gerada, sugerindo que a memória inclua este novo arco, simplesmente trocando a ordem dos vértices, ou seja, $S_{ts} = (o_1, o_2, \ldots, o_i, o_j, \ldots, o_{i+1}, \ldots, o_n)$. Caso o valor *fit* da solução temporária (L_{ts}) seja menor do que o que foi obtido na última iteração pela formiga (L_{ms}), então o processo de construção termina e a solução construída por esta formiga será a solução temporária, caso contrário, a memória da formiga será substituída pela solução temporária ($S_{ms} \leftarrow S_{ts}$) e o processo se repete ao selecionar o próximo vértice.

É importante fazer a substituição da solução na memória pela solução temporária, para que o processo admita a seleção de arcos que sejam localmente ruins, mas que possam melhorar a solução final obtida, caso contrário o processo seria muito semelhante ao 2-opt.

Apesar de a memória S_{ms} se alterar durante o processo de construção, o valor L_{ms} não será atualizado, caso contrário o processo de construção poderia ser forçado a acabar com uma solução pior que a encontrada na iteração anterior, e neste caso não haveria uma convergência.

Hlaing; Khine (2011) apresentam um aprimoramento do ACO que se baseia em duas alterações, a criação de uma lista de candidatos de vértices e uma adaptação dinâmica do parâmetro β.

A lista de candidatos é de fácil implementação, para cada vértice serão relacionados os vértices que estiverem mais próximos a este, no trabalho de Hlaing cada lista possui o tamanho igual a um quarto da quantidade de vértices presentes na instância. Durante a construção de cada solução será dada prioridade de seleção aos vértices que estiverem presentes na lista de candidatos, outros vértices fora da lista poderão ser selecionados caso todos os vértices da lista já estejam na solução parcial.

Para realizar a adaptação do parâmetro de relevância heurística, o autor calcula o nível de entropia (E') presente na matriz de feromônios, através das equações (3.28) e (3.29)

$$E(X) = -\sum_{i}^{r} P_i \log P_i, \tag{3.28}$$

$$E' = 1 - \frac{E_{max} - E_{current}}{E_{max}}. \tag{3.29}$$

O valor r é a quantidade de arestas presentes na instância, no caso simétrico

este valor será $r = n(n-1)/2$, e P_i é a probabilidade de ocorrência de uma determinada trilha na matriz de feromônios. O valor de entropia máximo (E_{max}) ocorre no começo do processo, quando o acúmulo de feromônios é igual em todas as arestas e portanto $P_i = 1/r$, o que implica $E_{max} = \log r$.

Por fim, após o cálculo da entropia da matriz de feromônios, é atualizado o valor de β de acordo com (3.30)

$$\beta = \begin{cases} 5, X < E' < 1 \\ 4, Y < E' < X \\ 3, Z < E' < Y \\ 2, 0 < E' < Z \end{cases} . \tag{3.30}$$

Os valores de X, Y e Z são definidos pelo usuário de forma heurística ou de acordo com o tamanho da instância.

Quanto mais dispersa é a matriz de feromônios mais as formigas tem informações sobre caminhos promissores do problema, e passa-se a dar maior valor à quantidade de feromônios depositados nestas arestas durante a construção de cada solução. O algoritmo proposto foi comparado ao AS, e se mostrou eficiente, encontrando a solução ótima em instâncias de até 140 vértices.

Um algoritmo bem mais teórico é apresentado por Bai *et al.* (2013), nele são realizadas várias alterações, inclusive na fórmula de probabilidade, onde, no lugar do valor heurístico, é utilizado o custo residual ($\bar{c}_{ij}$) relacionado a cada aresta, como apresentado em (3.31). Os valores u_i e v_j, são as variáveis duais relacionadas à solução do problema de designação com os custos c_{ij}

$$\bar{c}_{ij} = c_{ij} - u_i - v_j. \tag{3.31}$$

A fórmula de probabilidade a ser utilizada para construção de cada solução é dada por (3.32). Nesta equação $w(s^p(r), k)$ representa o peso relativo a seleção do vértice k como sendo o próximo a ser visitado na solução parcial s^p que termina em r, e $J_{adj}(s^p(r))$ uma vizinhança reduzida dos possíveis vértices a serem selecionados a seguir

$$p_{adj}(k/s^p(r), s^p) = \begin{cases} \frac{\tau_{rk}[w(s^p(r),k)]^\beta}{\sum_{u \in J_{adj}(s^p(r))} \tau_{ru}[w(s^p(r),u)]^\beta}, \text{ se } k \in J_{adj}(s^p(r)) \\ 0 \qquad\qquad\qquad\quad , \text{caso contrário} \end{cases} . \tag{3.32}$$

O peso é calculado pelas equações (3.33) e (3.34), com ΔG sendo a diferença entre o limite superior, dado pela melhor solução global $f(s^{gb})$, e o limite inferior, dado pela solução do problema de designação Z^*_{AP}, e w_{min} um parâmetro

$$w(s^p(r), k) = max \left\{ 1 - \frac{\sum_{(i,j) \in s^p(k)} \bar{c}_{ij}}{\Delta G}, w_{min} \right\}. \tag{3.33}$$

$$\Delta G = f(s^{gb}) - Z^*_{AP} \tag{3.34}$$

O conjunto $J_{adj}(s^p(r))$ é provavelmente o mais relevante para o algoritmo e seu desempenho, a sua construção está baseada no Teorema 1.

Teorema 1 *Dada uma solução parcial $s^p = \{i_1, i_2, \ldots, i_p\}$ e uma solução factível s, se $f(s) - Z^*_{AP} \leq \sum_{(i,j)\in s^p} \bar{c}_{ij}$ é satisfeito, então a seguinte inequação prevalece*

$$f(s) \leq Z^*(s^p). \tag{3.35}$$

Ou seja, o Teorema 1 permite prever quando um vértice não irá obter soluções melhores que a melhor global, e por isso o conjunto $J_{adj}(s^p(r))$ exclui estes vértices, como mostra (3.36)

$$J_{adj}(s^p(r)) = J(s^p(r)) - \left\{ k | \sum_{(i,j)\in s^p(k)} \bar{c}_{ij} \geq \Delta G \right\}. \tag{3.36}$$

$J(s^p(r))$ é simplesmente o conjunto dos vértices que não estão na lista tabu. Este algoritmo se mostra incrivelmente eficiente, obtendo soluções com erros menores que 0.1% em relação ao ótimo, também alcançou tempos de execução menores que 1 segundo, em instâncias de mais de 400 vértices.

Escario *et al.* (2015) detalham como funciona o *Ant Colony Extended* (ACE), um algoritmo muito promissor que supera, em diversos casos, os resultados obtidos por outros métodos já bem estabelecidos da literatura, como o ACS e MMAS. O algorimo é bem complexo para ser detalhado em poucos parágrafos, então aqui será descrita apenas os conceitos mais relevantes para a construção das soluções, que é o foco desta seção.

No ACE as formigas ainda utilizam o método da roleta para tomar suas decisões na construção das soluções, como descrito na Equação (3.37), porém, o valor da informação probabilística $\tau^{q_n}_{u_j}$ utilizada será diferente dependendo da formiga

$$P_\tau(u_i|q_n) = \frac{\tau^{q_n}_{u_i}}{\sum_{j\in N_\tau} \tau^{q_n}_{u_j}}. \tag{3.37}$$

Cada formiga será classificada como patrulheira ou forageira, ao calcular a probabilidade desta se mover do vértice q_n para o vértice u_j, dentre os possíveis vértices N_τ que não estão na lista tabu. As formigas forageiras necessariamente utilizam somente as informações contidas na matriz de feromônios para tomar suas decisões, enquanto que as formigas patrulheiras podem utilizar ou as informações

heurísticas ou novamente as informações dos feromônios, então, diferente do algoritmo original, uma única formiga não irá utilizar as duas informações simultaneamente para tomar cada decisão.

As formigas patrulheiras definem se utilizam informações heurísticas ou de feromônios de acordo com duas equações. Quando a decisão anterior foi tomada com base em feromônios, a equação a ser utilizada é (3.38), onde χ é um valor tomado aleatoriamente, NS o número de vértices que ainda serão selecionados pela formiga e χ um parâmetro. No segundo caso, a decisão é tomada com base heurística e utiliza-se a Equação (3.39), com γ_2 sendo também um parâmetro

$$\begin{cases} \chi < 1 - (1 - \gamma_1)^{1/NS} & \rightarrow P_\eta \\ \text{caso contrário} & \rightarrow P_\tau \end{cases}, \tag{3.38}$$

$$\begin{cases} \chi < \gamma_2 & \rightarrow P_\eta \\ \text{caso contrário} & \rightarrow P_\tau \end{cases}. \tag{3.39}$$

A probabilidade P_η, que é definida com base heurística é calculada utilizando o inverso da distância $d(c_i, c_k)$ entre o nó onde a formiga está atualmente (c_i), para cada um dos nós (c_k) que não estão na lista tabu (V) , como descrito nas equações (3.40) e (3.41)

$$P_\eta(u_i|q_n) = P_\eta(c_k|(c_i, c_j)) = \frac{\eta((c_i, c_j), c_k)}{\sum_{r \in N_\eta} \eta((c_i, c_j), c_r)}, \tag{3.40}$$

$$\eta((c_i, c_j), c_k) = \begin{cases} d(c_j, c_k)^{-\beta}, \text{ se } c_k \notin V \\ 0 \qquad\qquad , \text{ se } c_k \in V \end{cases}. \tag{3.41}$$

O ACE conta também com métodos de controle de população, estes controlam a quantidade de formigas forageiras e patrulheiras presentes em cada iteração e assim ponderam um equilíbrio entre busca e convergência de solução.

Neste capítulo foram ressaltados quatro pontos do ACO que são comumente alterados ao buscar melhorias na performance do algoritmo, mas claramente estas não são as únicas formas de se alterar o algoritmo. No restante deste trabalho é descrito o funcionamento de uma alteração diferente das anteriores, focada no posicionamento de cada formiga, também serão realizados testes para verificar as melhores configurações e os resultados obtidos.

4 MÉTODO PROPOSTO

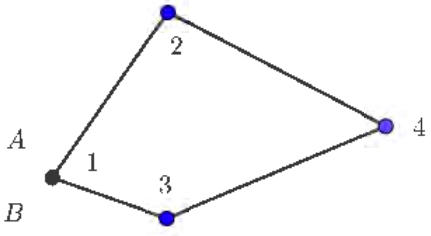
Uma iteração do algoritmo de colônia de formigas é representada basicamente pela construção do circuito de cada formiga. Sequencialmente, uma formiga de cada vez, constrói sua solução, e somente depois que todas terminarem este processo é realizada a atualização de feromônios. Este processo se distancia do comportamento natural das formigas, pois representa as formigas esperando o regresso de todas as outras ao formigueiro para poder retomar a busca por alimentos.

O algoritmo proposto tem o objetivo de representar de forma mais realista o método de busca de alimentos das formigas, dando maior ênfase à movimentação e posição de cada formiga. Sempre que uma formiga selecionar o próximo vértice a visitar em seu trajeto, é gravada a distância que ela deve percorrer para chegar até ele, a formiga que tem a menor distância a percorrer será a próxima a tomar uma decisão (escolher o próximo vértice). Assim que uma formiga completar seu circuito, iniciará imediatamente a construção do próximo, sem a necessidade de esperar que as outras formigas completem seus respectivos circuitos. Fazendo isso, espera-se que as formigas consigam acumular mais rapidamente os feromônios em caminhos mais curtos.

A implementação do algoritmo depende fundamentalmente do controle das distâncias percorridas por cada formiga, um vetor ($distancia$) é definido para cumprir esta função. Neste vetor serão armazenadas as distâncias que cada formiga deve percorrer para alcançar o próximo vértice escolhido em sua solução. Ao início de cada iteração toma-se o valor mínimo deste vetor (min), e cada entrada do vetor $distancia$ é subtraída por min, representando uma movimentação de todas as formigas em direção ao próximo vértice, todas percorrendo a mesma distância. Depois disto, pelo menos uma das formigas terá seu valor anulado no vetor $distancia$, simbolizando que ela chegou ao vértice destino e irá tomar a próxima decisão. As etapas de seleção de vértices e atualização de feromônios são similares as utilizadas no ACO original.

Para auxiliar a compreensão, a Figura 23 ilustra uma situação onde duas formigas, A e B, saem do mesmo vértice (1), com A indo em direção do vértice 2 e B em direção ao vértice 3. O comprimento das arestas, que ligam estes vértices, é repassado para o vetor $distancia$. A partir do momento que a formiga escolhe o próximo vértice do circuito, também faz o depósito de feromônios sobre a aresta que liga este vértice com o vértice no qual está posicionada.

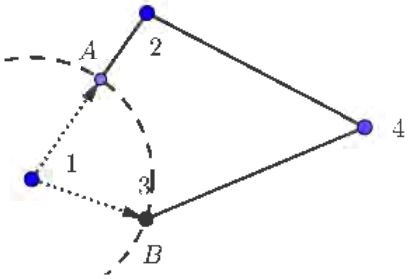
Ao avaliar os valores contidos no vetor $distancia$, nota-se que a formiga B está mais próxima do seu vértice alvo (3), então ambas as formigas se movimentam o



$distancia$	
A	B
5	3

FIGURA 23 – As formigas decidem o próximo vértice da solução, a distância é salva no vetor $distancia$. Fonte: Autor.

suficiente para que B alcance este vértice. Uma vez que B chega ao vértice 3, escolhe 4 para ser o próximo no circuito, atualizando o vetor $distancia$ na sequência com o comprimento da aresta $(3, 4)$, e também realizando o depósito de feromônios, como ilustrado na Figura 24.



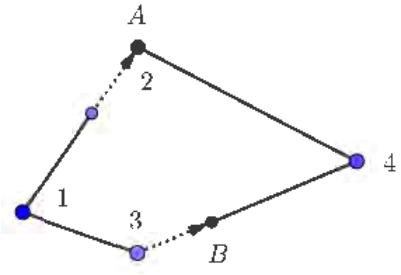
$distancia$	
A	B
2	0

↓

$distancia$	
A	B
2	5.88

FIGURA 24 – A formiga B chega ao vértice 3 e escolhe o próximo, alterando o vetor $distancia$. Fonte: Autor.

Para a próxima iteração, a formiga A percorreu parcialmente a aresta $(1, 2)$, e a parte restante é o menor valor do vetor $distancia$, então ambas as formigas se movimentam novamente, até que A alcance o próximo vértice, como ilustra a Figura 25.



$distancia$	
A	B
2	5.88

↓

$distancia$	
A	B
0	3.88

FIGURA 25 – As formigas continuam seus trajetos do ponto em que pararam na iteração anterior. Fonte: Autor.

O processo acima ilustrado se repete até que alguma formiga visite todos os vértices do grafo, e quando isso ocorre, sua solução é armazenada caso seja menor

que a melhor solução encontrada até o momento. Antes mesmo de começar a próxima iteração, iniciará um novo circuito, sem esperar que as outras formigas completem os seus.

Por causa das alterações propostas, não há mais um momento de atualização global de feromônios, então é necessária também uma adaptação em relação a evaporação. A aplicação da evaporação ocorrerá da mesma forma que o ACO original, o produto do coeficiente de evaporação ρ pela quantidade de feromônios de cada aresta τ_{ij}, a única correção a ser feita é o momento em que ela deverá ocorrer, então será reservada uma posição do vetor $distancia$ para controlar especificamente esta ocorrência, resta definir quanto tempo deve passar entre cada evaporação.

Intuitivamente falando, a evaporação deverá auxiliar o algoritmo a convergir para soluções melhores, logo, o intervalo de tempo entre evaporações não pode ser tão curto que não permita o acúmulo de feromônios nas arestas, mas também, não pode ser tão longo de forma que circuitos ruins não percam visibilidade com a dissipação de seus feromônios. Propõe-se então que este intervalo varie com o decorrer do algoritmo, sendo proporcional à melhor solução encontrada até o momento, esta proporcionalidade será controlada pelo parâmetro γ.

Outra pequena alteração ocorre em relação a quantidade de feromônios depositados. Inicialmente o algoritmo proposto foi executado com valores constantes de $\Delta\tau_{ij}$ aplicados a cada atualização, mas o mesmo não converge utilizando este método. Então é necessário ampliar a quantidade de feromônios depositados em arestas mais promissoras, e acelerar a convergência do algoritmo. Resultados melhores são encontrados ao utilizar uma fórmula parecida com a proposta no *ant cycle* (2.11), a única diferença é que no lugar de L^k, que representa o comprimento total da solução encontrada, será utilizado o valor l^k, o comprimento percorrido pela formiga k até o momento. Esta alteração é necessária, pois deseja-se realizar o depósito de feromônios no ato da escolha do próximo vértice. A variação de feromônios então será dada por (4.1)

$$\Delta\tau_{ij}^{k} = Q/l^{k}. \tag{4.1}$$

A probabilidade de escolha de cada nó ainda será dada pela Equação (2.4.1), o método de seleção original do ACO

$$p_{ij}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{\forall j \in P}[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}} & \text{se } j \in P \\ 0 \text{ caso contrário} \end{cases} . \tag{4.2}$$

Como o depósito de feromônio ocorre logo na escolha do próximo nó, a visibilidade de cada aresta também será atualizada neste momento, assim como quando ocorrer evaporação de feromônios.

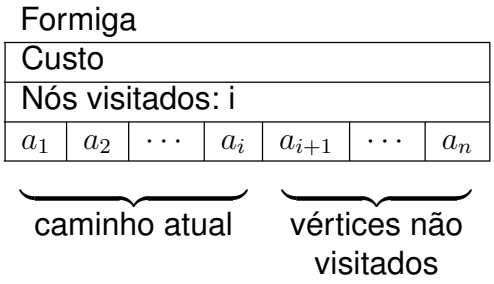


FIGURA 26 – Exemplo da estrutura utilizada. Fonte: Autor.

Por fim, para auxiliar a manipulação do programa será criada uma estrutura para a geração de cada formiga. Esta será composta por três informações: o custo total l^k da solução até então construída, a quantidade de vértices i já visitados, e um vetor v de n posições com valores de 1 até n, considerando n a quantidade de vértices na instância. Neste vetor será armazenada a solução construída por cada formiga, e também implementada a lista tabu, para isso, sempre que um novo vértice x é selecionado o valor de i é incrementado, o comprimento da aresta (v_i, x) é adicionado a l^k (caso $i = n - 1$ também deve ser somada a aresta que fecha o circuito (x, v_1)), e busca-se em v o índice j tal que $v_j = x$, por fim uma simples troca é realizada $troca(v_{i+1}, v_j)$. Desta forma, os primeiros i valores de v representam a solução parcial construída até então, e os outros $n - i$ valores são vértices a serem visitados por esta formiga, logo, tanto a solução quanto a lista tabu são atualizados com uma simples operação de troca, sem necessidade de alterações no tamanho das listas alocadas. Um exemplo é ilustrado na Figura 4.

Considerando um problema com n vértices e uma quantidade m de formigas operando, a determinação da próxima ação será encontrar o valor mínimo no vetor $distancia$ que é feito em $O(m)$, em seguida ocorrerá uma dentre três situações. Na primeira ocorre uma evaporação seguida de uma atualização completa da matriz visibilidade, que possui $O(2n^2)$. A segunda situação ocorre quando uma formiga encerra seu trajeto, a única operação trabalhosa a ser realizada neste caso é a geração de uma nova formiga, mesmo assim é uma operação linear de $O(n)$. Na última situação a formiga deve escolher o próximo nó a visitar, novamente os cálculos são bem específicos e, basicamente, dependendo do cálculo de probabilidades que, na pior das situações, possui $O(2(n-1))$. Portanto, no pior dos casos, uma iteração desta adaptação deve ocorrer em $O(2n^2 + m)$, mas esta deverá ser executada pelo menos n vezes para que uma formiga complete o seu caminho e o critério de parada reavaliado, levando a $O(2n^3 + mn)$.

Este será o algoritmo utilizado para realizar todos os testes presentes na próxima seção, assim como a comparação de resultados com o próprio ACO.

Algoritmo 12: Método proposto

```
Iniciar variáveis;
Enquanto Condição de parada não satisfeita fazer
    Definir próxima ação
    Se próxima ação for evaporação então
        Multiplicar a matriz de feromônios por ρ
        Atualizar matriz visibilidade
    fim
    caso contrário
        Se formiga completou o caminho então
            Depositar feromônios do nó atual para o inicial por 4.1
            Atualizar visibilidade do nó atual para o inicial
            Se este trajeto for o melhor já encontrado então
                Gravar o trajeto atual
            fim
            Reiniciar formiga atual
        fim
        caso contrário
            Determinar próximo nó a ser visitado por 2.4.1
            Depositar feromônios do nó atual para o próximo por 4.1
            Atualizar visibilidade do nó atual para o próximo
            Adicionar próximo nó ao fim do trajeto atual
        fim
    fim
fim
Retornar melhor circuito encontrado
```

4.1 ANÁLISE DE PARÂMETROS

A adaptação do ACO aqui proposta possui os mesmos parâmetros do algoritmo original e um adicional que é o intervalo entre evaporações. Para verificar qual deve ser a melhor configuração de parâmetros para o algoritmo serão realizadas diversas execuções do programa em duas instâncias tomadas do TSPLib (Discrete and Combinatorial Optimization, 2020). A primeira é o berlin52, uma instância pequena e com pontos uniformemente distribuídos (Figura 27), a segunda será o tsp225, uma instância com maior quantidade de pontos colocados propositalmente alinhados para formar a sigla “TSP” (Figura 28), o tamanho da instância ainda permite realizar vários testes para cada configuração de parâmetros.

Também será considerado que os problemas são simétricos, o que significa que o depósito de feromônios τ_{ij} na aresta que liga os nós i e j também ocorrerá para τ_{ji}.

O programa será executado em um computador comum, com sistema operacional Windows 10, com 8Gb de memória RAM, e um processador Intel i5-6400, 2,7 GHz. A linguagem de programação utilizada será VB.NET, por simples conveniência do

FIGURA 27 – Disposição de pontos na instância berlin52. Fonte: autor.

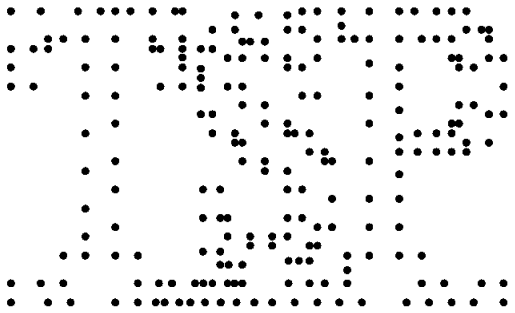


FIGURA 28 – Disposição de pontos na instância tsp225. Fonte: autor.

autor.

Os primeiros testes efetuados ponderam bons valores para os parâmetros α e β, estes são os parâmetros mais impactantes para o ACO, pois estão fortemente relacionados ao cálculo de probabilidades. Dada a relevância destes parâmetros, esta primeira bateria de testes irá abordar diversas combinações entre os parâmetros, são utilizados 5 valores distintos para α, sendo estes $\{0, 0.5, 1, 2, 5\}$, enquanto o conjunto de possibilidades para β compreende 6 valores, que são $\{0, 1, 2, 5, 10, 20\}$.

Os resultados dos testes estão apresentados nas Tabelas 1 e 2, onde cada célula contém o erro percentual da melhor solução encontrada e da média das soluções encontradas quando comparadas à melhor solução conhecida (*best know solution - BKS*) de cada uma das instâncias. Os valores BKS para as instâncias testadas são 7542 e 3919, para as instâncias berlin52 e tsp225 respectivamente. O último valor de cada célula é referente ao tempo médio utilizado em cada execução do algoritmo, e para cada combinação de parâmetros foram realizadas 500 execuções, sempre utilizando $Q = 100$, $\rho = 0.5$, $\gamma = 1$ e a quantidade n de formigas igual ao número de vértices na instância.

Nota-se nos testes efetuados que os melhores tempos de execução ocorrem com a mesma configuração de parâmetros para os valores $\alpha = 5$ e $\beta = 0$, o que é de se esperar. Quando β é nulo o cálculo de probabilidades depende estritamente do depósito

berlin52			α				
			0	0.5	1	2	5
β	0	Melhor	187.56%	155.97%	27.2%	169.02%	186.58%
		Média	216.28%	189.76%	55.47%	230.28%	250.43%
		Tempo Méd. (s)	0.37	0.44	0.98	0.3	**0.26**
	1	Melhor	91.59%	26.1%	0.01%	31.11%	84.12%
		Média	119.91%	40.01%	5.79%	65.49%	137.68%
		Tempo Méd. (s)	0.41	0.5	0.68	0.4	0.29
	2	Melhor	37.94%	5.74%	0.01%	6.2%	32.97%
		Média	55.27%	14.81%	3.07%	24.99%	66.64%
		Tempo Méd. (s)	0.41	0.46	0.58	0.41	0.31
	5	Melhor	6.03%	**0.0%**	**0.0%**	0.06%	4.73%
		Média	11.56%	3.96%	**2.71%**	7.54%	17.83%
		Tempo Méd. (s)	0.4	0.46	0.42	0.41	0.34
	10	Melhor	0.47%	0.06%	0.42%	**0.0%**	0.06%
		Média	5.92%	4.0%	3.72%	5.5%	8.52%
		Tempo Méd. (s)	0.4	0.39	0.46	0.33	0.29
	20	Melhor	1.59%	1.59%	1.59%	1.53%	**0.0%**
		Média	6.05%	5.75%	6.11%	6.45%	6.69%
		Tempo Méd. (s)	0.37	0.39	0.38	0.35	0.27

TABELA 1 – Testes acerca dos parâmetros α e β na instância berlin52.

de feromônios, isto significa que as formigas tenderão a escolher frequentemente algumas das soluções iniciais construídas, mesmo que estas não sejam boas soluções. Sem o fator heurístico as formigas não conseguem encontrar soluções melhores tão facilmente, e são levadas rapidamente para um mínimo local.

Em relação as soluções obtidas, não fica tão claro quais são os melhores parâmetros a serem utilizados. As melhores soluções médias obtidas possuem $\alpha = 1$ em ambas as instâncias, mas a melhor solução encontrada na instância menor utiliza $\beta = 5$, enquanto a melhor solução da instância maior toma $\beta = 20$. Na instância maior, enquanto $beta$ varia entre 5, 10 e 20, a solução média encontrada varia em menos de 0.5%, a mesma variação de $beta$ na instância menor faz a solução média encontrada variar em mais de 4%.

Para ponderar então um bom valor do parâmetro Q, serão utilizados os parâmetros $\alpha = 1$, $\beta = 5$, $\rho = 0.5$, $\gamma = 1$ e n igual a quantidade de vértices na instância. Serão testados os valores $Q = \{1, 100, 10000\}$, cada um durante 500 execuções também e os resultados apresentados na Tabela 3.

O parâmetro Q não se mostra tão impactante no algoritmo original do ACO, o mesmo ocorre com esta adaptação. Para a instância menor o algoritmo sempre foi capaz de encontrar o BKS, já em relação a instância maior a melhor solução foi encontrada com $Q = 10000$, e a segunda melhor solução com $Q = 1$. Não se pode

tsp225			α				
			0	0.5	1	2	5
β	0	Melhor	790.58%	760.88%	235.0%	732.68%	746.82%
		Média	843.35%	802.75%	317.6%	841.15%	874.84%
		Tempo Méd. (s)	3906	5354	19714	5156	**3575**
	1	Melhor	456.46%	96.14%	18.06%	120.54%	363.61%
		Média	502.75%	112.75%	31.38%	186.19%	449.75%
		Tempo Méd. (s)	4032	9198	9493	12003	4846
	2	Melhor	175.93%	34.75%	8.67%	30.92%	107.22%
		Média	200.28%	44.67%	16.27%	48.68%	152.69%
		Tempo Méd. (s)	4111	6977	9241	9601	6371
	5	Melhor	24.13%	10.46%	6.30%	6.63%	16.50%
		Média	31.99%	16.38%	10.81%	17.22%	28.37%
		Tempo Méd. (s)	4040	4717	6268	6060	4331
	10	Melhor	11.35%	6.43%	6.35%	5.66%	9.05%
		Média	15.48%	11.40%	**10.02%**	12.01%	16.96%
		Tempo Méd. (s)	3986	4291	5069	4731	3791
	20	Melhor	6.88%	7.14%	7.19%	**5.38%**	6.88%
		Média	12.24%	11.12%	11.20%	11.5%	13.9%
		Tempo Méd. (s)	4041	4079	4476	4110	3709

TABELA 2 – Testes acerca dos parâmetros α e β na instância tsp225.

		Q		
		1	100	10000
berlin52	Melhor	0.0%	0.0%	0.0%
	Média	**2.67%**	2.71%	3.19%
	Tempo Méd. (s)	0.47	**0.42**	0.5
tsp225	Melhor	5.84%	6.3%	**4.82%**
	Média	**10.25%**	10.81%	11.43%
	Tempo Méd. (s)	7.09	**6.26**	6.82

TABELA 3 – Testes acerca do parâmetro Q.

afirmar que valores mais altos deste parâmetro são mais propensos a encontrar boas soluções, muito pelo contrário, ao analisar as soluções médias obtidas, em ambas as instâncias valores maiores de Q levam a soluções ligeiramente piores.

Foi investigado também o quão relevante é a quantidade de formigas utilizadas no algoritmo. É intuitivo pensar que quanto maior essa quantidade mais fácil será encontrar soluções melhores, mas também será necessário maior tempo de processamento e memória para tal. Utilizando então os valores $m = \{N, N/2, N/10\}$, onde N representa a quantidade de nós presentes na instância, tem-se os resultados apresentados na Tabela 4. Os demais parâmetros utilizados serão $\alpha = 1$, $\beta = 5$, $Q = 100$, $\rho = 0.5$ e $\gamma = 1$.

As soluções obtidas nos testes presentes na Tabela 4 são muito semelhantes independente da quantidade de formigas, mas, ao utilizar 5 formigas na instância de 52

		Qtd. formigas		
		N	$N/2$	$N/10$
berlin52	Melhor	0.0%	0.0%	—
	Média	**2.71%**	2.9%	—
	Tempo Méd. (s)	**0.42**	0.45	—
tsp225	Melhor	6.3%	**4.82%**	5.79%
	Média	**10.81%**	11.43%	11.30%
	Tempo Méd. (s)	6.26	6.82	**4.90**

TABELA 4 – Testes acerca da quantidade de formigas utilizadas.

pontos, ocorre um problema no cálculo de probabilidades, o depósito de feromônios é muito fraco diante das evaporações, eventualmente os valores de feromônios acumulados serão tão baixos que o computador não poderá mais diferenciá-los. Observe que, diferente do que era esperado, as execuções com número de formigas igual a quantidade de vértices da instância foram um pouco mais rápidas que as que possuem metade, então a maior capacidade de busca e convergência pôde superar o custo computacional de manipulação destas formigas.

No algoritmo ACO as formigas são geralmente iniciadas de duas formas, ou cada formiga é inicialmente depositada em um vértice distinto da instância, ou o vértice inicial é tomado de forma aleatória. A seguir são realizados experimentos acerca da disposição inicial de cada formiga na instância. A primeira situação separa-se cada instância em $m = \{10, 20, 30\}$ regiões, utilizando o algoritmo k-médias MacQueen (1967), as formigas sempre iniciam suas soluções em algum dos centroides calculados, também foi testada uma iniciação nos vértices que compõe o envoltório convexo de cada instância. Os demais parâmetros seguem como utilizados nos testes para elaboração da Tabela 4.

		Disposição			
		$Meds10$	$Meds20$	$Meds30$	Envoltório
berlin52	Melhor	1.41%	0.15%	**0.0%**	2.24%
	Média	6.90%	5.36%	**4.03%**	7.04%
	Tempo Méd. (s)	0.44	0.42	**0.41**	0.5
tsp225	Melhor	—	**5.48%**	6.21%	—
	Média	—	**11.58%**	11.76%	—
	Tempo Méd. (s)	—	5.12	**4.14**	—

TABELA 5 – Testes acerca da disposição das formigas.

Na instância maior, a quantidade de formigas continua sendo um problema, tanto utilizando 10 medianas quanto os vértices do envoltório, a evaporação é forte demais para as formigas e o algoritmo não consegue completar a execução. Ao comparar os melhores resultados da Tabela 5 com os melhores resultados presentes

nos testes anteriores, houve uma piora discreta na instância tsp225 e uma piora mais significante na instância berlin52.

Por fim, os últimos dois parâmetros a serem investigados são γ e ρ, ambos relacionados ao processo de evaporação. Como este é um processo muito relevante para o ACO, ambos serão avaliados simultaneamente com os valores $\gamma = \{0.5, 0.8, 1, 1.2, 1.5, 3\}$ e $\rho = \{0.3, 0.5, 0.7, 0.9, 0.999\}$. Os parâmetros restantes serão fixados em $\alpha = 1$, $\beta = 5$, $Q = 100$ e número de formigas igual a quantidade de vértices na instância, ainda com 500 execuções para cada combinação de parâmetros. Os resultados estão expostos nas Tabelas 6 e 7.

berlin52			ρ				
			0.3	0.5	0.7	0.9	0.999
γ	0,5	Melhor	0.0%	0.0%	0.0%	0.0%	0.0%
		Média	3.12%	2.85%	**2.7%**	2.82%	3.04%
		Tempo Méd. (s)	0.49	0.51	0.48	0.49	0.46
	0.8	Melhor	0.0%	0.0%	0.0%	0.01%	0.0%
		Média	2.81%	2.74%	2.81%	2.85%	3.04%
		Tempo Méd. (s)	0.48	0.48	0.43	0.45	0.43
	1.0	Melhor	0.0%	0.0%	0.0%	0.0%	0.0%
		Média	2.82%	2.71%	2.82%	2.9%	3.18%
		Tempo Méd. (s)	0.45	0.42	0.43	0.45	0.46
	1.2	Melhor	0.01%	0.0%	0.0%	0.0%	0.0%
		Média	2.74%	2.75%	2.93%	2.82%	2.95%
		Tempo Méd. (s)	0.45	0.45	0.42	0.45	0.44
	1.5	Melhor	0.0%	0.0%	0.0%	0.0%	0.0%
		Média	2.86%	2.85%	2.87%	2.98%	2.99%
		Tempo Méd. (s)	0.45	0.43	0.42	0.45	0.44
	3.0	Melhor	0.01%	0.0%	0.0%	0.01%	0.01%
		Média	2.75%	2.81%	2.82%	2.97%	2.98%
		Tempo Méd. (s)	0.44	0.40	0.40	**0.39**	0.42

TABELA 6 – Testes acerca dos parâmetros ρ e γ na instância berlin52.

Nota-se que valores elevados de ρ e γ tem tempos de execução menores, isso porque os feromônios se acumulam mais rapidamente, porém, a estagnação ocorre mais frequentemente em mínimos locais. Também pode-se observar que a variação de γ não parece ser tão impactante quanto ρ, para melhoria das soluções obtidas pelo algoritmo, e a Tabela 7 evidencia que conforme os valores de ρ aumentam as soluções médias obtidas pioram.

A próxima seção, utilizará os resultados obtidos durante estes testes para comparar o ACO com a adaptação aqui proposta.

tsp225			ρ				
			0.3	0.5	0.7	0.9	0.999
γ	0,5	Melhor	5.07%	5.61%	4.97%	6.2%	6.22%
		Média	10.51%	10.56%	10.76%	11.25%	11.63%
		Tempo Méd. (s)	6.30	6.36	6.74	6.49	5.63
	0.8	Melhor	6.14%	5.61%	5.61%	5.23%	6.45%
		Média	10.48%	10.79%	10.84%	11.45%	11.66%
		Tempo Méd. (s)	5.87	6.13	6.74	5.53	**5.25**
	1.0	Melhor	5.53%	6.3%	4.95%	5.97%	6.88%
		Média	10.51%	10.81%	10.99%	11.48%	11.78%
		Tempo Méd. (s)	6.36	6.26	6.56	6.01	5.77
	1.2	Melhor	5.58%	5.97%	4.56%	6.04%	6.94%
		Média	**10.46%**	10.74%	11.02%	11.58%	11.73%
		Tempo Méd. (s)	6.34	6.43	6.01	5.78	5.40
	1.5	Melhor	6.04%	6.53%	5.86%	5.69%	6.45%
		Média	10.53%	10.89%	11.15%	11.45%	11.55%
		Tempo Méd. (s)	5.65	6.08	5.66	5.38	5.44
	3.0	Melhor	6.2%	6.37%	**4.54%**	6.37%	6.22%
		Média	10.76%	11.15%	11.61%	11.58%	11.78%
		Tempo Méd. (s)	5.83	5.53	5.97	5.42	6.03

TABELA 7 – Testes acerca dos parâmetros ρ e γ na instância tsp225.

4.2 RESULTADOS

Nesta seção, são apresentados testes comparativos entre o algoritmo *ant cycle* do ACO e a adaptação proposta neste trabalho, esta comparação foi feita sem finalidades competitivas e sim para avaliar o desempenho desta nova abordagem.

Todos os testes foram realizados no mesmo computador utilizado na seção anterior, durante a avaliação de parâmetros. Foram selecionadas mais instâncias, retiradas do site TSPlib (Discrete and Combinatorial Optimization, 2020), com tamanho variando de 76 a 783 vértices, estas são: eil76, rat99, bier127, ch150, gil262, lin318, pr439, rat575 e rat783. Em todas as instâncias serão utilizadas as mesmas configurações de parâmetros. No ACO é utilizada a configuração sugerida originalmente ($\alpha = 1, \beta = 5, \rho = 0.5, Q = 1$) (Dorigo *et al.*, 1999), enquanto que na adaptação proposta são utilizados os parâmetros que obtiveram maior estabilidade nas soluções encontradas na seção anterior, dando prioridade para os parâmetros com melhores soluções médias na instância com maior quantidade de vértices ($\alpha = 1, \beta = 10, \rho = 0.3, Q = 1, \gamma = 1.2$). Para ambos os algoritmos, o critério de parada será o mesmo, o algoritmo será encerrado após 2000 formigas falharem ao obter uma solução melhor do que a melhor já encontrada, além disto, a quantidade de formigas em cada instância será igual a quantidade de vértices presentes no problema, com cada formiga iniciando suas soluções em cada um dos vértices.

Na Tabela 8 são apresentados os resultados obtidos por cada um dos algoritmos, após serem executados 50 vezes em cada uma das instâncias, utilizando três critérios de comparação, o erro percentual da melhor solução encontrada e da solução média encontrada, em relação ao BKS e o tempo de execução médio de cada algoritmo. Também há uma coluna com a melhor solução conhecida (BKS), para utilizar como referência.

	BKS	ACO			Adap		
		Melhor	Média	Tempo méd.	Melhor	Média	Tempo méd.
eil76	538	**2.23%**	**4.27%**	**485**	3.53%	5.39%	692
rat99	1211	**6.27%**	**7.84%**	**667**	6.35%	8.25%	1141
bier127	118282	4.74%	6.31%	**1189**	**3.93%**	**5.15%**	2105
ch150	6528	**2.26%**	**3.95%**	**1604**	3.0%	4.45%	2454
gil262	2378	8.45%	11.6%	**3977**	**8.15%**	**11.35%**	7197
lin318	42029	10.14%	**12.5%**	**7117**	**9.64%**	12.88%	11986
pr439	107217	**9.8%**	**12.88%**	16220	12.91%	15.87%	**15593**
rat575	6773	**14.49%**	17.64%	**22892**	14.83%	**17.31%**	27855
rat783	8806	17.45%	19.46%	48177	**16.7%**	**18.83%**	**43740**

TABELA 8 – Comparativo entre o ACO e a adaptação proposta.

Na Tabela 8, também foram destacados qual dos algoritmos se mostrou superior em cada categoria, para cada uma das instâncias. Já era esperado que a adaptação fosse mais lenta que o ACO, afinal ela deve realizar operações adicionais para decidir qual a próxima formiga a tomar uma decisão, mas, surpreendentemente, ela se mostrou mais rápida em duas das instâncias com maior quantidade de pontos. De forma geral, as soluções obtidas em ambos os algoritmos são bem similares, mas, nas instâncias onde a adaptação obteve melhores soluções, a diferença entre os resultados obtidos pelos algoritmos foi ligeiramente menor.

Foram realizados testes combinando os algoritmos com o 2-OPT, para verificar o quão melhor as soluções podem ficar na vizinhança das soluções encontradas, porém, o 2-OPT será aplicado apenas uma vez ao fim do algoritmo, com a melhor solução que este encontrar, em cada uma das 50 execuções, para que a melhoria não seja dominante na busca pela solução. Os resultados são apresentados na Tabela 9.

Na nova série de testes a melhoria 2-OPT foi igualmente eficiente em ambos os algoritmos, tornando as soluções obtidas ainda mais próximas. Ainda é possível observar que desta vez a adaptação proposta não pôde vencer o ACO, em velocidade, para nenhuma das instâncias, e o motivo disto também não deve ser a execução do 2-OPT, pois o mesmo foi aplicado para ambos os algoritmos exatamente uma única vez, em cada execução. Isto pode significar que a adaptação está muito dependente dos valores aleatórios gerados para realizar uma boa convergência.

	BKS	ACO+2-OPT			Adap+2-OPT		
		Melhor	Média	Tempo méd.	Melhor	Média	Tempo méd.
eil76	538	**0.18%**	**2.41%**	**465**	1.3%	3.15%	689
rat99	1211	1.07%	**2.64%**	**722**	1.07%	2.8%	1206
bier127	118282	0.56%	2.14%	**1163**	**0.55%**	**1.97%**	2066
ch150	6528	**0.65%**	1.4%	**1524**	0.71%	**1.39%**	2471
gil262	2378	3.11%	**4.87%**	**3920**	**2.81%**	5.08%	7019
lin318	42029	2.97%	4.56%	**7320**	**1.93%**	**4.44%**	11402
pr439	107217	**2.44%**	**5.04%**	**15064**	2.53%	5.2%	16408
rat575	6773	**3.94%**	**5.64%**	**21664**	4.38%	5.72%	27905
rat783	8806	**3.99%**	**5.92%**	**48647**	4.61%	6.06%	49166

TABELA 9 – Comparativo entre o ACO e a adaptação proposta com aplicação da melhoria 2-OPT.

Baseando-se nos resultados descritos nas tabelas 8 e 9, o próximo capítulo irá pontuar algumas conclusões do autor e apontar possíveis focos de estudo para uma futura melhoria do trabalho.

5 CONSIDERAÇÕES FINAIS

Neste trabalho foi abordado o problema do caixeiro viajante, também descrevendo os modelos mais conhecidos para a resolução deste problema por métodos determinísticos, que são os modelos MTZ e de Dantzig-Fulkerson, além de mencionar o processo de resolução utilizando o método *branch and bound*. Após apontar o maior agravante do uso de métodos exatos para a solução do TSP, que seria o grande uso de recursos computacionais, são apresentados diversos métodos heurísticos e metaheurísticos, com a função de sanar este ponto negativo, as custas da solução ótima do problema. Na sequência é realizada uma revisão sistemática da literatura, com o objetivo de saber quais as alterações realizadas ao ACO na busca por melhorias para o algoritmo. Durante a revisão, são apontados quatro linhas principais que geralmente são foco das melhorias, estas são: alterações nas fórmulas de depósito de feromônios, alterações no método de construção de soluções, combinações de múltiplas heurísticas e uso de técnicas de paralelismo. Então o trabalho apresenta uma abordagem diferente, que não foi encontrada durante a revisão, esta abordagem tem como foco o posicionamento e movimentação das formigas pelo grafo. Após explicar como será implementado este novo algoritmo, são realizados testes para verificar qual deveria ser a melhor combinação de parâmetros a utilizar nesta adaptação, para então comparar os resultados obtidos pela adaptação com o algoritmo ACO.

De forma geral, os resultados obtidos pela adaptação proposta e o ACO são muito semelhantes, com exceção do tempo de execução, o ACO é visivelmente mais rápido que a adaptação, o que já era esperado, pois a adaptação precisa a toda iteração manipular um vetor que não existe no ACO originalmente. Apesar disto, em algumas das instâncias testadas, a adaptação pôde encontrar soluções e soluções médias melhores que o ACO, e foi 10% mais rápida na instância de 783 vértices. Os resultados apresentados nos testes mostram que esta abordagem pode ser uma melhoria para o ACO, mas é pouco consistente em obter resultados realmente bons.

Existem dois pontos no algoritmo que devem ser melhor trabalhados em trabalhos futuros, a fórmula de depósito de feromônios e o mecanismo de evaporação do algoritmo. O algoritmo foi escrito para ser o mais semelhante possível com o ACO, com exceção da movimentação das formigas, e por isso a fórmula de depósito de feromônios foi adaptada diretamente do algoritmo original. A fórmula de feromônios utilizada foi a que obteve melhores resultados, apesar disto, ela não se encaixa à ideia proposta. Esta fórmula utiliza o custo total da solução encontrada como um método de controle dos feromônios, logo, automaticamente soluções ruins recebem menos feromônios que soluções boas. Na adaptação o depósito de feromônios é realizado antes do circuito

se completar, então esta fórmula penaliza a seleção de arestas localmente ruins, que poderiam gerar soluções boas. Em relação a evaporação de feromônios, os testes realizados mostram que o parâmetro γ é pouco relevante na solução final obtida pela adaptação, isto é um pouco contraditório quando comparado ao parâmetro ρ que, nos mesmos experimentos, demonstra ter uma influência direta nos resultados obtidos.

Por fim, o método proposto tenta usar a ideia de que as formigas são independentes para se movimentar, e influenciam umas às outra somente através dos feromônios que deixam nas arestas, como se as formigas estivesse sendo executadas em paralelo. Logo, é possível que algoritmos que fazem uso de técnicas de paralelismo possam contribuir com esta abordagem, e isto pode ser um futuro foco do trabalho. Infelizmente, as publicações encontradas na revisão literária deste trabalho, que envolvem paralelismo, tem por objetivo tirar o máximo proveito dos recursos computacionais disponíveis, o que não pode ser aplicado à adaptação proposta.

REFERÊNCIAS

BAI, J.; YANG, G.-K.; CHEN, Y.-W.; HU, L.-S.; PAN, C.-C. A model induced max-min ant colony optimization for asymmetric traveling salesman problem. **Applied Soft Computing**, v. 13, n. 3, p. 1365–1375, 2013. Hybrid evolutionary systems for manufacturing processes. ISSN 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2012.04.008. Citado 1 vez na página 60.

BLUM, C. Ant colony optimization: Introduction and recent trends. **Physics of Life Reviews**, v. 2, n. 4, p. 353–373, 2005. ISSN 1571-0645. DOI: https://doi.org/10.1016/j.plrev.2005.10.001. Citado 1 vez na página 41.

BORYCZKA, U.; SZWARC, K. The adaptation of the harmony search algorithm to the ATSP with the evaluation of the influence of the pitch adjustment place on the quality of results. **Journal of Information and Telecommunication**, p. 1–17, jul. 2018. DOI: 10.1080/24751839.2018.1503149. Citado 1 vez na página 38.

BOSCHETTI, M. A.; MANIEZZO, V.; ROFFILLI, M.; BOLUFÉ RÖHLER, A. Matheuristics: Optimization, Simulation and Control. In_____. **Hybrid Metaheuristics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. P. 171–177. Citado 1 vez na página 8.

BOUSSAÏD, I.; LEPAGNOT, J.; SIARRY, P. A survey on optimization metaheuristics. **Information Sciences**, v. 237, p. 82–117, 2013. Prediction, Control and Diagnosis using Advanced Neural Computations. ISSN 0020-0255. DOI: https://doi.org/10.1016/j.ins.2013.02.041. Citado 1 vez na página 25.

CHEN, S.-M.; CHIEN, C.-Y. Solving the traveling salesman problem based on the genetic simulated annealing ant colony system with particle swarm optimization techniques. **Expert Systems with Applications**, v. 38, n. 12, p. 14439–14450, 2011. ISSN 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2011.04.163. Citado 1 vez na página 43.

CHRISTOFIDES, N. **Graph Theory: An Algorithmic Approach**. [S.l.]: Academic Press, 1975. (Computer science and applied mathematics : a series of monographs and textbooks). ISBN 9780121743505. Citado 1 vez na página 19.

CLERC, M. Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem. In: NEW Optimization Techniques in Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 219–239. ISBN 978-3-540-39930-8. DOI: 10.1007/978-3-540-39930-8_8. Citado 1 vez na página 35.

CROES, G. A. A Method for Solving Traveling-Salesman Problems. **Operations Research**, INFORMS, v. 6, n. 6, p. 791–812, 1958. ISSN 0030364X, 15265463. Disponível em: http://www.jstor.org/stable/167074. Citado 1 vez na página 21.

DANTZIG, G.; FULKERSON, R.; JOHNSON, S. Solution of a Large-Scale Traveling-Salesman Problem. **Journal of the Operations Research Society of America**, INFORMS, v. 2, n. 4, p. 393–410, 1954. ISSN 00963984. Citado 1 vez na página 11.

DAVIS, L. Applying Adaptive Algorithms to Epistatic Domains. In: PROCEEDINGS of the 9th International Joint Conference on Artificial Intelligence - Volume 1. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985. (IJCAI'85), p. 162–164. ISBN 0934613028. Citado 1 vez na página 33.

DISCRETE and Combinatorial Optimization. 2020. Disponível em: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Citado 3 vezes nas páginas 43, 67, 73.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant System: An Autocatalytic Optimizing Process Technical Report 91-016, fev. 1999. Citado 3 vezes nas páginas 26, 27, 73.

ESCARIO, J. B.; JIMENEZ, J. F.; GIRON-SIERRA, J. M. Ant Colony Extended: Experiments on the Travelling Salesman Problem. **Expert Systems with Applications**, v. 42, n. 1, p. 390–410, 2015. ISSN 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2014.07.054. Citado 1 vez na página 61.

FLOOD, M. M. The Traveling-Salesman Problem. **Operations Research**, INFORMS, v. 4, n. 1, p. 61–75, 1956. ISSN 0030364X, 15265463. Citado 1 vez na página 21.

GAMBARDELLA, L.; MONTEMANNI, R.; WEYLAND, D. Coupling ant colony systems with strong local searches. **European Journal of Operational Research**, v. 220, n. 3, p. 831–843, 2012. ISSN 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2012.02.038. Citado 1 vez na página 45.

GEEM, Z. W.; KIM, J.; LOGANATHAN, G. A New Heuristic Optimization Algorithm: Harmony Search. **Simulation**, v. 76, p. 60–68, fev. 2001. DOI: 10.1177/003754970107600201. Citado 1 vez na página 38.

GLOVER, F.; LAGUNA, M. **Tabu Search**. [S.l.]: Kluwer Academic Publishers, 1997. Disponível em: https://books.google.com.br/books?id=69XOswEACAAJ. Citado 1 vez na página 23.

GLOVER, F. Tabu search - Part I. **INFORMS Journal on Computing**, v. 2, p. 4–32, jan. 1990. Citado 1 vez na página 23.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675. Citado 1 vez na página 30.

GOLDBERG, D. E.; LINGLE, R. AllelesLociand the Traveling Salesman Problem. In: PROCEEDINGS of the 1st International Conference on Genetic Algorithms. USA: L. Erlbaum Associates Inc., 1985. P. 154–159. ISBN 0805804269. Citado 2 vezes nas páginas 33, 35.

GREENBERG, H. J. **Greedy Algorithms for Minimum Spanning Tree**. [S.l.], 1998. Citado 1 vez na página 19.

GÜLCÜ, u.; MAHI, M.; BAYKAN, Ö. K.; KODAZ, H. A Parallel Cooperative Hybrid Method Based on Ant Colony Optimization and 3-Opt Algorithm for Solving Traveling Salesman Problem. **Soft Comput.**, Springer-Verlag, Berlin, Heidelberg, v. 22, n. 5, p. 1669–1685, mar. 2018. ISSN 1432-7643. DOI: 10.1007/s00500-016-2432-3. Disponível em: https://doi.org/10.1007/s00500-016-2432-3. Citado 1 vez na página 55.

HLAING, Z.; KHINE, M. Solving Traveling Salesman Problem by Using Improved Ant Colony Optimization Algorithm. **International Journal of Information and Education Technology**, v. 1, p. 404–409, jan. 2011. DOI: 10.7763/IJIET.2011.V1.67. Citado 1 vez na página 59.

HOLZINGER, A.; PLASS, M.; HOLZINGER, K.; CRISAN, G. C.; PINTEA, C.; PALADE, V. Towards interactive Machine Learning (iML): Applying Ant Colony Algorithms to Solve the Traveling Salesman Problem with the Human-in-the-Loop Approach. In: v. 9817, p. 81–95. ISBN 978-3-319-45506-8. DOI: 10.1007/978-3-319-45507-5_6. Citado 1 vez na página 54.

JUN-MAN, K.; YI, Z. Application of an Improved Ant Colony Optimization on Generalized Traveling Salesman Problem. **Energy Procedia**, v. 17, p. 319–325, 2012. 2012 International Conference on Future Electrical Power and Energy System. ISSN 1876-6102. DOI: https://doi.org/10.1016/j.egypro.2012.02.101. Citado 1 vez na página 46.

KARABOGA, D. An Idea Based on Honey Bee Swarm for Numerical Optimization, Technical Report - TR06. **Technical Report, Erciyes University**, jan. 2005. Citado 1 vez na página 28.

KOLMOGOROV, V. Blossom V: A new implementation of a minimum cost perfect matching algorithm. **Mathematical Programming Computation**, v. 1, p. 43–67, jul. 2009. DOI: 10.1007/s12532-009-0002-8. Citado 1 vez na página 20.

LI, L.; CHENG, Y.; TAN, L.; NIU, B. A Discrete Artificial Bee Colony Algorithm for TSP Problem. In_____. **Bio-Inspired Computing and Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. P. 566–573. ISBN 978-3-642-24553-4. Citado 1 vez na página 29.

LIAO, E.; LIU, C. A Hierarchical Algorithm Based on Density Peaks Clustering and Ant Colony Optimization for Traveling Salesman Problem. **IEEE Access**, v. 6, p. 38921–38933, 2018. DOI: 10.1109/ACCESS.2018.2853129. Citado 1 vez na página 44.

LIN, S.; KERNIGHAN, B. W. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. **Oper. Res.**, INFORMS, Linthicum, MD, USA, v. 21, n. 2, p. 498–516, abr. 1973. ISSN 0030-364X. DOI: 10.1287/opre.21.2.498. Citado 1 vez na página 23.

LIU, Z.; LIU, T.; GAO, X. An Improved Ant Colony Optimization Algorithm Based on Pheromone Backtracking. In: p. 658–661. DOI: 10.1109/CSE.2011.116. Citado 1 vez na página 49.

LIZÁRRAGA, E.; CASTILLO, O.; SORIA, J. A Method to Solve the Traveling Salesman Problem Using Ant Colony Optimization Variants with Ant Set Partitioning. **Studies in Computational Intelligence**, v. 451, p. 237–246, jan. 2013. DOI: 10.1007/978-3-642-33021-6_19. Citado 1 vez na página 55.

LUCIC, P.; TEODOROVIĆ, D. Bee system: Modeling combinatorial optimization transportation engineering problems by swarm intelligence. **Preprints of the TRISTAN**

IV Triennial Symposium on Transportation Analysis, p. 441–445, jan. 2001. Citado 1 vez na página 28.

MACQUEEN, J. Some methods for classification and analysis of multivariate observations. In: PROCEEDINGS of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics. Berkeley, Calif.: University of California Press, 1967. P. 281–297. Citado 1 vez na página 71.

MATAI, R.; SINGH, S.; MITTAL, M. L. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. In: DAVENDRA, D. (Ed.). **Traveling Salesman Problem**. Rijeka: IntechOpen, 2010. cap. 1. DOI: 10.5772/12909. Disponível em: https://doi.org/10.5772/12909. Citado 1 vez na página 8.

MAVROVOUNIOTIS, M.; YANG, S. Ant colony optimization with direct communication for the traveling salesman problem. In: 2010 UK Workshop on Computational Intelligence (UKCI). [S.l.: s.n.], 2010. P. 1–6. DOI: 10.1109/UKCI.2010.5625608. Citado 1 vez na página 51.

METROPOLIS, N.; ROSENBLUTH, A. W.; ROSENBLUTH, M. N.; TELLER, A. H.; TELLER, E. Equation of State Calculations by Fast Computing Machines. **The Journal of Chemical Physics**, v. 21, n. 6, p. 1087–1092, jun. 1953. DOI: 10.1063/1.1699114. Citado 1 vez na página 24.

MILLER, C. E.; TUCKER, A. W.; ZEMLIN, R. A. Integer Programming Formulation of Traveling Salesman Problems. **J. ACM**, Association for Computing Machinery, New York, NY, USA, v. 7, n. 4, p. 326–329, out. 1960. ISSN 0004-5411. DOI: 10.1145/321043.321046. Citado 1 vez na página 13.

MOHSEN, A. Annealing Ant Colony Optimization with Mutation Operator for Solving TSP. **Computational Intelligence and Neuroscience**, v. 2016, nov. 2016. DOI: 10.1155/2016/8932896. Citado 1 vez na página 44.

NICHOLSON, T. **Optimization in Industry: Industrial applications**. [S.l.]: Aldine Atherton, 1971. (London Business School series). ISBN 9780202370026. Disponível em: https://books.google.com.br/books?id=MSTcxgEACAAJ. Citado 1 vez na página 14.

NING, J.; ZHANG, Q.; ZHANG, C.; ZHANG, B. A best-path-updating information-guided ant colony optimization algorithm. **Information Sciences**, v. 433-434, p. 142–162,

2018. ISSN 0020-0255. DOI: https://doi.org/10.1016/j.ins.2017.12.047. Citado 1 vez na página 47.

OLIVER, I. M.; SMITH, D. J.; HOLLAND, J. R. C. A Study of Permutation Crossover Operators on the Traveling Salesman Problem. In: PROCEEDINGS of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application. Cambridge, Massachusetts, USA: L. Erlbaum Associates Inc., 1987. P. 224–230. ISBN 0805801588. Citado 1 vez na página 34.

PEDEMONTE, M.; NESMACHNOW, S.; CANCELA, H. A survey on parallel ant colony optimization. **Applied Soft Computing**, v. 11, n. 8, p. 5181–5197, 2011. ISSN 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2011.05.042. Disponível em: http://www.sciencedirect.com/science/article/pii/S156849461100202X. Citado 1 vez na página 55.

RATANAVILISAGUL, C. Modified Ant Colony Optimization with pheromone mutation for travelling salesman problem. In: 2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON). [S.l.: s.n.], 2017. P. 411–414. DOI: 10.1109/ECTICon.2017.8096261. Citado 3 vezes nas páginas 53, 54.

RAZALI, N.; GERAGHTY, J. Genetic Algorithm Performance with Different Selection Strategies in Solving TSP. In: v. 2. Citado 1 vez na página 33.

SAHANA, S.; JAIN, D. An Improved Modular Hybrid Ant Colony Approach for Solving Traveling Salesman Problem. **GSTF INTERNATIONAL JOURNAL ON COMPUTING**, v. 1, jan. 2011. DOI: 10.5176/2010-2283_1.2.49. Citado 1 vez na página 43.

SÖRENSEN, K.; GLOVER, F. W. Metaheuristics. In: **Encyclopedia of Operations Research and Management Science**. Edição: Saul I. Gass e Michael C. Fu. Boston, MA: Springer US, 2013. P. 960–970. ISBN 978-1-4419-1153-7. DOI: 10.1007/978-1-4419-1153-7_1167. Disponível em: https://doi.org/10.1007/978-1-4419-1153-7_1167. Citado 1 vez na página 8.

SOUZA, A. L. d. **Teoria dos grafos e aplicações**. 2013. Diss. (Mestrado). Instituto de Ciências Exatas. Citado 1 vez na página 19.

STTZLE, T.; HOOS, H. Improving the Ant System: A Detailed Report on the MAX-MIN Ant System. In. Citado 1 vez na página 48.

TEIXEIRA, P. J. M. **Contagem e codificação de Árvore**. 2006. Disponível em: http://www.mat.ufg.br/bienal/2006/mini/p.jorge.contagem.pdf. Acesso em: 5 mai. 2020. Citado 1 vez na página 19.

TSENG, S.-P. An improved harmony search for travelling salesman problem. In: 2016 2nd IEEE International Conference on Computer and Communications (ICCC). [S.l.: s.n.], 2016. P. 299–302. Citado 1 vez na página 38.

TUBA, M.; JOVANOVIC, R.; JOVANOVIC, R. Improved ACO Algorithm with Pheromone Correction Strategy for the Traveling Salesman Problem. **International Journal of Computers, Communications & Control (IJCCC)**, v. 8, p. 477–485, abr. 2013. DOI: 10.15837/ijccc.2013.3.7. Citado 1 vez na página 50.

UCHIDA, A.; ITO, Y.; NAKANO, K. An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem. In: 2012 Third International Conference on Networking and Computing. [S.l.: s.n.], 2012. P. 94–102. DOI: 10.1109/ICNC.2012.22. Citado 1 vez na página 56.

WANG, R.-L.; ZHAO, L.-Q.; ZHOU, X.-F. Ant Colony Optimization with Memory and Its Application to Traveling Salesman Problem. **IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences**, E95.A, p. 639–645, mar. 2012. DOI: 10.1587/transfun.E95.A.639. Citado 1 vez na página 58.

YAN, Y.; SOHN, H.-s.; REYES, G. A modified ant system to achieve better balance between intensification and diversification for the traveling salesman problem. **Applied Soft Computing**, v. 60, p. 256–267, 2017. ISSN 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2017.06.049. Disponível em: http://www.sciencedirect.com/science/article/pii/S1568494617303927. Citado 1 vez na página 48.

YANG, J.; DING, R.; ZHANG, Y.; CONG, M.; WANG, F.; TANG, G. An improved ant colony optimization (I-ACO) method for the quasi travelling salesman problem (Quasi-TSP). **International Journal of Geographical Information Science**, v. 29, p. 1–18, mar. 2015. DOI: 10.1080/13658816.2015.1013960. Citado 1 vez na página 45.

YANG, X.; WANG, J. Application of improved ant colony optimization algorithm on traveling salesman problem. In: 2016 Chinese Control and Decision Conference (CCDC). [S.l.: s.n.], 2016. P. 2156–2160. DOI: 10.1109/CCDC.2016.7531342. Citado 1 vez na página 46.

YUN, H.; JEONG, S.-J.; KIM, K.-S. Advanced Harmony Search with Ant Colony Optimization for Solving the Traveling Salesman Problem. **Journal of Applied Mathematics**, v. 2013, p. 1–8, nov. 2013. DOI: 10.1155/2013/123738. Citado 1 vez na página 42.

ZHANG, Z.; FENG, Z. Two-stage updating pheromone for invariant ant colony optimization algorithm. **Expert Systems with Applications**, v. 39, n. 1, p. 706–712, 2012. ISSN 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2011.07.062. Citado 1 vez na página 52.

ZHOU, X.-F.; WANG, R.-L. SELF-EVOLVING ANT COLONY OPTIMIZATION AND ITS APPLICATION TO TRAVELING SALESMAN PROBLEM. **International Journal of Innovative Computing Information and Control**, v. 8, p. 8311–8321, 2012. Citado 1 vez na página 52.

APÊNDICES

Código 1 – Rotina que troca dois elementos de um vetor de lugar.

```
Sub Swap(vetor() As Integer, i As Integer, j As Integer)
    Dim aux As Single = vetor(i)
    vetor(i) = vetor(j)
    vetor(j) = aux
End Sub
```

Código 2 – Estrutura formiga utilizada no ACO e na adaptação.

```
Structure Formiga
    Dim Custo As Single
    Dim Caminho() As Integer
    Dim Comprimento As Integer

    Function AddPonto(i As Integer, Dist(,) As Single) As
       Integer
        If Comprimento = Caminho.Length Then
            Return -1
        End If

        For j = Comprimento To Caminho.Length - 1
            If Caminho(j) = i Then
                Swap(Caminho, Comprimento, j)
                Comprimento += 1
                AtualizaCusto(Dist)
                Return 0
            End If
        Next
        Return -1
    End Function

    Function AtualizaCusto(Dist(,) As Single) As Integer
        Custo = Custo + Dist(Caminho(Comprimento - 2),
           Caminho(Comprimento - 1))
        Return 0
    End Function

    Function Vizinhos() As Integer()
        Dim sol(Caminho.Length - Comprimento - 1) As Integer
        For i = 0 To Caminho.Length - Comprimento - 1
```

```
            sol(i) = Caminho(Comprimento + i)
        Next
        Return sol
    End Function

    Function TrajetoAtual() As Integer()
        Dim sol(Comprimento - 1) As Integer
        For i = 0 To Comprimento - 1
            sol(i) = Caminho(i)
        Next
        Return sol
    End Function

    Sub Inicial(v As Integer, tam As Integer)
        ReDim Caminho(tam - 1)
        For i = 0 To tam - 1
            Caminho(i) = i
        Next
        Swap(Caminho, 0, v)
        Comprimento = 1
        Custo = 0
    End Sub

    Sub InicialAleatorio(tam As Integer)
        Randomize()
        Inicial(Rnd() * (tam - 2) + 1, tam)
    End Sub
End Structure
```

Código 3 – Rotina que simula evaporação de feromônios, multiplicando a matriz "Fero"por um fator pré determinado.

```
Sub EvaporaFeromonio(Fero(,) As Double, fator As Single)
    For i = 0 To Fero.GetLength(0) - 1
        For j = 0 To Fero.GetLength(1) - 1
            Fero(i, j) = Fero(i, j) * fator
        Next
    Next
End Sub
```

Código 4 – Rotina que deposita feromônios na matriz "Fero"pelo caminho "cam"percorrido por uma formiga.

```
1  Sub AtualizaFeromonio(Fero(,) As Double, cam() As Integer,
       Dist(,) As Single, Q As Single)
2      Dim custo As Single = CamDist(Dist, cam)
3      For i = 0 To cam.Length - 1
4          Fero(cam(i), cam((i + 1) Mod cam.Length)) = Fero(cam(
               i), cam((i + 1) Mod cam.Length)) + Q / custo
5          Fero(cam((i + 1) Mod cam.Length), cam(i)) = Fero(cam(
               i), cam((i + 1) Mod cam.Length)) + Q / custo
6      Next
7  End Sub
```

Código 5 – Rotina que prepara a matriz de visibilidade "A"para o cálculo de probabilidades através da matriz de feromônios "Fero".

```
1  Sub AtualizaVisibilidade(A(,) As Double, Fero(,) As Double,
       Dist(,) As Single, alfa As Single, beta As Single)
2
3      Dim eta As Single
4      For i = 0 To A.GetLength(0) - 1
5          For j = 0 To A.GetLength(1) - 1
6              eta = 1 / Dist(i, j)
7              A(i, j) = (Fero(i, j) ^ alfa) * (eta ^ beta)
8              A(j, i) = (Fero(j, i) ^ alfa) * (eta ^ beta)
9          Next
10     Next
11 End Sub
```

Código 6 – Função que define qual o próximo nó a ser visitado por uma formiga "formi"através da matriz de visibilidade "A".

```
1  Function SelecaoProb(A(,) As Double, formi As Formiga) As
       Integer
2      Dim vizin() As Integer = formi.Vizinhos()
3      Dim atual As Integer = formi.Caminho(formi.Comprimento -
           1)
4      Dim prob(vizin.Length - 1) As Double
5      Dim rand As Double
6
7      prob(0) = A(atual, vizin(0))
8      For i = 1 To prob.Length - 1
```

```
9           prob(i) = prob(i - 1) + A(atual, vizin(i))
10      Next
11
12      If prob.Last = 0 Then
13          Return vizin(0)
14      End If
15
16      Randomize()
17      rand = Rnd() * prob.Last
18
19      For i = 0 To prob.Length - 1
20          If prob(i) > rand Then
21              Return vizin(i)
22          End If
23      Next
24      Return -1
25  End Function
```

Código 7 – Programa ACO, operando com uma quantidade "quant"de formigas iniciadas de forma aleatória entre os nós utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1   Function ColoniaFormigas(quant As Integer, Dist(,) As Single,
        alfa As Single, beta As Single, evap As Single, iter As
       Integer, Q As Single) As Integer()
2       Dim Npts As Integer = Dist.GetLength(0)
3       Dim A(Npts - 1, Npts - 1) As Double
4       Dim Fero(Npts - 1, Npts - 1) As Double
5       Dim formi(quant - 1) As Formiga
6       Dim best(Npts - 1), flag, prox As Integer
7       Dim Cbest As Single = Single.PositiveInfinity
8
9
10      For i = 0 To Npts - 1
11          For j = 0 To Npts - 1
12              Fero(i, j) = 0.01
13          Next
14      Next
15
16      For i = 0 To quant - 1
17          formi(i).InicialAleatorio(Npts)
```

```
18      Next
19
20      While flag < iter
21          AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
22          EvaporaFeromonio(Fero, 1 - evap)
23          For i = 0 To quant - 1
24              While formi(i).Comprimento <> Npts
25                  prox = SelecaoProb(A, formi(i))
26                  formi(i).AddPonto(prox, Dist)
27              End While
28
29              formi(i).Custo = formi(i).Custo + Dist(formi(i).
                    Caminho(formi(i).Comprimento - 1), formi(i).
                    Caminho(0))
30
31              If formi(i).Custo < Cbest Then
32                  Cbest = formi(i).Custo
33                  formi(i).Caminho.CopyTo(best, 0)
34                  flag = 0
35              End If
36          Next
37          For i = 0 To quant - 1
38              AtualizaFeromonio(Fero, formi(i).Caminho, Dist, Q
                    )
39              formi(i).InicialAleatorio(Npts)
40          Next
41          flag += 1
42      End While
43
44      Return best
45  End Function
```

Código 8 – Programa ACO, operando com uma quantidade de formigas igual a quantidade de nós presentes na instância, iniciadas cada uma em um destes nós, utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1   Function ColoniaFormigas2(Dist(,) As Single, alfa As Single,
        beta As Single, evap As Single, iter As Integer, Q As
        Single) As Integer()
2       Dim Npts As Integer = Dist.GetLength(0)
3       Dim A(Npts - 1, Npts - 1) As Double
```

```
4       Dim Fero(Npts - 1, Npts - 1) As Double
5       Dim formi(Npts - 1) As Formiga
6       Dim best(Npts - 1), flag, prox As Integer
7       Dim Cbest As Single = Single.PositiveInfinity
8
9
10      For i = 0 To Npts - 1
11          For j = 0 To Npts - 1
12              Fero(i, j) = 0.01
13          Next
14      Next
15
16      For i = 0 To Npts - 1
17          formi(i).Inicial(i, Npts)
18      Next
19
20      While flag < iter
21          AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
22          EvaporaFeromonio(Fero, evap)
23          For i = 0 To Npts - 1
24              While formi(i).Comprimento <> Npts
25                  prox = SelecaoProb(A, formi(i))
26                  formi(i).AddPonto(prox, Dist)
27              End While
28
29              formi(i).Custo = formi(i).Custo + Dist(formi(i).
                    Caminho(formi(i).Comprimento - 1), formi(i).
                    Caminho(0))
30              If formi(i).Custo < Cbest Then
31                  Cbest = formi(i).Custo
32                  formi(i).Caminho.CopyTo(best, 0)
33                  flag = 0
34              End If
35          Next
36          For i = 0 To Npts - 1
37              AtualizaFeromonio(Fero, formi(i).Caminho, Dist, Q
                    )
38              formi(i).Inicial(i, Npts)
39          Next
```

```
40          flag += 1
41      End While
42
43      Return best
44  End Function
```

Código 9 – Função que encontra o índice do ponto que possui menor ordenada em um vetor de pontos.

```
1  Function MenorY(pontos() As PointF) As Integer
2      Dim k As Integer
3      For i = 1 To pontos.Length - 1
4          If pontos(i).Y < pontos(k).Y Then
5              k = i
6          End If
7      Next
8      Return k
9  End Function
```

Código 10 – Função que calcula a distância euclidiana entre dois pontos.

```
1  Function Distancia(pt1 As PointF, pt2 As PointF) As Single
2      Return ((pt1.X - pt2.X) ^ 2 + (pt1.Y - pt2.Y) ^ 2) ^ 0.5
3  End Function
```

Código 11 – Função que calcula o cosseno do ângulo formado entre três pontos.

```
1  Function Cosseno(pt1 As PointF, pt2 As PointF, centro As
       PointF) As Single
2      Dim vet1, vet2 As PointF
3      Dim dist1, dist2 As Single
4
5      vet1 = New PointF(pt1.X - centro.X, pt1.Y - centro.Y)
6      vet2 = New PointF(pt2.X - centro.X, pt2.Y - centro.Y)
7
8      dist1 = Distancia(pt1, centro)
9      dist2 = Distancia(pt2, centro)
10
11     Return (vet1.X * vet2.X + vet1.Y * vet2.Y) / (dist1 *
           dist2)
12 End Function
```

Código 12 – Função que encontra o próximo ponto a entrar no envoltório convexo, considerando que os ponto estão sendo procurados no sentido anti-horário.

```
1  Function PassoEnvoltorio(pontos() As PointF, restantes As
       List(Of PointF), cam() As Integer) As Integer()
2      Dim p As PointF
3      Dim novo(cam.Length) As Integer
4      Dim ang1 As Single
5      Dim ang2 As Single
6
7      cam.CopyTo(novo, 0)
8
9      If cam.Length = 1 Then
10         ang2 = -1
11         For Each pt In restantes
12             ang1 = Cosseno(pt, New PointF(pontos(cam.Last).X
                   + 1, pontos(cam.Last).Y), pontos(cam.Last))
13             If ang1 > ang2 Then
14                 p = pt
15                 ang2 = ang1
16             End If
17         Next
18     Else
19         ang2 = 1
20         For Each pt In restantes
21             ang1 = Cosseno(pt, pontos(cam(cam.Length - 2)),
                   pontos(cam.Last))
22             If ang1 < ang2 Then
23                 p = pt
24                 ang2 = ang1
25             End If
26         Next
27     End If
28
29     novo(cam.Length) = Array.IndexOf(pontos, p)
30     Return novo
31 End Function
```

Código 13 – Função que verifica se determinado ponto está a direita de uma reta definida por outros dois pontos.

```
1  Function ADireita(ponto1 As PointF, ponto2 As PointF,
```

```
    avaliado As PointF) As Boolean
    Dim ang, lin, dir As Single

    ang = (ponto1.Y - ponto2.Y) / (ponto1.X - ponto2.X)
    lin = ponto1.Y - ponto1.X * ang
    dir = ang * avaliado.X + lin

    If ang > 0 Then
        If dir >= avaliado.Y Then
            Return True
        Else
            Return False
        End If
    Else
        If dir <= avaliado.Y Then
            Return True
        Else
            Return False
        End If
    End If
End Function
```

Código 14 – Função que determina os índices do pontos que estão no envoltório convexo de um vetor de pontos.

```
Function Envoltorio(pontos() As PointF) As Integer()
    Dim aux() As Integer
    Dim restantes As List(Of PointF) = pontos.ToList
    Dim cam As List(Of Integer) = New List(Of Integer)

    If pontos.Count = 1 Then
        Return {0}
    End If

    aux = {MenorY(pontos)}
    aux = PassoEnvoltorio(pontos, restantes, aux)

    restantes.Remove(pontos(aux(0)))
    restantes.Remove(pontos(aux(1)))

    While restantes.Count <> 0
```

```
17          aux = PassoEnvoltorio(pontos, restantes, aux)
18          restantes.Remove(pontos(aux.Last))
19          For Each pt In restantes.ToArray
20              If ADireita(pontos(aux.First), pontos(aux.Last),
                    pt) Then
21                  restantes.Remove(pt)
22              End If
23          Next
24      End While
25      Return aux
26  End Function
```

Código 15 – Programa ACO, operando com uma quantidade de formigas igual a quantidade de nós presentes no envoltório convexo do grafo, iniciadas cada uma em um destes nós presentes no envoltório, utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1   Function ColoniaFormigas3(Pontos() As PointF, Dist(,) As
        Single, alfa As Single, beta As Single, evap As Single,
        iter As Integer, Q As Single) As Integer()
2       Dim Npts As Integer = Dist.GetLength(0)
3       Dim A(Npts - 1, Npts - 1) As Double
4       Dim Fero(Npts - 1, Npts - 1) As Double
5       Dim env() As Integer = Envoltorio(Pontos)
6       Dim formi(env.Length) As Formiga
7       Dim best(Npts - 1), flag, prox As Integer
8       Dim Cbest As Single = Single.PositiveInfinity
9       Dim quant As Integer = env.Length
10
11
12      For i = 0 To Npts - 1
13          For j = 0 To Npts - 1
14              Fero(i, j) = 0.01
15          Next
16      Next
17
18      For i = 0 To quant - 1
19          formi(i).Inicial(env(i), Npts)
20      Next
21
22      While flag < iter
```

```
23          AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
24          EvaporaFeromonio(Fero, 1 - evap)
25          For i = 0 To quant - 1
26              While formi(i).Comprimento <> Npts
27                  prox = SelecaoProb(A, formi(i))
28                  formi(i).AddPonto(prox, Dist)
29              End While
30
31              formi(i).Custo = formi(i).Custo + Dist(formi(i).
                   Caminho(formi(i).Comprimento - 1), formi(i).
                   Caminho(0))
32              If formi(i).Custo < Cbest Then
33                  Cbest = formi(i).Custo
34                  formi(i).Caminho.CopyTo(best, 0)
35                  flag = 0
36              End If
37          Next
38          For i = 0 To quant - 1
39              AtualizaFeromonio(Fero, formi(i).Caminho, Dist, Q
                   )
40              formi(i).Inicial(env(i), Npts)
41          Next
42          flag += 1
43      End While
44
45      Return best
46  End Function
```

Código 16 – Função que retorna, dado um vetor de centroides, o índice daquele que mais se aproxima de um ponto definido.

```
1   Function Pertinencia(pt As PointF, Cent() As PointF) As
       Integer
2
3       Dim dist As Single = Single.PositiveInfinity
4       Dim aux As Single
5       Dim ret As Integer
6       For i = 0 To Cent.Length - 1
7           aux = Distancia(pt, Cent(i))
8           If aux < dist Then
9               dist = aux
```

```
10              ret = i
11          End If
12      Next
13      Return ret
14  End Function
```

Código 17 – Função que calcula a média de todos os pontos de um vetor indicados para um determinado grupo "k".

```
1  Function Media(pert() As Integer, k As Integer, Pontos() As
       PointF) As PointF
2
3      Dim qtd As Integer
4      Dim pt As PointF
5      pt = PointF.Empty
6
7      For i = 0 To Pontos.Count - 1
8          If pert(i) = k Then
9              qtd += 1
10             pt.X += Pontos(i).X
11             pt.Y += Pontos(i).Y
12         End If
13     Next
14
15     pt.X /= qtd
16     pt.Y /= qtd
17
18     Return pt
19 End Function
```

Código 18 – Algoritmo k-médias, organiza um vetor de pontos em "k"grupos distintos.

```
1  Function Kmedias(k As Integer, Pontos() As PointF) As Integer
       ()
2
3      Dim centroides(k - 1) As PointF
4      Dim pert(Pontos.Count - 1) As Integer
5      Dim flag As Integer
6      Dim aux As Integer
7      Dim ret(k - 1) As Integer
8
9      For i = 0 To k - 1
```

```
10          centroides(i) = Pontos(i)
11      Next
12
13      For i = 0 To Pontos.Count - 1
14          pert(i) = Pertinencia(Pontos(i), centroides)
15      Next
16
17      While flag = 0
18          flag = 1
19          For i = 0 To k - 1
20              centroides(i) = Media(pert, i, Pontos)
21          Next
22
23          For i = 0 To Pontos.Count - 1
24              aux = pert(i)
25              pert(i) = Pertinencia(Pontos(i), centroides)
26              If aux <> pert(i) Then
27                  flag = 0
28              End If
29          Next
30      End While
31
32      For i = 0 To k - 1
33          flag = Pertinencia(centroides(i), Pontos)
34          ret(i) = flag
35      Next
36
37      Return ret
38  End Function
```

Código 19 – Programa ACO, operando com uma quantidade de formigas "quant", iniciadas cada uma em um nó centróide de sua região definida pelo método k-médias, utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1   Function ColoniaFormigas4(quant As Integer, Pontos() As
       PointF, Dist(,) As Single, alfa As Single, beta As Single,
        evap As Single, iter As Integer, Q As Single) As Integer
       ()
2       Dim Npts As Integer = Dist.GetLength(0)
3       Dim A(Npts - 1, Npts - 1) As Double
4       Dim Fero(Npts - 1, Npts - 1) As Double
```

```
5       Dim env() As Integer = Kmedias(quant, Pontos)
6       Dim formi(env.Length) As Formiga
7       Dim best(Npts - 1), flag, prox As Integer
8       Dim Cbest As Single = Single.PositiveInfinity
9
10      For i = 0 To Npts - 1
11          For j = 0 To Npts - 1
12              Fero(i, j) = 0.01
13          Next
14      Next
15
16      For i = 0 To quant - 1
17          formi(i).Inicial(env(i), Npts)
18      Next
19
20      While flag < iter
21          AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
22          EvaporaFeromonio(Fero, 1 - evap)
23          For i = 0 To quant - 1
24              While formi(i).Comprimento <> Npts
25                  prox = SelecaoProb(A, formi(i))
26                  formi(i).AddPonto(prox, Dist)
27              End While
28
29              formi(i).Custo = formi(i).Custo + Dist(formi(i).
                    Caminho(formi(i).Comprimento - 1), formi(i).
                    Caminho(0))
30              If formi(i).Custo < Cbest Then
31                  Cbest = formi(i).Custo
32                  formi(i).Caminho.CopyTo(best, 0)
33                  flag = 0
34              End If
35          Next
36          For i = 0 To quant - 1
37              AtualizaFeromonio(Fero, formi(i).Caminho, Dist, Q
                    )
38              formi(i).Inicial(env(i), Npts)
39          Next
40          flag += 1
```

```
41      End While
42
43      Return best
44  End Function
```

Código 20 – Função que simula passagem de tempo na adaptação do ACO.

```
1   Function PassaTempo(ByRef tempo() As Single) As Integer
2       Dim indice As Integer
3       Dim menor As Single = tempo.Min
4
5       indice = Array.IndexOf(tempo, menor)
6       For i = 0 To tempo.Length - 1
7           tempo(i) -= menor
8       Next
9
10      Return indice
11  End Function
```

Código 21 – Programa ACO adaptado, operando com uma quantidade "quant"de formigas iniciadas de forma aleatória entre os nós utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1   Function FormigaTempo(quant As Integer, Dist(,) As Single,
       iter As Integer, alfa As Single, beta As Single, gama As
       Single) As Integer()
2       Dim Npts As Integer = Dist.GetLength(0)
3       Dim Fero(Npts - 1, Npts - 1), A(Npts - 1, Npts - 1) As
           Double
4       Dim formi(quant - 1) As Formiga
5       Dim best(Npts - 1), flag, prox, atual As Integer
6       Dim Cbest As Single = Single.PositiveInfinity
7       Dim tempo(quant) As Single
8       Dim Q, eta As Single
9
10      For i = 0 To Npts - 1
11          For j = 0 To Npts - 1
12              Fero(i, j) = 0.01
13          Next
14      Next
15
16      tempo(quant) = CamDist(Dist, CaminhoAleatorio(Dist))
```

```
17
18      AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
19
20      For i = 0 To quant - 1
21          formi(i).InicialAleatorio(Npts)
22      Next
23
24      While flag < iter
25          atual = PassaTempo(tempo)
26          If atual = quant Then
27              EvaporaFeromonio(Fero, 0.5)
28              AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
29              tempo(quant) = Cbest * gama
30          Else
31              If formi(atual).Comprimento = Npts Then
32                  formi(atual).Custo += Dist(formi(atual).
                        Caminho.Last, formi(atual).Caminho.First)
33                  eta = 1 / Dist(formi(atual).Caminho.Last,
                        formi(atual).Caminho.First)
34                  Q = 100 / formi(atual).Custo
35                  Fero(formi(atual).Caminho.Last, formi(atual).
                        Caminho.First) += Q
36                  Fero(formi(atual).Caminho.First, formi(atual)
                        .Caminho.Last) += Q
37                  A(formi(atual).Caminho.Last, formi(atual).
                        Caminho.First) = (Fero(formi(atual).
                        Caminho.Last, formi(atual).Caminho.First)
                        ^ alfa) * (eta ^ beta)
38                  A(formi(atual).Caminho.First, formi(atual).
                        Caminho.Last) = (Fero(formi(atual).Caminho
                        .First, formi(atual).Caminho.Last) ^ alfa)
                         * (eta ^ beta)
39                  If formi(atual).Custo < Cbest Then
40                      Cbest = formi(atual).Custo
41                      best = formi(atual).Caminho
42                      flag = 0
43                  End If
44                  formi(atual).InicialAleatorio(Npts)
45                  flag += 1
```

```
46              Else
47                  prox = SelecaoProb(A, formi(atual))
48                  tempo(atual) = Dist(formi(atual).Caminho(
                        formi(atual).Comprimento - 1), prox)
49                  eta = 1 / Dist(formi(atual).Caminho(formi(
                        atual).Comprimento - 1), prox)
50                  Q = 100 / (formi(atual).Custo + Dist(formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1), prox))
51                  Fero(formi(atual).Caminho(formi(atual).
                        Comprimento - 1), prox) += Q
52                  Fero(prox, formi(atual).Caminho(formi(atual).
                        Comprimento - 1)) += Q
53                  A(formi(atual).Caminho(formi(atual).
                        Comprimento - 1), prox) = (Fero(formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1), prox) ^ alfa) * (eta ^ beta)
54                  A(prox, formi(atual).Caminho(formi(atual).
                        Comprimento - 1)) = (Fero(prox, formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1)) ^ alfa) * (eta ^ beta)
55                  formi(atual).AddPonto(prox, Dist)
56              End If
57          End If
58      End While
59
60      Return best
61  End Function
```

Código 22 – Programa ACO adaptado, operando com uma quantidade de formigas igual a quantidade de nós presentes na instância, iniciadas cada uma em um destes nós, utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1  Function FormigaTempo2(Dist(,) As Single, iter As Integer,
       alfa As Single, beta As Single, gama As Single) As Integer
       ()
2       Dim Npts As Integer = Dist.GetLength(0)
3       Dim Fero(Npts - 1, Npts - 1), A(Npts - 1, Npts - 1) As
            Double
4       Dim formi(Npts - 1) As Formiga
5       Dim best(Npts - 1), flag, prox, atual As Integer
```

```
6       Dim Cbest As Single = Single.PositiveInfinity
7       Dim tempo(Npts) As Single
8       Dim Q As Single
9       Dim eta As Single
10
11      For i = 0 To Npts - 1
12          For j = 0 To Npts - 1
13              Fero(i, j) = 0.01
14          Next
15      Next
16
17      tempo(Npts) = CamDist(Dist, CaminhoAleatorio(Dist))
18
19      AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
20
21      For i = 0 To Npts - 1
22          formi(i).Inicial(i, Npts)
23      Next
24
25      While flag < iter
26          atual = PassaTempo(tempo)
27          If atual = Npts Then
28              EvaporaFeromonio(Fero, 0.5)
29              AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
30              tempo(Npts) = Cbest * gama
31          Else
32              If formi(atual).Comprimento = Npts Then
33                  formi(atual).Custo += Dist(formi(atual).
                        Caminho.Last, formi(atual).Caminho.First)
34                  eta = 1 / Dist(formi(atual).Caminho.Last,
                        formi(atual).Caminho.First)
35                  Q = 100 / formi(atual).Custo
36                  Fero(formi(atual).Caminho.Last, formi(atual).
                        Caminho.First) += Q
37                  Fero(formi(atual).Caminho.First, formi(atual)
                        .Caminho.Last) += Q
38                  A(formi(atual).Caminho.Last, formi(atual).
                        Caminho.First) = (Fero(formi(atual).
                        Caminho.Last, formi(atual).Caminho.First)
```

```
                    ^ alfa) * (eta ^ beta)
39                  A(formi(atual).Caminho.First, formi(atual).
                    Caminho.Last) = (Fero(formi(atual).Caminho
                    .First, formi(atual).Caminho.Last) ^ alfa)
                     * (eta ^ beta)
40                  If formi(atual).Custo < Cbest Then
41                      Cbest = formi(atual).Custo
42                      best = formi(atual).Caminho
43                      flag = 0
44                  End If
45                  formi(atual).Inicial(atual, Npts)
46                  flag += 1
47              Else
48                  prox = SelecaoProb(A, formi(atual))
49                  tempo(atual) = Dist(formi(atual).Caminho(
                    formi(atual).Comprimento - 1), prox)
50                  eta = 1 / Dist(formi(atual).Caminho(formi(
                    atual).Comprimento - 1), prox)
51                  Q = 100 / (formi(atual).Custo + Dist(formi(
                    atual).Caminho(formi(atual).Comprimento -
                    1), prox))
52                  Fero(formi(atual).Caminho(formi(atual).
                    Comprimento - 1), prox) += Q
53                  Fero(prox, formi(atual).Caminho(formi(atual).
                    Comprimento - 1)) += Q
54                  A(formi(atual).Caminho(formi(atual).
                    Comprimento - 1), prox) = (Fero(formi(
                    atual).Caminho(formi(atual).Comprimento -
                    1), prox) ^ alfa) * (eta ^ beta)
55                  A(prox, formi(atual).Caminho(formi(atual).
                    Comprimento - 1)) = (Fero(prox, formi(
                    atual).Caminho(formi(atual).Comprimento -
                    1)) ^ alfa) * (eta ^ beta)
56                  formi(atual).AddPonto(prox, Dist)
57              End If
58          End If
59      End While
60
61      Return best
```

```
62 End Function
```

Código 23 – Programa ACO adaptado, operando com uma quantidade de formigas igual a quantidade de nós presentes no envoltório convexo do grafo, iniciadas cada uma em um destes nós presentes no envoltório, utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1  Function FormigaTempo3(Pontos() As PointF, Dist(,) As Single,
       iter As Integer, alfa As Single, beta As Single, gama As
     Single) As Integer()
2      Dim Npts As Integer = Dist.GetLength(0)
3      Dim Fero(Npts - 1, Npts - 1), A(Npts - 1, Npts - 1) As
         Double
4      Dim env() As Integer = Envoltorio(Pontos)
5      Dim quant As Integer = env.Length
6      Dim formi(quant - 1) As Formiga
7      Dim best(Npts - 1), flag, prox, atual As Integer
8      Dim Cbest As Single = Single.PositiveInfinity
9      Dim tempo(quant) As Single
10     Dim Q, eta As Single
11
12     For i = 0 To Npts - 1
13         For j = 0 To Npts - 1
14             Fero(i, j) = 0.01
15         Next
16     Next
17
18     tempo(quant) = CamDist(Dist, CaminhoAleatorio(Dist))
19
20     AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
21
22     For i = 0 To quant - 1
23         formi(i).Inicial(env(i), Npts)
24     Next
25
26     While flag < iter
27         atual = PassaTempo(tempo)
28         If atual = quant Then
29             EvaporaFeromonio(Fero, 0.5)
30             AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
31             tempo(quant) = Cbest * gama
```

```
32          Else
33              If formi(atual).Comprimento = Npts Then
34                  formi(atual).Custo += Dist(formi(atual).
                       Caminho.Last, formi(atual).Caminho.First)
35                  eta = 1 / Dist(formi(atual).Caminho.Last,
                       formi(atual).Caminho.First)
36                  Q = 100 / formi(atual).Custo
37                  Fero(formi(atual).Caminho.Last, formi(atual).
                       Caminho.First) += Q
38                  Fero(formi(atual).Caminho.First, formi(atual)
                       .Caminho.Last) += Q
39                  A(formi(atual).Caminho.Last, formi(atual).
                       Caminho.First) = (Fero(formi(atual).
                       Caminho.Last, formi(atual).Caminho.First)
                       ^ alfa) * (eta ^ beta)
40                  A(formi(atual).Caminho.First, formi(atual).
                       Caminho.Last) = (Fero(formi(atual).Caminho
                       .First, formi(atual).Caminho.Last) ^ alfa)
                        * (eta ^ beta)
41                  If formi(atual).Custo < Cbest Then
42                      Cbest = formi(atual).Custo
43                      best = formi(atual).Caminho
44                      flag = 0
45                  End If
46                  formi(atual).Inicial(env(atual), Npts)
47                  flag += 1
48              Else
49                  prox = SelecaoProb(A, formi(atual))
50                  tempo(atual) = Dist(formi(atual).Caminho(
                       formi(atual).Comprimento - 1), prox)
51                  eta = 1 / Dist(formi(atual).Caminho(formi(
                       atual).Comprimento - 1), prox)
52                  Q = 100 / (formi(atual).Custo + Dist(formi(
                       atual).Caminho(formi(atual).Comprimento -
                       1), prox))
53                  Fero(formi(atual).Caminho(formi(atual).
                       Comprimento - 1), prox) += Q
54                  Fero(prox, formi(atual).Caminho(formi(atual).
                       Comprimento - 1)) += Q
```

```
55                  A(formi(atual).Caminho(formi(atual).
                        Comprimento - 1), prox) = (Fero(formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1), prox) ^ alfa) * (eta ^ beta)
56                  A(prox, formi(atual).Caminho(formi(atual).
                        Comprimento - 1)) = (Fero(prox, formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1)) ^ alfa) * (eta ^ beta)
57                  formi(atual).AddPonto(prox, Dist)
58              End If
59          End If
60      End While
61
62      Return best
63  End Function
```

Código 24 – Programa ACO adaptado, operando com uma quantidade de formigas "quant", iniciadas cada uma em um nó centróide de sua região definida pelo método k-médias, utilizando as distâncias "Dist"entre nós do grafo como entrada.

```
1   Function FormigaTempo4(quant As Integer, Pontos() As PointF,
        Dist(,) As Single, iter As Integer, alfa As Single, beta
        As Single, gama As Single) As Integer()
2       Dim Npts As Integer = Dist.GetLength(0)
3       Dim Fero(Npts - 1, Npts - 1), A(Npts - 1, Npts - 1) As
            Double
4       Dim env() As Integer = Kmedias(quant, Pontos)
5       Dim formi(quant - 1) As Formiga
6       Dim best(Npts - 1), flag, prox, atual As Integer
7       Dim Cbest As Single = Single.PositiveInfinity
8       Dim tempo(quant) As Single
9       Dim Q, eta As Single
10
11      For i = 0 To Npts - 1
12          For j = 0 To Npts - 1
13              Fero(i, j) = 0.01
14          Next
15      Next
16
17      tempo(quant) = CamDist(Dist, CaminhoAleatorio(Dist))
18
```

```
19      AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
20
21      For i = 0 To quant - 1
22          formi(i).Inicial(env(i), Npts)
23      Next
24
25      While flag < iter
26          atual = PassaTempo(tempo)
27          If atual = quant Then
28              EvaporaFeromonio(Fero, 0.5)
29              AtualizaVisibilidade(A, Fero, Dist, alfa, beta)
30              tempo(quant) = Cbest * gama
31          Else
32              If formi(atual).Comprimento = Npts Then
33                  formi(atual).Custo += Dist(formi(atual).
                        Caminho.Last, formi(atual).Caminho.First)
34                  eta = 1 / Dist(formi(atual).Caminho.Last,
                        formi(atual).Caminho.First)
35                  Q = 100 / formi(atual).Custo
36                  Fero(formi(atual).Caminho.Last, formi(atual).
                        Caminho.First) += Q
37                  Fero(formi(atual).Caminho.First, formi(atual)
                        .Caminho.Last) += Q
38                  A(formi(atual).Caminho.Last, formi(atual).
                        Caminho.First) = (Fero(formi(atual).
                        Caminho.Last, formi(atual).Caminho.First)
                        ^ alfa) * (eta ^ beta)
39                  A(formi(atual).Caminho.First, formi(atual).
                        Caminho.Last) = (Fero(formi(atual).Caminho
                        .First, formi(atual).Caminho.Last) ^ alfa)
                         * (eta ^ beta)
40                  If formi(atual).Custo < Cbest Then
41                      Cbest = formi(atual).Custo
42                      best = formi(atual).Caminho
43                      flag = 0
44                  End If
45                  formi(atual).Inicial(env(atual), Npts)
46                  flag += 1
47              Else
```

```
48                  prox = SelecaoProb(A, formi(atual))
49                  tempo(atual) = Dist(formi(atual).Caminho(
                        formi(atual).Comprimento - 1), prox)
50                  eta = 1 / Dist(formi(atual).Caminho(formi(
                        atual).Comprimento - 1), prox)
51                  Q = 100 / (formi(atual).Custo + Dist(formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1), prox))
52                  Fero(formi(atual).Caminho(formi(atual).
                        Comprimento - 1), prox) += Q
53                  Fero(prox, formi(atual).Caminho(formi(atual).
                        Comprimento - 1)) += Q
54                  A(formi(atual).Caminho(formi(atual).
                        Comprimento - 1), prox) = (Fero(formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1), prox) ^ alfa) * (eta ^ beta)
55                  A(prox, formi(atual).Caminho(formi(atual).
                        Comprimento - 1)) = (Fero(prox, formi(
                        atual).Caminho(formi(atual).Comprimento -
                        1)) ^ alfa) * (eta ^ beta)
56                  formi(atual).AddPonto(prox, Dist)
57              End If
58          End If
59      End While
60
61      Return best
62  End Function
63  End Module
```

Código 25 – Função que toma um trecho de um vetor, de números inteiros, e inverte a ordem a qual estes valores aparecem.

```
1  Function TrocaSequencia(i As Integer, j As Integer, cam() As
       Integer) As Integer()
2      Dim v(cam.Length - 1) As Integer
3      Dim n As Integer = j - i
4
5      cam.CopyTo(v, 0)
6      For k = 0 To n
7          v(i + k) = cam(j - k)
8      Next
```

```
9
10      Return v
11  End Function
```

Código 26 – Função que procura em um caminho duas rota que ao serem trocadas diminuem a distância total percorrida pelo caminho.

```
1  Function PassoDoisOpt(M(,) As Single, cam() As Integer) As
       Integer()
2       Dim v() As Integer = Nothing
3       Dim n As Integer = cam.Length - 1
4       Dim i As Integer
5
6       For i = 0 To n - 2
7           For j = i + 2 To n
8               If M(cam(i), cam(i + 1)) + M(cam(j), cam((j + 1)
                    Mod cam.Length)) > M(cam(i), cam(j)) + M(cam(i
                     + 1), cam((j + 1) Mod cam.Length)) Then
9                   v = TrocaSequencia(i + 1, j, cam)
10                  Return v
11              End If
12          Next
13      Next
14
15      Return v
16  End Function
```

Código 27 – Algoritmo 2-OPT, melhora um caminho pré-definido, trocando duas rotas por vez, tantas vezes quanto forem possíveis

```
1  Function DoisOpt(M(,) As Single, cam() As Integer) As Integer
       ()
2       Dim aux(cam.Length - 1) As Integer
3       cam.CopyTo(aux, 0)
4
5       While Not IsNothing(aux)
6           aux.CopyTo(cam, 0)
7           aux = PassoDoisOpt(M, cam)
8       End While
9       Return cam
10  End Function
```