Otimização por Colônia de Formigas Distribuída: Garantindo Consistência e Causalidade com Two-Phase Commit, Relógios de Lamport e RPC

Thomas Neuenschwander[1], **Thiago Rezende**[1], **Luigi Louback**[1], **Henrique Lara**[1]
Livia Xavier[1], **Eduardo Araújo**[1], **Tiago Lascasas**[1], **Rodrigo Drummond**[1], **Caio de Castro**[1]

[1] Departamento de Ciência da Computação — PUC Minas
Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte – MG – Brasil

{thomas.baron, thiago.rezende, luigi.louback, henrique.lara
livia.xavier, eduardo.araujo, tiago.lascasas,
rodrigo.drummond}@sga.pucminas.br

Abstract. *This work presents a distributed implementation for solving the Traveling Salesperson Problem (TSP) using the Ant Colony Optimization (ACO) meta-heuristic. Adopting a Master-Worker architecture via gRPC, the system addresses consistency challenges in asynchronous environments. The Two-Phase Commit (2PC) protocol was implemented to ensure atomic updates of the global pheromone matrix, mitigating stale data issues. Additionally, Lamport Logical Clocks were integrated to guarantee causal event ordering and enable debugging. Experimental results demonstrate the solution's scalability, maintaining polynomial time complexity where exact brute-force methods fail due to factorial explosion.*

Resumo. *Este trabalho apresenta a implementação de um sistema distribuído para a resolução do Problema do Caixeiro Viajante (TSP) utilizando a meta-heurística de Otimização por Colônia de Formigas (ACO). Adotando uma arquitetura Mestre-Escravo com comunicação via gRPC, o projeto foca na consistência em ambientes assíncronos. Para garantir a atomicidade na atualização da matriz de feromônios, implementou-se o protocolo Two-Phase Commit (2PC). Adicionalmente, Relógios Lógicos de Lamport foram integrados para assegurar a ordenação causal de eventos. Resultados experimentais demonstram a escalabilidade da solução frente à complexidade fatorial de métodos exatos.*

1. Introdução

O Problema do Caixeiro Viajante (TSP - *Traveling Salesperson Problem*) é um dos problemas mais estudados na computação combinatória [Wikipedia contributors 2025]. Ele consiste em encontrar o menor caminho que percorra um conjunto de cidades visitando cada uma exatamente uma vez e retornando à origem. Sendo um problema NP-difícil, métodos exatos tornam-se inviáveis à medida que o número de nós cresce, motivando o

uso de meta-heurísticas que buscam soluções sub-ótimas aceitáveis em tempo computacional razoável.

Uma das abordagens mais eficazes para o TSP é a Otimização por Colônia de Formigas (ACO - *Ant Colony Optimization*), proposta originalmente por Dorigo e Gambardella em 1997 [Dorigo and Gambardella 1997]. O ACO é uma meta-heurística baseada em populações de agentes que constroem soluções de forma probabilística e iterativa [Pei et al. 2012]. Embora eficiente, o ACO é computacionalmente intensivo. A natureza independente dos agentes (formigas) sugere um alto potencial de paralelização. No entanto, a distribuição desse algoritmo introduz desafios clássicos de sistemas distribuídos [Tanenbaum and Van Steen 2017], como a necessidade de sincronização global do estado (matriz de feromônios) e a dificuldade em ordenar eventos em um ambiente assíncrono.

ste trabalho apresenta a implementação de um sistema distribuído para execução do ACO aplicado ao TSP, seguindo princípios de alocação distribuída [Falaghi and Haghifam 2007]. O sistema adota uma arquitetura Mestre-Escravo (*Master-Worker*) comunicando-se via gRPC. As principais contribuições desta implementação não residem apenas na paralelização, mas nos mecanismos de robustez adotados: o uso do protocolo *Two-Phase Commit* (2PC) para garantir a consistência atômica da atualização dos feromônios entre iterações e a implementação de Relógios Lógicos de Lamport para a ordenação causal de eventos e análise de execução.

1.1. Inspiração Biológica e Funcionamento do ACO

A técnica ACO possui inspiração biológica direta no comportamento forrageiro de formigas reais. Muitas espécies de formigas são quase cegas e sua comunicação é estigmergica, ou seja, realizada indiretamente através de modificações no ambiente via deposição de feromônios químicos [Dorigo and Gambardella 1997].

O fundamento do algoritmo pode ser explicado pelo ”experimento da ponte binária”, realizado por Denebourg et al. (1990). No experimento, um formigueiro é conectado a uma fonte de alimento por duas pontes de comprimentos diferentes. Inicialmente, sem a presença de feromônios, as formigas escolhem o caminho aleatoriamente com igual probabilidade.

Conforme as formigas transitam, elas depositam feromônio. O caminho mais curto é percorrido mais rapidamente; consequentemente, ele recebe uma taxa de deposição de feromônio maior do que o caminho longo, onde o feromônio tende a evaporar antes de ser reforçado. Com o tempo, a maior concentração química na ponte curta atrai mais formigas, criando um ciclo de retroalimentação positiva que leva a colônia a convergir para a melhor rota.

Computacionalmente, simulamos este processo através de formigas artificiais que constroem soluções passo a passo. Em um ambiente distribuído, o desafio central torna-se sincronizar essa ”memória coletiva”(matriz de feromônios) entre múltiplos computadores, evitando que decisões sejam tomadas com base em informações obsoletas, problema que será abordado através dos mecanismos de consistência detalhados nas seções seguintes.

2. Arquitetura do Sistema

O sistema implementa uma arquitetura distribuída baseada no modelo *Master-Worker* para execução paralela do algoritmo ACO (*Ant Colony Optimization*). Esta seção des-

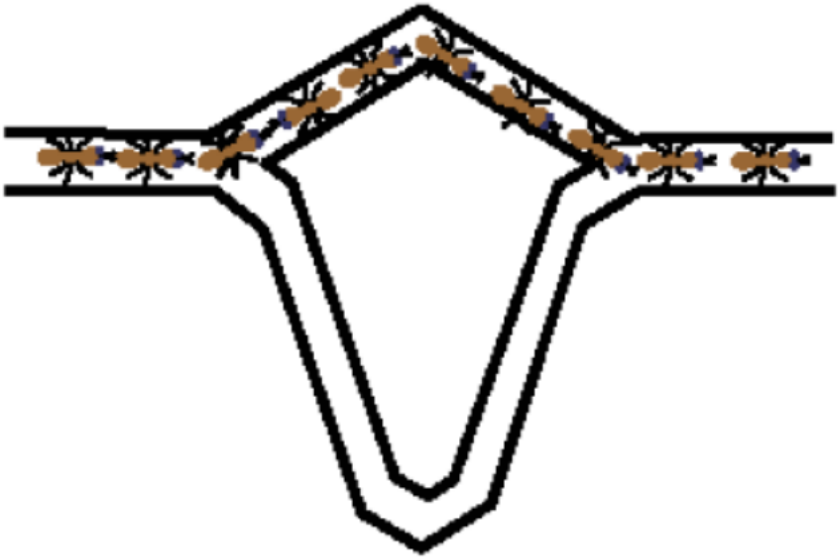

Figura 1. Ilustração do experimento da ponte binária. (a) Inicialmente a escolha é equiprovável. (b) Com o tempo, a evaporação e o reforço fazem a colônia convergir para o caminho mais curto. Fonte: Adaptado de [Siqueira 2025].

creve os componentes principais, os mecanismos de comunicação e o fluxo de dados que caracterizam a arquitetura proposta.

2.1. Modelo Master-Worker

A arquitetura adota o paradigma *Master-Worker*, onde um processo coordenador central (Mestre) delega tarefas computacionais a múltiplos processos executores (Workers) que operam de forma independente e paralela [Coulouris et al. 2011]. Este modelo é particularmente adequado para algoritmos meta-heurísticos como o ACO, onde múltiplas formigas artificiais podem explorar o espaço de soluções simultaneamente sem necessidade de sincronização constante.

No contexto deste trabalho, o Mestre atua como o detentor da "verdade" do sistema, mantendo o estado global compartilhado (matriz de feromônios) e coordenando o progresso iterativo do algoritmo. Os Workers, por sua vez, executam subconjuntos de formigas localmente, construindo soluções candidatas de forma independente e reportando apenas os resultados ao coordenador.

2.2. Componentes do Sistema

2.2.1. ACOMaster: O Coordenador

A classe `ACOMaster`, definida em `aco_master.py`, implementa o componente coordenador do sistema. Suas responsabilidades incluem:

- **Gerenciamento do Estado Global:** Mantém a matriz de feromônios τ_{ij} que representa a memória coletiva do algoritmo, armazenando a intensidade de feromônio em cada aresta (i, j) do grafo.

- **Coordenação de Iterações:** Controla o progresso do algoritmo através de iterações sucessivas, garantindo que todos os Workers completem suas tarefas antes de avançar para a próxima iteração.
- **Distribuição de Trabalho:** Responde a requisições dos Workers através do método `RequestWork`, fornecendo os dados necessários para execução local: matriz de feromônios atual, matriz de distâncias, parâmetros α e β, e número de formigas a executar.
- **Agregação de Soluções:** Recebe soluções dos Workers via `SubmitSolution`, mantém registro da melhor solução global encontrada e acumula contribuições para atualização dos feromônios.
- **Atualização Centralizada:** Ao final de cada iteração, aplica a regra de evaporação e deposição de feromônios de forma centralizada, garantindo consistência do estado global.
- **Protocolo Two-Phase Commit (2PC):** Implementa o protocolo de commit em duas fases para garantir atomicidade nas atualizações de feromônios [Boutros and Desai 1996]. O Mestre atua como coordenador da transação, enviando mensagens `PREPARE` aos Workers e, após receber votos afirmativos de todos os participantes, efetua o `COMMIT` da nova matriz de feromônios.
- **Sincronização Temporal:** Utiliza Relógios de Lamport (classe `LamportClock`) para ordenação causal de eventos distribuídos, permitindo desempate entre soluções de mesmo custo com base no *timestamp* lógico [Lamport 1978].

2.2.2. ACOWorker: O Executor

A classe `ACOWorker`, definida em `aco_worker.py`, implementa o componente executor. Cada Worker opera de forma autônoma, realizando as seguintes funções:

- **Requisição de Tarefas:** Solicita trabalho ao Mestre através de chamadas RPC `RequestWork`, recebendo os dados necessários para execução local.
- **Execução de Formigas:** Executa n formigas artificiais localmente, onde cada formiga constrói uma solução completa aplicando a regra de transição probabilística:

$$P_{ij} = \frac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{k \in \mathcal{N}_i} [\tau_{ik}]^{\alpha} \cdot [\eta_{ik}]^{\beta}}$$

  onde τ_{ij} é o feromônio, $\eta_{ij} = 1/d_{ij}$ é a informação heurística (inverso da distância), e $\mathcal{N}_i$ representa os nós vizinhos não visitados.
- **Seleção Local:** Identifica a melhor solução encontrada localmente (menor custo) entre todas as formigas executadas.
- **Submissão de Resultados:** Envia a melhor solução local ao Mestre via `SubmitSolution`, incluindo o caminho completo, custo total e *timestamp* de Lamport.
- **Participação no 2PC:** Cada Worker implementa um servidor gRPC que responde às mensagens do protocolo Two-Phase Commit. Na fase `PREPARE`, o Worker vota `YES` se completou a execução de suas formigas e está pronto para commitar. Na fase de decisão, recebe e aplica a matriz de feromônios atualizada em caso de `COMMIT`, ou descarta os resultados locais em caso de `ABORT`.

- **Sincronização Temporal:** Mantém seu próprio Relógio de Lamport, incrementando-o antes de enviar mensagens e atualizando-o ao receber respostas, garantindo ordenação causal consistente com o Mestre.

2.3. Comunicação via gRPC e Protocol Buffers

A comunicação entre os componentes é realizada através do framework gRPC (*Google Remote Procedure Call*) [The gRPC Authors 2024], que oferece chamadas de procedimento remoto eficientes e tipadas. A escolha do gRPC se justifica por sua performance superior em relação a alternativas baseadas em REST/HTTP, suporte nativo a *streaming* bidirecional e geração automática de código cliente/servidor.

2.3.1. Definição de Serviços

O arquivo `aco_distributed.proto` define dois serviços principais utilizando a linguagem Protocol Buffers (protobuf):

- **ACOMasterService:** Implementado pelo Mestre, expõe dois métodos RPC:
  - `RequestWork(WorkRequest) returns (WorkAssignment)`: Permite que Workers solicitem tarefas.
  - `SubmitSolution(Solution) returns (SolutionResponse)`: Permite que Workers enviem soluções.
- **TwoPhaseCommitService:** Implementado pelos Workers, expõe três métodos RPC para o protocolo 2PC:
  - `Prepare(PrepareRequest) returns (PrepareResponse)`: Fase de votação.
  - `Commit(CommitRequest) returns (CommitResponse)`: Fase de commit.
  - `Abort(AbortRequest) returns (AbortResponse)`: Fase de abort.

2.3.2. Mensagens Protocol Buffers

As estruturas de dados trocadas entre os componentes são definidas como mensagens protobuf, garantindo serialização eficiente e compatibilidade entre diferentes linguagens. As principais mensagens incluem:

- `WorkAssignment`: Contém matriz de feromônios serializada (`repeated double pheromone_matrix`), matriz de distâncias, parâmetros do algoritmo (α, β) e número de formigas a executar.
- `Solution`: Encapsula uma solução candidata com o caminho percorrido (`repeated int32 path`), custo total (`double cost`) e *timestamp* de Lamport.
- `PrepareRequest/Response`, `CommitRequest/Response`, `AbortRequest/Response`: Mensagens do protocolo 2PC, incluindo identificadores de transação, votos dos participantes e matrizes de feromônios atualizadas.

2.3.3. Geração de Código

A partir da definição protobuf, são gerados automaticamente os módulos Python `aco_distributed_pb2.py` (classes de mensagens) e `aco_distributed_pb2_grpc.py` (stubs de cliente e classes base de servidor). No código, o Mestre utiliza `add_ACOMasterServiceServicer_to_server` para registrar sua implementação, enquanto os Workers criam stubs via `ACOMasterServiceStub(channel)` para realizar chamadas RPC ao Mestre.

2.4. Fluxo de Dados

O fluxo de dados no sistema segue um padrão cíclico iterativo, onde cada iteração do algoritmo ACO compreende as seguintes etapas:

1. **Requisição de Trabalho:** Workers enviam `WorkRequest` ao Mestre, incluindo seu identificador único e *timestamp* de Lamport. O Mestre responde com `WorkAssignment` contendo a matriz de feromônios global atual, matriz de distâncias e parâmetros do algoritmo.
2. **Execução Paralela:** Cada Worker executa independentemente n formigas artificiais, utilizando a matriz de feromônios recebida como guia probabilístico. Esta fase é completamente paralela, sem comunicação entre Workers.
3. **Submissão de Soluções:** Workers enviam suas melhores soluções locais ao Mestre via `SubmitSolution`. O Mestre acumula todas as soluções recebidas, atualiza a melhor solução global (usando *timestamp* de Lamport para desempate) e aguarda a conclusão de todos os Workers esperados.
4. **Protocolo Two-Phase Commit:**
    - *Fase 1 - Prepare:* O Mestre envia `PrepareRequest` a todos os Workers. Cada Worker verifica se completou suas tarefas e responde com voto `YES` ou `NO`.
    - *Decisão:* Se todos os Workers votaram `YES`, o Mestre decide por `COMMIT`; caso contrário, decide por `ABORT`.
    - *Fase 2 - Commit/Abort:* Em caso de `COMMIT`, o Mestre atualiza localmente a matriz de feromônios aplicando evaporação ($\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$) e deposição ($\tau_{ij} \leftarrow \tau_{ij} + \sum_k \Delta\tau_{ij}^k$), e envia a matriz atualizada a todos os Workers via `CommitRequest`. Em caso de `ABORT`, envia `AbortRequest` e a iteração é descartada.
5. **Sincronização e Próxima Iteração:** Após commit bem-sucedido, o Mestre incrementa o contador de iteração e o ciclo se repete. Workers que recebem `WorkAssignment` com flag `finished=true` encerram sua execução.

Este fluxo garante que o Mestre mantém a "verdade" do sistema: a matriz de feromônios global é atualizada de forma atômica e consistente, enquanto os Workers processam subconjuntos de formigas de forma distribuída e paralela. A utilização do protocolo 2PC assegura que todos os participantes concordam sobre o estado final de cada iteração, prevenindo inconsistências causadas por falhas parciais ou latências de rede variáveis.

3. Algoritmos e Protocolos

3.1. Definição Formal do TSP

O Problema do Caixeiro Viajante (*Traveling Salesman Problem* - TSP) é um problema clássico de otimização combinatória NP-difícil. Formalmente, dado um grafo completo

$G = (V, E)$ com conjunto de vértices $V = \{v_1, v_2, \ldots, v_n\}$ e uma função de custo $d : V \times V \rightarrow R^+$ que atribui uma distância d_{ij} a cada aresta $(v_i, v_j) \in E$, o objetivo é encontrar um ciclo hamiltoniano de custo mínimo que visite cada vértice exatamente uma vez e retorne ao vértice inicial.

Matematicamente, o TSP pode ser formulado como:

$$\min \sum_{i=1}^{n} d_{\pi(i),\pi(i+1)} \tag{1}$$

sujeito a:

$$\pi : \{1, 2, \ldots, n\} \rightarrow V \text{ é uma permutação} \tag{2}$$

$$\pi(n+1) = \pi(1) \text{ (retorno ao vértice inicial)} \tag{3}$$

onde π representa uma solução candidata (tour) que define a ordem de visitação dos vértices. O espaço de busca possui $(n-1)!/2$ soluções possíveis para o TSP simétrico, tornando a enumeração exaustiva impraticável para instâncias de tamanho moderado ou grande.

No contexto deste trabalho, a matriz de distâncias $D = [d_{ij}]_{n \times n}$ é fornecida como entrada, onde d_{ij} representa o custo de deslocamento direto do vértice i ao vértice j. A implementação considera grafos simétricos ($d_{ij} = d_{ji}$) e completos, embora o algoritmo possa ser adaptado para grafos esparsos ou assimétricos.

3.2. O Algoritmo ACO

O algoritmo ACO (*Ant Colony Optimization*) é uma meta-heurística bioinspirada que simula o comportamento de formigas reais na busca por alimentos. No contexto computacional, formigas artificiais constroem soluções incrementalmente, guiadas por dois tipos de informação: feromônio (memória coletiva das boas soluções encontradas) e heurística local (informação específica do problema).

3.2.1. Construção de Soluções

Cada formiga artificial constrói uma solução completa aplicando iterativamente uma regra de transição probabilística. Partindo de um vértice inicial v_{start}, a formiga mantém um conjunto de vértices visitados $\mathcal{V}_{\text{visited}}$ e, a cada passo, seleciona o próximo vértice j dentre os vértices candidatos $\mathcal{N}_i = V \setminus \mathcal{V}_{\text{visited}}$ de acordo com a seguinte probabilidade:

$$P_{ij} = \begin{cases} \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{k \in \mathcal{N}_i} [\tau_{ik}]^\alpha \cdot [\eta_{ik}]^\beta} & \text{se } j \in \mathcal{N}_i \\ 0 & \text{caso contrário} \end{cases} \tag{4}$$

onde:

- τ_{ij} é a intensidade de feromônio na aresta (i, j), representando a memória coletiva de boas soluções que utilizaram essa aresta;

- $\eta_{ij} = 1/d_{ij}$ é a informação heurística, representando a atratividade local da aresta (inversamente proporcional à distância);
- $\alpha \geq 0$ é o parâmetro que controla a influência do feromônio;
- $\beta \geq 0$ é o parâmetro que controla a influência da informação heurística.

A escolha de α e β determina o equilíbrio entre exploração (busca em novas regiões) e explotação (refinamento de boas soluções conhecidas). Valores maiores de α intensificam a convergência para soluções já descobertas, enquanto valores maiores de β favorecem escolhas gulosas baseadas na distância.

3.2.2. Implementação no Código

A construção de soluções é implementada no método `run_ant` da classe `ACOWorker` (linhas 191-225 de `aco_worker.py`). O algoritmo segue os seguintes passos:

1. **Inicialização:** A formiga parte de um vértice inicial `start_node`, inicializando o conjunto de vértices visitados e o custo acumulado:

```
visited = [start_node]
total_cost = 0
current = start_node
```

2. **Construção Iterativa:** Enquanto existirem vértices não visitados ($|\mathcal{V}_{\text{visited}}| < n$), a formiga:
   - Identifica os vértices candidatos (vizinhos não visitados com aresta válida):

```
neighbors = []
for j in range(n):
    if j != current and j not in visited
       and distance_matrix[current][j] > 0:
        neighbors.append(j)
```

   - Calcula a atratividade de cada candidato aplicando a fórmula probabilística:

```
probs = []
for next_node in neighbors:
    tau = pheromone[current][next_node] ** alpha
    eta = (1.0 / distance_matrix[current][next_node]) ** beta
    probs.append(tau * eta)
```

   - Normaliza as probabilidades e seleciona o próximo vértice através de amostragem estocástica:

```
total = sum(probs)
if total == 0:
    next_node = random.choice(neighbors)
else:
    prob_norm = [p / total for p in probs]
    next_node = random.choices(neighbors,
                               weights=prob_norm, k=1)[0]
```

   - Atualiza o estado da formiga:

```
        visited.append(next_node)
        total_cost += distance_matrix[current][next_node]
        current = next_node
```

3. **Fechamento do Ciclo:** Se a formiga visitou todos os n vértices, o custo da aresta de retorno ao vértice inicial é adicionado, completando o ciclo hamiltoniano:
```
if len(visited) == n:
    total_cost += distance_matrix[current][start_node]
```
4. **Retorno:** O método retorna a solução construída (sequência de vértices visitados) e seu custo total.

3.2.3. Atualização de Feromônios

Após todas as formigas completarem suas soluções em uma iteração, a matriz de feromônios é atualizada em duas etapas:

1. **Evaporação:** Simula a degradação natural do feromônio, reduzindo a influência de soluções antigas:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij}, \quad \forall (i, j) \in E \tag{5}$$

   onde $\rho \in [0, 1]$ é a taxa de evaporação.
2. **Deposição:** Formigas que encontraram boas soluções depositam feromônio nas arestas utilizadas:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{6}$$

   onde m é o número de formigas e $\Delta\tau_{ij}^{k}$ é a quantidade de feromônio depositada pela formiga k:

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/C_k & \text{se a formiga } k \text{ usou a aresta } (i, j) \\ 0 & \text{caso contrário} \end{cases} \tag{7}$$

   sendo Q uma constante e C_k o custo total da solução encontrada pela formiga k.

No código, esta atualização é realizada centralizadamente pelo Mestre no método `_update_pheromones` (linhas 220-235 de `aco_master.py`):

```
# Evaporação
for i in range(self.n):
    for j in range(self.n):
        self.pheromone[i][j] *= (1 - self.rho)

# Deposição
for solution in self.solutions_current_iteration:
    path = solution[0]
    cost = solution[1]
    deposit = self.q / cost
    for idx in range(len(path)):
```

```
i = path[idx]
j = path[(idx + 1) % len(path)]
self.pheromone[i][j] += deposit
self.pheromone[j][i] += deposit
```

Este mecanismo de atualização implementa o princípio de reforço positivo: soluções de menor custo depositam mais feromônio (pois Q/C_k é inversamente proporcional ao custo), aumentando a probabilidade de que futuras formigas explorem regiões promissoras do espaço de busca. A evaporação, por sua vez, evita convergência prematura para ótimos locais, mantendo um equilíbrio entre exploração e explotação ao longo das iterações.

3.3. Consistência via Two-Phase Commit (2PC)

A manutenção da consistência dos feromônios entre múltiplos workers representa um desafio fundamental em implementações distribuídas do ACO. Neste trabalho,foi implementado o protocolo Two-Phase Commit (2PC) para garantir atomicidade nas atualizações da matriz de feromônios, assegurando que todos os workers mantenham uma visão consistente do estado global do algoritmo.

3.3.1. Motivação e Desafio

No contexto do ACO distribuído, cada iteração do algoritmo requer que: (i) múltiplos workers executem formigas em paralelo; (ii) as soluções sejam coletadas e processadas; e (iii) a matriz de feromônios seja atualizada e redistribuída. Sem um mecanismo de coordenação adequado, workers poderiam operar com versões divergentes da matriz de feromônios, comprometendo a convergência do algoritmo.

O protocolo 2PC foi escolhido por oferecer garantias de atomicidade com implementação relativamente simples, sendo adequado para o cenário com quantidade moderada de workers. A implementação utiliza gRPC como framework de comunicação, fornecendo RPC síncrono com serialização eficiente via Protocol Buffers.

3.3.2. Implementação do Protocolo

A implementaçao do 2PC segue o modelo clássico com coordenador centralizado (master) e participantes (workers), executado em duas fases distintas:

Fase 1 - Voting Phase: O coordenador envia mensagem `PREPARE` para todos os workers participantes, solicitando confirmação de prontidão para commit. Cada worker verifica seu estado interno através do método `is_ready_for_commit()`, retornando `VOTE_YES` se completou o processamento de suas formigas ou `VOTE_NO` caso contrário.

Fase 2 - Decision Phase: Baseado nos votos recebidos, o coordenador toma a decisão global. Se todos os workers votaram `YES`, o coordenador: (i) aplica a atualização dos feromônios localmente considerando todas as soluções coletadas; (ii) envia `COMMIT` com a matriz atualizada para todos os workers. Caso contrário, envia `ABORT` e a iteração é descartada.

3.3.3. Integração com Relógios de Lamport

Para garantir ordenação causal dos eventos do protocolo, foi integrado o 2PC com relógios lógicos de Lamport. Cada mensagem do protocolo (`PREPARE`, `VOTE`, `COMMIT/ABORT`, `ACK`) carrega um timestamp lógico, permitindo:

- Ordenação total dos eventos distribuídos
- Detecção de violações de causalidade
- Auditoria completa do protocolo para debugging

Os logs do sistema demonstram essa integração:

```
[2PC] Recebi PREPARE para transacao 1 | Lamport: 55
[2PC] Votando: YES (pronto para commitar)
[2PC] Recebi COMMIT para transacao 1 | Lamport: 63
[2PC] Feromonios atualizados localmente
[2PC] Transacao 1 COMMITADA
```

3.3.4. Tratamento de Falhas

Nossa implementação incorpora três mecanismos de tolerância a falhas:

1. Timeout Configurável: Todas as chamadas RPC possuem timeout de 5 segundos. Workers que não respondem dentro deste prazo são considerados como voto `NO`, garantindo que o sistema não fique bloqueado indefinidamente.

2. Retry Automático: Em caso de abort, o coordenador realiza até 3 tentativas de commit antes de descartar a iteração, tratando falhas transientes de rede:

3. Estado Consistente em Abort: Quando uma transação é abortada, todos os workers descartam soluções parciais e mantêm a matriz de feromônios anterior, garantindo consistência mesmo em cenários de falha.

Falha Permanente de Worker: Se um worker falha permanentemente, o coordenador considera seu voto como `NO`, resultando em `ABORT` da transação.

3.3.5. Garantias Fornecidas

A implementação do 2PC em nosso sistema fornece as seguintes garantias:

Atomicidade: Atualizações de feromônios são aplicadas em todos os workers ou em nenhum, nunca parcialmente.

Consistência: Todos os workers operam com a mesma versão da matriz de feromônios em cada iteração.

Isolamento: Transações de diferentes iterações não interferem entre si, executando sequencialmente.

Durabilidade: Embora não implementada com persistência em disco, a replicação da matriz em múltiplos workers oferece redundância contra falhas individuais.

3.4. Ordenação via Relógio de Lamport

Em sistemas distribuídos assíncronos, a ausência de uma referência temporal global torna impossível determinar a ordem exata em que eventos ocorreram em máquinas distintas baseando-se apenas em relógios físicos. Para mitigar esse problema e garantir uma propriedade de causalidade ("happens-before"), o sistema implementa Relógios Lógicos de Lamport.

A implementação baseia-se em um contador inteiro incremental mantido independentemente por cada processo (Mestre e Workers), encapsulado na classe `LamportClock`. Esta classe gerencia o tempo lógico seguindo as regras fundamentais do algoritmo de Lamport, garantindo *thread-safety* através de um bloqueio de exclusão mútua (`threading.Lock`), necessário devido à natureza concorrente do servidor gRPC.

3.4.1. Algoritmo e Implementação

O funcionamento do relógio lógico no sistema obedece às seguintes regras implementadas nos métodos da classe:

1. **Estado Inicial:** Cada processo inicializa seu contador local $C_i = 0$.
2. **Eventos Locais e Envios:** Antes da ocorrência de um evento significativo (como o envio de uma mensagem RPC), o processo incrementa seu contador:

$$C_i \leftarrow C_i + 1 \tag{8}$$

   Este valor atualizado é então anexado à mensagem como um metadado (*timestamp*). As mensagens do sistema (e.g., `WorkRequest`, `Solution`, `CommitRequest`) possuem um campo `int64 timestamp` para transportar esse valor.
3. **Recebimento de Mensagens:** Ao receber uma mensagem contendo um timestamp T_m, o processo receptor deve sincronizar seu relógio para refletir que o recebimento ocorre necessariamente após o envio. O método `update(received_time)` implementa a regra:

$$C_j \leftarrow \max(C_j, T_m) + 1 \tag{9}$$

3.4.2. Aplicação no Fluxo de Execução

A integração dos relógios de Lamport permeia todo o ciclo de vida da aplicação distribuída:

No Worker: Ao solicitar trabalho (`request_work`), o worker incrementa seu relógio e envia o timestamp ao Mestre. Ao receber a resposta (`WorkAssignment`), o worker extrai o timestamp do Mestre e atualiza seu relógio local. O mesmo ocorre durante a submissão de soluções e durante as fases de votação do protocolo *Two-Phase Commit*.

No Mestre: O Mestre utiliza os timestamps recebidos para duas funções críticas:

- **Log de Eventos Ordenado:** O Mestre mantém uma lista `event_log` onde registra as interações (requisições, submissões, commits). Ao final da execução, os timestamps de Lamport são utilizados para ordenar a saída, permitindo ao desenvolvedor visualizar a sequência causal correta dos eventos, independentemente da latência de rede ou da ordem de chegada física dos pacotes.
- **Critério de Desempate Determinístico:** Na recepção de soluções via `SubmitSolution`, pode ocorrer de dois workers encontrarem soluções com o mesmo custo exato. Para garantir determinismo na escolha da "melhor solução global", o Mestre utiliza o timestamp lógico como critério de desempate.

4. Desafios de Implementação e Soluções

A transição do modelo teórico do ACO para um sistema distribuído funcional impõe desafios práticos relacionados à natureza assíncrona da rede e à necessidade de consistência de dados. Esta seção detalha como a implementação abordou problemas de concorrência, dados obsoletos e rastreabilidade de eventos.

4.1. Desafio 1: Concorrência no Mestre

O uso do framework gRPC com *ThreadPoolExecutor* no componente Mestre permite o processamento simultâneo de múltiplas requisições (RPCS) vindas de diferentes Workers. No entanto, isso introduz condições de corrida críticas, pois múltiplos *threads* podem tentar acessar e modificar o estado global (matriz de feromônios e lista de soluções) simultaneamente.

Para mitigar esse risco, a implementação adota um mecanismo de exclusão mútua através da classe `threading.Lock` do Python. No arquivo `aco_master.py`, métodos críticos como `RequestWork`, `SubmitSolution` e a execução do protocolo 2PC são protegidos por blocos `with self.lock`. Isso garante a integridade dos dados compartilhados, assegurando, por exemplo, que a atualização da melhor solução global (`self.best_cost`) seja atômica e que a lista de soluções da iteração corrente não sofra corrupção durante escritas concorrentes.

4.2. Desafio 2: "Stale Data"(Dados obsoletos)

Em sistemas distribuídos, a latência de rede ou a heterogeneidade de hardware podem fazer com que workers fiquem dessincronizados. Um desafio comum é o recebimento de mensagens atrasadas ("stale data"), onde um worker tenta submeter uma solução referente a uma iteração anterior (k) quando o Mestre já avançou para a iteração $k + 1$. Aceitar esses dados violaria a lógica de atualização dos feromônios, que depende estritamente das soluções geradas com a matriz da iteração corrente.

A solução implementada atua em duas frentes:

1. **Validação Lógica:** No método `SubmitSolution` do Mestre, há uma verificação explícita: `if iteration != self.current_iteration`. Se a condição for verdadeira, a solução é rejeitada e o worker é notificado, prevenindo a contaminação do estado atual com dados antigos.

2. **Sincronização por Barreira:** O protocolo *Two-Phase Commit* (2PC) atua como uma barreira de sincronização global. O Mestre não inicia a próxima iteração até que a transação de commit da atual seja concluída com sucesso ou abortada e reiniciada, forçando um alinhamento temporal de todos os participantes antes de prosseguir.

4.3. Desafio 3: Debug em Sistemas Distribuídos

A depuração de algoritmos distribuídos é notoriamente complexa devido à falta de um relógio global e à imprevisibilidade da ordem de chegada das mensagens nos logs de console. A simples observação da saída padrão pode levar a conclusões errôneas sobre a causalidade dos eventos (quem causou o quê).

Para resolver este problema, o sistema implementa a classe `LamportClock` tanto no Mestre quanto nos Workers. Conforme detalhado no código, cada mensagem trocada (requisições de trabalho, submissões e votos do 2PC) carrega um *timestamp* lógico.

No Mestre, a solução para a "observabilidade"foi a criação de um `event_log` estruturado. Ao invés de confiar na ordem de impressão do console, o Mestre armazena tuplas contendo o timestamp de Lamport, o tipo de evento e o ID do worker. Ao final da execução, o método `print_event_log` ordena essa lista baseando-se no tempo lógico (C_i), permitindo aos desenvolvedores reconstruir a história causal exata da execução, independente da ordem física em que os pacotes de rede foram processados pelo sistema operacional.

5. Análise de Desempenho

Para validar o sistema distribuído, foram realizados experimentos comparando o algoritmo de Força Bruta (exato) com a implementação proposta do ACO Distribuído.

5.1. Ambiente e Metodologia

Os testes foram executados em um processador Intel Core i7-1360P (12 núcleos, 16 threads) com 16 GB de RAM, rodando Ubuntu 24.04. A implementação utilizou Python 3 e gRPC. Foram submetidos grafos de 5 a 14 nós, com limitação de tempo para o método exato.

5.2. Resultados e Discussão

A Tabela 1 apresenta os tempos e custos obtidos. A análise dos dados deve ser compreendida sob a ótica da complexidade computacional dos algoritmos envolvidos.

O método de Força Bruta possui complexidade fatorial, especificamente $O(n!)$, devido à necessidade de permutar todas as cidades possíveis para garantir a otimalidade. Para instâncias pequenas ($N \leq 10$), o número de permutações é computacionalmente trivial, fazendo com que o tempo de execução seja dominado apenas pelo processamento da CPU. Nestes casos, o Força Bruta superou o ACO, pois o sistema distribuído paga um "custo fixo"alto de comunicação (serialização gRPC e latência do protocolo 2PC) que não se justifica para problemas simples.

Entretanto, a Figura 2 ilustra o ponto de inflexão que ocorre a partir de 12 nós. Enquanto a curva do Força Bruta cresce exponencialmente, inviabilizando a execução

Tabela 1. Comparativo: Força Bruta vs. ACO Distribuído

Grafo	Tempo BF (s)	Tempo ACO (s)	Custo BF	Custo ACO	Gap (%)
5 nós	6.08	13.08	9.0	9.0	0.0%
10 nós	12.82	14.09	74.0	74.0	0.0%
12 nós	21.02	15.08	127.0	131.0	3.1%
14 nós	TIMEOUT	16.09	-	113.0	N/A

para 14 nós (*timeout*), o ACO mantém um comportamento polinomial, tipicamente $O(k \cdot m \cdot n^2)$ — onde k é o número de iterações e m o número de formigas.

O sistema distribuído demonstrou escalabilidade: o aumento no tamanho do grafo impactou minimamente o tempo total (de 13s para 16s), provando que o overhead de comunicação, embora alto inicialmente, dilui-se frente à complexidade do problema. O ACO sacrificou a otimalidade exata (Gap de 3.1% no caso de 12 nós) em troca da viabilidade de execução, validando a arquitetura para cenários onde a busca exaustiva é impossível.

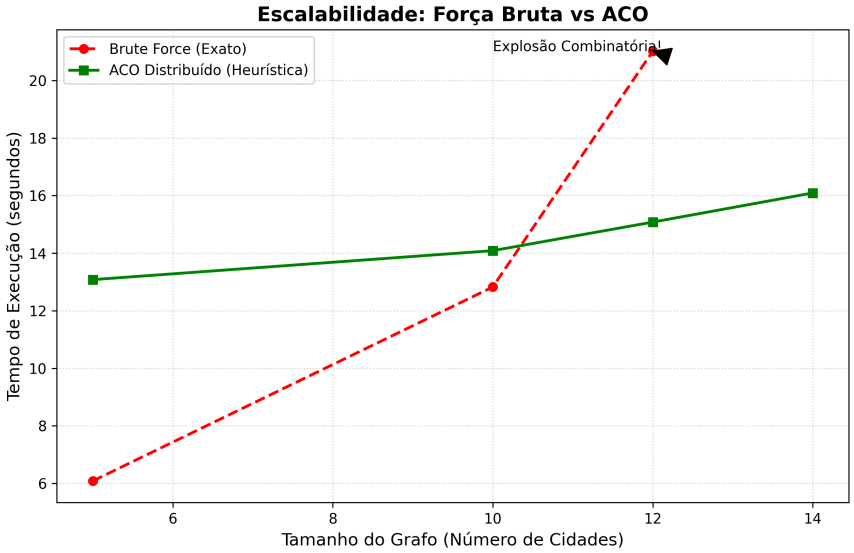

Figura 2. Escalabilidade: Crescimento exponencial ($O(n!)$) do Força Bruta vs. Polinomial do ACO Distribuído.

6. Conclusão

Este trabalho apresentou o desenvolvimento e a análise de um sistema distribuído para a resolução do Problema do Caixeiro Viajante utilizando a meta-heurística ACO. A implementação baseou-se no modelo Mestre-Escravo sobre gRPC, integrando mecanismos robustos de sistemas distribuídos: o protocolo *Two-Phase Commit* (2PC) para consistência e Relógios de Lamport para ordenação causal.

A análise experimental demonstrou que a arquitetura proposta é eficaz em cenários de alta complexidade combinatória. Enquanto métodos exatos de Força Bruta falharam

devido à complexidade fatorial $O(n!)$ em instâncias superiores a 12 nós, o ACO distribuído manteve tempos de execução estáveis, convergindo para soluções de alta qualidade (com gap inferior a 4% em relação ao ótimo) em tempo hábil.

Em relação aos mecanismos de coordenação, conclui-se que:

1. **Consistência via 2PC:** O protocolo garantiu que a atualização da matriz de feromônios fosse atômica entre iterações, eliminando o problema de "dados obsoletos"(*stale data*). Embora o 2PC tenha introduzido uma latência perceptível em instâncias pequenas — onde o Força Bruta foi mais rápido — esse custo de comunicação provou-se necessário e aceitável para garantir a convergência correta do algoritmo em um ambiente assíncrono.
2. **Observabilidade via Lamport:** A utilização de relógios lógicos foi fundamental para a validação do sistema, permitindo a ordenação determinística de eventos e facilitando a depuração de condições de corrida que relógios físicos não conseguiriam capturar.

Em suma, o trabalho evidencia o *trade-off* clássico da computação distribuída: troca-se a simplicidade e a baixa latência local pela escalabilidade e robustez. A solução desenvolvida mostrou-se viável para problemas NP-difíceis, onde a garantia de execução e a consistência dos dados superam o custo adicional de coordenação.

Referências

Boutros, B. and Desai, B. (1996). A two-phase commit protocol and its performance. In *Proceedings of 7th International Conference on Database and Expert Systems Applications (DEXA)*, pages 100–105.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley, 5th edition.

Dorigo, M. and Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66.

Falaghi, H. and Haghifam, M.-R. (2007). Aco based algorithm for distributed generation sources allocation and sizing in distribution systems. In *2007 IEEE Lausanne Power Tech*, pages 555–560.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Pei, Y., Wang, W., and Zhang, S. (2012). Basic ant colony optimization. In *2012 International Conference on Computer Science and Electronics Engineering*, volume 1, pages 665–667.

Siqueira, P. H. (2025). Metaheurísticas e aplicações: Material didático. `https://paulohscwb.github.io/metaheuristicas/`. Acessado em: 10 de dezembro de 2025.

Tanenbaum, A. S. and Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms*. Pearson Education, 3rd edition.

The gRPC Authors (2024). grpc: A high performance, open source universal rpc framework. `https://grpc.io/`. Acessado em: 10 de dezembro de 2025.

Wikipedia contributors (2025). Travelling salesman problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Travelling_salesman_problem. Acessado em: 10 de dezembro de 2025.