COMPUTAÇÃO DE ALTO DESEMPENHO
OTIMIZAÇÃO DE COLÔNIAS DE FORMIGAS

Prof. Ricardo Augusto Pereira Franco

Discentes:

Davi Teixeira (202202436)

Enzo Lemes Marques (202202438)

Lisandra Menezes (202202448)

Pedro Ribeiro Fernandes (202202458)

INTRODUÇÃO

A otimização por colônia de formigas (ACO) é uma metaheurística inspirada no comportamento natural de colônias de formigas ao buscarem alimento. Imagine uma colônia de formigas em busca de uma fonte de comida. Elas se movimentam aleatoriamente, deixando rastros de feromônio no caminho. As formigas que encontram um caminho mais curto e eficiente para a comida depositam mais feromônio, atraindo outras formigas para aquela rota. Com o tempo, o caminho mais curto se torna o mais percorrido, pois acumula mais feromônio e atrai mais formigas.

Dorigo et al., imitou esse processo natural para resolver problemas de otimização combinatória, explorando o espaço de busca de um problema, usando informações heurísticas e o nível de "feromônio" em cada caminho para guiar suas decisões. Os caminhos que levam a soluções melhores são "marcados" com mais feromônio, incentivando outros algoritmos a explorá-los.

Essa abordagem se torna particularmente útil para problemas NP-hard, como o problema do caixeiro viajante, onde encontrar a solução ótima de forma determinística é computacionalmente muito caro. A ACO, ao utilizar uma abordagem probabilística e colaborativa, consegue encontrar soluções de alta qualidade em um tempo computacional razoável.

OBJETIVOS

A partir da solução apresentada, o objetivo deste trabalho consiste em algumas etapas, que são principalmente, mas não se limitando a:

- Implementação do algoritmo da colônia de formigas em um problema real de logística
- Paralelização do algoritmo por meio de linguagens de alto desempenho
- Otimização do algoritmo por meio de diferentes técnicas de paralelização

Com isso, o foco é entregar uma solução factível e consistente para resolver problemas reais de logística, aplicando o algoritmo paralelizado e otimizado de forma que entregas e atividades logísticas de uma empresa de transportes fictícia possam ser realizadas com a maior eficiência possível, utilizando de dados públicos para realizar os testes.

LINGUAGEM

Usaremos para desenvolver as versões sequenciais, paralela em CPU e paralela em GPU a linguagem de programação C e API OpenMP e CUDA, respectivamente. Desenvolver aplicações de computação de alto desempenho com OpenMP, CUDA e C permite explorar os recursos de processadores multi-core e GPUs para acelerar tarefas computacionalmente intensivas. O OpenMP permite a utilização eficiente de múltiplos núcleos de CPU, enquanto o CUDA fornece um caminho para aproveitar a vasta capacidade de paralelismo das GPUs.

ANÁLISE DO PROBLEMA ESCOLHIDO

Inicialmente, o problema do caixeiro-viajante, consiste em determinar uma rota de menor custo, começando de um estado inicial, percorrendo todos os estados possíveis sem repetição, e retornando ao estado inicial. Esse problema é aparentemente simples de resolver e pode ser executado manualmente em casos de pequena escala.

Esse problema é um clássico amplamente estudado por matemáticos e cientistas ao longo do tempo. Apesar de sua aparente simplicidade, ele se torna extremamente complexo à medida que a quantidade de pontos a ser explorado aumenta, devido à sua natureza combinatória.

Em casos onde existe simetria na matriz, podemos formular o problema da seguinte forma:

$$r = \frac{(n-1)!}{2}$$

Figura 1: Equação de análise combinatória do problema do caixeiro-viajante, onde n é o número de cidades a serem percorridas.

Enquanto que, em matrizes assimétricas, formulamos da seguinte maneira:

$$r = (n-1)!$$

Figura 2: Equação de análise combinatória do problema do caixeiro-viajante.

Portanto, à medida que n aumenta, o número total de percursos possíveis cresce exponencialmente. Este crescimento rápido torna o problema do caixeiro-viajante computacionalmente desafiador para um número elevado de cidades. A tabela a seguir ilustra a expansão do total de percursos possíveis conforme o número de cidades aumenta:

Valores de n	**Número de Percursos Possíveis Matriz Simétrica**	**Número de Percursos Possíveis Matriz Assimétrica**
3	1	2
5	12	24

7	360	720
9	20.160	40.320
10	181.440	362.880

Tabela 1: Crescimento exponencial do número de percursos possíveis no problema de otimização combinatória

Para o nosso problema em questão, construímos matrizes genéricas. Na tabela abaixo podemos conferir a complexidade de cada matriz.

Valores de n	**Matriz Simétrica**
25	24! / 2
50	49! / 2
100	99! / 2

Tabela 2: Números de percursos possíveis criados para execução aplicação na solução criada.

IMPLEMENTAÇÃO SEQUENCIAL

O programa começa definindo parâmetros como o número de cidades, o número de formigas, e a distância máxima entre elas. Além disso, são ajustados parâmetros específicos do ACO, como alfa, beta, a taxa de evaporação do feromônio e a quantidade de feromônio aplicada por passagem de formiga, que influenciam na formação do caminho.

Uma matriz de distâncias entre as cidades é lida de um arquivo txt. Essa matriz é crucial para calcular as distâncias reais entre as cidades e inicializar os níveis de feromônio entre cada par delas. Cada formiga começa em uma cidade inicial aleatória, com estruturas de dados que registram a cidade atual, a próxima a visitar, o caminho percorrido, um controle de cidades visitadas e a sequência de cidades percorridas.

Durante cada iteração das formigas, elas se movem de cidade em cidade até visitar todas. A escolha da próxima cidade é baseada numa fórmula que combina a

quantidade de feromônio na aresta (preferindo rotas com mais feromônio) e a distância entre as cidades (preferindo caminhos mais curtos). Esse processo continua até todas as cidades serem visitadas, ajustando os parâmetros alfa e beta para influenciar essa escolha.

Após cada iteração de uma formiga, os feromônios são atualizados. Primeiro, ocorre a evaporação em todas as arestas, seguida pelo depósito de feromônio ao longo do caminho percorrido pela formiga. A quantidade de feromônio depositado é inversamente proporcional ao comprimento do caminho da formiga, incentivando caminhos mais curtos a receberem mais feromônio.

Por fim, selecionamos a melhor distância encontrada e imprimimos o tempo gasto para a execução do programa.

Valores de n	Melhor distância	Tempo
25	408.462555	3.407926 segundos
50	572.472900	12.063116 segundos
100	845.702881	43.423530 segundos

Tabela 3: Resultados do algoritmo sequencial usando Intel® Core™ i7-11800H de 11ª geração, cache de 24 MB, 8 núcleos, até 4,60 GHz.

IMPLEMENTAÇÃO PARALELA EM CPU

O programa inicia configurando parâmetros essenciais como o número de cidades e formigas, bem como valores específicos para o algoritmo de otimização com colônia de formigas (ACO), como alfa, beta, taxa de evaporação do feromônio e quantidade depositada por passagem de formiga. Após a leitura e inicialização de uma matriz de distâncias entre as cidades, o programa procede para a inicialização das formigas, cada uma posicionada em uma cidade aleatória com estruturas para rastrear sua rota e cidades já visitadas.

A paralelização é intensivamente aplicada utilizando a diretiva `#pragma omp parallel` do OpenMP, permitindo que múltiplas formigas operem simultaneamente,

aumentando significativamente a eficiência do processo. As formigas movem-se iterativamente de uma cidade a outra, onde a escolha da próxima cidade é influenciada por um modelo probabilístico que prioriza caminhos com maior concentração de feromônio e menor distância. Cada formiga atualiza um conjunto local de feromônios para evitar conflitos de escrita, usando a atomicidade nas operações para garantir a consistência dos dados durante as atualizações concorrentes.

Durante o movimento das formigas, várias iterações são realizadas de forma paralela, onde cada thread controla independentemente uma formiga, compartilhando informações de feromônios e distâncias de forma coordenada para otimizar as rotas exploradas. Após cada ciclo completo de todas as formigas (tour), os feromônios locais são sincronizados e combinados com os globais dentro de uma região crítica, garantindo a integridade dos dados evitando as condições de corrida.

Finalmente, o programa executa uma redução paralela para identificar e atualizar a menor distância percorrida entre todas as formigas. A evaporação dos feromônios é tratada fora das regiões paralelas para prevenir condições de corrida, assegurando que apenas um thread modifique os valores em um dado momento. Este enfoque paralelo reduz o tempo necessário para executar o algoritmo em comparação com a abordagem sequencial (ver tabela abaixo), destacando a eficácia do uso do OpenMP na otimização de processos intensivos em cálculos como o problema de otimização abordado aqui.

Valores de n	Melhor distância	Tempo
25	408.462585	2.083050 segundos
50	583.176758	8.330663 segundos
100	906.311829	31.147344 segundos

Tabela 4: Resultados do algoritmo OpenMP usando Intel® Core™ i7-11800H de 11ª geração, cache de 24 MB, 8 núcleos, até 4,60 GHz.

IMPLEMENTAÇÃO PARALELA EM GPU

Para a paralelização com a plataforma CUDA, inicialmente, uma matriz de distâncias foi carregada e validada contra parâmetros predefinidos para assegurar que os dados de entrada estivessem dentro dos limites aceitáveis. Esses dados foram então transferidos para a memória da GPU, que é mais adequada para operações de alta intensidade e paralelismo que caracterizam o ACO.

A paralelização foi realizada utilizando kernels CUDA, onde cada formiga foi associada a uma thread. Utilizou-se o gerador de números aleatórios CURAND para inicializar as posições das formigas, garantindo que cada formiga começasse sua jornada em uma cidade aleatória. Este procedimento de inicialização foi realizado em paralelo, com cada thread executando de forma independente, o que aumentou significativamente a eficiência do processo. O kernel responsável foi o inicializarFormigasCUDA.

A movimentação das formigas foi igualmente implementada em um ambiente paralelo. Cada thread calculou a próxima cidade a ser visitada com base na probabilidade influenciada pelos níveis de feromônio e pela distância, processos estes que exigem acesso simultâneo a múltiplos pontos de dados. Para garantir a consistência dos dados durante essas operações concorrentes, utilizou-se funções atômicas para atualizar os níveis de feromônio, evitando condições de corrida e garantindo a integridade das informações.

Durante cada iteração, os feromônios foram evaporados e depositados em um ciclo contínuo para refletir os caminhos explorados pelas formigas. Através da implementação de operações atômicas, a menor distância encontrada foi atualizada em tempo real, permitindo uma convergência rápida para a solução ótima ou próxima dela. A computação foi realizada inteiramente na GPU, e os resultados foram posteriormente transferidos de volta para a CPU, e a memória alocada para a GPU liberada.

Os resultados da tabela abaixo demonstram a eficácia da implementação paralela do ACO em CUDA, destacando uma redução significativa no tempo de execução em comparação com abordagens sequenciais.

Valores de n	Melhor distância	Tempo
25	407.936554	0.940340 segundos
50	583.426331	3.853429 segundos
100	854.655701	13.263616 segundos

Tabela 5: Resultados do algoritmo CUDA, RTX 3070 Notebook.

SPEEDUPS

Para complementar nossas análises, calculamos os speedups:

Speedup

O speedup absoluto é calculado comparando o tempo de execução da versão sequencial com o tempo de execução da versão paralela.
A fórmula é dada por:

$$SpeedupAbsoluto = \frac{TempoAntigo}{NovoTempo}$$

Onde:

S: Speedup

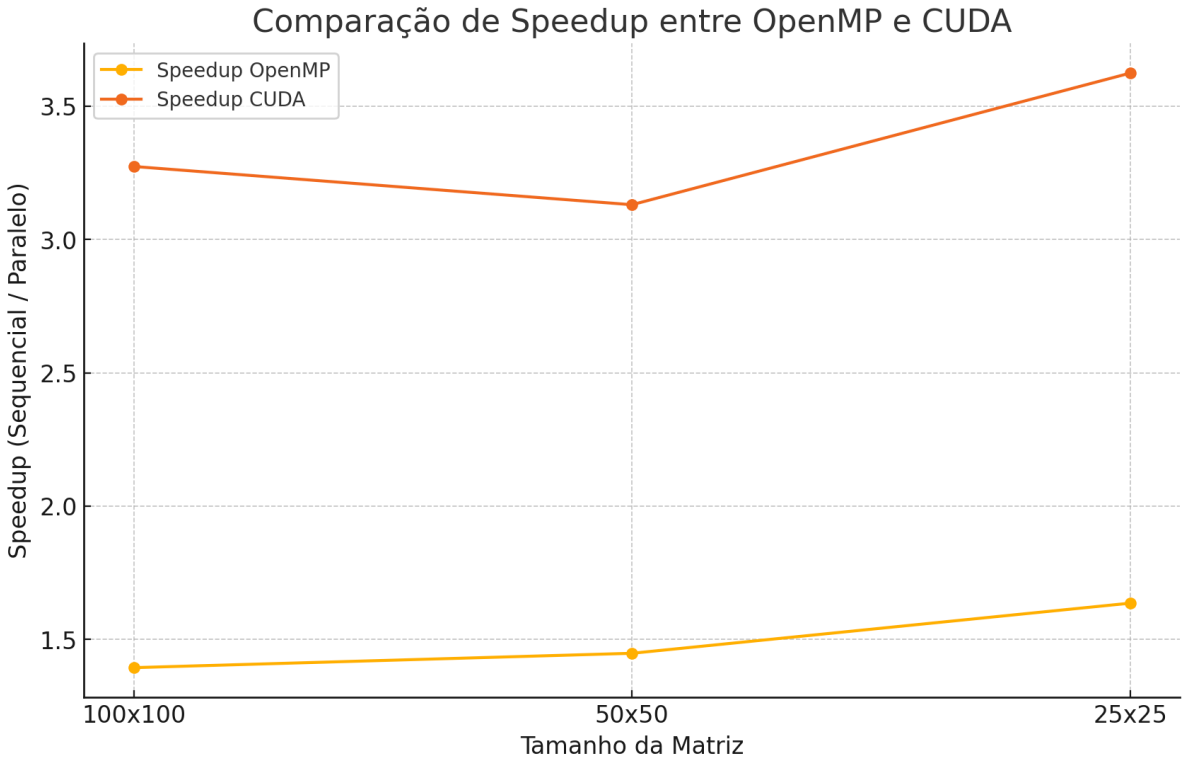
TempoAntigo: tempo de exec. sequencial

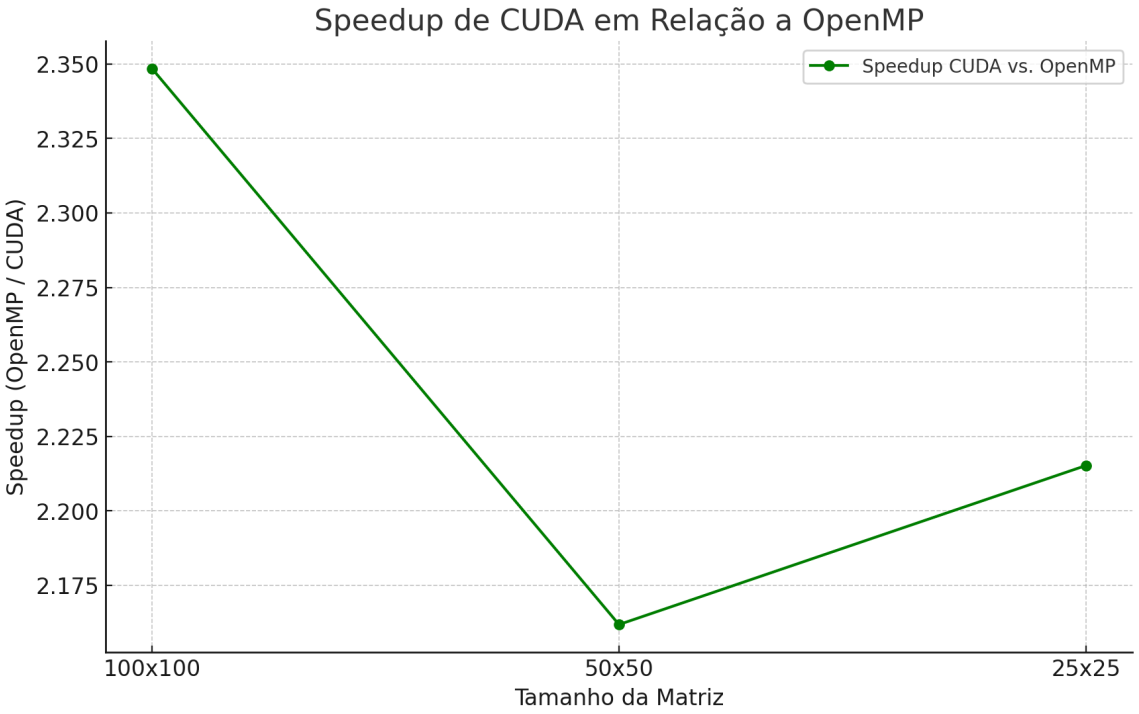
TempoNovo: tempo de exec. paralela

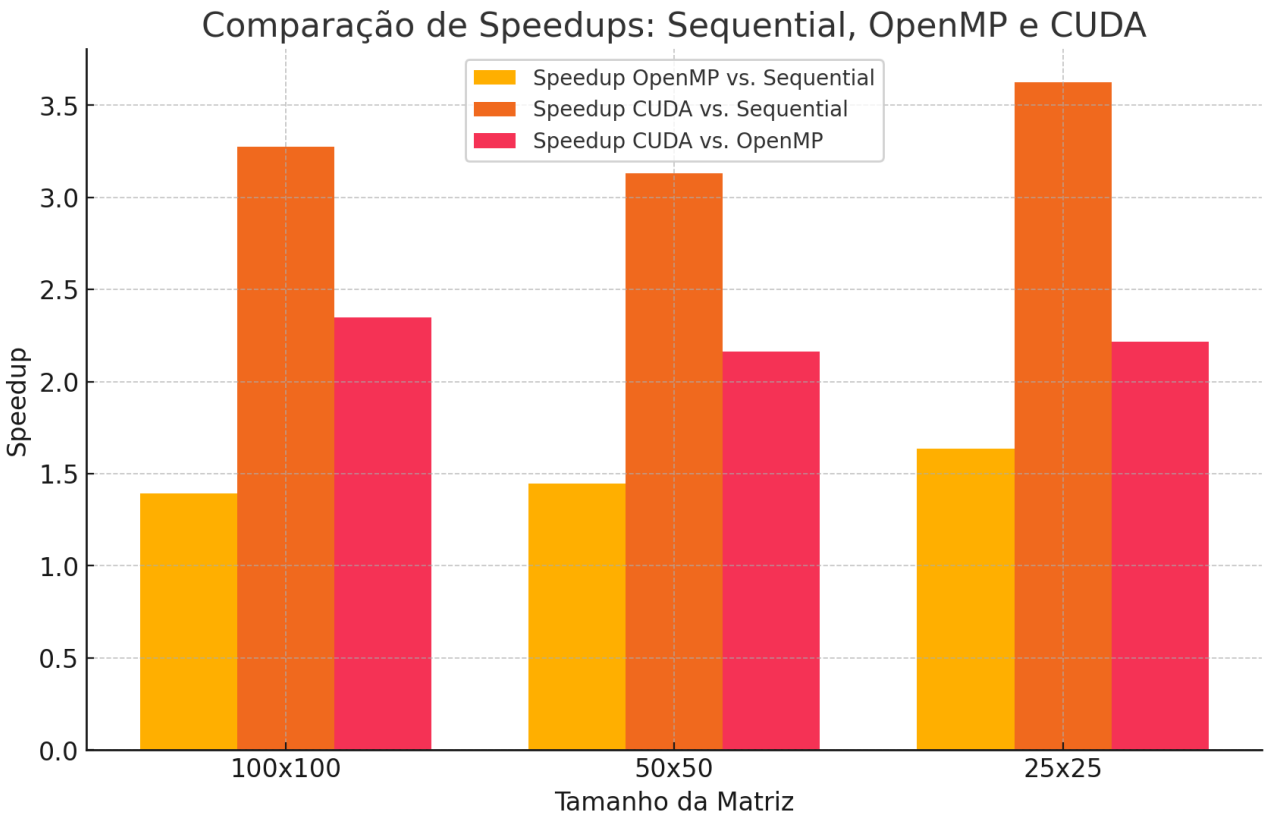
Dada a fórmula, os speedups para as implementações na matriz 100x100 (mais complexa) foram os seguintes:

	Cuda	OpenMP	Sequencial
Cuda	**1**	2.35	3.27
OpenMP	Não há speedup	**1**	1.39
Sequencial	Não há speedup	Não há speedup	1

Podemos também plotar gráficos comparando o speedup entre as diferentes implementações para as diferentes matrizes:







EFICIÊNCIA EM CPU

Com base nos tempos de execução fornecidos para as implementação sequencial e para OpenMP, podemos calcular a eficiência para cada tamanho de matriz usando 16 threads (número de threads usadas padrão pelo processador Intel® Core™ i7-11800H, que possui 8 núcleos, 16 se o Hyper-Threading estiver ativado (o que geralmente está nos processadores Intel)).

Dado que já calculamos os speedups para as diferentes matrizes do nosso código, temos que para calcular a eficiência a fórmula é dada por:

$$Eficiência = \frac{Speedup}{NumProcessadores}$$

Assim, temos que a eficiência para cada um dos casos é:

- Caso 1: Matriz 25x25, Eficiência de 10.23%
- Caso 2: Matriz 50x50, Eficiência de 9.05%
- Caso 3: Matriz 100x100, Eficiência de 8.71%

Essa baixa eficiência pode ser explicada pela quantidade de operações atômicas e zonas críticas que estão presentes no código para evitar condições de corrida. Vamos definir o que é uma secção crítica e como ela impacta a eficiência do código:

1. **Seções Críticas**:
    - **Definição**: Uma seção crítica é uma parte do código que deve ser executada por uma única thread por vez. Ela é usada para proteger os recursos compartilhados que não podem ser acessados por várias threads simultaneamente sem causar inconsistência de dados.
    - **Impacto no Desempenho**: Embora necessárias para garantir a correção dos resultados, as seções críticas reduzem o paralelismo efetivo porque serializam parte da execução. Isso significa que, mesmo em um ambiente multi-core, apenas uma thread pode executar essa parte do código por vez, criando um gargalo.

2. **Operações Atômicas**:
    - **Definição**: Operações atômicas são utilizadas para realizar uma única operação de update em uma variável compartilhada de maneira segura de thread, sem necessidade de bloquear todo um bloco de código. Exemplos comuns incluem incrementos, decrementos, ou atualizações baseadas em valores anteriores.
    - **Impacto no Desempenho**: As operações atômicas são mais leves que as seções críticas completas porque bloqueiam apenas o acesso à variável específica e não a todo um bloco de código. No entanto, elas ainda podem ser fonte de desaceleração se muitas threads tentarem acessar a mesma variável simultaneamente, levando a uma contenção intensa que pode reduzir o ganho de desempenho obtido pela paralelização.

Por conta desses atrasos de processamento, temos um tempo um pouco superior em relação ao tempo de execução do algoritmo sequencial. A métrica de eficiência é bastante impactada por esse baixo speedup derivado do tempo semelhante de processamento dessas duas técnicas e pelo número alto de threads do processador, sendo assim necessário reformular alguns trechos de código a fim de minimizar o uso de zonas críticas e de operações atômicas, mas claro, sempre tentando evitar condições de corrida na medida do possível, pois estas levam a resultados imprecisos do algoritmo.

CONCLUSÃO E ANÁLISE DO PROBLEMA

Percebemos portanto, que em nosso estudo uma meta-heurística como o ACO pode se beneficiar de implementações paralelas, pois podemos processar mais instâncias de um problema de uma só vez, dessa forma, aumentando ainda mais o desempenho dessas técnicas para a resolução de problemas reais. Como sabemos, algumas meta-heurísticas como AGs e ACOs demoram tempo considerável para convergir, mas, com o processamento paralelo que ganha mais força a cada dia, será cada vez mais fácil a utilização desses algoritmos para resolução de problemas de otimização combinatória, por exemplo.

Salienta-se também, a superioridade das GPUs (General Purpose Graphics Processing Units) utilizando CUDA, que oferecem vantagens significativas sobre as CPUs tradicionais e outras formas de paralelismo como o OpenMP. O emprego de GPGPUs em meta-heurísticas revela uma mudança paradigmática no processamento computacional moderno, marcada por uma capacidade maior de realizar operações paralelas intensivas. Isso é evidenciado pelos resultados substancialmente melhores obtidos com CUDA em termos de tempos de execução, reiterando a relevância das GPUs para enfrentar desafios computacionais atuais e futuros.

CÓDIGO

[1] dvzk1. ACO. GitHub, 2024. Disponível em: <https://github.com/dvzk1/ACO>.

REFERÊNCIAS

[1] Dorigo, M.; Birattari, M.; Stutzle, T. Ant colony optimization. Université Libre de Bruxelles, <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4129846>

[2] Freitas, F. Tendências de aplicações da otimização por colônia de formigas na programação de Job-shops. Pontifícia Universidade Católica do Paraná, 2010. <(PDF) b>Tendências de aplicações da otimização por colônia de formigas na progamação de JOB-SHOPS</b (researchgate.net)>

[3] Piotto, J. Problema do caixeiro viajante. Medium, 2024. Disponível em: <https://joaopiotto.medium.com/problema-do-caixeiro-viajante-98c275756788>. Acesso em: 18 jul. 2024.

[4] Silva, F. Tendências de aplicações da otimização por colônia de formigas na programação de JOB-SHOPS. ResearchGate, 2024. Disponível em: <https://www.researchgate.net/publication/123456789_Tendencias_de_aplicacoes_da_otimizacao_por_colonia_de_formigas_na_programacao_de_JOB-SHOPS>. Acesso em: 18 jul. 2024.