Curso de Sistemas de Informação
Universidade Estadual de Mato Grosso do Sul

# COLÔNIAS DE FORMIGAS PARA PORTFÓLIO DE PROJETOS

Igor Souza Patricio

Dr. Cleber Valgas Gomes Mira

Dourados - MS
2022

# COLÔNIAS DE FORMIGAS PARA PORTFÓLIO DE PROJETOS

Igor Souza Patricio

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Igor Souza Patricio e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Dourados, 22 de novembro de 2021.

Prof. Dr. Cleber Valgas Gomes Mira

P341c Patricio, Igor Souza
Colônias de formigas para portfólio de projetos / Igor Souza Patricio. – Dourados, MS: UEMS, 2022.

Trabalho de Conclusão de Curso (Graduação) – Sistemas de Informação – Universidade Estadual de Mato Grosso do Sul, 2022.
Orientador: Prof. Dr. Cleber Valgas Gomes Mira.

1. Project Portfolio Selection (PPS) 2. Portfólio de projetos 3. Otimização de colônia de formigas 4. IA I. Mira, Cleber Valgas Gomes II. Título

CDD 23. ed. - 006.3

Curso de Sistemas de Informação

Universidade Estadual de Mato Grosso do Sul

# COLÔNIAS DE FORMIGAS PARA PORTFÓLIO DE PROJETOS

**Igor Souza Patricio**

11 de 2022

**Banca Examinadora:**

Prof. Dr. Cleber Valgas Gomes Mira
Área de Computação – UEMS

Profa. Dr. Evandro Cesar Bracht
Área de Computação – UEMS

Profa. Delair Osvaldo Martinelli Júnior

Área de Computação – UEMS

# Lista de Ilustrações

# Lista de Tabelas

# Agradecimentos

Agradeço meus pais por me proporcionar a oportunidade de cursar o ensino superior, sempre me apoiando. Agradeço ao meu Orientador Professor Doutor Cleber Valgas Gomes Mira por ter me acompanhado durante todo o trabalho e me guiado para poder entregar um bom trabalho, eu aprendi muito com seus ensinamentos. Muito obrigado!

# Resumo

Neste trabalho é apresentada uma versão simplificada de um problema de otimização de portfólios de projetos, baseado em um problema real, de uma companhia de geração de energia elétrica brasileira. O problema é conhecido como *Project Portfolio Selection* (PPS) e consiste em agendar projetos, da melhor forma, seguindo as restrições estabelecidas. O objetivo deste trabalho é usar o algoritmo *Ant Colony Optimization* (ACO) para gerar soluções do problema e analisar os resultados gerados. Para gerar os resultados foi proposta uma modelagem original do problema que pode ser resolvida pelo ACO. Os experimentos foram executados em conjuntos de instâncias gerados aleatoriamente. Os resultados são analisados em relação o seu tempo de execução e comparamos a qualidade das soluções com o valor do limitante inferior.

**Palavras-chave**:Project Portfolio Selection, Portfólio de projeto, Otimização de colônia de formiga.

# Abstract

In this work we discuss about a simplified version of a problem known as Project Portfolio Selection (PPS) that is based on a real world problem of a brazilian electric power company [19]. The problem is consist on choosing from a set of projects which ones should be executed and when they should start depending on some restrictions. The objective of this work is to use an Ant Colony Optimization (ACO) algorithm to generate solutions of the problem and analyze those generated solutions. To generate the solutions we propose a modeling of the problem that can be solved by the ACO. The experiments were executed in the same machine and with the same data that was randomly generated.The results are analyzed regarding their running time and we compared the results with the lower bound.

**Keywords**: PPS, ACO, Project Portfolio Selectio.

# Sumário

# 1 Introdução

Decidir o agendamento de portfólio de projeto é um problema encontrado em organizações que é conhecido como *Project Portfolio Selection* [2] (abreviado como PPS). O problema consiste em determinar a ordem de execução de projetos que satisfaça os objetivos da empresa obedecendo certas regras.

O PPS pode ser descrito como um conjunto de projetos onde cada projeto possui custos e benefícios. Para realizar cada projeto é necessário algum recurso. Todos os recursos disponíveis são limitados. Um agendamento consiste em um subgrupo do conjunto de projetos que são agendados ao longo de um intervalo de tempo, e deve respeitar certas restrições [22]. É possível achar vários agendamentos, mas o principal problema é encontrar o melhor agendamento disponível.

Este é um problema que vem sendo muito estudado nas últimas duas décadas, e que pode ser aplicado em várias áreas. Devido a variedade de modelos e aplicações de PPS, várias abordagens podem ser utilizadas para resolver o problema. O principal objetivo do PPS é criar um portfólio de projetos que satisfaça os objetivos estratégicos da empresa além de ser ser válido respeitando o conjunto de restrições impostas. São objetivos comuns no agendamento de projetos: reduzir custos, controlar riscos, otimizar o agendamento de atividades, otimizar a alocação de recursos, lidar com incerteza e imprecisão [20].

O modelo de PPS usado neste trabalho é uma simplificação de um problema encontrado em uma usina hidroelétrica, onde os projetos são agendados para reduzir o risco de incidentes durante a geração de energia elétrica [19]. O problema conta com algumas restrições e o único objetivo considerado é reduzir ao máximo os riscos de incidentes durante a geração de energia elétrica dentro de um período de agendamento e execução. Neste trabalho será usado um método heurístico de colônia de formigas para resolver o PPS

Métodos heurísticos são técnicas que providenciam soluções aceitáveis em um tempo razoável para resolução de problemas difíceis e complexos. A maioria das meta-heurísticas imita metáforas naturais para resolver problemas de otimização complexos [26]. Devido ao problema PPS ser classificado como um problema NP-difícil, o que significa que não se sabe se pode ser resolvido em tempo polinomial [20], o uso de métodos heurísticos se mostra uma boa opção. Algumas das técnicas meta-heurísticas já usadas para resolver o PPS incluem Algoritmo gulosos (*Greedy heuristic*), Pesquisa em Vizinhança Variável (*Variable Neighborhood Search*), Algoritmos Genéticos (*Genetic algorithm*), Procedimento de Busca Guloso Aleatório e Adaptativo (*GRASP heuristic*), Algoritmo Competitivo Imperialista (*The imperialist competitive algorithm*) [20].

O algoritmo de colônia de formiga, *Ant Colony Optimization* (abreviado como ACO), é um método heurístico que se baseia no comportamento de colônia de formigas e de como ela aplica o uso de feromônios para a resolução de problemas de otimização [9]. O ACO tem recentemente chamado atenção da comunidade científica por ser usado principalmente para a resolução de problemas NP-difíceis [5].

A heurística do ACO foi criada a partir de observações feitas em como as colônias de formigas conseguem gerar um caminho do ninho até seu alimento. Foi descoberto que ao caminhar da comida até o ninho, a formiga deposita no chão uma substância química chamada

de feromônio, que influência no caminho escolhido por outras formigas fazendo com que elas priorizem andar onde o feromônio está mais concentrado, o que gera uma trilha de feromônios. Por meio desse método, as formigas conseguem encontrar um caminho para diferentes fontes de comida de maneira eficiente através de trilhas de feromônios [10].

Na Figura 1.1 é mostrado de maneira esquemática como as formigas da colônia lidam com um obstáculo. No caminho mostrado no item (a) da figura, as formigas se locomovem entre as posições E e A sem nenhum obstáculo. No item (b) da figura, é colocado um obstáculo que cria dois caminhos e faz com que as formigas escolham entre estes dois; inicialmente a chance das formigas escolherem qualquer um dos caminhos é igual. No item (c) da figura, as formigas passam mais vezes pelo caminho mais curto e aumentam a sua concentração de feromônio mais rapidamente neste caminho mais curto, o que o torna o caminho "preferido" pela colônia.

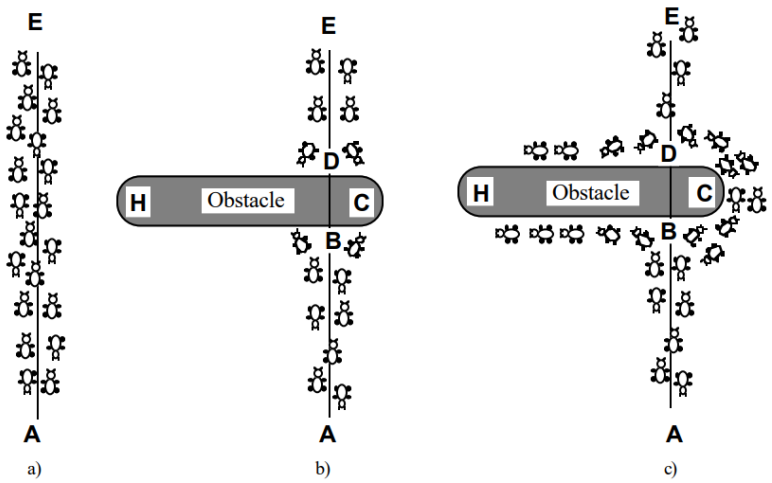


Figura 1.1 – Exemplo esquemático de uma colônia de formigas se movendo entre os pontos A e E [9]

No trabalho de Doerner e outros [3], o ACO é usado para encontrar soluções de um problema de agendamento de portfólio multiobjetivo e é comparado com outras heurísticas. O resultado é comparado com outros dois métodos: Monte Carlo e uma heurística descrita no trabalho. As comparações são feitas baseadas na qualidade das soluções e no numero de iterações necessárias para gerá-las. Os resultados indicam que o ACO encontra soluções satisfatórios para problemas grandes deste tipo.

Em outro trabalho de Doerner e outros [4] é usado *Pareto* ACO com a assistência do método de preprocessamento baseado em Programação Linear Inteira, onde é mostrado que é possível obter melhores resultados com o uso de preprocessamento antes de executar o ACO.

Neste trabalho nós iremos utilizar uma outra estrategia ACO distinta das utilizadas nos

trabalhos anteriores de Doerner para resolver uma versão especifica do problema PPS encontrada na indústria de geração de energia elétrica.

O restante deste trabalho é organizado da seguinte maneira. No Capítulo 2 é apresentado com detalhes o modelo de PPS usado, os detalhes do funcionamento da heurística ACO e o capítulo aborda também sobre as *frameworks* disponíveis para o ACO e qual será utilizada. No Capítulo 3 é apresentado como os experimentos foram realizados, e a análise dos resultados.

## 1.1 Objetivos

O objetivo do projeto é implementar o algoritmo de colônia de formigas para um problema de otimização de portfólios de projetos.

Os objetivos específicos são:

- Estudar um problema específico de agendamento de portfólio de projetos.
- Estudar o algoritmo de otimização com colônia de formigas.
- Modelar o PPS para ser resolvido usando o ACO.
- Gerar dados de entrada referente ao agendamento de portfólio de projeto para o algoritmo.
- Realizar experimentos computacionais e analisar os resultados gerados pela implementação do algoritmo.

## 1.2 Justificativa

O PPS é um problema comum em organizações governamentais e privadas. Infelizmente como já dito este problema é classificado como NP-difícil, então encontrar a melhor solução não é uma opção viável para instâncias grandes. Há diversas abordagens para resolver o problema PPS, como, por exemplo, o uso de métodos heurísticos que podem encontrar bons resultados em um tempo de execução razoável [19].

Definir um bom agendamento de portfólio é uma tarefa difícil para os gerentes de projeto da organização, já que é possível gerar uma grande quantidade de diferentes agendamentos com resultados diferentes.

ACO é um método heurístico que tem sido usado para a resolução de problemas de otimização, e tem recentemente chamado atenção da comunidade científica por conseguir soluções razoáveis para problemas NP-difíceis [5]. É interessante avaliar se o uso de uma heurística como a ACO pode encontrar boas soluções rapidamente para o problema PPS.

# 2 Revisão Bibliográfica

Neste capítulo será apresentado com mais detalhes o problema PPS e o algoritmo utilizado. Na Seção 2.1 é explicado o contexto do problema, seus conceitos e sua inspiração. A Seção 2.2 apresenta a formalização das características do problema, suas restrições, função-objetivo. A Seção 2.3 discute sobre o funcionamento do ACO e como ele é implementado, em que problemas ele é usado e como modelar um problema para utilizar o ACO. A Seção 2.5 aborda quais as *frameworks* disponíveis para a implementação do ACO e qual será utilizadas.

## 2.1 Contexto do problema

O problema estudado neste trabalho é uma versão simplificada do PPS que se trata de um caso real de uma companhia de geração de energia elétrica brasileira que é apresentado no trabalho de Mira e outros [19]. A seguir serão apresentadas informações referentes a essa companhia relevantes para o problema.

Neste PPS os projetos agendados são de gerenciamento de risco que são responsáveis pelo controle de riscos que podem surgir durante a geração de energia. Existem dois tipos de projetos os de manutenção e de não manutenção, projetos de manutenção podem resultar paradas em unidades geradoras de energia. Eles podem ser agendados com certa liberdade, respeitando as restrições.

Os projetos devem ser agendados em um mês dentro de um período de tempo chamado de *Planning Horizon* (PH), ou seja, o horizonte de planejamento.

O processo de geração de energia é organizado por localizações geográficas. Cada localização deve produzir uma quantidade miníma de energia para cumprir com as regulamentações do governo. A produção de energia de certo local é a soma da energia gerada por diversas usinas elétricas daquele local.

### 2.1.1 Entrada do problema

A entrada do problema é composta por um conjunto de projetos, um horizonte de planejamento, PH, os recursos disponíveis durante o PH e a matriz que indica quais projetos não podem ser executados simultaneamente.

O PH leva 60 meses (cinco anos). Para a resolução do problema é considerado também um horizonte de execução que dura mais cinco anos que é para garantir que registre a contribuição de todos os projetos que terminem após o fim do PH. Dobrar o PH é o suficiente porque nenhum projeto tem duração de mais de cinco anos.

### 2.1.2 Recursos disponíveis

Por questões de fiscalização, administrativas e governamentais, os recursos disponíveis são divididos em duas categorias: gastos operacionais (OPEX) e gastos capitais (CAPEX). O custo de cada projeto vem inteiramente de uma dessas categorias.

Os recursos disponíveis são divididos em duas séries anuais, onde a primeira série é equivalente a recursos anuais para gastos CAPEX, e o a segunda série representa os recursos anuais para gastos OPEX.

### 2.1.3 Projetos

Durante a geração de energia elétrica existem chances de acontecer problemas que podem impactar negativamente na geração de energia (defeito em equipamentos, falha em unidades geradoras e acidentes), o que é chamado de risco total. Os projetos são criados por especialistas com a intenção de reduzir ao máximo possível o risco de acontecer incidentes durante a geração de energia elétrica, o que faz, ao final de cada projeto, uma quantidade de risco ser diminuída do risco total. Os projetos são agendados para manter a menor quantidade de riscos durante a geração de energia elétrica em um horizonte de planejamento.

Um projeto é definido pela seguinte lista de parâmetros:

1. Identificador: número inteiro que identifica o projeto.
2. Risco reduzido: que representa a quantidade de risco diminuído no fim do projeto.
3. Tipo de projeto: indica se é um projeto de manutenção ou não manutenção.
4. Mês de parada: para projetos de manutenção, ele indica o mês em que começará o processo de parada.
5. Duração de parada: para projetos de manutenção, ele indica a duração de parada em meses a partir do mês de parada.
6. Categoria de recurso usado: o tipo de recurso que o projeto utiliza, pode ser OPEX ou CAPEX. Um projeto pode usar recursos de apenas uma categoria.
7. Sequência de custo mensal do projeto: descreve a quantidade de recursos que o projeto consumirá em cada mês de sua duração, começando no mês inicial. A duração de um projeto é o número de meses necessários para sua execução.

### 2.1.4 Projetos de Manutenção

Cada usina elétrica é associada a uma localização. Além de sua localização as usinas também são associadas a certas divisões administrativas. Ambas associações possuem influência na operação da usina elétrica, o que deve se refletir nas restrições do problema. Embora ambas associações possuem impacto na geração de energia elétrica o que deveria refletir nas restrições do problema, nos decidimos simplifica o modelo do problema ignorando tais associações.

Uma usina possui uma quantidade de unidades geradoras. Eventualmente essas unidades geradoras podem parar por diversos motivos como eventos climáticos e operações de manutenção. A execução de projetos de manutenção pode ser afetada por paradas simultâneas de unidades. Por exemplo, se uma unidade geradora parar para manutenção, outras unidades na mesma usina devem continuar funcionando, o que pode proibir execuções simultâneas de projetos de manutenção em unidades geradoras desta usina. A impossibilidade de execução simultânea de certos projetos de manutenção gera uma dependência de exclusão entre os projetos de manutenção. Apenas projetos de manutenção que causam a parada de unidades geradoras são afetados pela dependência de exclusão.

A representação de localização usada em Mira e outros [19] será substituída neste trabalho por uma matriz de projetos de manutenção, onde cada posição representa a relação entre dois projetos, onde 1 significa que não é possível realizar simultaneamente os dois projetos e 0 significa que é possível realizar simultaneamente os dois projetos.

## 2.2 Formalização do problema

Nesta seção será apresentada a formalização do problema e suas restrições.

Seja $I$ um grupo de projetos da entrada do problema de PPS, e $T$ o tamanho do horizonte de planejamento em meses. É assumido que T sempre seja um número inteiro de meses que possa ser dividido por 12. Para cada projeto $p \in I$ sua duração em meses é representada por $d_p$. Representamos o intervalo de $[1, n]$ pela notação $[n]$ em que *n* deve ser um inteiro positivo.

O portfólio $P$ é representado por um conjunto de pares $(p, m)$ onde $p$ é um projeto dentro de $I$ e $m$ é o mês em que o projeto $p$ é agendado para iniciar, isto é, $P \subseteq I \times [T]$. O conjunto de projetos presentes em um portfólio $P$ é representado por $I_P \subseteq I$.

Um portfólio $P$ é considerado válido caso obedeça às seguintes restrições:

**Agendamento consistente:** Um projeto não pode ser agendado para começar em dois meses diferentes.

$$\texttt{Se } (p, m_1) \in P \texttt{ e } (p, m_2) \in P \texttt{ então } m_1 = m_2\texttt{, para todo } p \in I_P. \tag{2.1}$$

Isso significa que um portfólio válido $P$ é uma função $P : I_P \mapsto [T]$. Vamos usar a notação de $m_p = P(p)$ para indicar o mês de início de execução de projeto e $e_p = m_p + d_p - 1$ será o último mês de execução do projeto $p$, quando $p \in I_P$.

**Recursos limitados:** Cada projeto possui um custo mensal de uso de recursos que começa no mês inicial e dura até o seu mês final. Todos os recursos gastos durante a execução do projeto são de apenas uma categoria. O consumo de recursos anual de certo tipo contando todos os projetos que estão em execução em certo ano não pode passar da quantidade disponível de recursos para aquele ano.

O conjunto da categoria de recursos é $Q = \{CAPEX, OPEX\}$. Para qualquer projeto $p \in I$, seja $r_{p,m}$ o consumo de recurso do projeto $p$ durante o m-ésimo mês de execução, onde $1 \leq m \leq d_p$. Para um portfólio $P$, seja $P_{t,q}$ o conjunto de todos os projetos de classe $q \in Q$ que

estão sendo executados no mês $t \in [2T]$, isto é $P_{t,q} = \{p \in I_P | (p,t) \in P \text{ e } t \in [m_p, e_p]\}$. Então

$$c_{P,q,t} = \sum_{p \in P_{t,q}} r_{p,t-m_p+1}, \qquad t \in [2T], q \in Q, \texttt{P é um portfólio} \tag{2.2}$$

onde $c_{P,q,t}$ é o consumo de recursos no mês $t$ para todos os projetos em execução da categoria $q$ que já estão agendados no portfólio $P$. Podemos agora totalizar o consumo de recursos de projetos da classificação $q$ para um ano $y$ em um portfólio $P$:

$$C_{P,y,q} = \sum_{t \in M_y} c_{P,q,t}, \qquad y \in [1, T/12], q \in Q, \tag{2.3}$$

onde $M_y = [12y - 11, 12y]$ é o intervalo de meses correspondente ao ano y do horizonte de planejamento, e $P$ é um portfólio.

Então se $S_{y,q}$ é o total de recursos disponíveis em um ano $y$ de classificação $q$, temos essa restrição:

$$C_{P,y,q} \leq S_{y,q} \texttt{ para todos } y \in [1, T/12], \quad q \in Q, P \texttt{ é um portfólio.} \tag{2.4}$$

**Restrição de manutenção:** Projetos de manutenção que resultam na parada das mesmas unidades geradoras não podem ter o período de parada ocorrendo simultaneamente. Seja um projeto de manutenção $p$ tal que seu mês de parada é representado por $c_p$ e a duração da parada é representada por $s_p$. Seja $A$ uma matriz quadrada de ordem $\|I\|$ tal que $a_{j,k} = 1$ quando os projetos $j$ e $k$ fazem manutenção com parada nas mesmas unidades geradoras; caso contrário, temos $a_{j,k} = 0$.

$$\texttt{Se } a_{j,k} = 1 \texttt{ e } j \in I_P \texttt{ e } k \in I_P\texttt{, então } [c_j, c_j + s_j - 1] \cap [c_k, c_k + s_k - 1] = \emptyset \tag{2.5}$$

### 2.2.1 Função objetivo

Um PPS pode ser agendado para cumprir diferentes objetivos (reduzir custos, recursos humanos, e riscos) dependendo do que a empresa deseja focar. O objetivo da função neste trabalho é encontrar um portfólio nos quais os projetos são agendados de tal forma a reduzir o risco na maior quantidade possível ao longo do tempo. A função objetivo é dada pela Fórmula 2.6:

$$\texttt{Minimizar } R(P) = 2T.R_{total} - \sum_{p \in I_P} r_p(2T - e_p) \tag{2.6}$$

onde $P$ é um portfólio e $R_{total}$ é o risco total a ser diminuído em um conjunto $I$ de projetos. A variável $r_p$ é o risco diminuído por um projeto $p \in I_P$, e $e_p$ representa o mês em que o projeto $p$ é finalizado.

Simplificando, a função objetivo usa o pior cenário onde nenhum risco é reduzido durante todo o tempo de agendamento e de execução de projetos, que é dado pela multiplicação de $2T.R_{total}$ e diminui desse valor a somatória dos riscos multiplicados pelo número de meses em que ele está reduzido de todos os projetos executados $\sum_{p \in I_P} r_p(2T - e_p)$. O resultado da função objetivo é o produto do risco não resolvido pelo tempo de agendamento e execução que existe para um portfólio $P$.

## 2.3 Metaheurística de Otimização de Colônia de Formigas

A heurística do ACO foi criada a partir de observações feitas em como as colônias de formigas conseguem gerar um caminho do ninho até seu alimento. Foi descoberto que ao caminhar da comida até o ninho a formiga deposita no chão uma substância química chamada de feromônio, que influência no caminho escolhido por outras formigas fazendo com que elas priorizem andar onde o feromônio está mais concentrado, o que gera uma trilha de feromônios. Por meio desse método, as formigas conseguem encontrar um caminho para diferentes fontes de comida de maneira eficiente através de trilhas de feromônios [10].

Um problema no qual pode ser aplicado o ACO deve ter as seguintes características conforme descrito por Dorigo [7, 8]:

- Um conjunto finito de componentes $C = \{c_1, c_2, \ldots, c_{N_C}\}$.
- $L = \{l_{c_i,c_j} | (c_i, c_j) \in \widetilde{C}\}$, onde $|L| \leq N_C^2$, um conjunto finito que corresponde as conexões dos elementos de $\widetilde{C}$, onde $\widetilde{C}$ é um subconjunto do produto cartesiano $C \times C$.
- $J_{c_i,c_j} \equiv J(l_{c_i,c_j}, t)$ é uma função que corresponde ao custo das conexões para cada $l_{c_i,c_j} \in L$, geralmente associado com um valor de tempo $t$.
- $\Omega \equiv \Omega(C, L, t)$ é o conjunto finito de restrições atribuídas aos elementos de $C$ e $L$.
- $s = \langle c_i, c_j, \ldots, c_k, \ldots \rangle$ é uma sequência de elementos em $C$ . Essa sequência pode também ser chamada de estado do problema. Se $S$ é o conjunto de todas as sequências possíveis, e se $\widetilde{S}$ é o conjunto de subsequências que são possíveis respeitando as restrições $\Omega(C, L, t)$, então $\widetilde{S}$ é um subconjunto de $S$. Os elementos de $\widetilde{S}$ são todas as soluções válidas para o problema. O tamanho de uma sequência $s$ é dado por $|s|$.
- Considerando dois estados $s_1$ e $s_2$, definimos uma estrutura de vizinhança como: o estado de $s_2$ é vizinho de $s_1$ caso os dois estejam em $S$, e possa ser chegado ao estado $s_2$ a partir do estado $s_1$ com apenas uma passo lógico, ou seja, se $c_1$ é o ultimo componente na sequência $s_1$, deve existir $c_2 \in C$ tal que $l_{c_1,c_2} \in L$ e $s_2 \equiv \langle s_1, c_2 \rangle$. A vizinhança de um estado $s$ é denotado por $\mathcal{N}_s$.
- $\psi$ é uma solução possível dentro de $\widetilde{S}$ que satisfaz todos os requerimentos do problema.
- $J_\psi(L, t)$ é o custo de cada solução $\psi$. $J_\psi(L, t)$ é a função que calcula o custos de$J_{c_i,c_j}$ para todas as conexões de uma solução $\psi$.

Podemos definir um modelo $B$ de ACO como $B = (S, \Omega, f)$ onde $S$ é um espaço de soluções, $\Omega$ é o conjunto de restrições e $f$ é uma função objetivo a ser reduzida [6].

Para resolver o problema $B$ é usado um conjunto de formigas, onde cada formiga criada é capaz de construir uma solução. Cada formiga possui uma memória que guarda o seu caminho percorrido para determinar se a solução é valida e o valor da função objetivo para está solução. As decisões das formigas são influenciadas por um valor heurístico $\eta$ e o valor de feromônio $\tau$ que existem nas ligações entre os componentes.

Os valores de $\eta$ são definidos de acordo com o problema e representam a atratividade entre as ligações de dois componentes, e geralmente permanece fixo durante toda a execução do

ACO. O $\tau$ representa os valores de feromônio contidos entre as ligações dos componentes, o que indica como o caminho foi avaliado por outras formigas. O $\tau$ é inicializado com um valor de $\tau_0$ em todas as ligações e é constantemente modificado durante a execução do ACO.

Uma formiga tem as seguintes características:

- Toda formiga procura pela solução válida de menor custo $\widehat{J} = \texttt{min}_{\psi}\ J_{\psi}(L, t)$.
- Toda formiga $k$ possui uma memória $\mathcal{M}^k$, ou seja uma subsequência que representa seu caminho atual. A memória é usada para construir e avaliar soluções possíveis e refazer o caminho de volta.
- Uma formiga $k$ em um estado $s_r = \langle s_{r-1}, i\rangle$ pode se mover para qualquer sequência $j$ em uma vizinhança possível $\mathcal{N}_i^k$ , onde $\mathcal{N}_i^k = \{j \mid (j \in \mathcal{N}_{s_r}) \wedge (\langle s_r, j\rangle \in \widetilde{S})\}$. Este movimento é decidido por uma regra de análise de probabilidade.
- Uma formiga $k$ pode ser atribuída a um estado inicial $s_s^k$ e uma ou mais condições de parada $e^k$.
- Uma formiga é iniciada em um estado inicial e move para um vizinho possível, construindo uma solução de modo incremental. O processo de construção de solução acaba quando para ao menos uma formiga $k$ tenha pelo menos uma condição de parada $e^k$ satisfeita.
- A regra de decisão por probabilidade dos movimentos da formiga é dada em função de: (i) um valor armazenado em um nó de uma estrutura $\mathcal{A}_i = [a_{i,j}]$ chamada de tabela de rota de formiga, que é obtida por meio de uma função da composição de feromônios disponíveis no nó e valor heurístico, (ii) a memória privada do caminho armazenado da formiga e (iii) as restrições do problema.
- Quando uma formiga move de um nó $i$ para um nó vizinho $j$, a formiga pode atualizar a trilha de feromônio $\tau_{i,j}$ no segmento $(i, j)$. Isso é chamado de atualização online de feromônio passo a passo.
- Assim que é criado um caminho, a formiga pode refaze-lo e atualizar a trilha de feromônio. Isto é chamado de atualização online de feromônio atrasada.
- Assim que a formiga constrói a solução e se for o caso, refaz o seu caminho de volta ao nó inicial, a formiga morre e sua memória é liberada.

Mesmo que uma formiga seja complexa o suficiente para encontrar uma solução válida sozinha, boas soluções só podem ser achadas com a interação entre várias formigas de uma colônia. As formigas usam informações privadas e do ambiente para a tomada de decisão e não tem sua decisão influenciada diretamente por uma outra formiga, mas pela informação que é deixada nas trilhas de feromônio. As formigas adaptam como o problema é representado, mas não são capazes de se readaptar sozinhas.

Para que a colônia de formigas não se encontre em uma solução sub-ótima, existe um procedimento chamado de evaporação global de feromônio, que automaticamente diminui a concentração de feromônios global. Isso faz com que as formigas explorem uma variedade maior de caminhos.

Opcionalmente também podem ser implementados *daemon actions* que são procedimentos globais não realizados pelas formigas e que atualizam a quantidade de feromônios dos caminhos. Como um exemplo prático, o *daemon* pode observar o caminho encontrado por cada formiga na colônia e optar por depositar feromônio extra nos arcos usados pela formiga que fez a melhor solução. Essas ações podem também ser chamadas de atualizações *offline* de feromônios.

**Algoritmo 1:** meta-heurístico_ACO($B(S, \Omega, f)$, numAnts, numIterations, $\alpha, \beta, \rho, \tau_0, \varphi$)

```
1 início
2     inicializar_parâmetros();
3     while numIterations > 0 do
4         geração_de_formigas_e_atividade(numAnts) ;
5         evaporação_de_feromônio();
6         daemon_actions(); // opcional
7         numIterations = numIterations - 1;
8     end
9 fim
```

O Algoritmo 1 *meta-heurístico_ACO* chama as funções responsáveis por construir a solução. O algoritmo recebe como parâmetro um problema $B$, numAnts, numIterations e um conjunto opcional de parâmetros que são inicializados na linha 2 pela função *inicializar_parâmetros()*. Caso não seja recebido o valor de algum desses parâmetros, é usado um valor predefinido pelo programador.

A trilha de feromônios $\tau$, valor de atratividade $\eta$ e a tabela de rota de formiga $\mathcal{A}$ são inicializados na linha 2 do Algoritmo 1. A maneira que $\mathcal{A}$ e $\eta$ são calculados é discutida na Fórmula 2.7.

Após a inicialização, o algoritmo entra em um laço que é executado um número de vezes que é representado por *numIterations*. Dentro do laço são chamadas as funções *geração_de_formigas(numAnts)* na linha 4, *evaporação_de_feromônio()* na linha 5 e *daemon _actions()* na linha 6 caso houver, e na linha 7 atualiza o número de iterações é atualizado. Não é especificado se as ações devem ser executadas em paralelo ou se necessário sincronizadas, deixando a critério do programador como esses procedimentos devem interagir.

**Algoritmo 2:** geração_de_formigas(numAnts)

```
1 início
2     while numAnts > 0 do
3         nova_formiga() ;
4         numAnts = numAnts - 1;
5     end
6 fim
```

O Algoritmo 2 *geração_de_formigas()* é responsável pela criação das formigas. Na linha 2 há um início de um laço de repetição que deve ser executado um número de vezes igual ao número de formigas(numAnts). A função *nova_formiga()* na linha 3 gera uma nova formiga capaz de criar uma solução. O parâmetro numAnts geralmente é equivalente ao número de componentes do problema $N_C$.

**Algoritmo 3:** Procedimento de nova_formiga()

```
1  início
2      inicializar_formiga();
3      M = atualizar_memória();
4      while estado_atual ≠ estado_alvo do
5          A = ler_tabela_de_rota_local() ;
6          P = probabilidades_computada_de_transição(A, M, Ω );
7          próximo_estado = aplica_decisão_da_formiga(P);
8          mover_para_próximo_estado(próximo_estado) ;
9          if atualização_online_de_feromônio_passo_a_passo then
10             deposita_feromônio_passo_a_passo();
11             atualiza_tabela_de_rota_de_formiga();
12         end
13         M = atualiza_estado_interno();
14     end
15     if atualização_online_de_feromônio_com_atraso then
16         foreach ligação_visitada ∈ M do
17             deposita_feromônio_com_atraso(ligação_visitada, M);
18             atualiza_tabela_de_rota();
19         end
20     end
21     S = melhor_sequência(M, S);
22     morre();
23 fim
```

O Algoritmo 3 nova_formiga() cria uma formiga $k$ que gera uma solução. Na linha 2 o método *inicializar_formiga()* posiciona uma formiga em um componente aleatório e este componente é adicionado na memória $\mathcal{M}^k$ pelo método da linha 3 *atualizar_memória()*.

Na linha 4 o algoritmo entra em um laço no qual em cada iteração a formiga $k$ escolhe um novo componente para se posicionar. Esse laço se repete até o estado atual da formiga ser o estado alvo que corresponde a uma solução válida. Então no começo do laço na linha 5 a formiga lê a tabela de rota e obtém o valor de $\mathcal{A}$ para todos os vizinhos do componente onde se encontra a formiga com a função *ler_tabela_de_rota_local()*. Com o valor obtido de $\mathcal{A}$, a memória da formiga ($\mathcal{M}$) e as restrições do problema ($\Omega$), é calculada a probabilidade da formiga escolher um próximo componente com a função *probabilidades_computada_de_transição($\mathcal{A}, \mathcal{M}, \Omega$)* da linha 6. O cálculo da probabilidade $\mathcal{P}$ será discutido mais a frente na Formula 2.8. O próximo estado para o qual a formiga deve se mover é escolhido por meio da probabilidade $\mathcal{P}$ na linhas 7. Finalmente, na linha 8, a formiga move-se para o próximo estado.

Caso a atualização de feromônios passo a passo esteja ativa na linha 9, então os feromônios na ligação entre os componentes são atualizados após cada iteração. Na linha 10 é realizada a atualização de feromônio passo a passo que é discutida na Formula 2.10 e logo em seguida na linha 11 a tabela de rota de formiga é atualizada de acordo com a Formula 2.7. No fim do laço na linha 13 o componente escolhido é adicionado na memória $\mathcal{M}^k$ da formiga $k$ e o laço acaba na linha 14.

Entre as linhas 15 e 20 a atualização online de feromônio com atraso é executado caso

ela esteja ativada. Na linha 16 executa um *foreach* que percorre em cada ligação visitada na solução formada pela formiga e deposita uma quantidade de feromônio baseada na qualidade da solução usando a função da linha 17 *deposita_feromônio_com_atraso(ligação_visitada, $\mathcal{M}$)*. A quantidade de feromônio depositada é dependente da solução criada pela formiga. Na linha 18 a tabela de rota de formigas $\mathcal{A}$ é atualizada. Os processos de atualização online de feromônio passo a passo e a atualização online de feromônio com atraso normalmente são mutualmente exclusivos, no entanto, caso os dois não estejam ativos, a atualização de feromônio é feita pelo *daemon actions*. A atualização de feromônios obedecem às Fórmulas 2.9, 2.10.

Na linha 21 caso a sequência $\mathcal{M}^k$ seja melhor do que a melhor sequência $S$ atual, então esta sequência $\mathcal{M}^k$, ou seja, a solução recém construída pela formiga, se torna a melhor sequência $S$. A formiga morre logo em seguida na linha 22.

Cada linha da tabela de rota de formiga $\mathcal{A}_i = [a_{i,j}(t)]$ do componente $i$, onde $\mathcal{N}_i$ é o conjunto de vizinhos do componente $i$, é construída usando as trilhas de feromônio de $\tau_{i,j}(t)$ e os valores da heurística $\eta_{i,j}$ da seguinte maneira:

$$a_{i,j} = \frac{[\tau_{i,j}(t)]^\alpha [\eta_{i,j}]^\beta}{\sum_{l \in \mathcal{N}_i} [\tau_{i,l}(t)]^\alpha [\eta_{i,l}]^\beta}, \quad \forall j \in \mathcal{N}_i \tag{2.7}$$

onde os valores de $\alpha$ e $\beta$ são usados para determinar o peso dos valores de feromônio ($\tau_{i,j}(t)$) e de heurística ($\eta_{i,j}$) respectivamente. O valor da heurística $\eta_{i,j}$ é determinado de acordo com o problema. Por exemplo, no problema do Caixeiro Viajante são usadas as distâncias entre as cidades tal que $\eta_{i,j} = 1/J_{c_i,c_j}$, onde $J_{c_i,c_j}$ representa a distância entre as cidades $c_i$ e $c_j$. Sempre que houver uma mudança em $\tau$ ou $\eta$, a tabela de rota de formiga deve ser atualizada.

A probabilidade de uma formiga $k$ escolher o próximo componente vizinho é representada por $p^k_{i,j}$, onde a formiga no componente $i$ deve escolher uma cidade vizinha $j \in \mathcal{N}^k_i$ e esta probabilidade da escolha é dada pela seguinte função:

$$p^k_{i,j} = \frac{a_{i,j}(t)}{\sum_{l \in \mathcal{N}^k_i} a_{i,l}(t)} \tag{2.8}$$

onde $\mathcal{N}^k_i \subseteq \mathcal{N}_i$ são os componentes vizinhos possíveis de um componente $i$ para uma formiga $k$.

Caso a atualização online de feromônio com atraso esteja ativa, todas as formigas depositam uma quantidade de feromônios após completarem sua solução. A quantidade de feromônio depositado é $\Delta\tau^k(t) = 1/J^k_\psi(t)$ para cada conexão $l_{i,j}$ dentro da solução da formiga $k$ em uma iteração $t$, e $J^k_\psi(t)$ corresponde a custo da solução $\psi^k(t)$:

$$\tau_{i,j}(t) \leftarrow \tau_{i,j}(t) + \Delta\tau^k(t), \quad \forall l_{i,j} \in \psi^k(t), k = 1, \ldots, m \tag{2.9}$$

onde o número de formigas $m$ geralmente é dado pelo número de elementos em $C$ e é mantido constante durante todo o procedimento.

Caso a atualização online de feromônio passo a passo esteja ativa, sempre que uma formiga se move de um componente $c_i$ para um componente $c_j$ ela atualiza o valor de feromônio $\tau_{i,j}$ da conexão $l_{i,j}$ seguindo a regra

$$\tau_{i,j}(t) \leftarrow (1 - \varphi)\tau_{i,j}(t) + \varphi\tau_0 \tag{2.10}$$

onde $0 < \varphi \leq 1$.

Sempre que as formigas montam suas soluções e atualizam o feromônio, é chamada a função de evaporação de feromônios que atualiza todas as conexões $l_{i,j}$. A quantidade de feromônio a ser diminuída é controlada pelo parâmetro $\rho$ e a regra de atualização é:

$$\tau_{i,j}(t) \leftarrow (1 - \rho)\tau_{i,j}(t) \tag{2.11}$$

onde $0 < \rho \leq 1$ .

Os paramentos usados geralmente são $\alpha = 1$, $\beta = 5$ e $\rho = 0.5$ e a quantidade inicial de feromônio $\tau_{i,j}(0)$ usada é uma constante inteira e positiva $\tau_0$ de valor baixo que é atribuído para todos os ligações.

O funcionamento do ACO pode ser representado pelo fluxograma mostrado na Figura 2.1. No fluxograma primeiro inicializamos os parâmetros do algoritmo e damos início ao nosso primeiro *loop* que verifica se a população de formigas é completa, no qual em cada iteração uma formiga é criada para gerar uma solução. Cada formiga é inicializada em uma componente aleatório e entra no próximo passo que é formar uma solução. Nesta etapa as formigas definem sua vizinhança e a probabilidade de um componente da mesma ser escolhido, e por fim adicionam este componente a sua solução com base na probabilidade gerada. Caso a atualização de feromônio *online* esteja ativa, o feromônio da ligação usada é incrementado. Após a solução estiver completa, ela é armazenada caso seja a melhor gerada até o momento, e por fim é efetuada a atualização *offline* de feromônio se esta estiver ativa. Este processo de gerar soluções é executado uma vez para cada formiga até o limite de tamanho de população de formiga seja atingido e então seguimos para o próximo passo que é a evaporação de feromônio. Após a evaporação do feromônio, é verificado se o número de iterações desejadas foram executadas. Caso o número não tenha sido atingido, o algoritmo volta a gerar soluções para todas as formigas de uma nova população; caso contrario, o algoritmo chega ao seu fim.

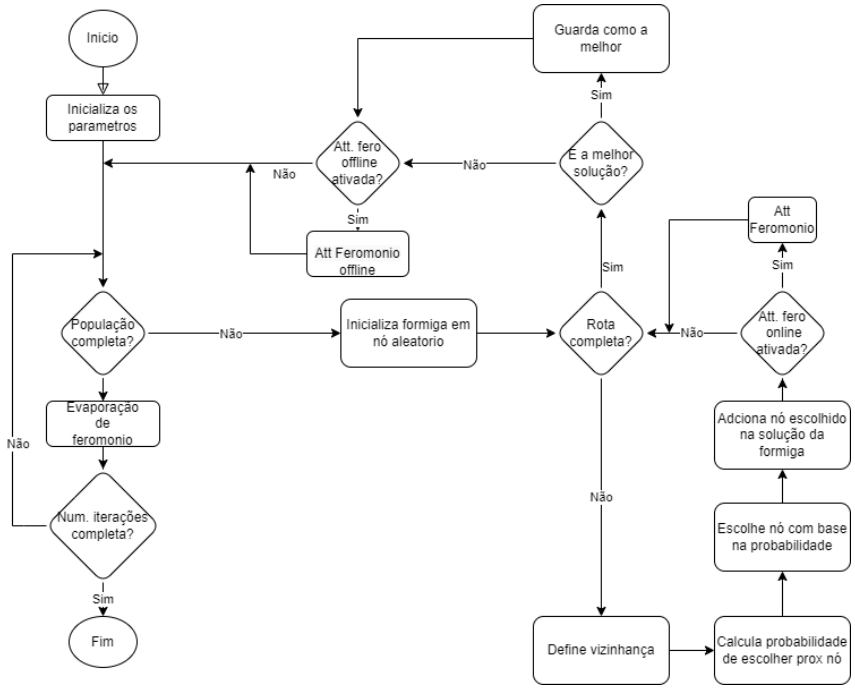


Figura 2.1 – Fluxograma ACO.

A heurística do ACO tem sido usada em diferentes aplicações como: *Data Mining* [23], *Shop Scheduling Problems* [1], *Quadratic Assignment Problem* [25], *Dynamic Vehicle Routing Problem* [21]. O ACO também é utilizado para resolver problemas de PPS multiobjetivo [4, 3].

### 2.3.1 Caixeiro Viajante

Uma das primeiras aplicações do ACO foi para resolver o problema do Caixeiro Viajante [9]. Nesta seção iremos apresentar um exemplo de como o ACO pode ser usado para resolver este problema. O Caixeiro Viajante é um problema que consiste em, dado um conjunto de cidades ligadas por rodovias, o caixeiro deve visitar todas as cidades e voltar ao ponto inicial usando o menor caminho sem passar pela mesma cidade mais de uma vez [12].

Usando os termos definidos na Seção 2.3, o problema do Caixeiro Viajante pode ser definido como um problema de ACO [7] tal que $C$ é o conjunto de cidades, $L$ é o conjunto de caminhos entre as as cidades de $C$ que estão totalmente conectadas, e $J$ é a matriz de distância entre as cidades tal que $J_{c_i,c_j}$ representa a distância entre as cidades $c_i$ e $c_j$. Formalmente, o problema do Caixeiro Viajante consiste em achar o menor circuito Hamiltoniano (ciclo em um grafo que passa por todos os vértices somente uma vez) em um grafo $G = (C, L)$. Caso um par de cidades que seja modelado no problema do Caixeiro Viajante não possua um caminho que as ligue, então podemos assumir que a aresta correspondente no grafo $G$ possui uma distância com valor infinito.

Considere uma instância do problema de Caixeiro Viajante para a qual o conjunto de cidades é o conjunto $C = \{1, 2, 3, 4\}$, as distâncias entre as cidades são encontradas na matriz $J$ dada a seguir:

$$\mathbf{J} = \begin{pmatrix} \infty & 30 & 10 & 15 \\ 30 & \infty & 20 & 5 \\ 10 & 20 & \infty & 40 \\ 15 & 5 & 40 & \infty \end{pmatrix}$$

Usaremos os valores de parâmetros $\alpha = 1$, $\beta = 1$, $\rho = 0.5$ e $\tau_0 = 1$ para o algoritmo do ACO. Os valores de $\eta$ são dados por $\eta_{i,j} = 1/J_{ci,cj}$ para cada par de cidades $c_i$ e $c_j$. Os valores de $\tau$ e $\eta$ são dados nas matrizes a seguir:

$$\tau = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \qquad \eta = \begin{pmatrix} \infty & 0.033 & 0.1 & 0.066 \\ 0.033 & \infty & 0.05 & 0.2 \\ 0.1 & 0.5 & \infty & 0.025 \\ 0.066 & 0.2 & 0.025 & \infty \end{pmatrix}.$$

Conforme os valores dos parâmetros $\alpha$ e $\beta$ junto dos valores das matrizes $\tau$ e $\eta$, usando a Fórmula 2.7 temos a seguinte tabela de rota de formiga:

$$\mathcal{A} = \begin{pmatrix} 0 & 0.165 & 0.502 & 0.331 \\ 0.116 & 0 & 0.176 & 0.706 \\ 0.571 & 0.285 & 0 & 0.142 \\ 0.226 & 0.687 & 0.085 & 0 \end{pmatrix}.$$

Vamos resolver uma iteração da formiga que começou na cidade 1. Usando os valores de $a_{c_1,cj}$ para todas as cidades $c_j$ vizinhas de $c_1$. Temos os seguintes valores de $A_{c_1} =$

$[0, 0.165, 0.502, 0.331]$. Com os valores de $A_{c_1}$ se determina a seguinte probabilidade da formiga escolher uma das cidades dentro da vizinhança possível $P_{c_1} = [0\%, 16.5\%, 50.3\%, 33.16\%]$. Neste caso a formiga irá priorizar a cidade 3 que possui $50.3\%$ de chance de ser escolhida, pois esta cidade é a que tem menor distância e além disso o feromônio tem o mesmo valor para todas as ligações.

Vamos considerar o mesmo cenário anterior, mas com o valor do $\tau_{1,2} = 5$, ou seja vamos aumentar o feromônio apenas para a ligação entre as cidades $c_1$ e $c_2$. Os novos valores da tabela de rota de formigas para a cidade $c_1$ são $A_{c_1} = [0, 0.498, 0.302, 0.199]$. Com a mudança no $\tau_{1,2}$ as probabilidades ficam $P_{c_1} = [0\%, 49.84\%, 30.23\%, 19.91\%]$, ou seja, mesmo sendo $J_{c1,c2}$ a ligação com maior distância, a formiga prioriza ir para a cidade $c_2$ por causa da trilha de feromônio.

Agora vamos considerar que duas formigas fizeram 2 caminhos distintos representados na memória $\mathcal{M}^1 = [c_1, c_3, c_2, c_4]$ e $\mathcal{M}^2 = [c_1, c_2, c_4, c_3]$. Temos $\Delta\tau^1 = 1/50$ e $\Delta\tau^2 = 1/85$ e após a atualização de feromônio a trilha de feromônios alterada pelas duas formigas é a seguinte:

$$\tau = \begin{pmatrix} 0 & 1.011 & 1.02 & 1 \\ 1 & 0 & 1 & 1.031 \\ 1.011 & 1.02 & 0 & 1 \\ 1.02 & 1 & 1.011 & 0 \end{pmatrix}$$

onde o feromônio só é atualizado onde as formigas passaram.

Depois que as formigas constroem a solução e depositam o feromônio, a função de evaporação de feromônio é ativada. Dado o valor de $\rho = 0.5$, a função de evaporação de feromônio vai reduzir o valor de feromônio pela metade para todas as ligações. A matriz de trilha $\tau$ é atualizada para

$$\tau = \begin{pmatrix} 0 & 0.505 & 0.51 & 0.5 \\ 0.5 & 0 & 0.5 & 0.515 \\ 0.505 & 0.51 & 0 & 0.5 \\ 0.51 & 0.5 & 0.505 & 0 \end{pmatrix}.$$

Vamos resolver uma instância real do problema do Caixeiro Viajante conhecida como `berlin52` encontrado no *The Symmetric Traveling Salesman Problem Instances* [27], que corresponde a 52 localizações dentro da cidade de Berlin, conforme apresentado na Figura 2.2. Este problema tem uma solução ótima conhecida de 7542m.

Para resolver esta instância será usado um programa chamado *aco-tsp* que implementa o ACO usando *python* disponível no *GitHub* [18]. Nos realizamos um experimento no qual o programa foi executado 3 vezes. Os parâmetros do ACO usados na execução são: numIterations = 80, numAnts = 50, $\alpha$ = 1, $\beta$ = 1, $\tau_0$ = 1, $\rho$ = 0.5. Os resultados obtidos ao final das 3 execuções foram os dos caminhos de distância 13929.686m (mostrado na Figura 2.3), 15458.266m (mostrado na Figura 2.4) e 15815.757m (mostrado na Figura 2.5). Usando diferentes valores de parâmetros para o programa ACO baseados nas características da instância, é possível encontrar soluções ainda melhores.

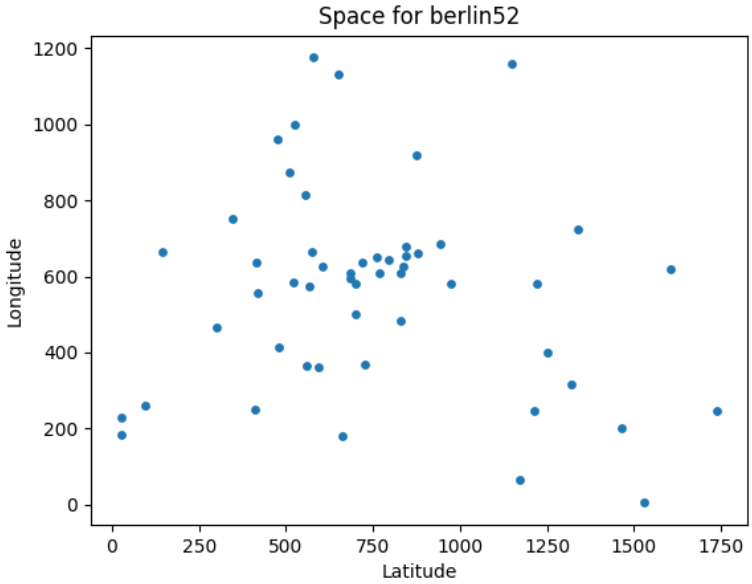


Figura 2.2 – As 52 localizações dentro de Berlin de acordo com a instância berlin52.

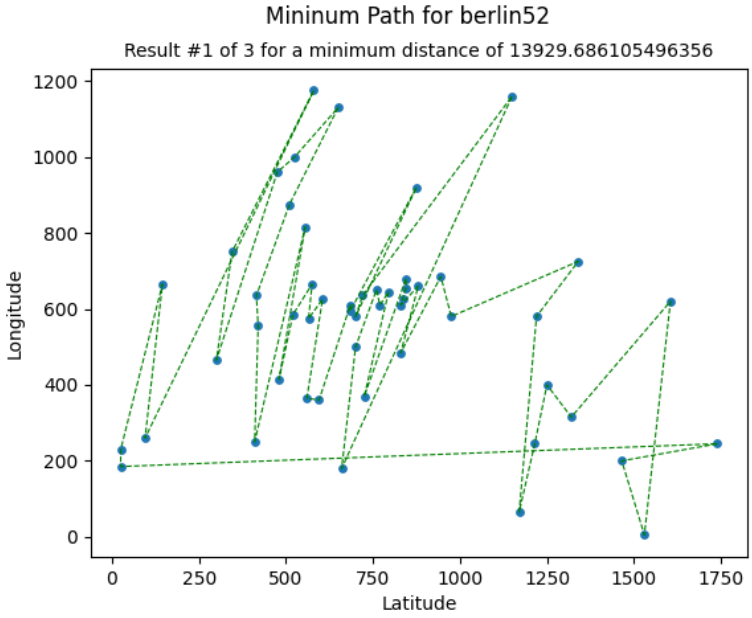


Figura 2.3 – Menor caminho gerado na execução 1 do programa aco-tsp.

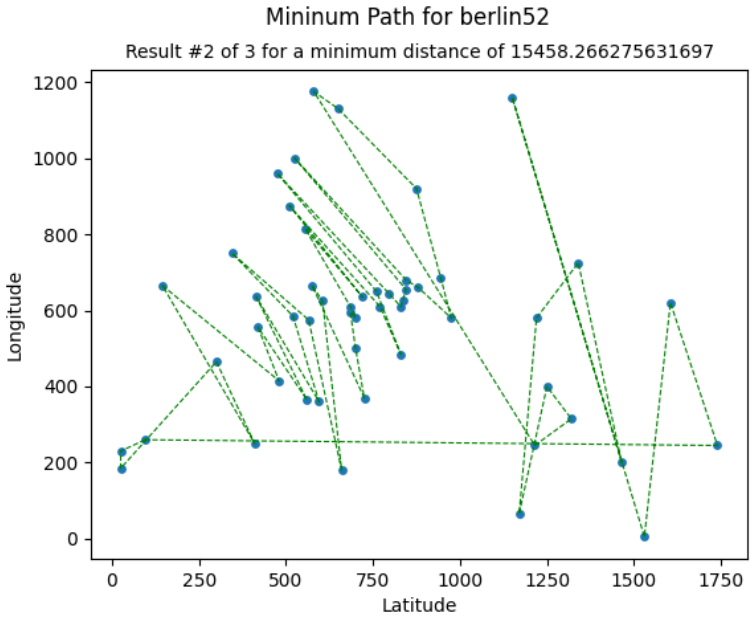


Figura 2.4 – Menor caminho gerado na execução 2 do programa aco-tsp.

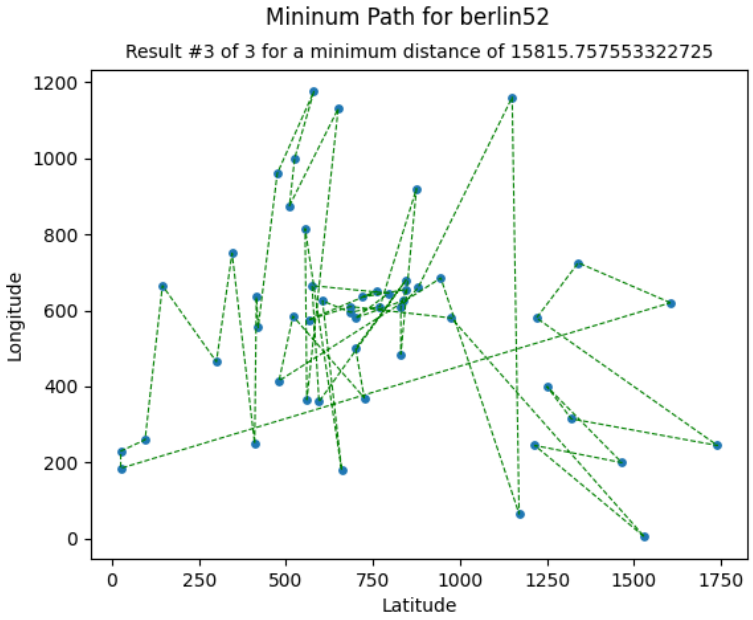


Figura 2.5 – Menor caminho gerado na execução 3 do programa aco-tsp.

## 2.4 Modelagem PPS para ACO

Nos trabalhos de Doerner e outros [3, 4], o problema de PPS é modelado para o ACO, porém ele é diferente do abordado neste trabalho, porque no problema de Doerner é decidido qual o melhor grupo de projetos a serem agendados com os recursos disponíveis sem a preocupação de quando os agendar, o que faz a modelagem para resolvê-lo ser diferente da que será utilizada neste trabalho.

Outra diferença encontrada no problema PPS estudado por Doerner é que ele possui 4 funções objetivo. A heurística de ACO proposta pelo autor baseia-se em definir uma formiga para cada uma destas funções objetivo. No nosso caso, ao contrário do problema definido por Doerner, nós temos uma única função objetivo e vamos implementar uma heurística na qual várias formigas tentam encontrar soluções para esta única função objetivo. Esta maneira de implementar o ACO é um dos aspectos originais do nosso trabalho.

Usando os conceitos definidos nas Seções 2.2 e 2.3, o problema PPS será modelado para ser resolvido com o uso do ACO. Para isso o PPS será definido tal que $C$ é o conjunto de projetos a serem agendados e dos meses dentro do horizonte de planejamento tal que o conjunto $C$ pode ser dividido em dois conjuntos $I$ de projetos e $[T+1]$ de meses. Seja $L$ o conjunto de conexões $(u, v)$ tais que ou $u \in I$ e $v \in [T+1]$ ou $u \in [T+1]$ e $v \in I$. Observe que o grafo $G(C, L)$ é um grafo bipartido completo.0

Nós iremos definir a função $J$ que mapeia um par de componentes $(u, v)$ pertencente a L para o conjunto dos números reais R da seguinte maneira:

$$J_{u,v} = \begin{cases} \frac{1}{r_u(2T+1-(v+d_u-1))}; \text{ se } u \in I \text{ e } v \in [T+1]; \\ \\ \frac{u}{r_v.d_v}; \text{ se } u \in [T+1] \text{ e } v \in I; \end{cases} \tag{2.12}$$

São usados $r_i$ para representar o risco reduzido por um projeto $i$ e $d_i$ para representar a duração de um projeto $i$. Para o caso no qual o argumento de entrada é $(u, v)$ para $u \in I$ e $v \in [T+1]$ , a função indica o quanto de risco o projeto resolve e por quanto tempo após o término de sua execução o risco está resolvido. Este valor representa o custo da conexão entre $u$ e $v$. Observe que o denominador de $J_{u,v}$ neste caso aparece na somatória da Fórmula 2.6 da função objetivo do problema PPS. Para o caso no qual o argumento de entrada é $(u, v)$ para $u \in [T+1]$ e $v \in I$, a função indica o custo da conexão entre $u$ e $v$ que representa a prioridade em que um projeto $v$ será agendado, ou seja quanto maior o risco que o projeto pode resolver e, quanto maior sua duração, será maior a chance de ser agendado nos meses iniciais do horizonte de planejamento. Além disso, a função $J$ captura a ideia de distância entre os componentes e, portanto, quanto menor o valor de $J_{u,v}$, melhor será a escolha desta conexão pelas formigas.

A solução do problema ACO é um caminho orientado no grafo $G(C, L)$ cujo os vértices são os componentes em $C$ e as arestas pertencem ao conjunto $L$ de conexões. Podemos representar um caminho orientado no grafo por meio de uma sequência de componentes $s$. O conjunto de todas as sequências que correspondem a caminhos no grafo é chamado de $S$. Caso a sequência $s$ satisfazer um conjunto de restrições $\Omega$, inclusive a propriedade de ser uma sequência que alterna entre projetos e meses, então é uma solução válida do problema do ACO. O conjunto de restrições $\Omega$ a ser considerado são as restrições para um portfólio ser válido: agendamento consistente, recursos limitados e restrição de manutenção. Além das restrições do problema PPS, é necessário também uma restrição de *início válido*: onde o primeiro componente $s_0$ na sequência/solução $s$ deve ser um projeto, ou seja $s_0 \in I$. Logo o conjunto $\widetilde{S}$ são todas sequências válidas que respeitam todas essas restrições.

Seja uma solução $\psi \in \widetilde{S}$ onde todos os projetos estão contidos em $\psi$. A solução para um PPS, no entanto, não é uma sequência de elementos, mas um conjunto de pares ordenados contendo projeto e mês. Logo para gerar uma solução $P$ de PPS a partir de uma solução $\psi$ de ACO, podemos construir o portfólio $P$ identificando cada par ordenado como $(\psi_i, \psi_{i+1})$ onde

$i \in \{1, 3, 5, 7, ..., (2.|I| - 1)\}$. Observe que duas ou mais sequências $\psi$ podem gerar uma única solução em PPS.

A função $J_\psi$ do trabalho de Doerner e outros [3, 4] não atende a função objetivo do problema PPS usada neste trabalho. Logo, por isso, nós definimos $J_\psi$ como

$$3T.R_{total} - \sum_{i \in \{1,3,5,7,...,(2|I|-1)\}} J^{-1}_{\psi_i, \psi_{(i+1)}} \tag{2.13}$$

o que é equivalente a função objetivo do problema PPS.

O valor heurístico de $\eta$ que influencia a formiga a decidir que componente será adicionado à sequência é definido pelo custo $J_{u,v}$ das ligações. Para as ligações entre projetos e meses temos $\eta_{i,t} = 1 - J_{i,t}$, o que identifica a atratividade de um projeto $i$ ser agendado em um mês $t$, e para as ligações de meses a projetos, temos $\eta_{t,i} = 1 - J_{t,i}$, o que indica a atratividade de uma formiga escolher o próximo projeto a ser agendado dado que houve um agendamento no mês $t$.

Como parâmetros para a execução do algoritmo, nós temos $\alpha$, $\beta$, $\rho$, $\varphi$, $\tau_0$, `numAnts` e `numIterations`. Esses valores podem ser alterados para gerar resultados diferentes. No trabalho de Doerner e outros [3] os parâmetros usados que geraram os melhores resultados foram $\alpha = 1$, $\beta = 1$, $\rho = 0.1$, $\varphi = 0.1$, $\tau_0 = 0.01$, `numAnts` $= 10$ e o parâmetro de `numIterations` que representa o número de iterações realizadas pelo algoritmo foi de 100, 200, 300, 400 e 500. Observe que é usado tanto a atualização online de feromônio quanto a atualização atrasada de feromônio no experimento de Doener. Por outro lado neste trabalho será utilizada apenas a atualização atrasada de feromônio. Inicialmente os valores dos parâmetros que nós iremos utilizar para a execução do nosso algoritmo serão os que foram apresentados no trabalho de Doener, e a partir deles, nós iremos encontrar quais são os melhores valores para o nosso problema.

## 2.5 Bibliotecas e Frameworks de ACO

Para implementar o nosso programa ACO e realizar o experimento será utilizado um *framework* de ACO. As *frameworks* encontradas foram *scikit-opt* [16], *formigueiro* [24] e *Isula* [13].

O *scikit-opt* é uma *framework* que disponibiliza um conjunto de algoritmos de enxame incluindo o ACO desenvolvido em *python* pelo programador Guofei disponível no *GitHub* com licença *MIT* atualmente na versão 0.6.5. A biblioteca disponibiliza alguns recursos para melhorar a execução como: função definida pelo usuário, execução continua (a biblioteca permite executar um algoritmo para 10 iterações e, em seguida, executar outras 20 iterações com base nas 10 iterações anteriores) e biblioteca ainda possui um recurso chamado *4-ways to accelerate* que possibilita o uso de vetorização, *multithreading*, multiprocessamento e uso de *cache* para acelerar a execução. A biblioteca também conta com uma boa documentação e alguns exemplos básicos que mostram como começar utiliza-la.

O *formigueiro* é uma *framework* feita para implementar algoritmos simples da heurística ACO desenvolvida em *python* pelo Dilson Lucas Pereira e está disponível no *GitHub*. Ela é uma *framework* simples sem uma documentação detalhada, mas que ainda assim pode implementar as principais características do ACO.

A *Isula* é uma *framework* feita para facilitar a implementação do ACO usando componentes básicos da heurística. A *framework* é desenvolvida em java pelo Carlos Gavidia e está

disponível no *GitHub* com licença *MIT* atualmente na versão 2.0.1. A *framework* tem suporte para os três algoritmos de maior performance do ACO: *Ant System*, *Ant Colony* e *Max-Min Ant System*, esses algoritmos representam diferentes maneiras de executar o ACO. Com o uso da *Isula* esses algoritmos podem ser implementados reusando e adaptando os componentes disponíveis na *framework*. Ela também possui um tutorial de como ser usada e uma boa documentação que conta com um site onde é disponibilizado um *javadoc* para cada um de seus componentes [14], além de exemplos para uso no Caixeiro Viajante, *Flow-Shop Scheduling Problem*, Problema de Cobertura de Conjuntos, Segmentação de imagem binária e *Image Clustering* [15].

Para o desenvolvimento do *software* será usada a *framework Isula* devido apresentar a possibilidade de implementar as principais funcionalidades do ACO de maneira fácil e eficaz, possuir uma melhor documentação se comparado com as demais *frameworks*, além de contar com diferentes exemplos que servem de guia para uso da *framework*.

# 3 Experimentos

Neste capítulo será abordado como foram realizado os experimentos, a escolha de parâmetros e como foram geradas as instâncias do problema. O capítulo apresenta também os resultados encontrados e a análise destes resultados.

## 3.1 Instâncias de Entrada

Para gerar as instâncias de entrada foi criado um programa em java que cria instâncias aleatórias dado um número de projetos e projetos de manutenção a serem criados. A instância gerada é armazenada em um arquivo JSON, que possui todas as informações do problema.

Os recursos OPEX e CAPEX são calculados a partir da soma dos custos de todos os projetos de cada categoria e então são armazenados em dois vetores com tamanho igual à duração em anos do horizonte de planejamento, considerado sempre como cinco anos. O total de recursos de cada categoria é dividido em cinco partes iguais para cada respectivo vetor. A quantidade de recursos disponível é igual à quantidade necessária para agendar todos os projetos.

As instâncias de entradas feitas são divididas pela sua quantidade de projetos; a saber escolhemos os valores de 30, 50, 100, 150 e 200 projetos. A categoria de recursos do projeto é gerada aleatoriamente, e o tipo dos projetos são distribuídos de maneira que de cada três projetos, um é de manutenção.

A duração dos projetos é gerada aleatoriamente entre os valores de 6 a 24 meses. O custo de cada mês do projeto é gerado de maneira aleatória, tal que o valor de custo para cada mês do projeto varia entre 1000 a 2500.

O risco que um projeto $p$ com duração de $d_p$ resolve é calculado da seguinte maneira:

$$r_p = 20 + rand(0, 5d_p), \tag{3.1}$$

onde $rand(i, j)$ é uma função que gera um número aleatório entre $i$ e $j - 1$ inclusive. O valor máximo de risco é definido pela duração do projeto multiplicado por um peso definido arbitrariamente para que o tamanho dos projetos possam impactar no risco padrão. O risco de cada projeto é decidido aleatoriamente e dessa maneira possibilita que alguns projetos menores possam competir com os maiores.

A parada dos projetos de manutenção são geradas a partir de um número aleatório com base na sua duração. O mês de parada, ou seja o início da parada é definido entre o primeiro mês da sua duração até a metade de sua duração. A duração de parada vária de um mês até o tempo de duração que sobra a partir do mês de parada.

A matriz que representa quais projetos de manutenção não podem ser agendados simultaneamente é gerada de maneira completamente aleatória.

## 3.2 Parâmetros de Heurística

Os parâmetros usados na configuração do ACO são os seguintes: $\alpha$ (importância do feromônio), $\beta$ (importância da informação heurística), $\rho$ (evaporação de feromônio).

Os valores dos parâmetros usados no ACO foram definidos a partir de experimentos realizados onde os parâmetros iniciais foram $\alpha = 1$, $\beta = 1$, $\rho = 0.1$, $\tau_0 = 0.01$, $\texttt{numAnts} = 10$ e a partir deles como foi discutido no fim do Capítulo 2.4 cada parâmetro foi alterado individualmente em busca dos melhores resultados. Os resultados destes experimentos se encontram no Apêndice A.

As execuções para determinar os parâmetros em suma maioria foram realizadas com o uso de uma instância de 300 projetos. A única exceção foi o uso de uma instância de 30 projetos para determinar o número de formigas. Após cada execução o valor da função objetivo foi manualmente adicionado a uma tabela e os parâmetros atualizados para o próximo experimento. As tabelas geradas foram usadas para definir os parâmetros para os experimentos finais.

Na execução dos experimentos não foi notada uma grande diferença nos resultados a partir da alteração dos parâmetros, com exceção ao número de formigas, então foram escolhidos os melhores resultados de cada parâmetro. E como o número de formigas foi o único parâmetro que apresentou uma alteração significativa nos resultados, o seu valor acabou sendo muito superior ao de outros trabalhos.

Os valores dos parâmetros com os melhores resultados são os apresentados na Tabela 3.1.

| Parâmetro | Valor | Descrição |
|---|---|---|
| $\alpha$ | 4 | Peso do feromônio |
| $\beta$ | 1,5 | Peso do valor heurístico |
| $\rho$ | 0,2 | Controle do valor de evaporação |
| $\tau_0$ | 3 | Feromônio inicial |
| $NFormigas$ | 5000 | Número de formigas |

Tabela 3.1 – Parâmetros da metaheurística ACO.

## 3.3 Execução dos experimentos

Para realizar o experimento computacional, foi realizada uma busca por bibliotecas e *frameworks* que implementam o ACO no GitHub e encontramos as seguintes *frameworks*: *scikit-opt*, *Formigueiro* e *isula*. Foi decidido o uso da *Isula*, escolha que é discutida no Capítulo 2.

Para auxiliar na implementação foi usada a *IntelliJ IDEA* que é um ambiente de desenvolvimento de software para *Java*. Ela disponibiliza facilidades para o desenvolvimento. Por exemplo ela facilita a depuração, gerencia dependências, e também ajuda na escrita do código.

Com o uso da *Isula* implementamos um programa em java responsável por rodar a metaheurística ACO, que usa um arquivo JSON como instância de entrada do problema. O formato do arquivo de entrada é o mostrado no Apêndice B. A saída gerada pelo experimento é armazenada em um arquivo JSON contendo uma lista de objetos que armazenam os resultados da execução, sendo eles: tempo de execução, a melhor solução gerada e o valor de sua função objetivo. Cada

objeto na saída representa uma execução do ACO. Para poder manipular os arquivos *json* foi necessário o uso de uma framework chamada *json-simple* que facilita a leitura e gravação em arquivos *json* [11].

Foi necessário implementar duas classes principais para poder utilizar a biblioteca *Isula*: *Environment* e *Ant*. A implementação do programa foi feita com base no exemplos disponíveis de implementação do TSP com a *Isula*.

A classe *Environment* é onde está contida todas as informações do problema como detalhes dos projetos, recursos disponíveis, horizonte de planejamento, matriz de feromônio e os valores de $J$ e $\eta$. Ela também é responsável por verificar a validade de soluções parciais para uma determinada restrição. As principais funções dessa classe podem ser encontradas no Apêndice C.2.

Os mesmos valores de $\eta$ são utilizados diversas vezes durante a execução do algoritmo, então para não ter a necessidade de recalcular os seus valores, eles são pre-calculados durante a chamada do construtor de um objeto da classe *Environment* e são armazenados em uma matriz que fica disponível em um atributo da classe *Environment*.

A classe *Ant* gera e armazena as soluções e para isso usa o *Environment* como parâmetro na maioria de suas funções para poder acessar as informações do problema e verificar as restrições. A principal função da classe é determinar quais são os componentes válidos para serem adicionados a solução e seus valores de heurística e feromônio entre as ligações. Ela também tem funções para determinar o valor da função objetivo de uma solução ou se ela está completa ou não. As principais funções dessa classe podem ser encontradas no Apêndice C.3.

Além dessas duas classes principais é necessário fazer algumas implementações de configurações da *framework*, como definir os valores de parâmetros que serão utilizados, inicializar uma colônia de formiga usando a classe *Ant* e *Environment*, definir o método de evaporação de feromônios, e usar a classe *AcoProblemSolver* para gerar as soluções. Um fragmento do código onde o ACO é executado pode ser encontrado no Apêndice C.1

Os experimentos foram realizados em um computador de mesa. Devido ao tempo necessário para fazer os experimentos, a quantidade de parâmetros e as limitações técnicas do computador não foi possível realizar muitos testes com os diferentes valores de parâmetros.

Na Tabela 3.2 são apresentadas as configurações da máquina utilizada para executar os experimentos.

| Sistema Operacional | Windows 10 64Bits |
|---|---|
| Memória Principal (RAM) | 8GB |
| CPU | 3,6GHz |
| Processador | AMD Ryzen™ 5 2400G |

Tabela 3.2 – Configurações da máquina utilizada nos experimentos.

## 3.4 Análise de Resultados

Na Tabela 3.3 temos os resultados obtidos pelos experimentos feitos com o ACO onde para cada entrada foram realizadas 10 execuções. A coluna de *Número de Projetos* identifica

a categoria da entrada pelo seu número de projetos. A coluna *Min FO* indica o menor valor encontrado da função objetivo, já a coluna *Max FO* indica o maior valor. A *Média FO* indica a média dos valores das função objetivo obtida com as 10 execuções. A *Razão LFO* é dada pela razão entre a *Média FO* e o limitante inferior da instância de entrada, ou seja, o valor da função objetivo de uma solução onde todos os projetos são agendados no mês 1. A *Média TE(s)* representa o tempo médio das execuções em segundos.

| Número de projetos | Min FO | Max FO | Média FO | Razão LFO | Média TE(s) |
|---|---|---|---|---|---|
| 30 | 158168,0 | 161674,0 | 160078,4 | 1,274013 | 831,5 |
| 50 | 288217,0 | 292387,0 | 290737,0 | 1,307712 | 1587,7 |
| 100 | 551026,0 | 562390,0 | 558050,0 | 1,334971 | 4540,8 |
| 150 | 815352,0 | 826220,0 | 822626,7 | 1,363425 | 10375,6 |
| 200 | 1117018,0 | 1123998,0 | 1120978,3 | 1,379751 | 18270,2 |

Tabela 3.3 – Resultados obtidos através dos experimentos ACO.

A *Razão LFO* indica quanto o valor da média da função objetivo está a mais do limitante inferior, ou seja, quanto menor o seu valor, melhores os resultados. Dada a tabela de resultados podemos observar que a melhor *Razão LFO* está na entrada de 30 projetos. Conforme o número de projetos é aumentado a *Razão LFO* aumenta também, o que implica que a qualidade das soluções acaba caindo conforme o tamanho da entrada aumenta, tomando como parâmetros de comparação apenas o limitante inferior. Observe que nós não sabemos qual é o valor da função objetivo para uma solução ótima destas instâncias de entrada, o que permitiria estimar melhor a qualidade das soluções geradas pela heurística. Outra observação que podemos identificar da tabela é que a diferença entre os valores de *Razão LFO*, entre as entradas com 150 e 200 projetos a diferença é ligeiramente menor.

Em relação ao tempo de execução, conforme o tamanho da entrada aumenta, a média de tempo também é aumentada consideravelmente acima da proporção de aumento de projetos. Esse aumento se dá devido à complexidade de execução do ACO que pode ser estimada em: $O(numIterations\ n^2 m)$ onde $numIterations$ é o número máximo de iterações, $n$ é o número de componentes e $m$ o número de formigas [17].

# 4 Conclusão

Neste trabalho foi apresentada uma versão simplificada de um problema PPS que foi baseado em um problema real de uma companhia geradora de energia elétrica, modelado para ser resolvido pelo ACO.

A análise dos resultados foi feita usando o valor do limitante inferior, o que não é uma maneira muito consistente de descobrir a qualidade das soluções encontradas pelo algoritmo. Uma alternativa seria ter o valor ótimo de alguma instância ou ter outro método implementado para poder encontrar a solução para poder ter uma perspectiva melhor da qualidade da solução, o que infelizmente não foi possível para este trabalho.

O maior desafio encontrado neste trabalho foi modelar o PPS para o ACO. Existem outros trabalhos com uma versão diferente do PPS resolvido por meio de ACO, mas infelizmente o nosso problema é diferente o suficiente para ser necessário criar a nossa própia modelagem. Além da diferença os trabalhos semelhantes não possuem muitos detalhes de como o PPS foi portado para o ACO. Então para modelar o problema foi usado de base como o ACO é implementado para resolver o TSP.

O ACO utilizado para resolver o problema foi a versão mais simples onde apenas a atualização de feromônios com atraso é implementada. Versões diferentes do algoritmo utilizando atualização *online* de feromônio passo a passo, ou até mesmo com a implementação de alguma *daemon action* poderiam gerar resultados diferentes.

A razão entre valor da média dos valores da função objetivo da solução e o limitante inferior é aumentada conforme o tamanho da instância de entrada é aumentada. Além disso, podemos identificar que o número de projetos na instância de entrada é o fator que causa maior impacto no tempo de execução médio do algoritmo.

Para trabalhos futuros seria interessante executar um programa para poder definir os valores ótimos de algumas instâncias para poder ter uma noção melhor da qualidade das soluções geradas pelo ACO. Poderia-se também tentar usar variações do ACO, ou até mesmo tentar usar outra maneira de modelar o PPS para ser executado pelo ACO.

# Referências


[1] C. Blum and M. Sampels. An ant colony optimization algorithm for shop scheduling problems. *Journal of Mathematical Modelling and Algorithms*, 3(3):285–308, 2004.

[2] A. F. Carazo. Multi-criteria project portfolio selection. In *Handbook on Project Management and Scheduling Vol. 2*, pages 709–728. Springer, 2015.

[3] K. Doerner, W. J. Gutjahr, R. F. Hartl, C. Strauss, and C. Stummer. Ant colony optimization in multiobjective portfolio selection. In *Proc. 4th Metaheuristics International Conference*, pages 243–248, 2001.

[4] K. F. Doerner, W. J. Gutjahr, R. F. Hartl, C. Strauss, and C. Stummer. Pareto ant colony optimization with ILP preprocessing in multiobjective project portfolio selection. *European Journal of Operational Research*, 171(3):830–841, 2006.

[5] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[6] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. *Theoretical computer science*, 344(2-3):243–278, 2005.

[7] M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477. IEEE, 1999.

[8] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial life*, 5(2):137–172, 1999.

[9] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[10] M. Dorigo and K. Socha. An introduction to ant colony optimization. In *Handbook of Approximation Algorithms and Metaheuristics, Second Edition*, pages 395–408. Chapman and Hall/CRC, 2018.

[11] Y. Fang. json-simple. `https://github.com/fangyidong/json-simple`. Último acesso em 01/09/2021.

[12] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, 1979.

[13] C. Gavidia-Calderon. Isula: A Framework for Ant Colony Algorithms. `https://github.com/cptanalatriste/isula`. Último acesso em 13/10/2021.

[14] C. Gavidia-Calderon. Javadoc of Isula. `http://cptanalatriste.github.io/isula/doc/`. Último acesso em 14/10/2021.

[15] C. Gavidia-Calderon and C. B. Castañon. Isula: A java framework for ant colony algorithms. *SoftwareX*, 11:100–400, 2020.

[16] Guofei. Python module scikit-opt. https://github.com/guofei9987/scikit-opt. Último acesso em 14/10/2021.

[17] M. Li. Efficiency improvement of ant colony optimization in solving the moderate ltsp. *Journal of Systems Engineering and Electronics*, 26(6):1300–1308, 2015.

[18] Y. Marín. Implementing Ant Colony Optimization (ACO) algorithm for a given Symmetric traveling salesman problem (TSP). https://github.com/yammadev/aco-tsp. Último acesso em 14/10/2021.

[19] C. Mira, P. R. Viadanna, M. A. Souza, A. Moura, J. Meidanis, G. A. C. Lima, and R. P. Bossolan. Project scheduling optimization in electrical power utilities. *Pesquisa Operacional*, 35(2):285–310, 2015.

[20] V. Mohagheghi, S. M. Mousavi, J. Antuchevičienė, and M. Mojtahedi. Project portfolio selection problems: a review of models, uncertainty approaches, solution techniques, and case studies. *Technological and Economic Development of Economy*, 25(6):1380–1412, 2019.

[21] R. Montemanni, L. M. Gambardella, A. E. Rizzoli, and A. V. Donati. Ant colony system for a dynamic vehicle routing problem. *Journal of combinatorial optimization*, 10(4):327–343, 2005.

[22] B. Naderi. The project portfolio selection and scheduling problem: mathematical model and algorithms. 2013.

[23] R. S. Parpinelli, H. S. Lopes, and A. A. Freitas. Data mining with an ant colony optimization algorithm. *IEEE transactions on evolutionary computation*, 6(4):321–332, 2002.

[24] D. L. Pereira. Formigueiro - A python Framework for Simple Ant Colony Optimization Algorithms. https://github.com/dilsonpereira/Formigueiro. Último acesso em 13/10/2021.

[25] T. Stützle and M. Dorigo. ACO algorithms for the quadratic assignment problem. *New ideas in optimization*, (C50):33, 1999.

[26] E.-G. Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[27] TESTDATA. MP-TESTDATA - The TSPLIB Symmetric Traveling Salesman Problem Instances. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/. Último acesso em 14/10/2021.

# Apêndices

# APÊNDICE A – Experimentos para determinar os parâmetros

Antes de realizar os experimentos foram testados valores variados para os parâmetros do ACO. O valor padrão dos parâmetros foram decididos como: $\rho = 0.1$, $\tau_0 = 1$, $\alpha = 1$, $\beta$, $NFormigas = 1000$. Devido a demora de execução dos, experimentos para definir o número de formigas foram feitos com uma instância de 30 projetos, enquanto os demais experimentos foram feitos com uma instancia de 300 projetos. Para decidir os melhores valores dos parâmetros foram realizadas execuções variando um parâmetro alvo, enquanto os demais são mantidos fixos, onde os resultados se encontram nas tabelas a seguir:

Tabela A.1 – Variação no parâmetro de evaporação.

| 300 projetos | |
|---|---|
| $\rho$ | Resultado 100 iterações |
| 0.01 | 1791188 |
| 0.2 | 1790681 |
| 0.3 | 1798527 |
| . 0.4 | 1794919 |
| 0.5 | 1791150 |

Tabela A.2 – Variação no parâmetro de feromônio inicial.

| 300 projetos | |
|---|---|
| $\tau_0$ | Resultado 100 iterações |
| 0.5 | 1792099 |
| 1.5 | 1788355 |
| 2 | 1792745 |
| 2.5 | 1783583 |
| 3 | 1782110 |

Tabela A.3 – Variação no parâmetro do peso heurístico.

| 300 projetos | |
|---|---|
| $\beta$ | Resultado 100 iterações |
| 1.5 | 1777907 |
| 2 | 1791476 |
| 2.5 | 1787173 |
| 3 | 1793796 |
| 3.5 | 1785912 |
| 4 | 1789622 |
| 5 | 1787383 |

Tabela A.4 – Variação no parâmetro do peso feromônio.

| 300 projetos | |
|---|---|
| $\alpha$ | Resultado 100 iterações |
| 1.5 | 1784417 |
| 2 | 1790309 |
| 2.5 | 1787102 |
| 3 | 1791640 |
| 3.5 | 1790081 |
| 4 | 1772726 |
| 5 | 1790316 |

Tabela A.5 – Variação no número de formigas.

| 30 projetos | |
|---|---|
| N Formigas | Resultado 100 iterações |
| 100 | 138277 |
| 1000 | 136032 |
| 5000 | 134295 |
| 10000 | 134383 |
| 20000 | 135597 |

# APÊNDICE B – Exemplo de instância


```
{
  "Limits":[
    {
      "Classification":"CAPEX",
      "Values":[31300, 31300,31300,31300,31300]
    },
    {
      "Classification":"OPEX",
      "Values":[18529, 18529,18529,18529,18529]
    }
  ],
  "Projects":[
    {
      "Costs":[1448, 2373,1841,1515,2149,1661,1383,1360,1625,1093,1057,1610,2145,1912,2455,1079,2377,1652],
      "Classification":"OPEX",
      "Halting Duration":-1,
      "Risk":23,
      "Duration":18,
      "Id":1,
      "Maintenance":"N",
      "Halting Start Month":-1
    },
    {
      "Costs":[ 1173,2418,1350,2447,1839,2009,1774,2166,2090,1551,1325,1262,1038,1204,1577],
      "Classification":"CAPEX",
      "Halting Duration":-1,
      "Risk":81,
      "Duration":15,
      "Id":2,
      "Maintenance":"N",
      "Halting Start Month":-1
    },
    {
      "Costs":[1962, 1793,1797,2463,1973,1179,1617,1565,1863,1675,1946,2292,2081],
      "Classification":"CAPEX",
      "Halting Duration":-1,
      "Risk":28,
      "Duration":13,
```

```
      "Id":3,
      "Maintenance":"N",
      "Halting Start Month":-1
    },
    {
      "Costs":[2328, 2328,1972,1915,1948,1372,2405,1077,1843,1394,1995,1324,1028,2189,1563,1566,2445,2184,1162,1762,1306],
      "Classification":"CAPEX",
      "Halting Duration":-1,
      "Risk":117,
      "Duration":21,
      "Id":4,
      "Maintenance":"N",
      "Halting Start Month":-1
    },
    {
      "Costs":[1655, 2206,2208,2144,2098,1531,2266,2071,1723,1406,1724,1348,1329,2435,2394,2359],
      "Classification":"CAPEX",
      "Halting Duration":-1,
      "Risk":67,
      "Duration":16,
      "Id":5,
      "Maintenance":"N",
      "Halting Start Month":-1
    },
    {
      "Costs":[1234, 2179,1170,1016,1000,1698,1151,1254,1009,1797,1695,2015,1233,1608,2077,1744,2302,2426],
      "Classification":"CAPEX",
      "Halting Duration":-1,
      "Risk":32,
      "Duration":18,
      "Id":6,
      "Maintenance":"N",
      "Halting Start Month":-1
    },
    {
      "Costs":[1792, 1646,1111,2011,1423,2480],
      "Classification":"CAPEX",
      "Halting Duration":-1,
      "Risk":34,
      "Duration":6,
      "Id":7,
      "Maintenance":"N",
      "Halting Start Month":-1
```

```
    },
    {
      "Costs":[1601, 2208,1409,1328,2054,1143,2028,2071,1525,2390,214
        4,1524,1062,1189,1836,1314,1950],
      "Classification":"OPEX",
      "Halting Duration":8,
      "Risk":97,
      "Duration":17,
      "Id":8,
      "Maintenance":"L",
      "Halting Start Month":4
    },
    {
      "Costs":[1439, 1867,1764,1632,1387,1941,1999,1162,1256,1896,115
        9,1601,2271,1648],
      "Classification":"OPEX",
      "Halting Duration":9,
      "Risk":34,
      "Duration":14,
      "Id":9,
      "Maintenance":"L",
      "Halting Start Month":4
    },
    {
      "Costs":[1204, 1362,2390,2113,1030,2016],
      "Classification":"OPEX",
      "Halting Duration":2,
      "Risk":25,
      "Duration":6,
      "Id":10,
      "Maintenance":"L",
      "Halting Start Month":3
    }
  ],
  "Restriction":[
    [0,0,1],
    [0,0,1],
    [1,1,0]
  ]
}
```

# C Código fonte

## C.1 Código fonte de execução do algoritmo ACO

```
ConfiguracaoAcoPPS config = new ConfiguracaoAcoPPS();
AntColony<Integer, AmbientePps> colonia = getAntColony(config);

AcoProblemSolver<Integer, AmbientePps> solver = new AcoProblemSolver<>();
solver.initialize(prob, colonia, config);
solver.addDaemonActions(new StartPheromoneMatrix<>(),
new PerformEvaporation<>());

solver.addDaemonActions(getPheromoneUpdatePolicy());

solver.getAntColony().addAntPolicies(new RandomNodeSelection<>());
solver.solveProblem();
```

## C.2 Classe Ant

A função *isSolutionReady* retorna verdadeiro caso a solução esteja completa e a função *getSolutionCost* retorna o valor da função objetivo para a função gerada epla formiga.

```
public boolean isSolutionReady(AmbientePps ambientePps){
      return getCurrentIndex()== (ambientePps.getNumeroDeProjetos() * 2 );
   }

   //Retorna o custo da solucao
   @Override
   public double getSolutionCost(AmbientePps ambientePps, List<Integer>
       list) {
      //Valores necessarioa pra calcular J_psi
      double riscoTotal = (double) 3 * this.horizonteDePlanejamento *
          ambientePps.getRiscoTotal();
      double somatoria = 0;//somatoria do (J_psi,psi+1)^-1 para os projetos
          agendados

      //calcula a somatoria
      for(int i = 0; i < list.size(); i+=2){
         //J_(i, i+1) ^ -1
         somatoria +=
            Math.pow(ambientePps.getCustoProjetoParaMes(list.get(i) -
            this.horizonteDePlanejamento,(list.get(i+1))+1), -1);
      }
```

```
      return (riscoTotal - somatoria);

   }
```

A função *getNeighbourhood* faz parte da classe *Ant* e ela retorna os componentes disponíveis para serem adicionados a solução.

```
public List<Integer> getNeighbourhood(AmbientePps ambientePps) {
      List<Integer> vizinhanca = new ArrayList<Integer>(), solucao =
         getSolution();
      int componenteAtual = getCurrentIndex() - 1;//index componente atual
     //Escolhe uma vizinhanca de meses
      if(componenteEhProjeto(componenteAtual)){
         int idProjeto =
           MetodosGlobais.converteComponenteParaIDPPS(
             solucao.get(componenteAtual),
               this.horizonteDePlanejamento);
         int mes;
         int[] recursoDisponivel =
             ambientePps.calculaOrcamentoDisponivel(solucao,
             ambientePps.getTipoCusto(idProjeto));

         for(int componenteMes = 0; componenteMes <
             this.horizonteDePlanejamento + this.horizonteDeExecucao;
             componenteMes++){
            mes =
                MetodosGlobais.converteComponenteParaMesPPS(componenteMes);
            if(ambientePps.verificaRestricaoDeCusto(idProjeto,
                recursoDisponivel, mes) &&
                   ambientePps.verificaRestricaoManutencao(idProjeto,
                       solucao, mes)){
               vizinhanca.add(componenteMes);
            }
         }
      }else{//Escolhe uma vizinhanca de projeto
         for (int idProjeto : ambientePps.getListaProjetos().keySet()) {
            if(!(isNodeVisited(MetodosGlobais.converteIDPPSParaComponente(
              idProjeto, this.horizonteDePlanejamento)))){
               vizinhanca.add(MetodosGlobais.converteIDPPSParaComponente(
               idProjeto, this.horizonteDePlanejamento));
            }
         }
      }

      return vizinhanca;
   }
```

As próximas duas funções são responsáveis por interagir com a matriz de feromônio que está no *Environment*. A função *getPheromoneTrailValue* retorna o valor de feromônio na ligação entre

dois componentes, já a função *setPheromoneTrailValue* altera o valor de feromônio entre dois componentes para o valor passado na função.

```
@Override
   public Double getPheromoneTrailValue(Integer componenteDaSolucao,
      Integer posicaoNaSolucao, AmbientePps ambientePps) {
      if(posicaoNaSolucao == 0){
         double[][] matriz = ambientePps.getPheromoneMatrix();
         return matriz[componenteDaSolucao][componenteDaSolucao];
      }else{
         int componenteAnterior = this.getSolution().get(posicaoNaSolucao -
            1);
         double[][] matriz = ambientePps.getPheromoneMatrix();
         return matriz[componenteAnterior][componenteDaSolucao];
      }
   }

   @Override
   public void setPheromoneTrailValue(Integer componenteDaSolucao, Integer
      posicaoNaSolucao, AmbientePps ambientePps, Double novoPheromonio) {
      if(posicaoNaSolucao == 0) {
         double[][] matriz = ambientePps.getPheromoneMatrix();
         matriz[componenteDaSolucao][componenteDaSolucao] = novoPheromonio;
      }else{
         int componenteAnterior = this.getSolution().get(posicaoNaSolucao -
            1);
         double[][] matriz = ambientePps.getPheromoneMatrix();
         matriz[componenteAnterior][componenteDaSolucao] = novoPheromonio;
      }
   }
```

A função *getHeuristicValue* retorna o valor heurístico de se adicionar determinado componente a solução.

```
@Override
   public Double getHeuristicValue(Integer componenteDaSolucao, Integer
      posicaoNaSolucao, AmbientePps ambientePps) {
      double valorHeuristico = 0;
      int idProjeto;
      List<Integer> solucao = this.getSolution();

      if(componenteEhProjeto(posicaoNaSolucao)){ //calcula a heuristica de
         adcionar um mes
         idProjeto =
            MetodosGlobais.converteComponenteParaIDPPS(componenteDaSolucao,
               this.horizonteDePlanejamento);
         valorHeuristico = 1 -
            ambientePps.getCustoMesParaProjeto(idProjeto);
      }else{// calcula a heuristica de acionar um projeto
         int mes =
```

```
            MetodosGlobais.converteComponenteParaMesPPS(componenteDaSolucao);
         int componenteAnterior = posicaoNaSolucao - 1;
         idProjeto =
           MetodosGlobais.converteComponenteParaIDPPS(
             solucao.get(componenteAnterior),
               this.horizonteDePlanejamento);
         valorHeuristico =
            ambientePps.getHeuristica(solucao.get(componenteAnterior),
            componenteDaSolucao);
      }

      return valorHeuristico;
   }
```

## C.3 Classe Environment

A função *createPheromoneMatrix* cria a matriz de feromônio com as informações do problema.

```
   @Override
   protected double[][] createPheromoneMatrix() {
      if (this.listaProjetos != null) {
         int numeroDeComponentes = (this.horizontePlanejamentoMeses +
            this.horizonteDeExecucao) + this.listaProjetos.size();
         return new double[numeroDeComponentes][numeroDeComponentes];
      }
      else{
         return new double[0][];
      }
   }
```

As funções *getCustoProjetoParaMes* e *getCustoMesParaProjeto* tem como retorno o custo $J$ de adicionar um mês ou projeto respectivamente.

```
public double getCustoProjetoParaMes(int projeto, int mes){
      double retorno; // valor a ser retornado

      //informacoes do projeto
      int risco = this.listaProjetos.get(projeto).getRisco();
      int duracao = this.listaProjetos.get(projeto).getDuracao();
      //Horizonte de planejamento em meses
      int mesesTotais = this.horizontePlanejamentoMeses;

      //Formula da funcao J para projeto -> mes
      retorno = (double) (risco*(2*mesesTotais + 1 - (mes + duracao - 1)));
      retorno = 1/retorno;
```

```
      return retorno;
   }

   public double getCustoMesParaProjeto(int projeto){
      double retorno;// valor a ser retornado

      //informacoes do projeto
      int risco = this.listaProjetos.get(projeto).getRisco();
      int duracao = this.listaProjetos.get(projeto).getDuracao();

      //Formula da funcao J de mes para projeto
      retorno =(double) (risco * duracao);
      retorno = 1/retorno;

      return retorno;
   }
```

A função *verificaRestricaoDeCusto* verifica se um projeto pode ser agendado em determinado mês respeitando as restrições de custo. A função *verificaRestricaoManutencao* verifica se um projeto pode ser agendado em determinado mês respeitando a restrição de manutenção.

```
public boolean verificaRestricaoDeCusto(int id, int[] recursoDisponivel,
    int mes){
      int[] custoAnualProj =
         this.listaProjetos.get(id).getOrcamentoAnualDoProjeto(mes);

      if(mes > this.horizontePlanejamentoMeses) {
         return true;
      }
      for(int anoRecursoDisponivel = mesesParaAno(mes), anoCustoAnualProj =
         0;
      anoRecursoDisponivel < this.horizontePlanejamentoAnos &&
         anoCustoAnualProj < custoAnualProj.length;
      anoRecursoDisponivel++, anoCustoAnualProj++){
         if((recursoDisponivel[anoRecursoDisponivel] -
            custoAnualProj[anoCustoAnualProj]) < 0){
            return false;
         }
      }
      return true;

   }

   boolean verificaRestricaoManutencao(int idProj, List<Integer> solucao,
      int mes){
      boolean podeAgendar = true;
      if(mes > this.horizontePlanejamentoMeses){
         return podeAgendar;
      }
```

```
      if(listaProjetos.get(idProj).verificaSeProjetoEhDeManutencao()) {
         MetodosGlobais mg = new MetodosGlobais();
         int idProj2, mes2;
         int posicaoListaManutencao =
             listaIdProjetosManutencao.indexOf(idProj);

         for (int i = 0; i < restricao[posicaoListaManutencao].length; i++){
            if (restricao[posicaoListaManutencao][i] == 1 &&
                  solucao.contains(mg.converteIDPPSParaComponente(
                    this.listaIdProjetosManutencao.get(i),
                    this.horizontePlanejamentoMeses))){
               idProj2 = this.listaIdProjetosManutencao.get(i);
               mes2 = mg.converteComponenteParaMesPPS(
                solucao.get(solucao.indexOf(
                  mg.converteIDPPSParaComponente(idProj2,
                     this.horizontePlanejamentoMeses)) + 1));
               if(paradaCoincidem(idProj, mes, idProj2, mes2)){
                  podeAgendar = false;
                  break;
               }
            }
         }
      }

      return podeAgendar;
   }
```