



Universidade Federal do ABC
Centro de Matemática, Computação & Cognição
Bacharelado em Ciência da Computação

**O Escalonador dos Papéis: Abordagens
heurísticas com colônias de formigas para o
escalonamento flexível de trabalhos no contexto
da indústria gráfica**

Matheus Antonio Ribeiro da Silva

Santo André - SP, Junho de 2025

Matheus Antonio Ribeiro da Silva

O Escalonador dos Papéis: Abordagens heurísticas com colônias de formigas para o escalonamento flexível de trabalhos no contexto da indústria gráfica

Projeto de Graduação em Computação apresentado como parte dos requisitos necessários para a obtenção do Título de Bacharel em Ciência da Computação.

Universidade Federal do ABC – UFABC
Centro de Matemática, Computação & Cognição
Bacharelado em Ciência da Computação

Orientador: Prof. Dr. Mario Leston Rey

Santo André - SP
Junho de 2025

*Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.*

Agradecimentos

Encarando o *prompt* do editor de texto, prestes a digitar o que vem a seguir, me ocorreu o quão privilegiado eu sou por ter tantas pessoas a agradecer. Tamanho é o privilégio a ponto de eu temer o peso na consciência que sentiria se, por um lapso, me esquecesse de alguém.

Sou grato, em princípio, a todos os meus familiares. Em especial, meus pais, Edson e Claudiana, cujos esforços para proporcionar as condições que me permitiram chegar até aqui foram incalculáveis. Amo vocês.

Ao meu orientador, Mario, cuja visão e experiência guiaram este trabalho que nos é motivo de muito orgulho. A segurança que me passou nos momentos de incerteza e a liberdade e incentivo para imprimir o meu estilo foram de suma importância no desenvolvimento deste texto. Jamais alcançaria o meu pleno potencial sem a sua orientação.

Aos meus amigos, pela rede de apoio que me possibilitaram os momentos de respiro necessários durante o desgastante processo de escrita de um trabalho como esse. A importância de vocês em minha vida é muito maior do que são capazes de imaginar. Prometo que vou estar mais presente a partir de agora.

A todos os professores com quem tive a oportunidade de aprender. Sou quem sou por causa de vocês. Uma homenagem especial aos meus mentores, padrinhos e queridos amigos Edvaldo Santos e Caroline Milone, que me querem bem como a um filho. Também preciso agradecer a dois Rodrigos, por manifestarem em mim o interesse genuíno pela Matemática. O primeiro, Rodrigo Ramos, desconstruiu tudo que achei que sabia. Ainda posso ouvir sua voz martelando em meus ouvidos o mantra de que “*toda função é uma relação, mas nem toda relação é uma função*”. Ao professor Rodrigo de Souza, agradeço, dentre outras coisas, por ter me feito acreditar que a matemática universitária seria simples quando me apresentou à minha primeira derivada. O tempo e as disciplinas de Cálculo me mostraram que eu estava errado, mas foi o bastante para eu ter certeza do caminho que queria seguir. Espero um dia, quem sabe, compartilhar da nobre missão de passar adiante o conhecimento que adquiri.

À Ameni Azzouz, Lucas Berterottiere e Janis Sebastian Neufeld que gentilmente me cederam seus artigos durante a fase de revisão bibliográfica deste trabalho, quando não consegui por nenhum outro meio. Gestos simples como esses contribuem ativamente na luta pela democratização do acesso ao conhecimento acadêmico-científico. Pelo papel nesta luta, dedico este trabalho a Alexandra Elbakyan e Aaron Swartz.

À Ecalc Software e a todos os meus queridos colegas de profissão pelo seu papel fundamental na minha trajetória como (pretenso) cientista da computação. Eu jamais teria condições de passar por essa etapa sem a confiança, autonomia e flexibilidade proporcionadas. Um agradecimento especial ao CTO Caio Rodrigues, pela estabilidade concedida; aos camaradas da minha *guilda* Vitor Suzarte e Viviane Almeida, pelo companheirismo; e ao CEO Valdir Filho (que em breve poderá se gabar de empregar um cientista), por compartilhar seu vasto conhecimento sobre o setor gráfico. Suas contribuições com este trabalho são indispensáveis e farei com que colham frutos do investimento em mim realizado.

Minha gratidão à Universidade Federal do ABC por proporcionar essa experiência que ampliou minha visão de mundo, bem como pelos recursos e estrutura oferecida. Foram sete anos dos quais não esquecerei. Um agradecimento especial ao Prof. Dr. Maycon Sambinelli e aos colegas do laboratório de Teoria da Computação, Otimização, Combinatória e Algoritmos (TOCA) que me concederam o ambiente perfeito para trabalhar na reta final deste trabalho. A comunidade acadêmica há de concordar que o senso de pertencimento pode ser um sentimento difícil de cultivar neste nosso modelo pedagógico e sou grato por ter conseguido.

Por fim, com o destaque que lhe é devido, agradeço e dedico este trabalho à pessoa que está tão ansiosa quanto eu pela sua conclusão. Ao amor da minha vida, Evelin, sou grato pelo carinho, cuidado e incentivo. O fardo é muito mais leve quando quem está nas trincheiras ao teu lado te escuta falar sobre otimização combinatória (mesmo que nenhuma palavra lhe faça sentido) com a mesma paixão com que escutaria, se você lhe recitasse um poema. Sei que sacrifiquei muito do nosso tempo juntos em prol deste trabalho e lamento por isso. Prometo que vou compensar.

A todos, meu sincero obrigado.

*"Formigas são tão parecidas com
os seres humanos que chega a ser constrangedor.
Elas cultivam fungos, criam pulgões como gado,
lançam exércitos à guerra,
usam sprays químicos para alarmar
e confundir inimigos,
capturam escravos,
praticam trabalho infantil,
trocaram informações incessantemente.
Elas fazem tudo, menos assistir televisão."*

(Lewis Thomas)

Resumo

Este trabalho possui dois principais objetos de estudo. O primeiro é o problema de escalonamento flexível orientado a trabalho (FJSSP), estudado sob a perspectiva da indústria gráfica. O segundo é o método metaheurístico de otimização por colônia de formigas (ACO), para o qual propomos um modelo formal alternativo capaz de abstrair cinco de suas variantes: AS, EAS, RBAS, MMAS e ACS. A partir da combinação desses objetos de estudo, foi desenvolvido, em C#, um conjunto de 33 algoritmos com diferentes variações de heurísticas construtivas, abordagens de execução e algoritmos ACO, com o objetivo de resolver o FJSSP minimizando o *makespan*. Os testes realizados com instâncias clássicas da literatura indicaram que alguns dos algoritmos desenvolvidos são capazes de gerar soluções competitivas em termos de tempo de conclusão. Além disso, foi proposto um novo conjunto de instâncias de grande porte para o FJSSP, visando avaliar o desempenho das abordagens no contexto da indústria gráfica. Os resultados sugerem que, sob certas condições, o uso do ACO é uma alternativa viável para aplicações práticas, contribuindo para a redução de atrasos e o aumento da eficiência produtiva.

Palavras-chaves: escalonamento de tarefas; metaheurísticas; otimização combinatória; indústria gráfica; problema de escalonamento flexível orientado a trabalhos (FJSSP); otimização por colônia de formigas (ACO); Sistema de Formigas (AS); Sistema de Formigas Elitista (EAS); Sistema de Formigas com Classificação (RBAS); Sistema de Formigas MAX-MIN (MMAS); Sistema de Colônia de Formigas (ACS).

Abstract

This study addresses two main components. The first is the *Flexible Job Shop Scheduling Problem* (FJSSP), analyzed in the context of the printing industry. The second is the *Ant Colony Optimization* (ACO) metaheuristic, for which we propose an alternative formal model capable of representing five of its variants: AS, EAS, RBAS, MMAS, and ACS. By combining these components, a set of 33 algorithms was developed in C#, incorporating different constructive heuristics, execution strategies, and ACO variants, aiming to solve the FJSSP while minimizing the *makespan*. Experimental results on classical benchmark instances demonstrate that some of the proposed algorithms can produce competitive solutions in terms of completion time. Furthermore, a new set of large-scale FJSSP instances was introduced to assess the algorithms' performance in a realistic industrial scenario. The findings suggest that, under specific conditions, ACO provides a viable approach for practical applications, contributing to delay reduction and improved production efficiency.

Keywords: task scheduling; metaheuristics; combinatorial optimization; printing industry; Flexible Job Shop Scheduling Problem (FJSSP); Ant Colony Optimization (ACO); Ant System (AS); Elitist Ant System (EAS); Rank-Based Ant System (RBAS); MAX-MIN Ant System (MMAS); Ant Colony System (ACS).

Listas de ilustrações

Figura 1 – Representação gráfica de um digrafo.	12
Figura 2 – Representação de um digrafo com arcos paralelos e antiparalelos.	13
Figura 3 – Representação de um digrafo G (esq.) e de seu subdigrafo $G - \{b, d\}$ (dir.).	14
Figura 4 – Um digrafo com um caminho simples destacado em vermelho.	15
Figura 5 – Digrafo representando tarefas e suas relações de precedência.	16
Figura 6 – Digrafo acíclico ponderado pelo tempo de processamento das tarefas.	18
Figura 7 – Um digrafo acíclico ponderado com um caminho crítico até g destacado em vermelho e um caminho crítico até f em azul.	19
Figura 8 – Digrafo acíclico ponderado (esq.) e uma ramificação geradora w -ótima (dir.).	20
Figura 9 – Diagrama Gantt de um escalonamento de 7 tarefas em 5 máquinas.	23
Figura 10 – Digrafo acíclico representando as restrições de precedência.	27
Figura 11 – Diagrama Gantt de um escalonamento ótimo com restrições de precedência.	28
Figura 12 – Diagrama Gantt de uma solução para o JSSP.	37
Figura 13 – Digrafo disjuntivo para um problema de escalonamento orientado a trabalho.	42
Figura 14 – Digrafo conjuntivo induzido por uma seleção completa e consistente.	43
Figura 15 – Diagrama Gantt de uma solução para o FJSSP.	50
Figura 16 – Digrafo disjuntivo flexível para a instância enunciada no Exemplo 4.5.	51
Figura 17 – Formigas distribuídas por todo o caminho (esq.), tendem a convergir para o caminho mais curto (dir.) devido a uma maior concentração de feromônio	56
Figura 18 – Uma formiga mantendo uma solução $\pi = \langle i \rangle$ escolhendo um movimento viável sobre a vizinhança $N(\pi) = \{g, j, k\}$.	65
Figura 19 – Arcabouço de um algoritmo de otimização por colônia de formigas.	80
Figura 20 – Relatório do perfilador de desempenho indicando o gargalo de execução em ACSV0-p	90
Figura 21 – Diagrama de classes do namespace Scheduling.Solver	94
Figura 22 – Relatório do perfilador de desempenho que sugere a eficiência da nova implementação	113
Figura 23 – Gerenciador de testes da IDE Visual Studio	115
Figura 24 – Tempo médio de CPU do conjunto V0 para as instâncias de Fattahi et al. (2007)	127

Figura 25 – Tempo médio de CPU do conjunto V0 para as instâncias de Brandimarte (1993)	127
Figura 26 – <i>Makespan</i> médio do conjunto V0 para as instâncias de Fattahi et al. (2007)	128
Figura 27 – <i>Makespan</i> médio do conjunto V0 para as instâncias de Brandimarte (1993)	128
Figura 28 – Tempo médio de CPU do conjunto V1 para as instâncias de Fattahi et al. (2007)	129
Figura 29 – Tempo médio de CPU do conjunto V1 para as instâncias de Brandimarte (1993)	129
Figura 30 – <i>Makespan</i> médio do conjunto V1 para as instâncias de Fattahi et al. (2007)	130
Figura 31 – <i>Makespan</i> médio do conjunto V1 para as instâncias de Brandimarte (1993)	130
Figura 32 – Tempo médio de CPU do conjunto V2 para as instâncias de Fattahi et al. (2007)	132
Figura 33 – Tempo médio de CPU do conjunto V2 para as instâncias de Brandimarte (1993)	132
Figura 34 – Tempo médio de CPU do conjunto V2 para as instâncias de Dauzère-Pérès e Paulli (1997)	133
Figura 35 – <i>Makespan</i> médio do conjunto V2 para as instâncias de Fattahi et al. (2007)	134
Figura 36 – <i>Makespan</i> médio do conjunto V2 para as instâncias de Brandimarte (1993)	134
Figura 37 – <i>Makespan</i> médio do conjunto V2 para as instâncias Dauzère-Pérès e Paulli (1997)	135
Figura 38 – Tempo médio de CPU do conjunto TOPV2 para as instâncias sdata de Hurink et al. (1994)	136
Figura 39 – <i>Makespan</i> médio do conjunto TOPV2 para as instâncias sdata de Hurink et al. (1994)	137
Figura 40 – Tempo médio de CPU do conjunto TOPV2 para as instâncias vdata de Hurink et al. (1994)	137
Figura 41 – <i>Makespan</i> médio do conjunto TOPV2 para as instâncias vdata de Hurink et al. (1994)	138
Figura 42 – Tempo médio de CPU do conjunto V3 para as instâncias de Fattahi et al. (2007)	139
Figura 43 – Tempo médio de CPU do conjunto V3 para as instâncias de Brandimarte (1993)	139
Figura 44 – Tempo médio de CPU do conjunto V3 para as instâncias de Dauzère-Pérès e Paulli (1997)	140
Figura 45 – <i>Makespan</i> médio do conjunto V3 para as instâncias de Fattahi et al. (2007)	140

Figura 46 – <i>Makespan</i> médio do conjunto V3 para as instâncias de de Brandimarte (1993)	141
Figura 47 – <i>Makespan</i> médio de V3 para as instâncias de Dauzère-Pérès e Paulli (1997)	141
Figura 48 – Tempo médio de CPU do conjunto TOPV3 para as instâncias sdata de Hurink et al. (1994)	142
Figura 49 – <i>Makespan</i> médio do conjunto TOPV3 para as instâncias sdata de Hurink et al. (1994)	142
Figura 50 – Tempo médio de CPU do conjunto TOPV3 para as instâncias vdata de Hurink et al. (1994)	143
Figura 51 – <i>Makespan</i> médio do conjunto TOPV3 para as instâncias vdata de Hurink et al. (1994)	143
Figura 52 – <i>Gap</i> percentual médio por conjunto de algoritmos e conjunto de instâncias.	144
Figura 53 – Captura de tela do monitor de recursos do sistema operacional.	145
Figura 54 – Estimativas de densidade por <i>kernel</i> dos <i>gaps</i> médios para as instâncias de Fattahi et al. (2007)	146
Figura 55 – Estimativas de densidade por <i>kernel</i> dos <i>gaps</i> médios para as instâncias de Brandimarte (1993)	146
Figura 56 – Estimativas de densidade por <i>kernel</i> dos <i>gaps</i> médios para as instâncias de Dauzère-Pérès e Paulli (1997)	147
Figura 57 – Estimativas de densidade por <i>kernel</i> dos <i>gaps</i> médios para as instâncias sdata de Hurink et al. (1994)	147
Figura 58 – Estimativas de densidade por <i>kernel</i> dos <i>gaps</i> médios para as instâncias vdata de Hurink et al. (1994)	148
Figura 59 – Tempo médio de CPU de V2 para a instância RibeiroSuzarte1	150
Figura 60 – <i>Makespan</i> médio de V2 para a instância RibeiroSuzarte1	150
Figura 61 – Tempo médio de CPU de V2 para a instância RibeiroSuzarte2	151
Figura 62 – <i>Makespan</i> médio de V2 para a instância RibeiroSuzarte2	151
Figura 63 – Tempo médio de CPU de V2 para a instância RibeiroSuzarte3	152
Figura 64 – <i>Makespan</i> médio de V2 para a instância RibeiroSuzarte3	152
Figura 65 – Tempo médio de CPU de ACSV2 por número de iterações	153
Figura 66 – <i>Makespan</i> médio de CPU de ACSV2 por número de iterações	154

Sumário

1	INTRODUÇÃO	1
1.1	Justificativa	1
1.2	Objetivos	2
1.3	Estrutura da monografia	2
2	PRELIMINARES	4
2.1	Conjuntos	4
2.1.1	Subconjuntos e maximalidade	4
2.1.2	Operações sobre conjuntos	5
2.2	Relações e funções	6
2.2.1	Produto cartesiano	6
2.2.2	Função ou relação, eis a questão	7
2.3	Algoritmos	8
2.4	Sequências	9
2.4.1	Prefixos, subsequências, permutações e truco	11
2.5	Digrafos	12
2.5.1	Subdigrafo	13
2.5.2	Caminhos e ciclos	14
2.5.3	Digrafos acíclicos	16
2.5.4	Digrafos ponderados	17
2.5.5	Caminho críticos	18
3	PROBLEMAS DE ESCALONAMENTO	22
3.1	Escalonamento em máquinas idênticas	22
3.1.1	Algoritmo de Graham	24
3.2	Escalonamento com restrições de precedência em máquina única	26
3.2.1	Algoritmo de Lawler	29
3.2.2	Antimatróides	31
4	ESCALONAMENTO NA INDÚSTRIA GRÁFICA	33
4.1	Escalonamento orientado a trabalho	34
4.1.1	Métodos exatos	37
4.1.2	Heurísticas construtivas	39
4.1.3	Digrafo disjuntivo	40
4.2	Escalonamento flexível orientado a trabalho	46
4.2.1	Digrafo disjuntivo flexível	50

4.2.2	Construindo soluções viáveis	51
5	OTIMIZAÇÃO POR COLÔNIA DE FORMIGAS	55
5.1	Problemas de otimização e estruturas de vizinhança	56
5.2	Seguir o rastro, você deve!	62
5.3	Fui ao mercado com um problema de otimização...	65
5.4	Formigas de todas as colônias, uni-vos!	67
5.5	Algoritmos ACO e suas variantes	70
5.5.1	Sistema de formigas	70
5.5.2	Sistema de formigas elitista	73
5.5.3	Sistema de formigas com classificação	74
5.5.4	Sistema de formigas MAX-MIN	75
5.5.5	Sistema de colônia de formigas	76
5.6	Que o feromônio esteja com você	78
5.7	A escala e a formiga	83
6	IMPLEMENTAÇÃO	86
6.1	Representação do problema e instâncias	86
6.2	Com quantos digrafos disjuntivos se faz um aquecedor elétrico?	89
6.3	De volta ao básico	92
6.3.1	<i>Ant against the machine</i>	93
6.3.1.1	Implementação do sistema de formigas	103
6.3.1.2	Implementação do sistema de formigas elitista	105
6.3.1.3	Implementação do sistema de formigas com classificação	106
6.3.1.4	Implementação do sistema de formigas MAX-MIN	107
6.3.1.5	Implementação do sistema de colônia de formigas	108
6.4	Rejeite a linearidade. Abrace a generalização.	113
7	RESULTADOS E DISCUSSÕES	118
7.1	Metodologia	118
7.2	Instâncias	121
7.2.1	Instâncias de Fattahi et al. (2007)	122
7.2.2	Instâncias de Brandimarte (1993)	122
7.2.3	Instâncias de Dauzère-Pérès e Paulli (1997)	123
7.2.4	Instâncias de Hurink et al. (1994)	124
7.3	Experimentos computacionais	126
7.3.1	Testes preliminares	126
7.3.2	<i>That's why he's the GOAT</i>	131
7.3.3	Quem generaliza os generalizadores?	138
7.3.4	Síntese	143

7.4	Novo conjunto de instâncias	148
7.4.1	RibeiroSuzarte1	149
7.4.2	RibeiroSuzarte2	151
7.4.3	RibeiroSuzarte3	152
7.4.4	RibeiroSuzarte4	153
8	CONCLUSÕES	155
8.1	Contribuições	156
8.2	Limitações	157
8.3	Trabalhos Futuros	157
	REFERÊNCIAS	159
	Índice	163

1 Introdução

A **Pesquisa Operacional** (comumente referida como **PO**) versa sobre a condução e coordenação das operações de uma organização. Segundo [Hillier e Lieberman \(2014\)](#) o seu êxito em resolver problemas militares de logística e produção durante a Segunda Guerra Mundial despertou o interesse de setores da indústria para a aplicação de modelos matemáticos como suporte nas tomadas de decisão envolvendo a gestão eficiente de recursos.

Atualmente, as áreas de planejamento e controle de produção (**PCP**) e de gestão de projetos têm se beneficiado diretamente dos avanços em **PO**, uma vez que sua área de atuação envolve a alocação dos recursos disponíveis em função do tempo. [Pinedo \(2016\)](#) caracteriza essa atividade como um processo de **scheduling**

“O scheduling, como processo de tomada de decisão, desempenha um papel importante na maioria dos sistemas de manufatura e produção, bem como na maioria dos ambientes de processamento de informações. Também é importante em ambientes de transporte e distribuição e em outros tipos de indústrias de serviços.” ([PINEDO, 2016](#))

Problemas do gênero são conhecidos na literatura como problemas de **escalonamento** (ou ainda, de **agendamento**) e se propõem a determinar quais recursos devem processar cada trabalho e a sequência em que esse processamento deve acontecer.

A indústria gráfica é responsável por produzir materiais impressos e embalagens em larga escala, o que torna o planejamento da produção uma etapa extremamente complexa e vital para o cumprimento de contratos dentro dos prazos estabelecidos. Além disso, essas empresas têm passado por uma pressão cada vez maior pela redução do impacto ambiental da sua operação, de modo que é de grande interesse desse segmento encontrar meios para uma gestão mais eficiente dos recursos utilizados, como a energia e o combustível consumidos pelas máquinas e a água utilizada para a lavagem do maquinário entre um trabalho e outro.

1.1 Justificativa

Este trabalho se justifica pela relevância prática e acadêmica do problema abordado. Do ponto de vista prático, a implementação de um método eficiente de alocação de recursos pode representar uma redução significativa nos custos de produção, além de contribuir para a sustentabilidade ambiental ao minimizar o consumo de energia

e a produção de resíduos. Isso impulsionou o interesse de uma empresa que presta serviços para a indústria gráfica em otimizar o processo de *scheduling* das ordens de produção de seus clientes. A empresa, que já dispõe de uma solução *ad hoc* desenvolvida nesse contexto, busca uma alternativa otimizada que apresente um desempenho superior.

No âmbito acadêmico, este trabalho se insere como uma plataforma para a formalização de um modelo abstrato alternativo para descrição de uma técnica heurística específica aplicada à solução de problemas de otimização. Ao estruturar o modelo sob uma perspectiva teórica, o estudo visa contribuir para um melhor entendimento da técnica, além de potencialmente abrir caminho para aprimoramentos ou generalizações aplicáveis a diferentes classes de problemas.

Além disso, a abordagem proposta busca equilibrar rigor teórico e aplicabilidade prática. Dessa forma, este estudo não apenas visa contribuir para gerar impactos diretos no setor produtivo, como também se posiciona como uma alternativa na literatura disponível sobre técnicas de escalonamento e otimização heurística.

1.2 Objetivos

Este trabalho tem dois grandes objetivos. O primeiro é estudar e implementar as técnicas de escalonamento de trabalhos mais adequadas às particularidades da indústria gráfica. O segundo, não menos importante, é a formalização de um modelo abstrato para a descrição de algoritmos que resolvam problemas de otimização combinatória por meio de colônias de formigas.

Mais especificamente, combinando esses objetivos, busca-se desenvolver um algoritmo de otimização por colônia de formigas capaz de definir o escalonamento e a alocação de um conjunto de trabalhos de forma eficiente, buscando minimizar sua data de conclusão.

1.3 Estrutura da monografia

O presente trabalho está estruturado da seguinte forma: no Capítulo 2, são discutidos conceitos preliminares e notações utilizados no decorrer do trabalho. Em seguida, o Capítulo 3 apresenta uma breve introdução aos problemas de escalonamento, preparando o leitor com os conhecimentos necessários para discutir os problemas de escalonamento no contexto da indústria gráfica no Capítulo 4. Neste ponto, começamos abordando uma versão mais simples do problema antes de encerrar o capítulo com o problema de interesse.

Com o problema bem definido, passamos a olhar para os meios de resolvê-lo, dedicando o Capítulo 5 à tarefa de formalizar o método de otimização por colônia

de formigas. Ao longo deste capítulo, conhecemos os algoritmos que estruturam esse método e apresentamos a modelagem para problemas clássicos de otimização combinatória. Ao final, estamos habilitados a, finalmente, reunir o problema e a solução, descrevendo a aplicação do método no problema de interesse.

Colocando um fim na teoria, o Capítulo 6 apresenta os detalhes de implementação dos algoritmos estudados. Em seguida, no Capítulo 7 detalha a metodologia do trabalho, o delineamento experimental e apresenta os resultados obtidos. Por fim, encerramos com o Capítulo 8 falando sobre as conclusões, limitações, contribuições e trabalhos futuros.

2 Preliminares

Admitindo que o prezado leitor esteja familiarizado com grande parte do que vem a seguir, dedicaremos as próximas seções à tarefa de estabelecer notações e apresentar conceitos preliminares fundamentais para este trabalho. Para os interessados em explorar tais conceitos com maior profundidade, veja ([CAPUTI; MIRANDA, 2023](#)) e ([BONDY; MURTY, 2008](#)).

2.1 Conjuntos

A teoria ingênua dos conjuntos descreve um **conjunto** como sendo uma coleção qualquer de elementos. Tanto a coleção de todos os livros escritos por J. R. R. Tolkien quanto a coleção de todos os números pares são exemplos do que conjuntos podem representar. Para expressar essas coleções de forma clara e consistente, convencionou-se utilizar letras maiúsculas para denotar conjuntos e letras minúsculas para representar seus elementos.

A noção mais fundamental dentro da teoria dos conjuntos é a de pertinência, denotada por \in , que é um predicado binário envolvendo elementos e conjuntos. Escrevemos $a \in A$ para indicar que o elemento a **pertence** ao conjunto A , ou ainda, que A **contém**¹ a . A negação dessa afirmação é denotada por $a \notin A$. O conjunto que não contém nenhum elemento é dito **vazio** e é denotado por \emptyset . Assim, $a \notin \emptyset$ para cada elemento a .

Podemos descrever um conjunto de forma **declarativa**, listando diretamente os seus elementos. Por exemplo, $N = \{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ representa o conjunto de todos os *naipes* do baralho que deve ser entendido como, para cada elemento a , temos que $a \in N$ se, e só se, $a = \spadesuit$ ou $a = \heartsuit$ ou $a = \clubsuit$ ou $a = \diamondsuit$. Essa tarefa, no entanto, se torna hercúlea quando o conjunto possui um número muito grande de elementos. Nesses casos, é mais conveniente recorrer a uma descrição **predicativa**, que utiliza um critério ou condição que todos os elementos do conjunto devem satisfazer. Por exemplo, $P = \{n \in N \mid n \text{ é preto}\}$ representa o conjunto de todos os *naipes* pretos do baralho, isto é, o conjunto $\{\spadesuit, \clubsuit\}$.

2.1.1 Subconjuntos e maximalidade

Seja X um conjunto qualquer. Dizemos que um conjunto Y é um **subconjunto** de X , denotado por $Y \subseteq X$, se todo elemento de Y também pertence a X . Decorre dessa

¹ Como matemáticos são conhecidos por um grande senso de humor, $A \ni a$ é uma notação alternativa que captura essa noção.

definição que todo conjunto é subconjunto de si mesmo. Ademais, se X, Y , e Z são conjuntos tais que $Z \subseteq Y$ e $Y \subseteq X$, então $Z \subseteq X$.

A noção de igualdade é fundamental, e não seria diferente no caso de conjuntos. Sejam X e Y conjuntos. Dizemos que X e Y são **iguais**, denotado $X = Y$, se $Y \subseteq X$ e $X \subseteq Y$. Por outro lado, se $Y \subseteq X$ e $X \neq Y$, então dizemos que Y é um subconjunto **próprio** de X , o que é denotado por $Y \subset X$. Por exemplo, no contexto dos conjuntos de *naipes* apresentados anteriormente, podemos observar que P é um subconjunto próprio de N .

O conjunto dos números naturais, denotado por² $\mathbb{N} := \{0, 1, 2, \dots\}$, é geralmente o primeiro conjunto numérico que aprendemos, devido à sua relação intrínseca com a contagem e a ordenação. O conjunto dos números reais é denotado por \mathbb{R} . É claro que $\mathbb{N} \subset \mathbb{R}$. Ambos os conjuntos são amplamente utilizados neste trabalho, assim como os subconjuntos $\mathbb{N}_+ := \{n \in \mathbb{N} \mid n > 0\}$ (os naturais sem o zero), $\mathbb{R}_+ := \{r \in \mathbb{R} \mid r \geq 0\}$ (os reais não negativos) e $\mathbb{R}_{++} := \{r \in \mathbb{R} \mid r > 0\}$ (os reais estritamente positivos). Além disso, outro subconjunto frequentemente utilizado neste trabalho é o subconjunto dos $k \in \mathbb{N}$ primeiros números naturais $[k] = \{0, 1, \dots, k - 1\}$. Note que, desta definição, segue que $[0] = \emptyset$.

O conjunto formado por todos os subconjuntos de um conjunto X é chamado de **conjunto das partes** de X e é denotado por 2^X . Essa notação reflete o fato de que, se X é um conjunto finito com $n \in \mathbb{N}$ elementos, então o conjunto 2^X terá exatamente 2^n elementos. O conjunto das partes desempenha um papel fundamental em diversas áreas da matemática, sendo amplamente utilizado no estudo da combinatória e da teoria dos conjuntos.

Outra noção importante que será usada mais adiante é a de **maximalidade**. Seja X um conjunto e $\mathcal{P} \subseteq 2^X$. Dizemos que um conjunto $Y \in \mathcal{P}$ é **maximal** em \mathcal{P} se para cada $Z \in \mathcal{P}$, se $Y \subseteq Z$, então $Y = Z$. Note que \mathcal{P} pode conter mais de um elemento maximal. Por exemplo, considere o conjunto \mathcal{M} dos subconjuntos M de \mathbb{N} tais que (i) $1 \notin M$ e (ii) para todo $n \in \mathbb{N}$, se $n \notin M$, então $n + 2 \notin M$. Note que $\emptyset, \{2\}, \{0, 2\}$ são elementos de \mathcal{M} e que $\{n \in \mathbb{N} \mid n \text{ é par}\}$ é o único elemento maximal de \mathcal{M} . Dizemos que um conjunto $M \subseteq X$ é **maximal** em relação a uma propriedade p (sobre conjuntos) se M é um elemento maximal de $\{Y \subseteq X \mid p(Y)\}$.

2.1.2 Operações sobre conjuntos

Sejam A e B dois conjuntos quaisquer. As operações fundamentais de união, interseção e diferença permitem combiná-los para formar novos conjuntos. De forma sucinta, a **união** de A e B , denotada $A \cup B$, é o conjunto formado pelos elementos em A ,

² Sim! Como típicos cientistas da computação, assumiremos que 0 é um número natural.

em B ou em ambos os conjuntos. Já a **interseção** de A e B , denotada $A \cap B$, é formada pelos elementos contidos simultaneamente em A e em B . Se A e B não compartilharem elementos, isso é, $A \cap B = \emptyset$, dizemos que A e B são **disjuntos**.

A **diferença** $A \setminus B$ é o conjunto dos elementos que estão em A mas não estão em B . Se A é subconjunto de um conjunto X e x é um elemento de X , então escrevemos $A - x$ no lugar de $A \setminus \{x\}$. Pela coerência deste trabalho, adotaremos convenção semelhante para denotar a união com um conjunto unitário, escrevendo $A + x$ no lugar de $A \cup \{x\}$.

2.2 Relações e funções

Como o leitor testemunhará mais adiante, o presente trabalho busca empregar uma abordagem funcional às definições e algoritmos apresentados. Portanto, a noção de função e suas notações merecem uma seção de destaque neste capítulo. Para defini-las de maneira precisa, contudo, é necessário introduzir previamente o conceito de produto cartesiano.

2.2.1 Produto cartesiano

Sejam A e B conjuntos quaisquer. O **produto cartesiano** entre A e B , denotado por $A \times B$, é o conjunto de todos os pares ordenados formados por elementos de A seguidos por elementos de B , ou seja,

$$A \times B = \{(a, b) \mid a \in A \text{ e } b \in B\}.$$

Por exemplo, o produto cartesiano entre os conjuntos $X = \{1, 2\}$ e $Y = \{a, b, c\}$ é dado por

$$X \times Y = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}.$$

O produto cartesiano é especialmente importante para a noção de relação. Formalmente, uma **relação** entre A e B é um subconjunto $R \subseteq A \times B$ utilizado para codificar uma associação entre elementos dos dois conjuntos. Note que cada escolha de subconjunto de $A \times B$ determina uma relação diferente entre esses conjuntos.

Por exemplo. Considere o conjunto dos *naipes* $N = \{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ mencionado anteriormente e o conjunto de cores $C = \{\square, \blacksquare\}$. A relação

$$R := \{(\spadesuit, \blacksquare), (\heartsuit, \square), (\clubsuit, \blacksquare), (\diamondsuit, \square)\} \subseteq N \times C$$

relaciona cada *naipe* à sua cor correspondente.

Por fim, toda relação $R \subseteq A \times B$ está associada a outra relação conhecida como **relação inversa**, denotada por R^{-1}

$$R^{-1} = \{(b, a) \in B \times A \mid (a, b) \in R\}.$$

2.2.2 Função ou relação, eis a questão

Dados dois conjuntos A e B , uma **função parcial** de A em B é uma tripla (A, f, B) tal que $f \subseteq A \times B$ (portanto, uma relação entre A e B) que satisfaz a seguinte propriedade:

para todo $a \in A$, existe *no máximo* um elemento $b \in B$ tal que $(a, b) \in f$.

O conjunto f é chamado de **gráfico** da função. Uma forma mais compacta e amplamente usada neste trabalho de denotar o mesmo é

$$f: A \rightharpoonup B.$$

Seja $f: A \rightharpoonup B$ uma função parcial qualquer. O **domínio** de f é o conjunto

$$\text{Dom}(f) := \{a \in A \mid \exists b \in B: (a, b) \in f\}.$$

Dizemos que $f: A \rightharpoonup B$ é uma **função** quando $\text{Dom}(f) = A$ e, neste caso, escrevemos $f: A \rightarrow B$. Para cada $a \in \text{Dom}(f)$ escrevemos $f(a)$ para denotar o elemento $b \in B$ tal que $(a, b) \in f$. Às vezes, por uma questão de estilo, escrevemos alternativamente f_a no lugar de $f(a)$. O leitor deve ficar atento em relação à alternância desta notação. A **imagem** de f é o conjunto

$$\text{Im}(f) := \{f(a) \mid a \in \text{Dom}(f)\}$$

de todos os elementos em B relacionados a algum elemento no domínio através de f .³ Por fim, a **restrição** de f a um subconjunto $C \subseteq A$ é a função $f|_C: C \cap \text{Dom}(f) \rightarrow B$ tal que $f|_C := \{(a, f(a)) \mid a \in C \cap \text{Dom}(f)\}$.

Frequentemente neste trabalho, construiremos funções parciais a partir de outras, simplesmente modificando ou definindo a imagem de alguns elementos do domínio. Sejam $f: A \rightharpoonup B$ e $g: C \rightharpoonup B$ funções parciais. Considere a função parcial $f': A \cup C \rightharpoonup B$ tal que

$$f' := (f \setminus \{(a, f(a)) \mid a \in \text{Dom}(f) \cap A \cap C\}) \cup \{(a, g(a)) \mid a \in \text{Dom}(g) \cap C\}, \quad (2.1)$$

A partir deste ponto, escrevemos $f[C \ni a \mapsto g(a)]$ para denotar a função f' . Quando C for um conjunto unitário, isto é, se existe um elemento a tal que $C = \{a\}$, então escrevemos simplesmente $f[a \mapsto g(a)]$ no lugar de $f[\{a\} \ni c \mapsto g(c)]$.

No caso de funções parciais cuja imagem é um conjunto bem definido para a soma, isto é, quando sempre é possível somar dois elementos da imagem, peço licença ao leitor para estabelecer a seguinte notação. Seja $f: A \rightharpoonup B$ uma função parcial tal

³ Quando o contexto permitir vamos confundir uma função com o seu gráfico.

que $Im(f)$ está bem definida para a soma. Seja também $C \subseteq Dom(f)$. Para simplificar a notação de somatórios, adotamos a convenção de escrever $f(C)$ para representar

$$\sum_{a \in C} f(a).$$

É evidente que $f(\emptyset) = 0$.

Seja $f: A \rightharpoonup B$ uma função parcial. Se A e B são dois conjuntos com uma relação de ordem bem definida, estamos habilitados a discutir a noção de monotonicidade. Uma função parcial é **monótona** quando sua aplicação preserva uma certa ordem. Dizemos que f é **monótona não-decrescente** se, para todo $a_1 \leq a_2 \in Dom(f)$, temos que $f(a_1) \leq f(a_2)$. A função f é dita **estritamente crescente** se, além de ser monótona, satisfaz a desigualdade estrita $f(a_1) < f(a_2)$ para todo $a_1 < a_2 \in Dom(f)$.

Uma função $f: A \rightarrow B$ é dita **injetora** se todo par de elementos distintos no domínio é levado a imagens distintas, isso é, para todo $a, a' \in A$

$$f(a) = f(a') \implies a = a'.$$

Caso a imagem de f seja exatamente o conjunto B , ou seja, $Im(f) = B$, então a função f é dita **sobrejetora**. Quando f for simultaneamente injetora e sobrejetora, dizemos que f é **bijetora**.

Seja $f: A \rightharpoonup B$ uma função parcial. A **imagem inversa do conjunto** $C \subseteq B$ pela **função parcial** f , é o conjunto de todo $a \in A$ tal que $f(a) \in C$, isso é,

$$f^{-1}(C) := \{a \in A \mid f(a) \in C\}.$$

Como de costume, escrevemos $f^{-1}(b)$ no lugar de $f^{-1}(\{b\})$ para cada $b \in B$.

2.3 Algoritmos

Não abordaremos neste trabalho a definição formal de um algoritmo. Para nossos fins, a citação curta, porém sintética de Dasgupta et al. (2006) é mais do que suficiente para entender com o que estamos lidando.

“[...] procedimentos precisos, inequívocos, mecânicos, eficientes, corretos — em suma, algoritmos [...]” (DASGUPTA et al., 2006)

Como mencionado anteriormente, este trabalho se apoia fortemente sobre o paradigma funcional, tanto em essência quanto em estilo. Em virtude disso, com certo atrevimento, confundiremos a notação de função para citar a “assinatura”⁴ de algoritmos. Para uma demonstração da notação, o algoritmo $mdc: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ será descrito a seguir:

⁴ Por “assinatura” entenda o nome de um algoritmo, seguido do domínio da entrada e do domínio da saída.

Algoritmo 1 – Algoritmo de Euclides $\text{mdc}: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ para determinar o máximo divisor comum entre dois números

O presente algoritmo recebe dois números $a, b \in \mathbb{N}$ e calcula recursivamente o maior divisor comum entre eles. Para tal, há dois casos simples a considerar:

Caso 1: $b = 0$.

Neste caso, simplesmente devolva a e encerre o algoritmo.

Caso 2: $b \neq 0$.

Seja $r \in \mathbb{R}$ o resto da divisão inteira entre a e b . Devolva $\text{mdc}(b, r)$ e encerre o algoritmo.

Apesar de fundamental para este trabalho, outro aspecto da teoria da computação, que simplesmente admitiremos que o leitor possui certa familiaridade, é a teoria da complexidade dos algoritmos. A **teoria da complexidade** tem como principal função classificar os tipos de problemas computacionais e formalizar as bases que delimitam o que computadores podem ou não fazer na prática. Para se aprofundar em conceitos abordados mais adiante, como algoritmos polinomiais, classes de complexidade P e NP ou a notação *Big-O*, consulte ([CORMEN; LEISERSON, 2022](#)).

2.4 Sequências

O leitor perceberá que o problema central deste trabalho envolve a manipulação de objetos que carregam consigo alguma relação de ordem. Em razão disso, nesta seção, trataremos de formalizar o objeto matemático que captura essa noção.

Uma **sequência** σ de elementos tomados de um conjunto Σ é uma função

$$\sigma: [k] \rightarrow \Sigma, \quad (2.2)$$

onde $k \in \mathbb{N}$, denotado por $|\sigma|$, é dito o **comprimento** de σ . Existe uma única sequência de comprimento 0, chamada sequência **vazia**, denotada por ϵ . Os elementos que compõem uma sequência são denominados **termos**. Para economizar caracteres, adotaremos a notação $\underline{\sigma}$ para representar conjunto $Im(\sigma)$ dos termos de uma sequência σ . A forma mais habitual de expressar uma sequência é listando seus termos ordenadamente

$$\sigma(0), \sigma(1), \sigma(2), \dots, \sigma(k-1),$$

ou ainda,

$$\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_{k-1}.$$

Quando for necessário evitar ambiguidades, a sequência $\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_{k-1}$ será denotada por $\langle \sigma_0, \sigma_1, \sigma_2, \dots, \sigma_{k-1} \rangle$.

Antes de prosseguir, façamos um esforço para convencer o leitor do poder de representação deste objeto. Considere os conjuntos

$$C = \{J, Q, K, A, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \quad \text{e} \quad N = \{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}.$$

O produto cartesiano $C \times N$ é o conjunto de pares ordenados que representa um baralho francês de 52 cartas⁵. Uma função $\pi: [52] \rightarrow (C \times N)$ representa uma sequência qualquer de cartas. Se considerarmos tão somente as permutações, ou seja, as funções π bijetoras, a sequência

$$A\spadesuit, 2\spadesuit, 3\spadesuit, \dots, J\spadesuit, Q\spadesuit, K\spadesuit, A\heartsuit, \dots, K\heartsuit, A\clubsuit, \dots, K\clubsuit, A\diamondsuit, 2\diamondsuit, \dots, J\diamondsuit, Q\diamondsuit, K\diamondsuit$$

que corresponde à sequência ordenada padrão de um baralho, é apenas uma dentre as $52!$ possíveis. Acredite em mim, caro leitor, $52!$ é um número colossal. Se programarmos um computador para exibir uma permutação π distinta a cada segundo, o tempo necessário para concluir a execução desse programa superaria, por muito, a idade estimada do universo⁶.

Retomando ao que interessa, definimos o conjunto de todas as sequências de elementos sobre um conjunto Σ , denominado **fecho de Kleene**, como sendo o conjunto⁷

$$\Sigma^* := \bigcup_{k \in \mathbb{N}} ([k] \rightarrow \Sigma).$$

Há algo de intrigante nisso: se Σ fosse o conjunto de todos os símbolos que a humanidade já usou para se comunicar, então Σ^* abrigaria não só sequências ininteligíveis de símbolos, mas também coisas como este trabalho, os escritos na Pedra de Roseta, os livros que se perderam com a biblioteca de Alexandria, e até mesmo os livros que sequer foram escritos. Note também que, qualquer que seja a natureza de Σ , a sequência vazia ϵ sempre estará em Σ^* .

Sejam σ, π duas sequências sobre Σ , com $p = |\sigma|$ e $q = |\pi|$. A **concatenação de σ e π** , denotada por $\sigma \cdot \pi$, é a sequência $\mu: [p+q] \rightarrow \Sigma$ cujos termos são

$$\mu_i := \begin{cases} \sigma_i & \text{se } i < p, \text{ e} \\ \pi_{i-p} & \text{caso contrário,} \end{cases} \quad \text{para todo } i \in [p+q].$$

Em outras palavras, $\sigma \cdot \pi$ é a sequência

$$\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_{p-1}, \pi_0, \pi_1, \pi_2, \dots, \pi_{q-1},$$

isso é, os elementos de σ aparecem primeiro, seguidos pelos de π , preservando a ordem de cada sequência.

⁵ Por uma questão estética, escrevemos cn , no lugar do par ordenado $(c, n) \in C \times N$.

⁶ De acordo com o modelo Λ CDM o universo possui aproximadamente 13,8 bilhões de anos.

⁷ Se esta definição lhe causar estranheza, lembre-se que sequências são funções, e funções por sua vez, são essencialmente conjuntos de pares ordenados.

2.4.1 Prefixos, subsequências, permutações e truco

Sejam $\sigma, \pi \in \Sigma^*$ sequências de elementos em Σ . Diremos que π é um **prefixo** de σ , denotado por $\pi \sqsubseteq \sigma$, se existir uma sequência $\mu \in \Sigma^*$ (denominada **sufixo** de σ) tal que $\sigma = \pi \cdot \mu$. Note que a sequência vazia ϵ é prefixo e sufixo de qualquer sequência, o que implica que toda sequência é prefixo e sufixo de si mesma, ou seja,

$$\epsilon \cdot \sigma = \sigma \cdot \epsilon = \sigma.$$

Outra noção importante é a de subsequência. Informalmente, uma subsequência é a sequência resultante da remoção de alguns termos da sequência original, sem alterar a ordem dos elementos restantes.

Seja $\sigma: [k] \rightarrow \Sigma$ uma sequência de comprimento k sobre Σ . Uma **subsequência** de σ é uma sequência

$$\sigma \circ \lambda: [j] \rightarrow \Sigma$$

de comprimento $j \leq k$, obtida pela composição de σ com uma função estritamente crescente $\lambda: [j] \rightarrow [k]$. Em outras palavras, uma subsequência de σ é a sequência

$$\sigma_{\lambda(0)}, \sigma_{\lambda(1)}, \sigma_{\lambda(2)}, \dots, \sigma_{\lambda(j-1)}.$$

Quando $j = k$, dizemos que $\sigma \circ \lambda$ é uma **permutação** de σ .

Considere, por exemplo, a sequência $\langle \clubsuit, \heartsuit, \spadesuit, \diamondsuit \rangle$, que representa, da esquerda para a direita, a hierarquia dos *naipes* em um jogo de cartas muito popular na América Latina chamado truco. A composição de ϕ com a função

$$\lambda_1 = \{(0, 0), (1, 1)\}$$

produz a subsequência $\langle \clubsuit, \heartsuit \rangle$ conhecida no jogo como **casal maior**. Já a composição com a função

$$\lambda_2 = \{(0, 1), (1, 3)\}$$

produz a subsequência $\langle \heartsuit, \diamondsuit \rangle$, denominada **casal vermelho**.

Uma vez que o formalismo das sequências está posto à mesa, apresentaremos algumas convenções que facilitarão nossa vida daqui em diante. Seja σ uma sequência qualquer de elementos tomados de um conjunto $\Sigma \neq \emptyset$. Quando for conveniente, escreveremos simplesmente $\sigma\pi$ no lugar de $\sigma \cdot \pi$. Também confundiremos deliberadamente um elemento $e \in \Sigma$ com a sequência unitária $\langle e \rangle$ a fim de escrever σe no lugar de $\sigma \cdot \langle e \rangle$, por exemplo. Por fim, seja $k = |\sigma|$. Para inteiros $0 \leq i \leq j < k$ escreveremos:

- $\sigma_{i..j}$ no lugar de $\langle \sigma_i \sigma_{i+1}, \dots, \sigma_{j-1}, \sigma_j \rangle$;
- $\sigma_{..j}$ no lugar de $\sigma_{0..j}$; e

- $\sigma_{i..}$ no lugar de $\sigma_{i..(k-1)}$.

Com o mesmo intuito, se $s = \sigma_i, t = \sigma_j$ para inteiros $0 \leq i \leq j < k$, escreveremos $\sigma_{s..t}$ no lugar de $\sigma_{i..j}$, bem como $\sigma_{s..}$ no lugar de $\sigma_{i..}$ e $\sigma_{..t}$ no lugar de $\sigma_{..j}$.

2.5 Digrafos

Entre as noções fundamentais deste trabalho destaca-se a de digrafo, uma estrutura combinatória usada para representar relações binárias entre objetos, conforme será definido a seguir. Um **digrafo** G é uma tripla (V, A, ψ) , onde

- V é um conjunto não-vazio e finito cujos elementos são chamados de **vértices**;
- A é um conjunto finito e disjunto de V , cujos elementos chamam-se **arcos**; e
- $\psi: A \rightarrow (V \times V) \setminus \{(v, v) \mid v \in V\}$ é uma função, chamada de **função de incidência**.

A função de incidência associa cada arco $a \in A$ a um par ordenado de vértices distintos $\psi(a) = (u, v)$. Os vértices u e v são chamados de **pontas** de a , sendo u a **cauda** e v a **cabeça** do arco, também representados por $*a$ e $a*$, respectivamente. Quando estiver claro no contexto, usaremos simplesmente V , A e ψ para representar os conjuntos de vértices, de arcos e a função de incidência de um digrafo G . Caso contrário, esses objetos serão adornados com o nome do digrafo, aparecendo como V_G , A_G e ψ_G .

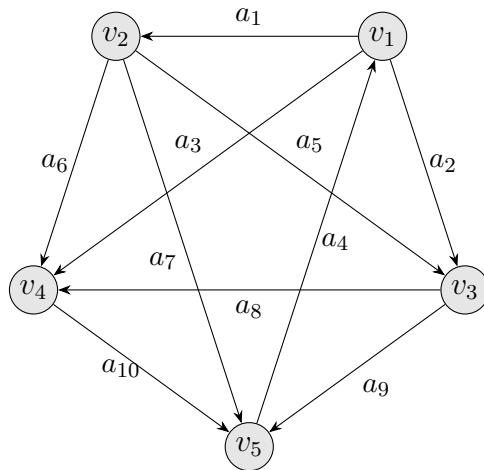


Figura 1 – Representação gráfica de um digrafo.

Um **digrafo** pode ser representado graficamente por círculos e setas, de modo que exista uma bijeção entre os círculos e os vértices, assim como uma bijeção entre as setas e os arcos. Para cada arco a do digrafo, a seta que representa a parte do círculo

correspondente ao vértice $*a$ e termina no círculo que representa o vértice $a*$. A Figura 1 ilustra um digrafo descrito pelos seguintes componentes:

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

$$A = \{a_1, a_2, \dots, a_{10}\}$$

$$\psi = \{(a_1, (v_1, v_2)), (a_2, (v_1, v_3)), (a_3, (v_1, v_4)), (a_4, (v_5, v_1)), (a_5, (v_2, v_3)), \\ (a_6, (v_2, v_4)), (a_7, (v_2, v_5)), (a_8, (v_3, v_4)), (a_9, (v_3, v_5)), (a_{10}, (v_4, v_5))\}$$

Observe que a definição de digrafo não diz nada sobre a injetividade de ψ . Em outras palavras, nada impede que existam arcos distintos, digamos a e b , cujas extremidades coincidam, isto é, $\psi(a) = \psi(b)$. Nesse caso, dizemos que os arcos a e b são **paralelos**. Por outro lado, caso $a, b \in A$ sejam arcos distintos tais que existem vértices $u, v \in V$ onde $\psi(a) = (u, v)$ e $\psi(b) = (v, u)$, diremos que a e b são arcos **antiparalelos**. Um digrafo é dito **simples** se não possui arcos paralelos. Em digrafos simples, cada arco pode ser identificado univocamente por suas pontas, o que motiva a seguinte convenção usada para simplificar a notação: um arco a tal que $\psi(a) = (u, v)$ num digrafo simples $G = (V, A, \psi)$ será simplesmente denotado por uv . Nestes casos, podemos jogar a função de incidência para debaixo do tapete e escrever somente (V, A) para definir G .

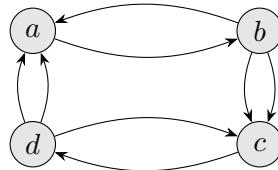


Figura 2 – Representação de um digrafo com arcos paralelos e antiparalelos.

Seja G um digrafo. A **vizinhança de entrada** de um vértice $v \in V$, denotada por $N^-(v)$, é o conjunto das caudas dos arcos cuja cabeça é v . Por outro lado, a **vizinhança de saída** de v , denotada por $N^+(v)$, é o conjunto das cabeças dos arcos cuja cauda é v . É comum denominar os elementos de $N^+(v)$ de **sucessores** de v . De forma análoga, os elementos de $N^-(v)$ são chamados de **predecessores** de v . O vértice v é dito uma **fonte** quando não tem predecessores e um **sorvedouro** quando não tem sucessores. Como de costume, quando não estiver claro no contexto, os conjuntos que representam as vizinhanças de um vértice serão adornados pelo digrafo G a que pertencem.

2.5.1 Subdigrafo

Sabendo que um digrafo é um objeto matemático modelado por conjuntos e funções, o leitor pode se perguntar o que obtemos ao restringir esses conjuntos. A resposta é que, quase certamente, obteremos um subdigrafo. Formalmente, um digrafo H é um **subdigrafo** de um digrafo G — denotado por $H \subseteq G$ — se atender às seguintes condições:

- (i) $V_H \subseteq V_G$,
- (ii) $A_H \subseteq A_G$, e, principalmente,
- (iii) $\psi_H = \psi_{G|_{A_H}}$ é uma restrição de ψ_G ao conjunto de arcos A_H .

Se $H \subseteq G$ mas $H \neq G$, diremos que H é um **subdigrafo próprio** de G , denotando por $H \subset G$. Por fim, dizemos que $H \subseteq G$ é um **subdigrafo gerador** de G , se $V_H = V_G$.



Figura 3 – Representação de um digrafo G (esq.) e de seu subdigrafo $G - \{b, d\}$ (dir.).

Seja G um digrafo qualquer. A forma mais trivial de se obter um subdigrafo de G , como ilustrado pela Figura 3, é **removendo vértices**. Seja $V' \subset V_G$ um conjunto de vértices de G . O subdigrafo $G - V'$ obtido após a remoção dos vértices em V' , é por definição um digrafo tal que $V_{G-V'} := V_G \setminus V'$ e $A_{G-V'} := \{a \in A_G \mid \psi(a) \in V_{G-V'} \times V_{G-V'}\}$ é um conjunto de arcos cujas pontas estão em $V_{G-V'}$. Analogamente, dado um subconjunto $A' \subseteq A_G$, a **remoção dos arcos** A' gera o subdigrafo $G - A'$, tal que $V_{G-A'} := V_G$ e $A_{G-A'} := A_G \setminus A'$. Naturalmente, quando V' e A' forem conjuntos unitários, digamos $\{v\}$ e $\{a\}$, escreveremos $G - v$ no lugar de $G - \{v\}$ e $G - a$ no lugar de $G - \{a\}$.

2.5.2 Caminhos e ciclos

Um caminho em um digrafo é formado por uma sequência alternada de vértices e arcos, sendo que cada arco tem como cauda o vértice que o precede e como cabeça o vértice posterior na sequência. Formalmente, um **caminho de v_0 até v_k** (ou ainda, um v_0v_k -caminho) em um digrafo G é uma sequência

$$v_0, a_1, v_1, \dots, a_k, v_k$$

em que k , dito o **comprimento**⁸ do caminho, é um inteiro não-negativo e

- (i) $v_0, v_1, \dots, v_k \in V$,

⁸ Note que, embora k seja o comprimento do caminho, a sequência alternada que representa o caminho tem comprimento $2k + 1$.

- (ii) $a_1, \dots, a_k \in A$, e
- (iii) $\psi(a_i) = (v_{i-1}, v_i)$ para cada $i \in [k] - 0$.

Seja $\rho = \langle v_0, a_1, v_1, \dots, a_k, v_k \rangle$ um caminho. Dizemos que v_0 é o **início** e v_k é o **término** de ρ . O conjunto $V(\rho) = \{v_0, v_1, \dots, v_k\}$ associado a ρ reúne os vértices que compõem o caminho. Analogamente, o conjunto $A(\rho) = \{a_1, \dots, a_k\}$ contém os arcos de ρ . Dizemos que ρ é um caminho **não trivial** se $k > 0$. Neste caso, ρ pode ser identificado tão somente pelos seus arcos, então escreveremos a_1, a_2, \dots, a_k no lugar de $v_0, a_1, v_1, \dots, a_k, v_k$ quando for conveniente.

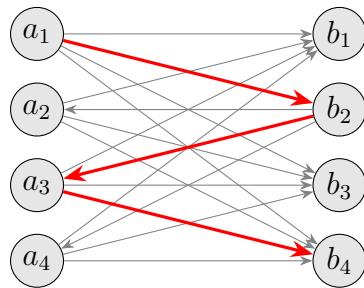


Figura 4 – Um digrafo com um caminho simples destacado em vermelho.

Um caminho $\rho = \langle v_0, a_1, v_1, \dots, a_k, v_k \rangle$ é **simples** se os seus vértices são dois a dois distintos, isto é, ρ é simples se $v_i \neq v_j$ para cada $i \neq j \in \{0, 1, \dots, k\}$. Se $V(\rho) = V$, então ρ é dito um **caminho hamiltoniano**. Um caminho $v_0, a_1, v_1, \dots, a_k, v_k$ é um **ciclo** se $v_0, \dots, a_{k-1}, v_{k-1}$ é um caminho simples e $v_k = v_0$. É claro que, quando o digrafo é simples, é possível identificar um caminho tão somente pelos seus vértices. Nesse caso, escrevemos $\langle v_0, v_1, \dots, v_k \rangle$ em vez de $\langle v_0, a_1, v_1, \dots, a_k, v_k \rangle$. A Figura 4 ilustra, destacado em vermelho no digrafo, o caminho simples $\langle a_1, b_2, a_3, b_4 \rangle$.

A concatenação dos caminhos

$$\rho = \langle v_0, a_1, v_1, \dots, a_k, v_k \rangle \text{ e } \sigma = \langle v_k, a_{k+1}, v_{k+1}, \dots, a_l, v_l \rangle,$$

denotada por $\rho \cdot \sigma$, é o caminho $\langle v_0, a_1, v_1, \dots, a_k, v_k, a_{k+1}, v_{k+1}, \dots, a_l, v_l \rangle$. Seja ρ um caminho que termina em um vértice u e que a seja um arco tal que $\psi(a) = (u, v)$. Adotaremos a seguinte convenção para simplificar a notação. Escreveremos $\rho \cdot a$ para denotar $\rho \cdot \langle u, a, v \rangle$. A mesma convenção se aplicará ao concatenar um caminho a um arco. Ademais, se ρ é um caminho em um digrafo simples, isso é, ψ é bijetora, então, podemos escrever $\rho \cdot v$ para denotar $\rho \cdot a$. Analogamente, convenção similar será utilizada ao concatenar um caminho a um vértice. Como de costume, quando for conveniente, omitiremos o símbolo de concatenação, escrevendo $\rho\sigma$ em vez de $\rho \cdot \sigma$.

Frequentemente estaremos interessados em comparar propriedades de diferentes caminhos que tenham as mesmas extremidades. Isso nos leva a denotar por \mathbb{P}_v o conjunto

de todos os caminhos que terminam em um vértice $v \in V$. Por exemplo, conforme ilustra o digrafo na Figura 4, enquanto \mathbb{P}_{a_1} contém apenas o caminho trivial $\langle a_1 \rangle$, o conjunto \mathbb{P}_{b_4} contém os caminhos $\langle a_2, b_4 \rangle$, $\langle b_2, a_4, b_4 \rangle$ e $\langle a_1, b_2, a_3, b_4 \rangle$.

2.5.3 Digrafos acíclicos

Digrafos acíclicos desempenham um papel fundamental neste trabalho. Suponha dada uma coleção de tarefas e uma relação de precedência entre elas. A relação de precedência deve satisfazer a seguinte propriedade de consistência: cada tarefa da coleção só pode ser executada desde que todas as tarefas das quais ela depende já tenham sido executadas previamente.

Considere um digrafo cujo conjunto dos vértices é o conjunto das tarefas e o conjunto dos arcos é a relação de precedência. A noção que captura a propriedade de consistência é a de um digrafo acíclico (ilustrado na Figura 5), cuja definição exibimos a seguir. Um digrafo é dito **acíclico** se não contém ciclos ou, de maneira equivalente, o início e o término de todo caminho não trivial no digrafo são distintos.

Uma **ordenação acíclica parcial** de um digrafo G é uma sequência

$$v_0, v_1, \dots, v_{k-1}$$

de $k \geq 0$ vértices, dois a dois distintos, de G tal que $N^-(v_i) \subseteq \{v_0, \dots, v_{i-1}\}$ para cada $i \in \{0, \dots, k-1\}$. Uma ordenação acíclica parcial é simplesmente uma **ordenação acíclica** quando $k = |V|$.

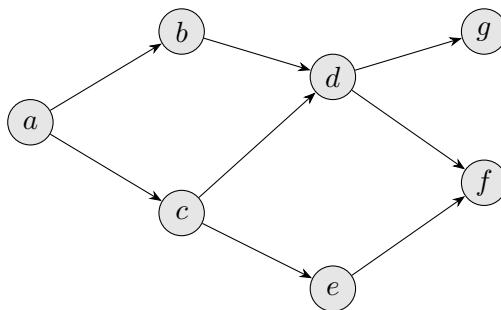


Figura 5 – Digrafo representando tarefas e suas relações de precedência.

As sequências $\langle a, c, b, d, e, g, f \rangle$, $\langle a, b, c, d, g, e, f \rangle$ e $\langle a, c, b, d, g, e, f \rangle$ são ordenações acíclicas igualmente válidas para o digrafo da Figura 5. Em geral, a depender de sua estrutura, pode existir um número não polinomial (em relação ao número de vértices) de ordenações acíclicas em um mesmo digrafo. Mas quando podemos garantir que existe ao menos uma delas? A resposta desta pergunta aparece no Teorema 2.5.1, que, apesar de bem conhecido, é fundamental para a consistência deste trabalho.

Teorema 2.5.1. Um digrafo é acíclico se, e só se, admite uma ordenação acíclica.

Prova. É claro que se um digrafo G admite uma ordenação acíclica, então é um digrafo acíclico. Para ver a recíproca, suponha então que G é um digrafo acíclico. Seja $\pi = \langle v_0, \dots, v_{k-1} \rangle$ uma ordenação acíclica parcial de G cujo comprimento é máximo. Vamos mostrar que $k = |V|$.

Suponha para uma contradição que $k < |V|$ e note que, neste caso, o conjunto $V \setminus \underline{\pi}$ é não-vazio. Seja ρ um caminho simples de comprimento máximo que não usa nenhum dos vértices do conjunto $\underline{\pi}$. Seja v_k o início de ρ . Vamos mostrar que todo antecessor de v_k é termo de π .

Suponha que não. Então existe $u \notin \{v_0, \dots, v_k\}$ que é um antecessor de v_k . Temos dois casos. Se u não é um dos vértices de ρ , então $u\rho$ é um caminho simples que não usa nenhum dos vértices do conjunto $\{v_0, \dots, v_{k-1}\}$, o que contraria a escolha de ρ . Assim, u é um dos vértices de ρ . No entanto, nesse caso⁹, $\rho_{v_k \dots u} \cdot v_k$ é um ciclo de G , o que é uma contradição, pois G é acíclico. Logo, todo antecessor de v_k está em $\{v_0, \dots, v_{k-1}\}$ e, portanto, é um termo de π .

Ora, se este é o caso, então $\pi\rho$ é uma ordenação acíclica parcial de G que contraria a escolha de π . Concluímos, assim, que $k = |V|$ e, portanto, π é uma ordenação acíclica de G . \square

2.5.4 Digrafos ponderados

Considere o cenário descrito no início da Subseção 2.5.3, mas agora com um novo ingrediente: neste caso, as tarefas possuem durações distintas. Diante disso, com o objetivo de calcular o tempo necessário para concluir um conjunto de tarefas, associamos cada arco a um número não negativo que representa o tempo necessário para partir de uma tarefa para a seguinte. Naturalmente, o modelo matemático que codifica essa noção é um digrafo ponderado. Um **digrafo ponderado** é um par (G, w) , onde G é um digrafo e $w : A \rightarrow \mathbb{R}$ é uma função que o pondera, chamada pelos mais íntimos de **função peso**.

O **custo** de um caminho ρ — que sem pudor algum, denotamos com abuso de notação por $w(\rho)$ — é a soma dos pesos de cada arco no caminho. Mais especificamente, se $\rho = \langle v_0, a_1, v_1, \dots, a_k, v_k \rangle$ é um caminho em um digrafo ponderado (G, w) , seu custo é

$$w(\rho) := \sum_{i=1}^k w(a_i).$$

Note que decorre da definição que o custo de todo caminho trivial é 0.

⁹ Lembre-se que $\rho_{v_k \dots u}$ denota a subsequência de ρ que vai de v_k até u

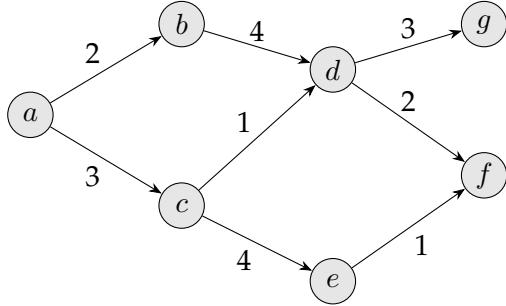


Figura 6 – Digrafo acíclico ponderado pelo tempo de processamento das tarefas.

As propriedades acerca do custo de um caminho estão intimamente relacionadas, como veremos mais adiante, com o problema que estamos prestes a explorar.

2.5.5 Caminho críticos

Nesta seção, vamos estudar o problema de, dado um digrafo ponderado (G, w) , determinar para cada vértice v de G um caminho ρ que termina em v e tem custo máximo. Antes de prosseguir, faz-se uma observação preliminar importante.

Vamos mostrar que podemos admitir que G é *simples*. De fato, suponha que G possui dois arcos paralelos $a_1 \neq a_2$ e que $w(a_1) \leq w(a_2)$. Considere o digrafo $G - a_1$ obtido de G através da remoção do arco a_1 . É imediato verificar que, para cada vértice v de G o custo de um caminho de custo máximo em G que termina em v coincide com o custo de um caminho de custo máximo de $G - a_1$ que termina em v . Portanto, sem perda de generalidade, vamos supor, nesta seção, que todo digrafo é simples. Assim, o princípio universal da preguiça permite que, durante esta seção, escrevamos “digrafo” no lugar de “digrafo simples”.

Seja (G, w) um digrafo ponderado acíclico. Note que, para cada vértice v de G o conjunto \mathbb{P}_v dos caminhos de G que terminam em v é finito – pois G é acíclico – e não vazio, uma vez que $\langle v \rangle$ sempre pertence a \mathbb{P}_v . Dito isso, é conveniente definir a função

$$\delta_G(v) := \max\{w(\rho) \mid \rho \in \mathbb{P}_v\},$$

que associa cada vértice v de G ao máximo dentre os custos dos caminhos que terminam em v . Quando estiver claro no contexto, escreveremos simplesmente δ no lugar de δ_G . Quando a função peso, w , é não negativa, isto é, quando $w(a) \geq 0$, para cada arco a de G , temos, evidentemente, que δ também é não negativa. Remetendo-se à motivação original para esta discussão, o valor de $\delta(t)$ para um certo vértice t representando uma tarefa, pode ser entendido como o menor instante em que essa tarefa pode ser iniciada sem que a propriedade de consistência seja violada.

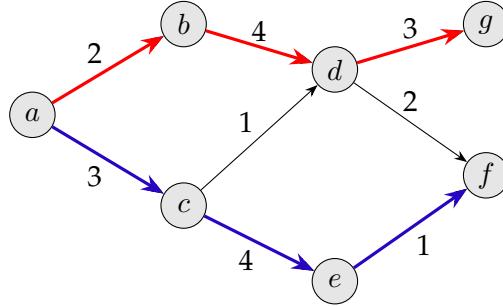


Figura 7 – Um digrafo acíclico ponderado com um caminho crítico até g destacado em vermelho e um caminho crítico até f em azul.

Para todo $v \in V_G$, um caminho $\rho \in \mathbb{P}_v$ tal que $w(\rho) = \delta(v)$ é dito **crítico** até v . A Figura 7 destaca em vermelho o caminho $\langle a, b, d, g \rangle$ crítico até g e em azul o caminho $\langle a, c, e, f \rangle$ crítico até f . O próximo teorema estabelece algumas propriedades básicas sobre caminhos críticos, contudo, precisamos de uma definição preliminar. Seja v um vértice de G . Um elemento u^* em $N^-(v)$ tal que $\delta(u^*) + w(u^*v) \geq \delta(u) + w(uv)$ para cada u em $N^-(v)$ é dito um **predecessor crítico** de v .

Teorema 2.5.2. *Seja (G, w) um digrafo ponderado e acíclico. Para cada $v \in V_G$, se u^* é um predecessor crítico de v e ρ é um caminho crítico até u^* , então*

- (i) ρv é um caminho crítico até v , e
- (ii) $\delta(v) = \max\{\delta(u) + w(uv) \mid u \in N^-(v)\}$.

Prova. Seja $v \in V_G$ e suponha que u^* é um predecessor crítico de v e que ρ é um caminho crítico até u^* . Primeiro, mostraremos que ρv é um caminho crítico até v . Note que, como G é acíclico, então v não é um dos vértices de ρ e, consequentemente, ρv é um elemento de \mathbb{P}_v . Seja φ um caminho que termina em v , ou seja, $\varphi \in \mathbb{P}_v$. Vamos mostrar que $w(\varphi) \leq w(\rho v)$, o que implica que ρv é crítico até v .

Como o resultado é evidente se φ é um caminho trivial, então suponha que este não é o caso. Seja u o vértice que precede v em φ e considere o prefixo $\varphi_{..u}$ de φ que termina em u . Então

$$w(\varphi) = w(\varphi_{..u}) + w(uv) \leq \delta(u) + w(uv) \leq \delta(u^*) + w(u^*v) = w(\rho) + w(u^*v) = w(\rho v),$$

onde a primeira desigualdade vem da definição de δ (pois $\varphi_{..u} \in \mathbb{P}_u$) e a segunda, da definição de predecessor crítico. Logo, ρv é crítico até v e, assim, $w(\rho v) = \delta(v)$, o que prova (i). A desigualdade acima estabelece que $\delta(v) \geq \delta(u) + w(uv)$, para todo $u \in N^-(v)$, o que prova (ii). \square

É conveniente introduzir a seguinte definição para facilitar o enunciado do próximo corolário. Seja G um digrafo e ρ um caminho de G . Para simplificar, vamos escrever $i(\rho)$ para denotar o início de ρ .

Corolário 2.5.1. Seja (G, w) um digrafo ponderado e acíclico. Se $w(x) \geq 0$ para todo $x \in \text{Dom}(w)$, então para cada vértice t de G e para cada $\rho \in \mathbb{P}_t$ vale que

$$\delta(t) \geq \delta(i(\rho)). \quad (2.3)$$

Prova. Suponha que $w \geq 0$. Seja $t \in V_G$. Vamos mostrar, por indução em $|\rho|$, que

$$\delta(t) \geq \delta(i(\rho)) + w(\rho)$$

para cada $\rho \in \mathbb{P}_t$. O resultado é evidente se $|\rho| = 0$. Suponha então que $|\rho| > 0$. Seja ρ' um caminho tal que $\rho = i(\rho) \cdot \rho'$. Então,

$$\begin{aligned} \delta(t) &\geq \delta(i(\rho')) + w(\rho') && \text{por hipótese de indução} \\ &\geq \delta(i(\rho)) + w(i(\rho)i(\rho')) + w(\rho') && \text{pelo Teorema 2.5.2} \\ &= \delta(i(\rho)) + w(\rho) && \text{pela definição de } \rho. \end{aligned}$$

Agora, como w é não-negativo, então $\delta(t) \geq \delta(i(\rho))$. □

Os resultados do Teorema 2.5.2 implicam em um algoritmo linear para determinar um caminho crítico até cada um dos vértices de um digrafo ponderado e acíclico. O Algoritmo 2, enunciado¹⁰ abaixo, depende da noção de ramificação que vamos definir agora. Seja G um digrafo. Uma **ramificação** é um subdigrafo acíclico B de G tal que para cada $v \in V_B$ há um único caminho maximal em B que termina em v . Uma ramificação de G é **geradora** se contém todos os vértices de G .

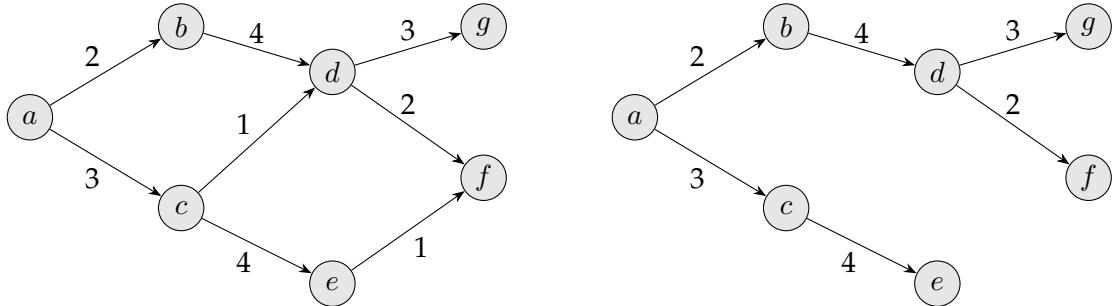


Figura 8 – Digrafo acíclico ponderado (esq.) e uma ramificação geradora w -ótima (dir.).

Em um digrafo ponderado (G, w) dizemos que uma ramificação B é **w -ótima** se para cada $v \in V_B$, o único caminho maximal de B que termina em v é um caminho crítico até v em G . A Figura 8 destaca uma ramificação geradora w -ótima.

¹⁰ O estilo adotado na descrição dos algoritmos iterativos deste trabalho foi disseminada pelo Prof. Dr. Paulo Feofilof.

Algoritmo 2 – Algoritmo ramificao-w* para construção de uma ramificação geradora w -ótima

O algoritmo recebe um digrafo ponderado (G, w) e devolve um par (B, γ) tal que B é uma ramificação geradora w -ótima de G e $\gamma : V_G \rightarrow \mathbb{R}^+$ é uma função tal que $\gamma = \delta$.

Construa uma ordenação acíclica, $\langle v_0, v_1, \dots, v_{n-1} \rangle$, de G . Em uma iteração $i \in [n]$ arbitrária do algoritmo, admita que B é uma ramificação w -ótima cujo conjunto dos vértices V_B é $\{v_0, v_1, \dots, v_{i-1}\}$ e $\gamma : V_B \rightarrow \mathbb{R}_{++}$ é tal que $\gamma = \delta|_{V_B}$.

Considere o próximo vértice, v_i , na ordenação acíclica.

Caso 1: $N_G^-(v_i) = \emptyset$.

Seja $B' = (V_B \cup \{v_i\}, A_B)$ a ramificação obtida de B através da adição do vértice v_i e $\gamma' := \gamma[v_i \mapsto 0]$. É claro que B' é w -ótima e $\gamma' = \delta|_{V_{B'}}$.

Caso 2: $N_G^-(v_i) \neq \emptyset$.

Selecione um predecessor $u^* \in N_G^-(v_i)$ tal que

$$\gamma(u^*) + w(u^*) \geq \gamma(u) + w(u)$$

para cada $u \in N_G^-(v_i)$. Observe que $N_G^-(v_i) \subseteq V_B$ pois $\langle v_0, \dots, v_{n-1} \rangle$ é uma ordenação acíclica. Note que como $\gamma = \delta|_{V_B}$, então u^* é um predecessor crítico de v_i . Por hipótese, o caminho maximal de B que termina em u^* é crítico até u^* . Dessa forma, a ramificação B' obtida de B através da adição do vértice v_i e do arco u^*v_i é w -ótima. Ademais, $\gamma' := \gamma[v_i \mapsto \gamma(u^*) + w(u^*v_i)]$ é tal que $\gamma' = \delta|_{V_{B'}}$.

Na primeira iteração, comece com $B = (\{v_0\}, \emptyset)$ e $\delta = \{(v_0, 0)\}$.

3 Problemas de escalonamento

Dedicaremos este capítulo a um breve estudo dos problemas de escalonamento e alocação. Breve, pois, seria muita presunção reduzir o estudo de uma área com décadas de descobertas a um simples capítulo. A popularidade desta área se dá pela sua grande aplicabilidade: esse tipo de problema tem aplicações em inúmeros sistemas produtivos, gestão de projetos e até mesmo no funcionamento de sistemas operacionais multitarefas.

Em geral, problemas de escalonamento consistem na alocação de recursos para um certo conjunto de atividades ou tarefas ao longo do tempo. Por recursos, entenda qualquer coisa necessária para desempenhar a tarefa, como a matéria-prima para a fabricação de um produto ou até mesmo o maquinário utilizado para desempenhar uma certa atividade, caso ao qual nos restringiremos aqui. Começaremos com um problema de escalonamento muito simples, mas que, surpreendentemente, é NP-difícil ([GAREY; JOHNSON, 1978](#)).

3.1 Escalonamento em máquinas idênticas

Informalmente, este problema consiste em, dado um conjunto de tarefas, determinar em qual máquina no maquinário disponível cada tarefa será executada. O problema possui esse nome, pois, assumimos que o maquinário é homogêneo, de modo que a duração de cada tarefa é a mesma independente da máquina escolhida. Dentre todas as possibilidades, buscamos encontrar a alocação em que o tempo gasto para executar todas as tarefas seja o menor possível.

Seja M um conjunto finito e não vazio de **máquinas** e J um conjunto finito e não vazio de **tarefas**. A cada tarefa j está associada, por uma função $p: J \rightarrow \mathbb{R}_{++}$, um número real estritamente positivo, $p(j)$, chamado **duração** de j . Um **escalonamento parcial** é uma função parcial $\pi: J \rightharpoonup M$ que associa uma tarefa $j \in J$ a uma única máquina $\pi(j) \in M$. Se π for total, isto é, uma função $\pi: J \rightarrow M$, dizemos que π é simplesmente um **escalonamento**.

Dado um escalonamento parcial π , a **carga de trabalho** de uma máquina $m \in M$ é a imagem inversa $\pi^{-1}(m)$, que consiste no conjunto das tarefas alocadas em m . Isto posto, a **ocupação** de uma máquina $m \in M$ em um escalonamento parcial π é a soma

$$p(\pi^{-1}(m)) = \sum_{j \in \pi^{-1}(m)} p(j)$$

da duração dos trabalhos executados em m .

O *makespan* de um escalonamento parcial $\pi: J \rightharpoonup M$ é a maior ocupação de uma máquina no escalonamento, isso é,

$$\nu_p(\pi) := \max\{p(\pi^{-1}(m)) \mid m \in M\}. \quad (3.1)$$

O objetivo final de um problema de escalonamento em máquinas idênticas, representado pela tripla (J, M, p) , é encontrar um escalonamento cujo *makespan* seja mínimo. Um escalonamento com essa propriedade é dito **ótimo** e o seu *makespan* é denotado por¹

$$\text{opt}(J, M, p) := \min\{\nu_p(\pi) \mid \pi: J \rightarrow M\}. \quad (3.2)$$

Exemplo 3.1 – Escalonamento de 7 tarefas em 5 máquinas idênticas.

Considere o problema de escalonamento em máquinas idênticas representado pela tripla

$$I = (\{a, b, c, d, e, f, g\}, \{M_1, M_2, M_3, M_4, M_5\}, p),$$

onde p é a função

$$p := \{(a, 8), (b, 3), (c, 17), (d, 11), (e, 4), (f, 11), (g, 15)\}.$$

A função

$$\pi := \{(a, M_5), (b, M_3), (c, M_1), (d, M_3), (e, M_5), (f, M_4), (g, M_2)\}$$

é um escalonamento para I .

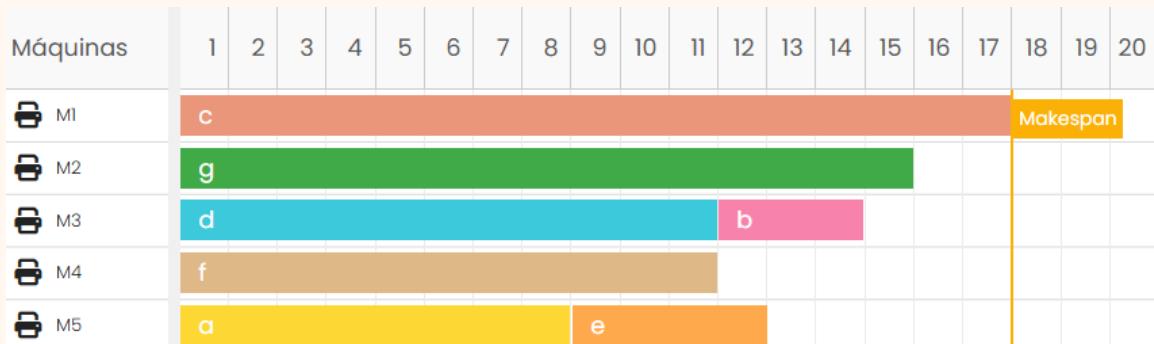


Figura 9 – Diagrama Gantt de um escalonamento de 7 tarefas em 5 máquinas.

Podemos representar escalonamentos graficamente por meio de um diagrama Gantt. Um diagrama Gantt é um gráfico que representa as tarefas por meio de barras, cuja largura é proporcional à sua duração. A Figura 9 ilustra o diagrama Gantt de um

¹ Note que o ótimo não é definido para escalonamentos parciais.

escalonamento para a instância exibida no Exemplo 3.1, cujo *makespan* — evidenciado pela linha amarela — é 17.

A seguir, apresentaremos um algoritmo guloso para a construção de soluções aproximadas em tempo polinomial.

3.1.1 Algoritmo de Graham

É bem sabido na literatura que o problema de escalonamento em máquinas idênticas é um problema NP-difícil (GAREY; JOHNSON, 1978), o que significa que não conhecemos um algoritmo eficiente para encontrar um escalonamento ótimo. No entanto, como veremos a seguir, Ronald L. Graham² apresentou um algoritmo guloso para a construção de soluções para esse problema, que possui propriedades interessantes.

Algoritmo 3 – Algoritmo graham para construção de um escalonamento em máquinas idênticas

O algoritmo recebe uma instância (J, M, p) de um problema de escalonamento. Como resultado, o algoritmo produz um escalonamento π tal que

$$\nu_p(\pi) \leq 2 \cdot \text{opt}(J, M, p).$$

A primeira iteração começa com a função vazia \emptyset . Admita que em uma iteração arbitrária, $\pi: J \rightarrow M$ é um escalonamento parcial. Seja $U = J \setminus \pi^{-1}(M)$ o conjunto das tarefas que ainda não foram escalonadas. Há dois casos a considerar:

Caso 1: $U = \emptyset$.

Neste caso, devolva π e encerre o algoritmo.

Caso 2: $U \neq \emptyset$.

Tome a máquina $k \in M$ menos ocupada, isso é, tal que

$$p(\pi^{-1}(k)) \leq p(\pi^{-1}(m))$$

para todo $m \in M$. Escolha seu $j \in U$ favorito e comece uma nova iteração com $\pi[j \mapsto k]$ no lugar de π .

Antes de prosseguirmos, gostaria de chamar a atenção do leitor para uma proposição que, embora seja importante, é tão trivial que caberia no rodapé desta

² Um ótimo matemático, sem dúvida. Mas quem o viu fazendo malabares sabe que ali estava o verdadeiro talento.

página.

Proposição 3.1.1. Se (J, M, p) é uma instância de um problema de escalonamento em máquinas idênticas, então $\text{opt}(J, M, p) \geq \frac{1}{|M|}p(J)$

Prova. Seja $\pi: J \rightarrow M$ um escalonamento parcial qualquer. Note que,

$$p(J) = \sum_{m \in M} p(\pi^{-1}(m)) \leq \sum_{m \in M} \max\{p(\pi^{-1}(m')) \mid m' \in M\} = \sum_{m \in M} \nu_p(\pi),$$

onde

$$\nu_p(\pi) \geq \frac{1}{|M|}p(J).$$

O resultado segue da arbitrariedade de π . \square

O Teorema 3.1.1 prova a correção do Algoritmo 3 mostrando que ele se enquadra como um algoritmo de aproximação. Um **algoritmo de aproximação** é um algoritmo que produz, em tempo polinomial uma solução “quase” ótima. Por “quase”, entenda que conseguimos demonstrar que o valor da função objetivo desta solução é limitado por um fator ϵ do valor da solução ótima. Uma solução com essa característica é dita uma ϵ -aproximação³. No nosso contexto, o teorema afirma que o Algoritmo 3 produz uma 2-aproximação, isso é, um escalonamento π tal que $\nu_p(\pi) \leq 2 \cdot \text{opt}(J, M, p)$.

Teorema 3.1.1. O algoritmo graham produz uma 2-aproximação.

Prova. Seja (J, M, p) uma instância de um problema de escalonamento em máquinas idênticas submetido ao algoritmo graham. Vamos demonstrar que a seguinte propriedade é uma invariante do algoritmo graham

$$\nu_p(\pi) \leq 2 \cdot \text{opt}(J, M, p). \quad (3.3)$$

Note que a função vazia satisfaz trivialmente esta propriedade.

Suponha que $\pi: J \rightarrow M$ é um escalonamento parcial no início de uma iteração arbitrária do algoritmo que satisfaz 3.3. Se ocorre o Caso 1, por hipótese, o resultado é evidente. Suponha então que ocorre o Caso 2, isto é, $U \neq \emptyset$. Provaremos que o escalonamento parcial $\pi[j \mapsto k]$ produzido ao final desta iteração também satisfaz 3.3. Antes, pelo bem do leitor, encurtaremos a notação para a ocupação de uma máquina, escrevendo a partir de agora $\varrho_m(\varphi)$ no lugar de $p(\varphi^{-1}(m))$ para todo $m \in M$ e todo $\varphi: J \rightarrow M$.

Provaremos primeiro que a ocupação da máquina k é limitada superiormente pelo ótimo. Como k é tal que $\varrho_k(\pi) \leq \varrho_m(\pi)$ para todo $m \in M$, então

$$|M|\varrho_k(\pi) \leq \sum_{m \in M} \varrho_m(\pi) \leq p(J),$$

³ Não nos aprofundaremos mais do que isso. Para mais detalhes, vide (VAZIRANI, 2002).

onde temos que $\varrho_k(\pi) \leq \frac{p(j)}{|M|}$. Isto, combinado com o resultado da Proposição 3.1.1, permite concluir que

$$\varrho_k(\pi) \leq \text{opt}(J, M, p). \quad (3.4)$$

Seja $\pi' = \pi[j \mapsto k]$. Note que $\varrho_k(\pi') = \varrho_k(\pi) + p(j)$. Segue daí que

$$\begin{aligned} \nu_p(\pi') &:= \max\{\varrho_m(\pi') \mid m \in M\} \\ &= \max(\{\varrho_m(\pi') \mid m \in M \setminus \{k\}\} \cup \{\varrho_k(\pi')\}) \\ &= \max(\{\varrho_m(\pi) \mid m \in M \setminus \{k\}\} \cup \{\varrho_k(\pi) + p(j)\}) \\ &\leq \max\{2 \cdot \text{opt}(J, M, p), \text{opt}(J, M, p) + p(j)\} \\ &\leq \max\{2 \cdot \text{opt}(J, M, p), 2 \cdot \text{opt}(J, M, p)\} \\ &= 2 \cdot \text{opt}(J, M, p), \end{aligned}$$

onde a primeira desigualdade decorre da hipótese de indução e de 3.4, e a segunda do fato de que $p(j) \leq \text{opt}(J, M, p)$. \square

O algoritmo de Graham descreve um modelo de algoritmo recorrente na literatura dos problemas de escalonamento, conhecido como **escalonamento por lista** (do inglês, list scheduling – LS). Uma versão bem conhecida deste algoritmo, chamada **duração mais longa** (do inglês, longest processing time – LPT), inclui uma regra de despacho em graham, escolhendo a cada iteração $j = \operatorname{argmax}\{p(u) \mid u \in U\}$. Esta ligeira modificação faz com que este algoritmo obtenha resultados melhores que graham, produzindo $\frac{4}{3}$ -aproximações (GRAHAM, 1969).

Encontrar soluções aproximadas é ótimo, mas encontrar soluções exatas é melhor ainda. A seguir, veremos que nem todos os problemas de escalonamento são NP-difíceis. Estudaremos um problema de escalonamento cujo ótimo pode ser encontrado em tempo polinomial.

3.2 Escalonamento com restrições de precedência em máquina única

O leitor pode se recordar da motivação apresentada para a definição de um digrafo acíclico na Seção 2.5, que sintetiza um problema de escalonamento com restrições de precedência em máquina única. Nesta seção, formalizaremos esse problema.

O problema de escalonamento com restrições de precedência em máquina única, como o nome sugere, difere do problema apresentado anteriormente tanto pela restrição de precedência entre as tarefas quanto pelo fato de que apenas uma máquina está disponível para sua execução.

Seja R um digrafo acíclico cujo conjunto de vértices, V_R , é chamado de conjunto das **tarefas**. O digrafo R , denominado **digrafo das restrições de precedência**, é tal que para todo par de tarefas distintas $j_1, j_2 \in V_R$, se existe um j_1j_2 -caminho em R , então a tarefa j_1 deve ser executada antes de j_2 . Além disso, seja $p: V_R \rightarrow \mathbb{R}_{++}$ uma função que associa a cada tarefa $j \in V_R$ sua respectiva **duração**, $p(j)$.

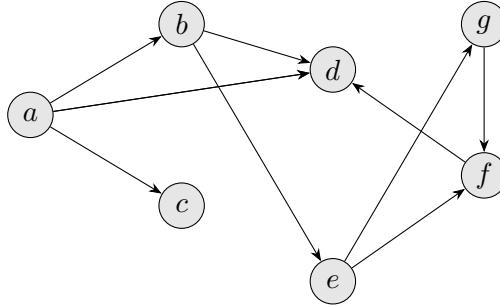


Figura 10 – Digrafo acíclico representando as restrições de precedência.

Um escalonamento parcial consiste na escolha de uma ordem de execução de algumas tarefas que seja compatível com as restrições de precedência. Formalmente, um **escalonamento parcial** é uma ordenação acíclica parcial π de tarefas em R . Quando $|\pi| = |V_R|$, isso é, quando π for uma ordenação acíclica, dizemos que π é simplesmente um **escalonamento**.

É bem verdade que qualquer ordenação acíclica do digrafo R descreve uma ordem de execução válida. No entanto, existe um número não exponencial delas (em relação a $|V_R|$) no caso geral, o que torna a busca por uma ordenação com alguma característica específica um problema intratável. Contudo, se incluirmos penalidades associadas ao instante de conclusão das tarefas, tornamos esse problema manejável. Isso indica que soluções que sofram as menores penalidades possíveis exigem um maior cuidado na escolha da ordem de execução, mas nos dão uma direção para seguir.

Seja π um escalonamento parcial. O instante de conclusão de π é dado por

$$c(\pi) := \sum_{i=0}^{|\pi|-1} p(\pi_i).$$

Por consequência, se $j \in \underline{\pi}$ é uma tarefa escalonada, o instante de conclusão de j em π é $c(\pi_{..j})$. A **penalidade** $f: V_R \rightarrow (\mathbb{R}_+ \rightarrow \mathbb{R}_{++})$ é uma função que, dada uma tarefa $j \in V_R$, devolve uma função monótona $f_j: \mathbb{R}_+ \rightarrow \mathbb{R}_{++}$ associada a j . Informalmente, espera-se de f_j que quanto maior for o instante de conclusão $c \in \mathbb{R}_{++}$ de uma tarefa j , maior deve ser a penalidade $f_j(c)$. Finalmente, o **custo** $w(\pi)$ de um escalonamento parcial π é, por definição

$$w(\pi) := \max\{f_j(c(\pi_{..j})) \mid j \in \pi\}.$$

Um problema de escalonamento com restrições de precedência em máquina única, descrito por uma tripla (R, p, f) , tem como objetivo determinar um escalonamento π de custo mínimo. Como de costume, um escalonamento com essa característica é dito **ótimo**.

Exemplo 3.2 – Escalonamento de 7 tarefas com restrições de precedência

Seja

$$R = (\{a, b, c, d, e, f, g\}, \{ab, ac, ad, bd, be, ef, eg, gf, fd\})$$

um digrafo acíclico — ilustrado na Figura 10 — que representa a restrição de precedência de um conjunto de tarefas. Admita que a duração $d(j)$ de cada tarefa $j \in V_R$ é dada conforme a função

$$d := \{(a, 4), (b, 6), (c, 3), (d, 2), (e, 1), (f, 2), (g, 5)\}$$

e que as tarefas estão sujeitas às funções monótonas de penalidade

$$\begin{aligned} \ell_a(x) &:= x & \ell_b(x) &:= x^2 & \ell_c(x) &:= x^3 \\ \ell_d(x) &:= 100 & \ell_e(x) &:= 0 & \ell_f(x) &:= \frac{x}{2} \\ \ell_g(x) &:= x^2 + 1. \end{aligned}$$

para todo $x \in \mathbb{R}_+$.

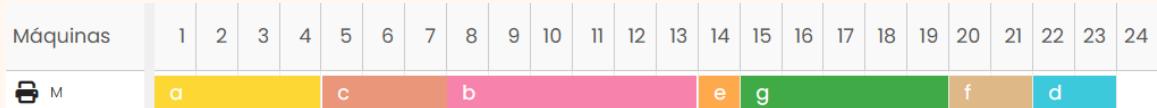


Figura 11 – Diagrama Gantt de um escalonamento ótimo com restrições de precedência.

A ordenação acíclica $\pi = \langle a, c, b, e, g, f, d \rangle$ é um *escalonamento* para uma instância (P, d, ℓ) , cujo custo é a penalidade $\ell_c(7) = 34$, por terminar a tarefa c no instante $c(\pi_{..c}) = 7$.

A Figura 11 ilustra o escalonamento π apresentado no Exemplo 3.2 através de um diagrama Gantt. A seguir, veremos o algoritmo responsável pela construção de π , e descobriremos que π é, sobretudo, um *escalonamento ótimo*, pois não há outro escalonamento sob as mesmas restrições com custo menor.

3.2.1 Algoritmo de Lawler

Antes de enunciar o algoritmo, uma observação. Por uma conveniência na descrição do algoritmo e na demonstração de sua correção, admitiremos, apenas neste ponto do trabalho, a existência do digrafo (\emptyset, \emptyset) . Este abuso calculado servirá como condição de parada do algoritmo. Em virtude disso, também assumiremos que $G - V_G$ é uma operação válida para qualquer digrafo G neste contexto. Isto posto, eis o algoritmo guloso de Lawler, capaz de produzir um escalonamento ótimo em tempo polinomial.

Algoritmo 4 – Algoritmo lawler para construção de um escalonamento com precedências em máquina única

O algoritmo recebe o problema de escalonamento (R, p, f) e devolve uma ordenação acíclica que representa um escalonamento ótimo.

Admita que cada iteração o algoritmo mantém um par (D, π) , sendo D um subdigrafo acíclico de R e π uma ordenação parcial acíclica de R . A primeira iteração começa com (R, ϵ) . Em uma iteração arbitrária do algoritmo, há dois casos a considerar:

Caso 1: $V_D = \emptyset$.

Neste caso, devolva π e encerre o algoritmo.

Caso 2: $V_D \neq \emptyset$.

Tome

$$j = \operatorname{argmin}\{f_u(p(V_D)) \mid u \text{ é um sorvedouro em } D\},$$

onde $p(V_D)$ é a soma da duração das tarefas em D .

Inicie uma nova iteração com $(D - j, j\pi)$ no lugar de (D, π) .

A correção deste algoritmo é verificada pelo Teorema 3.2.1.

Teorema 3.2.1. *O algoritmo lawler produz uma solução ótima.*

Prova. Seja $L = (R, p, f)$ a instância do problema de escalonamento submetida a lawler. Diremos que uma ordenação parcial acíclica β é **promissora** se existe uma sequência de tarefas, digamos α , tal que $\alpha\beta$ é um escalonamento ótimo de L . Para demonstrar a correção do algoritmo, devemos mostrar que no início de cada iteração, π é uma ordenação parcial acíclica promissora.

Considere uma iteração arbitrária de lawler. Suponha que π é uma ordenação parcial acíclica promissora. Note que se o Caso 1 ocorre, o resultado segue trivialmente

da hipótese. Suponha então que $V_D \neq \emptyset$ para que o Caso 2 ocorra e seja j a tarefa escolhida de forma gulosa pelo algoritmo. Mostraremos que $j\pi$ também é promissora.

Por hipótese, existe uma sequência, digamos σ e um vértice u tal que $\pi^* = \sigma u \pi$ é um escalonamento ótimo para L . Note que, se $u = j$ não há nada a provar; então suponha o contrário. Neste caso, existem sequências de vértices, digamos α, β , tais que $\pi^* = \alpha j \beta u \pi$ é ótimo. Note que a escolha gulosa de j implica que $f_j(c(\pi_{..u})) \leq f_u(c(\pi_{..u}))$. O Lema 3.2.1 garante que executar j depois de u é possível e não aumenta o custo do escalonamento, isso é, $\pi' = \alpha \beta u j \pi$ também é uma ordenação acíclica de R , cujo custo $w(\pi') \leq w(\pi^*)$. Segue daí que o prefixo $\alpha \beta u$ é testemunha de que $j\pi$ é uma ordenação parcial acíclica promissora. \square

Lema 3.2.1. *Seja (R, p, f) uma instância de um problema de escalonamento com restrições de precedência em máquina única. Seja $D \subseteq R$ um subdigrafo acíclico, e sejam $u, j \in V_D$ duas tarefas distintas, com j sendo um sorvedouro. Se $\pi = \alpha j \gamma u \beta$ é uma ordenação acíclica de R , então*

- (i) $\alpha \gamma u j \beta$ é uma ordenação acíclica de R ; e
- (ii) $f_j(c(\pi_{..u})) \leq f_u(c(\pi_{..u}))$ implica em $w(\alpha \gamma u j \beta) \leq w(\pi)$.

Prova. Suponha que $\pi = \alpha j \gamma u \beta$ é uma ordenação acíclica de R e seja $\pi' = \alpha \gamma u j \beta$. Primeiro, mostraremos que π' também é uma ordenação acíclica de R .

Note que $\underline{\beta} = V_R - V_D$, pois π é uma ordenação acíclica e $u, j \in V_D$. Como j é um sorvedouro em D , então qualquer sucessor de j em R deve ser um termo de β e, portanto, u não é sucessor de j . Esse fato, somado à aciclicidade de β , implica que π' é uma ordenação acíclica de R , como desejado.

Agora suponha que $f_j(c(\pi_{..u})) \leq f_u(c(\pi_{..u}))$. Mostraremos que $w(\pi') \leq w(\pi)$. Para tal, note que:

1. $f_u(c(\pi'_{..u})) \leq f_u(c(\pi_{..u}))$, pois $|\pi'_{..u}| \leq |\pi_{..u}|$;
2. $f_j(c(\pi'_{..j})) \leq f_u(c(\pi_{..u}))$, uma vez que $c(\pi'_{..j}) = c(\pi_{..u})$; e
3. se $x \in V_D \setminus \{u, j\}$, então $c(\pi'_{..x}) \leq c(\pi_{..x})$, donde $f_x(c(\pi'_{..x})) \leq f_x(c(\pi_{..x}))$.

Segue daí que

$$\begin{aligned} w(\pi) &= \max\{f_x(c(\pi_{..x})) \mid x \in V_D\} \\ &= \max(\{f_x(c(\pi_{..x})) \mid x \in V_D \setminus \{u, j\}\} \cup \{f_u(c(\pi_{..u})), f_j(c(\pi_{..j}))\}) \\ &\geq \max(\{f_x(c(\pi'_{..x})) \mid x \in V_D \setminus \{u, j\}\} \cup \{f_u(c(\pi'_{..u})), f_j(c(\pi'_{..j}))\}) \\ &= w(\pi'). \end{aligned}$$

\square

3.2.2 Antimatróides

Um aspecto interessante da demonstração do Teorema 3.2.1 é que ela decorre naturalmente do fato de que lawler é um caso particular da caracterização algorítmica gulosa de um *antimatróide* (BOYD; FAIGLE, 1990). Antimatróides são estruturas combinatorias abstratas relacionadas a ordens parciais e fechamentos de conjuntos. Formalmente, um **antimatróide** é um par (E, \mathcal{L}) , onde E é um conjunto finito de elementos, e $\mathcal{L} \subseteq E^*$ é um subconjunto não vazio de sequências finitas sobre E que satisfaz as seguintes propriedades:

- (i) para todo $\alpha \in E^*$ e todo $x \in E$, se $\alpha x \in \mathcal{L}$ então $\alpha \in \mathcal{L}$; e
- (ii) para todo $\alpha, \beta \in E^*$, se $\alpha, \beta \in \mathcal{L}$ e $\underline{\alpha} \not\subseteq \underline{\beta}$ então existe $a \in \underline{\alpha}$ tal que $\beta a \in \mathcal{L}$.

Considere a noção de ordenação parcial promissora, mencionada na demonstração do Teorema 3.2.1. Se \mathcal{L} for o conjunto de todas essas sequências em ordem reversa, a relação entre a propriedade (ii) e o argumento da demonstração fica evidente⁴. O Algoritmo 5 generaliza a propriedade descrita pelo Teorema 3.2.1 para antimatróides.

Algoritmo 5 – Algoritmo greedy para construção de soluções ótimas em antimatróides

O algoritmo recebe um antimatróide (E, \mathcal{L}) e uma função $w: \mathcal{L} \rightarrow \mathbb{R}$, e devolve uma sequência em \mathcal{L} que minimiza w .

Cada iteração do algoritmo começa com uma sequência σ . A primeira iteração começa com $\sigma = \epsilon$.

Caso 1: $E \setminus \underline{\sigma} = \emptyset$.

Neste caso, devolva σ e encerre o algoritmo.

Caso 2: $E \setminus \underline{\sigma} \neq \emptyset$.

Tome $s \in E \setminus \underline{\sigma}$ tal que:

- (i) $\sigma s \in \mathcal{L}$, e
- (ii) $w(\sigma s) \leq w(\sigma z)$ para todo $\sigma z \in \mathcal{L}$.

Inicie uma nova iteração com σs no lugar de σ .

⁴ Caso não esteja convencido, escreva $u\beta j\alpha$ no lugar de α , π no lugar de β e j no lugar de a e veja a matemágica acontecer.

Continuaremos abordando problemas de escalonamento no próximo capítulo, dando o devido destaque ao problema de interesse deste trabalho.

4 Escalonamento na indústria gráfica

O processo de planejamento de produção na indústria gráfica se inicia quando o profissional do setor de planejamento e controle de produção (PCP) recebe uma **ordem de serviço** (OS) com as especificações do produto a ser produzido. Com base nessas especificações, o profissional sequencia os processos produtivos necessários e os distribui na programação das máquinas disponíveis em cada **estágio produtivo** correspondente.

Os processos nesta indústria devem ser executados em sequências específicas, que variam conforme os acabamentos solicitados pelo cliente. Por exemplo, a produção das cartelas de um álbum de figurinhas adesivas envolve as seguintes etapas:

1. Impressão das figuras em uma impressora;
2. Aplicação de uma camada de lamination para proteção e brilho;
3. Corte e vinco em uma máquina de corte para destacar as figuras;
4. Etapa de embalagem, realizada manualmente ou em uma máquina apropriada.

No caso do álbum que receberia a colagem dos adesivos, os processos empregados seriam diferentes, determinados pelas especificações definidas junto ao cliente. Essa característica classifica o problema como uma variante do problema de **escalonamento orientado a trabalho**, mais especificamente o **escalonamento flexível orientado a trabalho**, onde cada trabalho deve passar por diferentes estágios de produção e cada estágio pode ter mais de uma **máquina** operando em paralelo (por exemplo, um estágio de impressão com impressoras digitais e offset funcionando simultaneamente).

O papel do profissional de PCP é realizar o escalonamento da produção, respeitando a ordem de processamento, as capacidades de armazenamento da empresa, a programação das máquinas e a disponibilidade de matéria-prima em estoque. Além disso, outros fatores podem influenciar sua tomada de decisão, como os prazos acordados com os clientes, que definem uma noção de **prioridade** para cada OS e, consequentemente, a ordem de processamento em cada máquina.

O tempo necessário para a conclusão de cada processo é estimado com base nas quantidades envolvidas e na velocidade da máquina. Por exemplo, uma impressora capaz de imprimir 500 folhas por minuto deve levar 1 hora para concluir a impressão de uma OS com 30.000 folhas. A **conclusão** de um trabalho ocorre no instante em que o último processo é finalizado. Assim, o objetivo do setor de PCP é organizar a produção

de modo a **minimizar** o instante de conclusão dos trabalhos programados, garantindo que sejam entregues dentro dos prazos estabelecidos.

A seguir, neste capítulo, formalizaremos este problema de escalonamento. Começando pela sua versão mais simples, na Seção 4.1, descrevemos os seus componentes, seguido de uma formulação como um programa linear. Em sequência, abordamos métodos heurísticos e apresentamos uma formulação baseada em digrafos muito conhecida na literatura. Após o leitor adquirir familiaridade com o problema e suas especificidades, abordamos a versão de interesse na Seção 4.2.

4.1 Escalonamento orientado a trabalho

O **escalonamento orientado a trabalho** (do inglês, job shop scheduling problem – JSSP) é um problema mais complexo que os anteriores e que depende de muitos ingredientes. Formalmente, uma instância do JSSP é uma 8-upla

$$\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p),$$

onde

- J é um conjunto finito e não-vazio, cujos elementos são denominados de **trabalhos**, e que representam os trabalhos que devem ser escalonados;
- O é um conjunto finito e não vazio, cujos elementos são denominados de **operações**, e que constituem as etapas para a conclusão de um trabalho; e
- M é um conjunto finito e não-vazio, cujos elementos são denominados de **máquinas**, e que constituem as máquinas disponíveis para processar as operações;
- $r : J \rightarrow \mathbb{N}$ é a função **instante de liberação** que indica que um trabalho $j \in J$ pode ser executado em qualquer momento m , desde que $m \geq r_j$;
- $n : J \rightarrow \mathbb{N}_+$ é uma função que estabelece o número de operações que formam um trabalho;
- $\mu : O \rightarrow M$ é a função **alocação de máquina** que estabelece para cada operação $o \in O$, a máquina $\mu(o)$ responsável pela execução de o ;
- $p : O \rightarrow \mathbb{R}_{++}$ é uma função que estabelece a **duração** $p(o) > 0$ de uma operação $o \in O$; e
- $\pi : J \rightarrow [n_j] \rightarrow O$ é uma função que associa cada $j \in J$ a uma sequência finita de operações π_j , chamada de **rota de processamento de j** , tal que:

1. π_j é injetora, ou seja, as operações que formam um trabalho j são duas a duas distintas; e
2. $\pi_j \cap \pi_k = \emptyset$ para cada $j \neq k \in J$, ou seja, uma operação pode fazer parte de no máximo um trabalho.

Considerando as rotas de processamento, por uma questão de conveniência, vamos extrair dois subconjuntos muito úteis do conjunto das operações: o conjunto das **operações iniciais** $O_{\perp} := \{\pi_j(0) \mid j \in J\}$ e o conjunto das **operações terminais** $O_{\top} := \{\pi_j(n_j - 1) \mid j \in J\}$. Seus respectivos complementos em O , o conjunto das **operações não iniciais** $O \setminus O_{\perp}$ e o das **operações não terminais** $O \setminus O_{\top}$, serão denotados por $\overline{O_{\perp}}$ e $\overline{O_{\top}}$, respectivamente. Seja

$$\sigma := \{(\pi_j(i), \pi_j(i + 1)) \in O \times O \mid i \in [n_j - 2]\}.$$

um conjunto de pares de operações adjacentes em cada sequência. Note que $\sigma: \overline{O_{\top}} \rightarrow O$, bem como $\sigma^{-1}: \overline{O_{\perp}} \rightarrow O$. Assim, para cada operação não terminal $o \in \overline{O_{\top}}$, $\sigma(o)$ é a operação que a sucede e, analogamente, $\sigma^{-1}(o)$ é a operação que precede cada operação não inicial $o \in \overline{O_{\perp}}$.

Exemplo 4.1 – Instância do JSSP com 10 operações e 4 máquinas.

O exemplo a seguir é baseado em uma instância descrita por [Brucker e Knust \(2011\)](#). Seja $J = \{j_1, j_2, j_3, j_4\}$, $O = \{o_i \mid 1 \leq i \leq 10\}$ e $M = \{m_1, m_2, m_3, m_4\}$. Admita que o número de operações n_j , o instante de liberação r_j e a rota de processamento π_j de cada trabalho $j \in J$ estão descritos na Tabela 1. Além disso, a alocação de máquina μ_o e a respectiva duração p_o de cada operação $o \in O$ constam na Tabela 2.

j	n_j	r_j	π_j
j_1	3	5	$\langle o_1, o_2, o_3 \rangle$
j_2	2	3	$\langle o_4, o_5 \rangle$
j_3	3	2	$\langle o_6, o_7, o_8 \rangle$
j_4	2	0	$\langle o_9, o_{10} \rangle$

Tabela 1 – Número de operações, instante de liberação e rotas de processamento.

o	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9	o_{10}
μ_o	m_1	m_4	m_2	m_4	m_2	m_1	m_4	m_2	m_4	m_3
p_o	2	2	1	2	7	4	5	2	3	7

Tabela 2 – Alocação de máquina e os respectivos tempos de processamento.

Podemos, agora, introduzir a noção de uma solução viável para o problema do escalonamento, que é uma função que associa um instante de início para cada

operação. Informalmente, tal função deve satisfazer as seguintes condições, capturadas formalmente pelas condições (4.1), (4.2), e (4.3) a seguir: (i) a operação inicial de cada trabalho só pode ser executada após a liberação do trabalho; (ii) cada operação não inicial só pode ser iniciada após o término da sua predecessora; e (iii) duas operações alocadas em uma mesma máquina não podem ser executadas simultaneamente.

É conveniente introduzir o conjunto Γ cujos elementos são os pares de operações que são executadas numa mesma máquina, ou seja,

$$\Gamma := \{(o, o') \in O \times O \mid o \neq o', \mu_o = \mu_{o'}\}.$$

Observe que Γ é *simétrico*: se $(o, o') \in \Gamma$, então $(o', o) \in \Gamma$.

Dizemos que uma função $s: O \rightarrow \mathbb{R}_+$, chamada de **instante de início**, é um **escalonamento viável** para \mathfrak{J} se as restrições

$$s_{\pi_j(0)} \geq r_j \text{ para todo } j \in J; \quad (4.1)$$

$$s_o + p_o \leq s_{\sigma(o)} \text{ para todo } o \in \overline{O_T}; \text{ e} \quad (4.2)$$

$$s_o + p_o \leq s_{o'} \text{ ou } s_{o'} + p_{o'} \leq s_o \text{ para cada } (o, o') \in \Gamma. \quad (4.3)$$

são satisfeitas. O termo $s(o) + p(o)$, que será referido como **instante de conclusão (término)** de uma operação $o \in O$, surge de uma hipótese que vale a pena mencionar: admitimos que a disponibilidade para a execução de uma operação é irrestrita, isto é, as máquinas ociosas estão sempre disponíveis e, uma vez iniciada a execução, não há interrupções. Vejamos, a seguir, uma proposição sobre escalonamentos viáveis.

Proposição 4.1.1. *Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ uma instância do JSSP. Para todo $(o, o') \in \Gamma$, se $s: O \rightarrow \mathbb{R}_+$ é um escalonamento viável para \mathfrak{J} , então exatamente uma das condições de (4.3) é satisfeita.*

Prova. Suponha que s é um escalonamento viável para \mathfrak{J} e que $(o, o') \in \Gamma$. Como $p: O \rightarrow \mathbb{R}_{++}$, não é possível que ambas as condições $s_o + p_o \leq s_{o'}$ e $s_{o'} + p_{o'} \leq s_o$ em (4.3) se cumpram simultaneamente. De fato, se esse fosse o caso, então

$$s_o < s_o + p_o \leq s_{o'} < s_{o'} + p_{o'} \leq s_o$$

o que é uma contradição. \square

Finalmente, seja s um escalonamento viável. O **makespan** de s é o número

$$C_{max}(s) := \max\{s(o) + p(o) \mid o \in O\}$$

(ou apenas C_{max} , quando o contexto permitir), que corresponde ao maior instante de conclusão de uma operação em s . Isto posto, o objetivo do problema de escalonamento orientado a trabalho consiste em determinar um escalonamento viável cujo makespan é mínimo, ou seja, encontrar a sequência de processamento que permita completar todos os trabalhos no menor instante possível.

Exemplo 4.2 – Solução viável para o JSSP

Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ a instância do JSSP enunciada no Exemplo 4.1. A função

$$s := \{(o_1, 6), (o_2, 11), (o_3, 14), (o_4, 3), (o_5, 5), (o_6, 2), (o_7, 6), (o_8, 12), (o_9, 0), (o_{10}, 3)\}$$

é um escalonamento viável para \mathfrak{J} . A Figura 12 ilustra o escalonamento s em um diagrama Gantt.

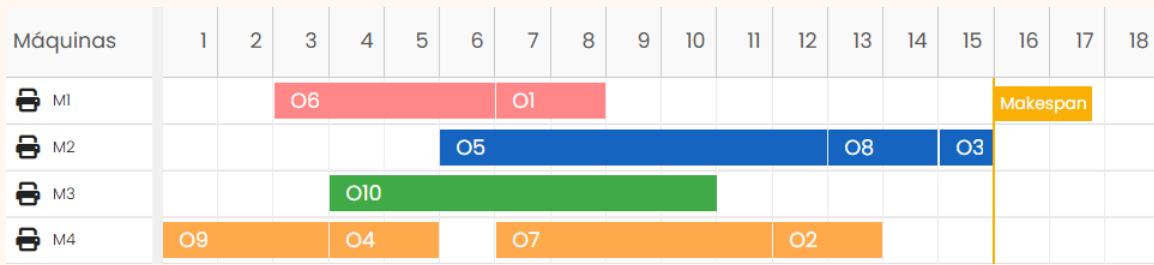


Figura 12 – Diagrama Gantt de uma solução para o JSSP.

Fonte: Adaptado de (BRUCKER; KNUST, 2011)

O *makespan* $C_{\max}(s)$ de s é $s(o_3) + p(o_3) = 15$.

4.1.1 Métodos exatos

Mostraremos agora como modelar o problema do escalonamento orientado a trabalho como um problema de programação inteira mista. Como um primeiro passo, considere o seguinte problema, que equivale ao problema de encontrar um escalonamento viável cujo *makespan* é mínimo:

$$\begin{aligned}
 & \min \quad \lambda \\
 \text{sujeito a} \quad & s_{\pi_j(0)} - r_j \geq 0 \text{ para todo } j \in J \\
 & s_o + p_o - s_{\sigma(o)} \leq 0 \text{ para todo } o \in \overline{O_T} \\
 & s_o + p_o \leq s_{o'} \text{ ou } s_{o'} + p_{o'} \leq s_o \text{ para cada } (o, o') \in \Gamma \\
 & s(o) + p(o) - \lambda \leq 0 \text{ para cada } o \in O \\
 & \lambda \in \mathbb{R}_+.
 \end{aligned} \tag{4.4}$$

Note que uma solução ótima, s, λ , para o problema acima satisfaz:

$$s_o \leq \max\{r_j \mid j \in J\} + p(O)$$

para cada $o \in O$. Para modelar a restrição (4.3), definimos

$$\alpha := 2(\max\{r_j \mid j \in J\} + p(O)).$$

Introduzimos, para cada par de operações $(o, o') \in \Gamma$, variáveis binárias de ordem:

$$y_{(o,o')} \in \{0, 1\},$$

onde:

- $y_{(o,o')} = 1$ indica que a operação o é processada antes de o' ,
- $y_{(o',o)} = 1$ indica que a operação o' é processada antes de o .

Para garantir que apenas uma das alternativas seja válida, adicionamos a restrição:

$$y_{(o,o')} + y_{(o',o)} = 1 \quad \text{para cada } (o, o') \in \Gamma.$$

A formulação completa é então dada por:

$$\begin{aligned} \min \quad & \lambda \\ \text{sujeito a} \quad & s_{\pi_j(0)} - r_j \geq 0 \text{ para todo } j \in J \\ & s_o + p_o - s_{\sigma(o)} \leq 0 \text{ para todo } o \in \overline{O_T} \\ & s_o + p_o - s_{o'} - \alpha(1 - y_{(o,o')}) \leq 0 \text{ para cada } (o, o') \in \Gamma \\ & s_{o'} + p_{o'} - s_o - \alpha(1 - y_{(o',o)}) \leq 0 \text{ para cada } (o, o') \in \Gamma \\ & s(o) + p(o) - \lambda \leq 0 \text{ para cada } o \in O \\ & y_{(o,o')} \in \{0, 1\} \text{ para cada } (o, o') \in \Gamma \\ & y_{(o,o')} + y_{(o',o)} = 1 \text{ para cada } (o, o') \in \Gamma \\ & \lambda \in \mathbb{R}_+. \end{aligned} \tag{4.5}$$

Considere uma solução viável s, y, λ para o programa inteiro misto. Seja $(o, o') \in \Gamma$ e suponha, sem perda de generalidade, que $y_{(o,o')} = 1$ e $y_{(o',o)} = 0$. Então, $s_o + p_o \leq s_{o'}$ e $s_{o'} + p_{o'} - s_o \leq \max\{r_j \mid j \in J\} + p(O) + p_{o'} \leq \alpha$.

Esta formulação do problema pode ser submetida a um solver¹ para obter soluções exatas para este problema. Algumas instâncias de pequeno e médio porte podem ser resolvidas até mesmo por softwares gratuitos como o IBM ILOG CPLEX COMMUNITY EDITION. No entanto, na prática, essa abordagem se mostra inviável, uma vez que o JSSP é um problema NP-difícil (SOTSKOV; SHAKLEVICH, 1995). Embora inviável para instâncias de grande porte (DAUZÈRE-PÉRÈS et al., 2024), modelos de programas inteiros mistos são especialmente úteis na obtenção de limitantes, que ajudam a avaliar a qualidade de algoritmos heurísticos. Abordaremos esse tópico na seção a seguir.

¹ Software dedicado à resolução de problemas matemáticos.

4.1.2 Heurísticas construtivas

Podemos implementar facilmente um algoritmo de escalonamento por lista para construção de soluções viáveis em tempo polinomial. Essa heurística, chamada **máquina menos carregada** (do inglês, *least loaded machine* — LLM), pode ser vista como uma adequação do Algoritmo 3 para o contexto do JSSP. Antes de enunciá-lo, faz-se necessária a noção de *escalonamento parcial*, que veremos agora.

Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ uma instância do JSSP. Diremos que uma função parcial $z: O \rightarrow \mathbb{R}_+$ é um **escalonamento parcial** de \mathfrak{J} se existe um escalonamento viável s para \mathfrak{J} tal que $z = s|_{Dom(z)}$. Ademais, dado um escalonamento parcial z de \mathfrak{J} , para cada $j \in J$ e cada $i \in [n_j]$, diremos que a operação $\pi_j(i) \in O \setminus Dom(z)$ está **liberada** em z se toda operação predecessora está no domínio do escalonamento parcial, isso é, $\{\pi_j(k) \in O \mid 0 \leq k < i\} \subseteq Dom(z)$.

Também definimos para cada máquina $m \in M$, e cada escalonamento parcial $z: O \rightarrow \mathbb{R}_+$ a **carga de trabalho** $W_m(z)$ da máquina m em z como sendo o conjunto das operações alocadas em m que já foram escalonadas por z , isto é,

$$W_m(z) := \{o \in Dom(z) \mid \mu_o = m\}.$$

Baseado nisso, definiremos a noção de ocupação de máquina. Informalmente, a ocupação de uma máquina em um determinado escalonamento corresponde ao tempo total em que uma máquina está ocupada, seja aguardando para iniciar a execução de uma operação, seja de fato executando-a. Formalmente, a **ocupação** $c_m(z)$ de uma máquina $m \in M$ no escalonamento parcial z corresponde ao maior instante de término dentre as operações na carga de trabalho $W_m(z)$, isso é,

$$c_m(z) := \max\{s(o) + p(o) \mid o \in W_m(z)\}.$$

Eis o Algoritmo LLM-JSSP, capaz de construir iterativamente escalonamentos viáveis em tempo polinomial simplesmente escolhendo as operações liberadas da máquina menos ocupada disponível.

Algoritmo 6 – Algoritmo LLM-JSSP para construção de um escalonamento viável

O algoritmo recebe uma instância $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ do JSSP e, como resultado, produz um escalonamento viável para \mathfrak{J} .

Cada iteração começa com um escalonamento parcial $s: O \rightarrow \mathbb{R}_+$ e um conjunto $L \subseteq O$ de operações liberadas em s . A primeira iteração começa com $s := \emptyset$ e $L := \{\pi_j(0) \mid j \in J\}$. Cada iteração consiste no seguinte:

Caso 1: $L = \emptyset$.

Neste caso, devolva s e encerre o algoritmo.

Caso 2: $L \neq \emptyset$.

Seja $M' := \{\mu_o \mid o \in L\}$ o conjunto das máquinas alocadas para as operações liberadas. Tome a máquina $k \in M'$ menos ocupada, isso é, tal que

$$c_k(s) \leq c_m(s)$$

para todo $m \in M'$. Seja também $O_k := \{o \in L \mid \mu_o = k\}$ o subconjunto das operações liberadas que estão alocadas na máquina k . Escolha uma operação $o \in O_k$ que te deixe contente e considere o número

$$\delta_o := \begin{cases} r_j & \text{se } \exists j \in J \text{ tal que } o = \pi_j(0) \\ s(\sigma^{-1}(o)) + p(\sigma^{-1}(o)) & \text{caso contrário.} \end{cases}$$

Seja^a:

$$L' := \begin{cases} L - o & \text{se } o \in O_{\top} \\ (L - o) + \sigma(o) & \text{caso contrário.} \end{cases}$$

Comece uma nova iteração com $s[o \mapsto \max\{\delta_o, c_k(s)\}]$ no lugar de s e L' no lugar de L .

^a Lembre-se que, O_{\top} é o conjunto das operações terminais e, portanto, não tem uma sucessora. Lembre-se também das notações para remoção e união envolvendo conjuntos unitários (vide a Subseção 2.1.2)

4.1.3 Digrafo disjuntivo

Podemos enxergar uma instância do problema de escalonamento orientado a trabalho através de um digrafo ponderado, construído de tal forma que, encontrar um escalonamento viável que minimiza o *makespan* equivale ao problema de extrair deste digrafo um subdigrafo acíclico gerador que minimize os caminhos críticos.

Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ uma instância do JSSP. Construiremos um digrafo que codifica, em sua estrutura, as restrições (4.1), (4.2) e (4.3) do problema. Para tal, definimos os seguintes conjuntos:

- $\widehat{O} := \{\perp, \top\} \cup O$, denominado **conjunto estendido das operações**. Neste conjunto, além das operações em O , temos as operações fictícias \perp, \top ;
- $C := C_\perp \cup \sigma \cup C_\top$, denominado conjunto das **conjunções**. Ele será formado pela união dos conjuntos:
 1. $C_\perp := \{\perp\} \times O_\perp$ de pares de associando a operação fictícia \perp a operação inicial de cada trabalho. Esses objetos serão utilizados para garantir que o digrafo possuirá exatamente uma fonte e para representar a restrição (4.1) do problema;
 2. σ de pares entre as operações e suas sucessoras. Esse conjunto de pares descreve a restrição (4.2) do problema;
 3. $C_\top := O_\top \times \{\top\}$ de pares entre as operações terminais e a operação fictícia \top . Esses objetos garantirão que o digrafo possui exatamente um sorvedouro.
- $D := \Gamma$, denominado conjunto das **disjunções**, que é essencialmente o conjunto Γ de pares de operações executadas em uma mesma máquina. Disjunções têm como função descrever a restrição (4.3) do problema². Vamos sempre admitir que os conjuntos D e C são disjuntos, embora isso não imponha nenhuma limitação.

Finalmente, o **digrafo disjuntivo** sobre \mathfrak{J} , ilustrado na Figura 13, é a 4-upla

$$\mathfrak{D} = (\widehat{O}, C, D, w),$$

onde $(\widehat{O}, C \cup D)$ é um digrafo e $w: C \cup D \rightarrow \mathbb{R}_+$ é uma **função peso** que associa cada arco $o_1 o_2 \in C \cup D$ a um real não negativo

$$w(o_1 o_2) := \begin{cases} r(j) & \text{se existe } j \in J \text{ tal que } (o_1, o_2) = (\perp, \pi_j(0)) \\ p(o_1) & \text{se } o_1 o_2 \in \sigma \cup C_\top \\ 0 & \text{caso contrário.} \end{cases}$$

² Note que a simetria de Γ implica na existência de arcos antiparalelos, e, por consequência, \mathfrak{D} tem ciclos.

Exemplo 4.3 – Digrafo disjuntivo de uma instância do JSSP

Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ a instância do JSSP enunciada no Exemplo 4.1. A Figura 13 ilustra um digrafo disjuntivo \mathfrak{D} sobre \mathfrak{J} .

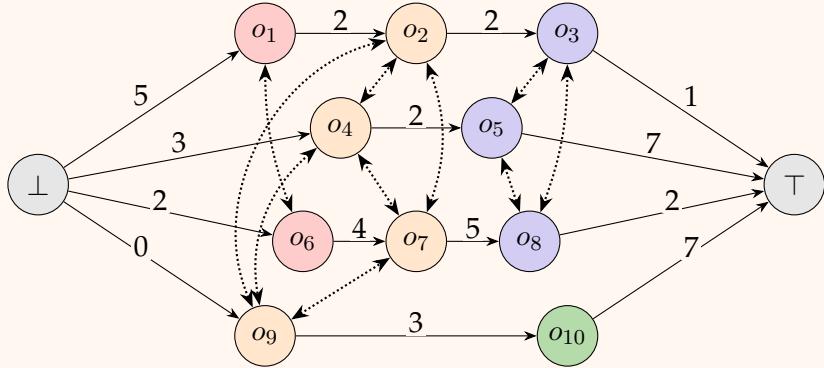


Figura 13 – Digrafo disjuntivo para um problema de escalonamento orientado a trabalho.

Fonte: Adaptado de (BRUCKER; KNUST, 2011)

As setas preenchidas representam os arcos conjuntivos e as setas tracejadas representam pares antiparalelos de arcos disjuntivos. Vértices pintados com as mesmas cores representam operações alocadas na mesma máquina. Acima de cada arco, consta o valor da função peso w , exceto nos arcos disjuntivos, cujo peso é sempre 0.

O modelo acima sugere uma associação entre o custo de caminhos críticos no digrafo disjuntivo e o instante de conclusão das operações terminais dos caminhos. Logo, como o digrafo possui exatamente um sorvedouro, o custo de um caminho crítico até T está diretamente relacionado ao *makespan* do escalonamento. Essa associação se torna mais evidente quando colocamos luz sobre uma certa impossibilidade: do ponto de vista do digrafo disjuntivo, a noção de caminho crítico não está bem definida, devido à presença de ciclos causados pelas disjunções. Por outro lado, a Proposição 4.1.1 aponta: uma solução viável não pode satisfazer simultaneamente as duas condições da restrição (4.3) do problema. Isto indica que o sucesso dessa abordagem depende da nossa capacidade de transformar o digrafo disjuntivo em um digrafo acíclico, uma vez que destruir arcos antiparalelos significa escolher qual das condições da restrição (4.3) vamos preservar.

Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ uma instância do JSSP e $\mathfrak{D} = (\widehat{O}, C, D, w)$ um digrafo disjuntivo sobre \mathfrak{J} . Uma **seleção** em \mathfrak{D} é um subconjunto não vazio $S \subset D$ de

arcos disjuntivos de \mathfrak{D} tal que

$$\text{se } o_1o_2 \in S, \text{ então } o_2o_1 \notin S. \quad (4.6)$$

Uma seleção S induz a 4-upla $\mathfrak{D}_S = (\widehat{O}, C, S, \gamma)$, chamada de **digrafo conjuntivo induzido por S** , onde $(\widehat{O}, C \cup S)$ é um digrafo, ponderado pela função $\gamma: C \cup S \rightarrow \mathbb{R}_+$, tal que

$$\gamma(o_1o_2) := \begin{cases} w(o_1o_2) & \text{se } o_1o_2 \in C \\ p(o_1) & \text{se } o_1o_2 \in S, \end{cases}$$

para todo $o_1o_2 \in C \cup S$. Note que C e S são conjuntos disjuntos.

Uma seleção S é **completa** se S é maximal, e **consistente** se o digrafo conjuntivo \mathfrak{D}_S é acíclico. Note que afirmar que uma seleção S é completa simplesmente quer dizer que exatamente um dentre os arcos o_1o_2 e o_2o_1 está em S para cada $(o_1, o_2) \in \Gamma$. Por fim, diremos que o **custo** da seleção S é o custo $\delta_{\mathfrak{D}_S}(\top)$ de um caminho crítico até \top em \mathfrak{D}_S .

Exemplo 4.4 – Seleção completa e consistente e um digrafo conjuntivo para uma instância do JSSP

Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, p)$ a instância do JSSP enunciada no Exemplo 4.1. Seja também $\mathfrak{D} = (\widehat{O}, C, D, w)$ o digrafo disjuntivo abordado no Exemplo 4.3.

A seleção completa e consistente

$$S = \{o_6o_1, o_5o_3, o_5o_8, o_8o_3, o_9o_4, o_4o_2, o_7o_2, o_4o_7, o_9o_7, o_9o_2\}$$

induz o digrafo conjuntivo $\mathfrak{D}_S = (\widehat{O}, C, S, w)$, ilustrado na Figura 14.

O caminho $\rho = \langle \perp, o_9, o_4, o_5, o_8, o_3, \top \rangle$ em \mathfrak{D}_S , cujos arcos estão destacados em vermelho, é crítico até \top . Em virtude disso, o custo da seleção S é $\delta_{\mathfrak{D}_S} = 15$.

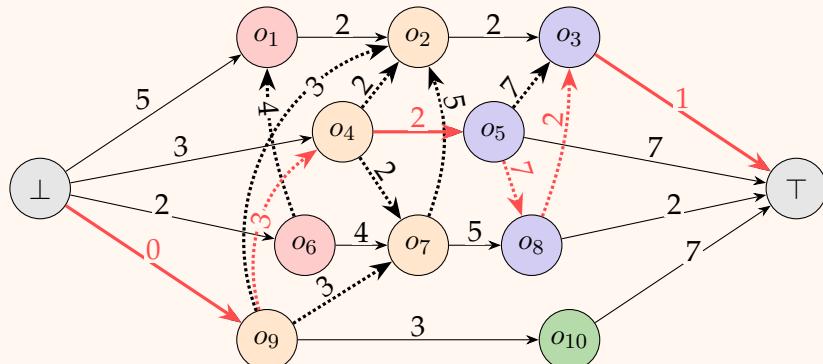


Figura 14 – Digrafo conjuntivo induzido por uma seleção completa e consistente.

Fonte: Adaptado de (BRUCKER; KNUST, 2011)

Vamos adotar a seguinte convenção. Suponha que s é um escalonamento viável para uma instância \mathfrak{J} do JSSP. Então,

$$s(\top) := C_{\max}(s). \quad (4.7)$$

O leitor concordará comigo que isso é razoável, uma vez que o instante de término de um dos predecessores críticos de \top em \mathfrak{D}_S é exatamente o $C_{\max}(s)$. Também assumiremos que $s(\perp) = 0$.

Proposição 4.1.2. *Se s é uma solução viável para uma instância \mathfrak{J} do JSSP, então existe uma seleção completa e consistente S de arcos em um digrafo disjuntivo $\mathfrak{D} = (\widehat{O}, C, D, w)$ sobre \mathfrak{J} tal que $C_{\max}(s) \geq \delta_{\mathfrak{D}_S}(\top)$.*

Prova. Seja s uma solução viável para uma instância \mathfrak{J} do JSSP e

$$S := \{uv \in \Gamma \mid s(v) \geq s(u) + p(u)\}.$$

Observe que S é uma seleção completa. De fato, pela Proposição 4.1.1, $uv \in S$ implica que $vu \notin S$ e, portanto, S é uma seleção. Ademais, $|\{uv, vu\} \cap \Gamma| \geq 1$ para cada $uv \in \Gamma$ e, assim, S é uma seleção completa.

Seja $\mathfrak{D} = (\widehat{O}, C, D, w)$ um digrafo disjuntivo sobre \mathfrak{J} . Vamos mostrar que \mathfrak{D}_S é acíclico. Suponha, por contradição, que \mathfrak{D}_S possui um ciclo $\langle u_0, u_1, \dots, u_k \rangle$. Nesse caso, $u_0 = u_k$ e $k \geq 2$. É claro que, pela definição de \mathfrak{D}_S , $u_0 \neq \perp$ e $u_k \neq \top$. Observe que $p \in O \rightarrow \mathbb{R}_{++}$. Assim, $s(u_0) = s(u_k) \geq s(u_{k-1}) + p(u_{k-1}) > s(u_{k-1}) > s(u_0)$. Essa contradição mostra que \mathfrak{D}_S é acíclico. Concluímos, assim, que S é uma seleção consistente e maximal.

Vamos agora provar que $C_{\max}(s) \geq \delta_{\mathfrak{D}_S}(\top)$. Para isso, é suficiente estabelecer que

$$C_{\max}(s) \geq s(u) + \gamma(\rho) \quad (4.8)$$

para todo vértice $u \in \widehat{O}$ e todo caminho ρ de u até \top em \mathfrak{D}_S . A prova é por indução no comprimento de ρ . Seja ρ um caminho de um vértice $u \in \widehat{O}$ até \top . Suponha, primeiro, que $|\rho| = 0$. Nesse caso, $u = \top$ e $\gamma(\rho) = 0$, donde $C_{\max}(s) = s(\top) + \gamma(\rho)$. Suponha, agora, que $|\rho| \geq 1$. Seja ρ' um caminho até \top tal que $\rho = u\rho'$. Seja v o início de ρ' . Por hipótese de indução, $C_{\max} \geq s(v) + \gamma(\rho')$. Temos que lidar com dois casos:

Caso 1: O arco uv está em C_{\perp} .

Nesse caso, $u = \perp$ e $\gamma(uv) = r_j$, onde $j \in J$ é tal que $v \in \underline{\pi}_j$. Observe que, como s é viável, então $s(v) \geq r_j = s(\perp) + r_j$. Agora,

$$\begin{aligned} C_{\max}(s) &\geq s(v) + \gamma(\rho') && \text{por hipótese de indução} \\ &\geq s(\perp) + r_j + \gamma(\rho') && \text{pois } s \text{ é viável} \\ &= s(\perp) + \gamma(\rho) && \text{pela definição de } \rho. \end{aligned}$$

Caso 2: O arco uv está em $\sigma \cup S \cup C_{\top}$.

Nesse caso,

$$\begin{aligned} C_{\max}(s) &\geq s(v) + \gamma(\rho') && \text{por hipótese de indução} \\ &\geq s(u) + p(u) + \gamma(\rho') && \text{pois } s \text{ é viável} \\ &= s(u) + \gamma(\rho) && \text{pela definição de } \rho. \end{aligned}$$

Isso completa a prova de (4.8)

Seja ρ um caminho crítico até \top em \mathfrak{D}_S . Note que para cada vértice u de \mathfrak{D}_S existe um caminho de u até \top . Isso, combinado com o fato de que γ não assume valores negativos, permite admitir que o início de ρ é o vértice \perp . Assim, de acordo com (4.8), temos que $C_{\max}(s) \geq s(\perp) + \gamma(\rho) = 0 + \delta_{\mathfrak{D}_S}(\top) = \delta_{\mathfrak{D}_S}(\top)$, como queríamos. \square

Proposição 4.1.3. *Seja $\mathfrak{D} = (\widehat{O}, C, D, w)$ um digrafo disjuntivo construído sobre uma instância \mathfrak{J} do JSSP e S uma seleção de arcos disjuntivos de \mathfrak{D} . Se S é completa e consistente, então existe um escalonamento viável s para \mathfrak{J} tal que $C_{\max}(s) \leq \delta_{\mathfrak{D}_S}(\top)$.*

Prova. Suponha que S é completa e consistente. Para abreviar, vamos escrever δ_S no lugar de $\delta_{\mathfrak{D}_S}$. Vamos mostrar que existe um escalonamento viável s para \mathfrak{J} tal que $C_{\max}(s) = \delta_S(\top)$. O digrafo conjuntivo $\mathfrak{D}_S = (\widehat{O}, C, S, \gamma)$ é acíclico (pois S é consistente por hipótese), assim, pelo Teorema 2.5.2,

$$\delta_S(v) = \max\{\delta_S(u) + \gamma(uv) \mid u \in N_{\mathfrak{D}_S}^-(v)\}$$

para todo vértice $v \in \widehat{O}$.

Vamos mostrar que $s = \delta_{S|O}$ é um escalonamento viável tal que $C_{\max}(s) \leq \delta_S(\top)$. Primeiro, vamos verificar esta última desigualdade. Note que $O = \overline{O_{\top}} \cup O_{\top}$. Além disso, como \top é um sorvedouro de \mathfrak{D}_S , $\gamma : C \cup S \rightarrow \mathbb{R}_{++}$ e para cada $u \in \widehat{O}$ existe um caminho de u até \top , então, pelo Corolário 2.5.1, temos que $\delta_S(\top) \geq \delta_S(u)$ para cada $u \in \widehat{O}$. Ademais, $\delta_S(o) + \gamma(o\top) \leq \delta_S(\top)$ para cada $o \in O_{\top}$. Assim,

$$\begin{aligned} C_{\max}(s) &= \max\{s(o) + p(o) \mid o \in O\} \\ &= \max(\{\delta_S(o) + \gamma(o\sigma_o) \mid o \in \overline{O_{\top}}\} \cup \{\delta_S(o) + \gamma(o\top) \mid o \in O_{\top}\}) \\ &\leq \max(\{\delta_S(\sigma_o) \mid o \in \overline{O_{\top}}\} \cup \{\delta_S(o) + \gamma(o\top) \mid o \in O_{\top}\}) \\ &\leq \delta_S(\top). \end{aligned}$$

Vamos, agora, à prova de que s satisfaz as restrições (4.1), (4.2), (4.3).

Por construção, temos que $o\sigma_o$ é um arco de \mathfrak{D}_S para toda operação não terminal $o \in \overline{O_{\top}}$. Assim,

$$s(\sigma(o)) = \delta(\sigma(o)) \geq \delta(o) + \gamma(o\sigma_o) = s(o) + p(o)$$

para cada $o \in \overline{O_{\top}}$ e, portanto, s satisfaz a restrição (4.2).

Já a restrição (4.1) é facilmente verificada observando-se se $j \in J$, então $\perp\pi_j(0)$ é um arco de \mathfrak{D}_S e, portanto,

$$s(\pi_j(0)) = \delta(\pi_j(0)) \geq \delta(\perp) + \gamma(\perp\pi_j(0)) = r_j.$$

Finalmente, para a restrição (4.3), verificaremos que $s(o_1) \geq s(o_2) + p(o_2)$ ou $s(o_2) \geq s(o_1) + p(o_1)$ para todo $(o_1, o_2) \in \Gamma$. Para tal, note que, devido à completude de S , temos que $o_1 o_2 \in S$ ou $o_2 o_1 \in S$, mas não ambas. Suponha, sem perda de generalidade, $o_1 o_2 \in S$. Então,

$$s(o_2) = \delta(o_2) \geq \delta(o_1) + \gamma(o_1 o_2) = s(o_1) + p(o_1),$$

como queríamos. \square

Teorema 4.1.1. *Seja \mathfrak{J} uma instância do JSSP e \mathfrak{D} um digrafo disjuntivo de \mathfrak{J} . Se \mathbb{S} é o conjunto de todas as soluções viáveis para \mathfrak{J} e \mathcal{S} é o conjunto de todas as seleções completas e consistentes em \mathfrak{D} , então*

$$\min_{s \in \mathbb{S}} C_{\max}(s) = \min_{S \in \mathcal{S}} \delta_{\mathfrak{D}_S}(\top)$$

Prova. Suponha que \mathbb{S} é o conjunto de todas as soluções viáveis para \mathfrak{J} e \mathcal{S} é o conjunto de todas as seleções completas e consistentes em \mathfrak{D} .

Seja $s \in \mathbb{S}$. Pela Proposição 4.1.2, existe uma seleção completa e consistente $S \in \mathcal{S}$ tal que $C_{\max}(s) \geq \delta_{\mathfrak{D}_S}(\top)$, donde

$$\min_{s \in \mathbb{S}} C_{\max}(s) \geq \min_{S \in \mathcal{S}} \delta_{\mathfrak{D}_S}(\top).$$

Seja agora $S \in \mathcal{S}$. Pela Proposição 4.1.3, existe $s \in \mathbb{S}$ tal que $C_{\max}(s) \leq \delta_{\mathfrak{D}_S}(\top)$, donde

$$\min_{s \in \mathbb{S}} C_{\max}(s) \leq \min_{S \in \mathcal{S}} \delta_{\mathfrak{D}_S}(\top).$$

\square

O resultado do Teorema 4.1.1 garante que o problema de encontrar um escalonamento viável s mínimo para uma instância \mathfrak{J} do JSSP é redutível ao problema de extrair uma seleção S completa, consistente e de custo mínimo do digrafo disjuntivo (G, w) associado a \mathfrak{J} .

4.2 Escalonamento flexível orientado a trabalho

Generalizando o problema descrito anteriormente, o **escalonamento flexível orientado a trabalho** (do inglês, flexible job shop scheduling problem – FJSSP) surge da

seguinte premissa: as operações deixam de estar associadas a máquinas e passam a estar associadas a um estágio produtivo. Um estágio consiste em um conjunto de máquinas, não necessariamente idênticas, capazes de processar operações com características similares, como tipos específicos de processamento ou requisitos técnicos. Os trabalhos precisam ser processados em uma sequência específica de estágios produtivos, sendo que o tempo gasto em cada estágio depende diretamente da máquina alocada para realizar a operação.

Portanto, resolver um escalonamento flexível envolve a solução de dois subproblemas:

1. **Alocação:** processo de selecionar qual máquina, em cada estágio produtivo, será responsável por processar as operações de cada trabalho;
2. **Escalonamento:** processo de determinar o instante de início do processamento de cada operação de cada trabalho.

De forma muito similar ao que foi discutido na Seção 4.1, formalmente uma instância do FJSSP é uma 8-upla

$$\mathfrak{F} = (J, O, M, r, n, \pi, \Phi, \varrho)$$

onde J, O, M, r, n e π são exatamente os mesmos objetos apresentados anteriormente. Literalmente copiando a seção anterior, temos que

- J é um conjunto finito e não-vazio, cujos elementos são denominados de **trabalhos**, e que representam os trabalhos que devem ser escalonados;
- O é um conjunto finito e não vazio, cujos elementos são denominados de **operações**, e que constituem as etapas para a conclusão de um trabalho; e
- M é um conjunto finito e não-vazio, cujos elementos são denominados de **máquinas**, e que constituem as máquinas disponíveis para processar as operações;
- $r : J \rightarrow \mathbb{N}$ é a função **instante de liberação** que indica que um trabalho $j \in J$ pode ser executado em qualquer momento m , desde que $m \geq r_j$;
- $n : J \rightarrow \mathbb{N}_+$ é uma função que estabelece o número de operações que formam um trabalho;
- $p : O \rightarrow \mathbb{R}_{++}$ é uma função que estabelece a **duração** $p(o) > 0$ de uma operação $o \in O$; e
- $\pi : J \rightarrow [n_j] \rightarrow O$ é uma função que associa cada $j \in J$ a uma sequência finita de operações π_j , chamada de **rota de processamento de j** , tal que:

1. π_j é injetora, ou seja, as operações que formam um trabalho j são duas a duas distintas; e
2. $\underline{\pi}_j \cap \underline{\pi}_k = \emptyset$ para cada $j \neq k \in J$, ou seja, uma operação pode fazer parte de no máximo um trabalho.

A diferença entre os problemas reside nos objetos Φ e ϱ , cuja definição a seguir reflete a flexibilidade admitida. Como já mencionado, as operações podem ser executadas em qualquer máquina dentro de um determinado conjunto de máquinas, de acordo com os requisitos tecnológicos que envolvem sua execução. Em razão disso, a função

$$\Phi: O \rightarrow 2^M \setminus \{\emptyset\}$$

define um conjunto $\Phi(o) \subseteq M$ de **máquinas compatíveis** com cada operação $o \in O$.

Admitindo que as máquinas compatíveis com uma operação podem não ser idênticas, ajustamos a noção de duração de uma operação. Dada uma operação $o \in O$, a função $\varrho: O \rightarrow (M \rightarrow \mathbb{R}_{++})$ devolve uma função parcial $\varrho_o: M \rightarrow \mathbb{R}_{++}$, definida sobre as combinações de operações e máquinas compatíveis, cuja imagem corresponde a **duração** $\varrho_o(m)$ da execução de o em uma máquina $m \in \Phi(o)$.

Usaremos livremente no contexto do FJSSP alguns objetos definidos na seção anterior para o JSSP. São eles: os conjuntos O_{\perp} e O_{\top} , respectivamente, de **operações iniciais** e **terminais**; os complementos de O_{\perp} e O_{\top} em O , que representam os conjuntos $\overline{O_{\perp}}$ das **operações não iniciais** e $\overline{O_{\top}}$ das **operações não terminais**; e a função $\sigma: \overline{O_{\top}} \rightarrow O$ que devolve a **operação sucessora** de uma operação não terminal, bem como $\sigma^{-1}: \overline{O_{\perp}} \rightarrow O$ que devolve a **operação predecessora** de uma operação não inicial.

Exemplo 4.5 – Instância do FJSSP com 4 operações e 2 máquinas.

O exemplo a seguir é baseado na instância SFJS2 de Fattahi et al. (2007). Seja $J = \{j_1, j_2\}$, $O = \{o_1, o_2, o_3, o_4\}$ e $M = \{m_1, m_2\}$. O número de operações n_j , o instante de liberação r_j e a rota de processamento π_j de cada trabalho $j \in J$ estão descritos na Tabela 3. A duração $\varrho_o(m)$ de cada operação $o \in O$ em cada máquina $m \in M$ consta^a na Tabela 4.

j	n_j	r_j	π_j
j_1	2	0	$\langle o_1, o_2 \rangle$
j_2	2	15	$\langle o_3, o_4 \rangle$

Tabela 3 – Número de operações, instante de liberação e rotas de processamento.

o/m	m_1	m_2
o_1	43	∞
o_2	64	71
o_3	21	35
o_4	∞	43

Tabela 4 – Duração de cada operação em cada máquina.

^a Assumimos a totalidade de ϱ denotando $\varrho_o(m) = \infty$ se $m \notin \Phi(o)$

A noção de uma solução viável para o FJSSP deve levar em conta a solução dos subproblemas de alocação e escalonamento mencionados acima. Informalmente, a solução do subproblema de alocação deve ser tal que cada operação esteja alocada em exatamente uma máquina compatível. Já o subproblema de escalonamento é essencialmente o mesmo problema abordado pelo JSSP: encontrar um instante de início para cada operação, evitando conflitos de máquina e respeitando as rotas de processamento.

Analogamente ao que fizemos com o JSSP, introduzimos o conjunto Ψ de triplas que representam a possibilidade de duas operações distintas estarem alocadas em uma mesma máquina:

$$\Psi := \{(o, m, o') \in O \times M \times O \mid m \in \Phi(o) \cap \Phi(o')\} \setminus \{(o, m, o) \mid o \in O, m \in M\}.$$

Uma **solução** para uma instância \mathfrak{F} do FJSSP consiste em um par (μ, s) . A função **alocação de máquina** $\mu: O \rightarrow M$, é uma solução para o subproblema de alocação, determinando para cada operação $o \in O$, qual máquina $m \in M$ será reservada à sua execução. Por sua vez, a função $s: O \rightarrow \mathbb{R}_+$ denominada **instante de início**, soluciona o subproblema de escalonamento, associando cada operação $o \in O$ ao instante s_o , que determina quando sua execução deve iniciar.

Dizemos que (μ, s) é uma **solução viável** para \mathfrak{F} se satisfaz

$$\mu_o \in \Phi(o) \text{ para cada } o \in O; \text{ e} \quad (4.9)$$

$$s_{\pi_j(0)} \geq r_j \text{ para todo } j \in J; \quad (4.10)$$

$$s_o + \varrho_o(\mu_o) \leq s_{\sigma(o)} \text{ para todo } o \in \overline{O_T}; \text{ e} \quad (4.11)$$

$$\mu_o = \mu_{o'} = m \implies s_o + \varrho_o(m) \leq s_{o'} \text{ ou } s_{o'} + \varrho_{o'}(m) \leq s_o \text{ para cada } (o, m, o') \in \Psi. \quad (4.12)$$

O objetivo do FJSSP, tal qual no JSSP, é encontrar uma solução viável (μ, s) que minimize o *makespan* do escalonamento³

$$C_{\max}(\mu, s) := \max\{s(o) + \varrho_o(\mu(o)) \mid o \in O\}.$$

Exemplo 4.6 – Solução viável para o FJSSP

Seja $\mathfrak{F} = (J, O, M, r, n, \pi, \Phi, \varrho)$ a instância do JSSP enunciada no Exemplo 4.5. As funções

$$\mu := \{(o_1, m_1), (o_2, m_1), (o_3, m_2), (o_4, m_2)\} \quad \text{e} \quad s := \{(o_1, 0), (o_2, 43), (o_3, 15), (o_4, 50)\}$$

dão uma solução viável para \mathfrak{F} , cujo *makespan* é $C_{\max}(\mu, s) = s(o_2) + \varrho_{o_2}(m_1) = 107$. A Figura 15 ilustra o diagrama Gantt que representa (μ, s) .

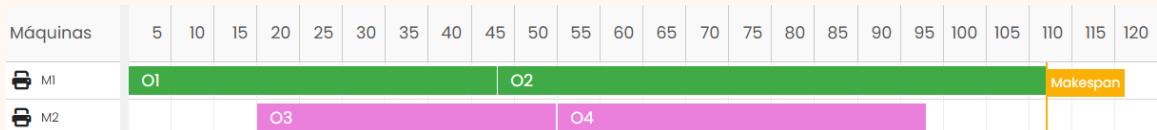


Figura 15 – Diagrama Gantt de uma solução para o FJSSP.

É claro que o FJSSP é um problema tão difícil de resolver quanto o JSSP, já que se trata de uma generalização. Isso implica que como o JSSP é um problema NP-difícil, o FJSSP também é. Além do mais, não abordaremos o modelo de programação inteira mista para o FJSSP, uma vez que ele é muito parecido com o que foi apresentado na seção anterior e apresenta as mesmas limitações. Vejamos a partir de agora abordagens heurísticas para o problema.

4.2.1 Digrafo disjuntivo flexível

Vimos na Seção 4.1.3 que o modelo de digrafo disjuntivo do JSSP pode ser uma estrutura combinatória útil no projeto de algoritmos construtivos para resolver o

³ Lembre-se que, assim como no JSSP, $s(o) + \varrho_o(\mu(o))$ é o instante de conclusão da operação o no escalonamento (μ, s) .

problema. Contudo, no decorrer do desenvolvimento deste trabalho, essa abordagem não se mostrou muito vantajosa. Existem inúmeras generalizações do digrafo disjuntivo para o FJSSP. A título de completude, o Exemplo 4.7 descreve de forma sucinta o modelo de [Rossi e Dini \(2007\)](#), cuja implementação realizada neste trabalho, como veremos mais adiante no Capítulo 6, se mostrou infrutífera.

Exemplo 4.7 – Digrafo disjuntivo flexível

O digrafo disjuntivo flexível de [Rossi e Dini \(2007\)](#) é algo parecido com este animal ilustrado na Figura 16. Assim como sua contraparte para o JSSP — vide Seção 4.1 — o conjunto dos vértices é composto pelas operações de cada trabalho e as operações fictícias \perp e \top . Da mesma forma, são colocados arcos conjuntivos (setas pretas na figura) entre \perp e as operações iniciais; entre as operações terminais e \top ; e entre cada operação e sua sucessora. Já os arcos disjuntivos (setas pontilhadas coloridas), além de conectar cada par de operações simultaneamente compatíveis a uma mesma máquina (ou seja, existe um arco para cada $(o, m, o') \in \Psi$), também conecta \perp e \top a cada operação $o \in O$, tantas vezes quanto $|\Phi(o)|$.

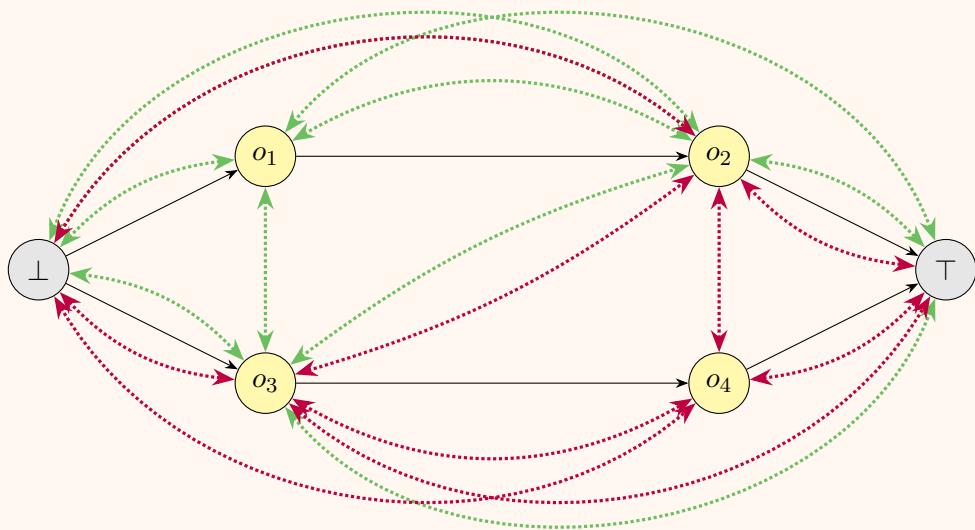


Figura 16 – Digrafo disjuntivo flexível para a instância enunciada no Exemplo 4.5.

4.2.2 Construindo soluções viáveis

As abordagens que resolvem primeiramente o subproblema de alocação e depois o subproblema de escalonamento são denominadas **hierárquicas**. Segundo [Brucker e Knust \(2011\)](#), ao se obter uma alocação μ para uma instância $\mathfrak{F} = (J, O, M, r, n, \pi, \Phi, \varrho)$ do

FJSSP, é possível resolver o segundo subproblema definindo uma função $\varphi: O \rightarrow \mathbb{R}_{++}$, pondo $\varphi(o) = \varrho_o(\mu(o))$ para cada operação $o \in O$ e submetendo ao problema original. Em outras palavras, se s é uma solução viável para a instância $\mathfrak{J} = (J, O, M, r, n, \pi, \mu, \varphi)$ do JSSP, então (μ, s) é uma solução viável para \mathfrak{J} .

Por outro lado, Genova et al. (2015) argumentam que, apesar de as abordagens hierárquicas apresentarem uma redução na complexidade de implementação por resolverem um subproblema de cada vez, as abordagens ditas **não hierárquicas** (ou **integradas**) frequentemente produzem resultados superiores. Essa diferença de desempenho pode ser explicada pelo fato de que a fixação prévia de uma alocação de máquinas restringe o espaço de busca, tornando boas soluções inacessíveis durante o processo de escalonamento.

O Algoritmo 7 apresenta uma adaptação da heurística LLM (Algoritmo 6) para admitir a flexibilidade de máquinas. O leitor perceberá que se trata de uma abordagem não hierárquica, uma vez que os dois subproblemas são resolvidos simultaneamente. Para descrevê-lo adequadamente, como de costume, apresentaremos a seguir as noções de solução parcial e ocupação de máquina no contexto do FJSSP.

Seja $\mathfrak{J} = (J, O, M, r, n, \pi, \Phi, \varrho)$ uma instância do FJSSP. Considere também as funções parciais $a: O \rightarrow M$ e $z: O \rightarrow \mathbb{R}_+$. Diremos que (a, z) é uma **solução parcial** para \mathfrak{J} se existe uma solução viável (μ, s) para \mathfrak{J} tal que $a = \mu|_{Dom(a)}$ e $z = s|_{Dom(z)}$. Dada uma solução parcial (a, z) para \mathfrak{J} , para todo $j \in J$ e todo $i \in [n_j]$, diremos que a operação $\pi_j(i) \in O \setminus Dom(a) \cap Dom(z)$ está **liberada** em (a, z) se toda operação que precede $\pi_j(i)$ foi alocada e escalonada, isto é, $\{\pi_j(k) \in O \mid 0 \leq k < i\} \subseteq Dom(a) \cap Dom(z)$.

Seja (a, z) uma solução parcial para \mathfrak{J} . A **carga de trabalho** $W_m(a, z)$ de uma máquina $m \in M$ em (a, z) é

$$W_m(a, z) := \{o \in Dom(a) \cap Dom(z) \mid a(o) = m\},$$

para todo $m \in M$. Para encurtar a notação, escreveremos $C(o)$ para denotar o instante de término $s(o) + \varrho(o, \mu(o))$, de uma operação $o \in Dom(s) \cap Dom(\mu)$. Assim, definimos para toda máquina $m \in M$, a **ocupação** de m na solução parcial (a, z) como o maior instante de término dentre as operações alocadas em m

$$c_m(a, z) := \max\{C(o) \mid o \in W_m(a, z)\}.$$

Eis o Algoritmo LLM-FJSSP para construção de uma solução viável para o FJSSP em tempo polinomial. Sua estratégia consiste em estender uma solução parcial a cada iteração, escolhendo dentre as operações liberadas aquela que causa o menor acréscimo na ocupação de uma máquina.

Algoritmo 7 – Algoritmo LLM-FJSSP para construção de uma solução viável

O algoritmo recebe uma instância $\mathfrak{F} = (J, O, M, r, n, \pi, \Phi, \varrho)$ do FJSSP e devolve uma solução viável para \mathfrak{F} .

Cada iteração começa com uma solução parcial (μ, s) e um conjunto $L \subseteq O$ de operações liberadas em (μ, s) . A primeira iteração começa com uma solução vazia (\emptyset, \emptyset) e $L := \{\pi_j(0) \mid j \in J\}$. Em uma iteração arbitrária, há dois casos a considerar:

Caso 1: $L = \emptyset$.

Neste caso, devolva (μ, s) e encerre o algoritmo.

Caso 2: $L \neq \emptyset$.

Seja $A = \{(o, m) \in O \times M \mid o \in L, m \in \Phi(o)\}$ o conjunto que representa as alocações de máquina para as operações liberadas. Tome uma alocação $(i, k) \in A$ que cause o menor impacto em (μ, s) , isso é, tal que

$$c_k(\mu, s) + \varrho_i(k) \leq c_m(\mu, s) + \varrho_o(m)$$

para todo $(o, m) \in A$.

Seja^a $j \in J$ o trabalho associado a i (isso é, tal que $i \in \underline{\pi_j}$) e considere o número

$$\delta_i := \begin{cases} r_j & \text{se } i \in O_{\perp} \\ C(\sigma^{-1}(i)) & \text{caso contrário.} \end{cases}$$

Tome também o conjunto:

$$L' := \begin{cases} L - v & \text{se } u \notin \overline{O_{\top}} \\ (L - v) + \sigma(v) & \text{caso contrário.} \end{cases}$$

Finalmente, comece uma nova iteração com L' no lugar de L e (μ', s') no lugar de (μ, s) , onde $\mu' = \mu[i \mapsto k]$ e $s' = s[i \mapsto \max\{\delta_i, c_k(\mu, s)\}]$.

^a Lembre-se que, $\underline{\pi_j}$ é o conjunto das operações de um determinado $j \in J$; O_{\perp} é o conjunto das operações iniciais; e $\overline{O_{\top}}$ é o conjunto das operações não terminais.

Peço licença ao leitor para um pequeno desvio de rota, pois o abuso de cafeína me levou a implementar uma variação deste algoritmo. Podemos modificar facilmente o Algoritmo 7 para se adequar a uma versão um pouco mais geral do FJSSP, que não será abordada com a devida profundidade por motivos óbvios. Admita que nos deparamos com uma situação em que, no lugar das rotas de processamento sequenciais π , a precedência entre as operações é dada por um digrafo acíclico $R = (O, P)$, onde

$P \subset O \times O$ é um conjunto de pares que representa a relação de precedência. O Algoritmo 8, inspirado no Algoritmo de Kahn, constrói uma solução viável para uma instância $\widehat{\mathfrak{F}} = (J, O, M, r, n, R, \Phi, \varrho)$ deste problema.

Algoritmo 8 – Algoritmo LLM-FJSSP2 para construção de uma solução viável para $\widehat{\mathfrak{F}}$

O algoritmo recebe uma instância $\widehat{\mathfrak{F}} = (J, O, M, r, n, R, \Phi, \varrho)$ do problema descrito e devolve uma solução viável para $\widehat{\mathfrak{F}}$.

Cada iteração começa com uma solução parcial (μ, s) para $\widehat{\mathfrak{F}}$ e uma função $\Upsilon: O \rightarrow \mathbb{N}$. Essencialmente, $\Upsilon(o)$ é um contador que indica a quantidade de operações predecessoras de uma operação $o \in O$ que ainda foram escalonadas em (μ, s) . A primeira iteração começa com (\emptyset, \emptyset) e $\Upsilon := \{(o, |N_R^-(o)|) \mid o \in O\}$.

Em uma iteração arbitrária, tome o conjunto ^a

$$L := \{o \in \text{Dom}(\Upsilon) \mid \Upsilon(o) = 0\} \setminus \text{Dom}(\mu) \cap \text{Dom}(s)$$

de operações liberadas em (μ, s) . Como de costume, há dois casos a considerar:

Caso 1: $L = \emptyset$.

Neste caso, devolva (μ, s) e encerre o algoritmo.

Caso 2: $L \neq \emptyset$.

Novamente, seja $A = \{(o, m) \in O \times M \mid o \in L, m \in \Phi(o)\}$. Tome uma alocação gulosa $(i, k) \in A$, isso é, tal que

$$c_k(\mu, s) + \varrho_i(k) \leq c_m(\mu, s) + \varrho_o(m)$$

para todo $(o, m) \in A$. Seja também $j \in J$ o trabalho associado a i e considere o número

$$\delta_i = \max(\{C(l) \mid l \in N_R^-(i)\} \cup \{r_j\}).$$

Comece uma nova iteração com $\Upsilon' = \Upsilon[N_R^+(i) \ni l \mapsto \Upsilon(i) - 1]$ no lugar de Υ e (μ', s') no lugar de (μ, s) , onde $\mu' = \mu[i \mapsto k]$ e $s' = s[i \mapsto \max\{\delta_i, c_k(\mu, s)\}]$.

^a É evidente, caro leitor, que calcular esse conjunto a cada iteração é uma escolha puramente didática. Idealmente, em uma implementação, mantemos esse conjunto através das iterações, assim como no Algoritmo 7.

5 Otimização por colônia de formigas

Otimização por colônia de formigas (do inglês, ant colony optimization – ACO) é uma técnica probabilística de otimização combinatória inspirada no comportamento de forrageamento¹ das formigas. O primeiro algoritmo ACO, proposto em 1992 por Marco Dorigo em sua tese de doutorado, foi aplicado ao problema do caixeiro-viajante. Mais tarde, a técnica foi descrita como uma *metaheurística*². Desde então, essa técnica tem sido amplamente desenvolvida e utilizada para encontrar soluções satisfatórias em diversos problemas de otimização combinatória.

De acordo com Dorigo et al. (2006), biólogos que se dedicaram a estudar o comportamento exploratório das formigas observaram que elas utilizam um mecanismo biológico chamado **feromônio** como ferramenta de comunicação. O feromônio é uma substância química produzida pelas formigas e que se deposita por onde elas passam, formando uma trilha. Outras formigas conseguem detectar o cheiro do feromônio ao seu redor e, ao escolher qual caminho seguir, tendem a optar por direções que apresentam alta concentração de feromônio. Outro aspecto importante é a evaporação do feromônio com o decorrer do tempo, que provoca o desaparecimento da trilha em regiões menos visitadas.

No experimento da ponte dupla ilustrado na Figura 17, uma colônia de formigas foi separada de uma fonte de alimento por dois caminhos de diferentes comprimentos. Inicialmente, as formigas exploraram livremente ambos os caminhos até encontrarem a fonte de comida. No entanto, com o passar do tempo, observou-se que a rota das formigas convergiu para o caminho mais curto entre a colônia e a fonte de alimento, impulsionada pela presença do feromônio deixado ao longo do percurso.

¹ Busca e exploração de recursos alimentares

² Uma *metaheurística* é um procedimento heurístico genérico aplicável a uma variedade de problemas distintos.

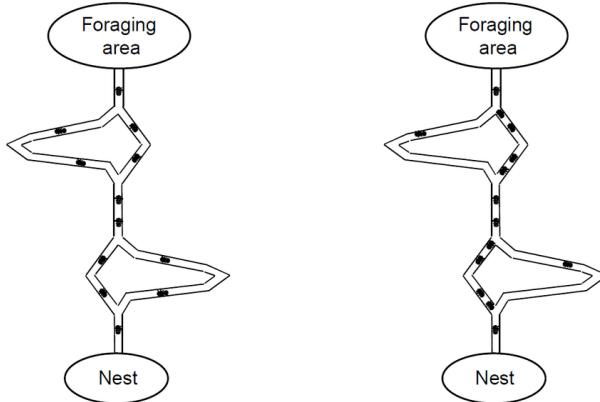


Figura 17 – Formigas distribuídas por todo o caminho (esq.), tendem a convergir para o caminho mais curto (dir.) devido a uma maior concentração de feromônio

Fonte: Retirado de ([DORIGO et al., 1999](#))

Inspirado por esse mecanismo biológico, o algoritmo **sistema de formigas** foi desenvolvido. Desde então, diversas variantes deste algoritmo vêm sendo discutidas na literatura. Neste trabalho, além do sistema de formigas, abordaremos teoricamente e apresentaremos implementações das variantes **sistema de formigas elitista** ([DORIGO et al., 1996](#)), **sistema de formigas com classificação** ([BULLNHEIMER et al., 1999](#)), **sistema de formigas MAX-MIN** ([STUTZLE; HOOS, 2000](#)) e **sistema de colônia de formigas** ([DORIGO; GAMBARDELLA, 1997](#)).

Neste capítulo, apresentaremos um modelo formal abstrato que captura as generalidades de um algoritmo ACO, detalhando o processo algorítmico subjacente. Em paralelo, exploraremos algumas de suas aplicações em problemas combinatórios clássicos. Contudo, primeiro precisamos formalizar o que de fato é um problema de otimização combinatória.

5.1 Problemas de otimização e estruturas de vizinhança

Uma descrição adequada para os algoritmos estudados neste trabalho envolve uma formalização igualmente adequada do que é um problema de otimização combinatória. No modelo que veremos a seguir, as soluções são representadas por sequências de elementos abstratos, denominados componentes de solução. O objetivo final de um problema de otimização é determinar, através do crivo de uma função objetivo, uma solução que satisfaça as restrições particulares do problema e minimize o valor da função objetivo estabelecida.

Formalizar um problema de otimização combinatória com o devido rigor, não é uma tarefa muito difícil. Tudo o que o caro leitor precisa fazer é definir três ingredientes fundamentais, que abordaremos abstratamente a seguir. Seja S um conjunto finito e não

vazio de elementos denominados de **componentes**, e $\Pi \subseteq S^*$, denominado **espaço de soluções**³, um conjunto finito, não vazio e fechado sob prefixos — isto é, se $\pi \in \Pi$ então todo prefixo π' de π também pertence a Π . Além disso, seja $c: \Pi \rightarrow \mathbb{R}$ uma função que associa a cada elemento π de Π um **custo**, $c(\pi)$. A tripla

$$(S, \Pi, c)$$

será carinhosamente chamada de **problema de otimização combinatória** (ou, de acordo com o princípio universal da preguiça, simplesmente um **problema de otimização**).

Devemos atenção especial a um subconjunto $\Sigma \subseteq \Pi$ formado pelas sequências que não são prefixo de nenhuma outra sequência distinta em Π . Em outras palavras, Σ contém exatamente as sequências $\sigma \in \Pi$ tais que, para todo $\pi \in \Pi$, se $\sigma \sqsubseteq \pi$, então $\sigma = \pi$. Chamaremos Σ de **conjunto das soluções viáveis**, pois informalmente ele representa as soluções completas do problema. Seu complemento em Π , $\bar{\Sigma} := \Pi \setminus \Sigma$, será chamado de **conjunto das soluções parciais**, contendo as sequências que ainda podem ser estendidas até uma solução viável.

O fato de Π ser fechado sob prefixos, induz à noção de estrutura de vizinhança. Formalmente, uma **estrutura de vizinhança para** $\mathfrak{P} = (S, \Pi, c)$ é uma função

$$\begin{aligned} N_{\mathfrak{P}}: \Pi &\rightarrow 2^S \\ \pi &\mapsto \{s \in S \mid \pi \cdot s \in \Pi\} \end{aligned} \tag{5.1}$$

que associa cada sequência $\pi \in \Pi$ a um subconjunto $N_{\mathfrak{P}}(\pi)$ de componentes de solução, denominado **vizinhança de** π , de elementos que permitem estender π para gerar um novo elemento do espaço de soluções. É claro que $N(\pi) = \emptyset$ se, e só se, $\pi \in \Sigma$.

Destacamos agora a importância da *função de estrutura de vizinhança* do ponto de vista algorítmico. Os algoritmos em que estamos interessados devem ter um consumo de tempo polinomial no tamanho do conjunto dos componentes. Do ponto de vista formal, é conveniente definir o espaço de soluções e, a partir dele, extrair o conjunto das soluções viáveis e das soluções parciais. No entanto, do ponto de vista **algorítmico** (considerando que buscamos algoritmos com consumo de tempo polinomial no tamanho do conjunto dos componentes), é evidente, devido ao tamanho do espaço de soluções para a classe de problemas em que estamos interessados, os famigerados NP-difícies, que um algoritmo *não pode tratar um problema de otimização combinatória na forma como definido acima*.

Os algoritmos que estudaremos têm uma natureza simples: cada iteração começa com uma solução π , a qual é estendida (caso π ainda não seja viável) para uma nova solução $\pi \cdot s$, onde s é um elemento da vizinhança de π . A operação de escolha deste elemento pode ser modelada por um **oráculo** que seleciona um elemento de $N(\pi)$, como observado no Algoritmo 9. Nos exemplos que discutiremos, é possível determinar o

³ ou ainda, **espaço de busca**

conjunto $N(\pi)$ para cada solução parcial π em tempo polinomial em $|S|$, o que implica que o tamanho de $N(\pi)$ também é polinomial em $|S|$.

Assim, para fins algorítmicos, adotaremos a seguinte definição: um **problema de otimização combinatória** é representado por uma tripla (S, N, c) , onde:

- S é o conjunto de **componentes**,
- N é a **função de estrutura de vizinhança**,
- c é a **função de custo**, que associa a cada solução viável π um custo $c(\pi) \in \mathbb{R}$.

Algoritmo 9 – Algoritmo feasible para a construção de uma solução viável

O algoritmo feasible recebe um problema de otimização combinatória $\mathfrak{P} = (S, N, c)$ e uma função $\text{choice} : 2^S \rightarrow S$ e devolve uma solução viável de \mathfrak{P} .

Cada iteração começa com uma solução $\pi \in \Sigma$. A primeira iteração começa com $\pi = \epsilon$. Cada iteração consiste no seguinte:

Caso 1: $N(\pi) = \emptyset$.

Devolva π e pare.

Caso 2: $N(\pi) \neq \emptyset$.

Seja $s := \text{choice}(N(\pi))$.

Comece uma nova iteração com $\pi \cdot s$ no lugar de π .

O objetivo, como adiantamos anteriormente, é encontrar uma **solução viável ótima** $\sigma^* \in \Sigma$ cujo custo $c(\sigma^*)$ seja mínimo, ou seja, $c(\sigma^*) \leq c(\sigma)$ para todo $\sigma \in \Sigma$. Embora a formalização que acompanhamos seja relativamente simples, não podemos dizer o mesmo do processo de busca pela solução viável ótima. A dificuldade reside no fato de que alguns problemas de otimização combinatória, ditos NP-difíceis, parecem⁴ ser computacionalmente intratáveis. Para tais problemas, com os melhores algoritmos exatos que conhecemos, o tempo necessário para encontrar uma solução viável ótima cresce exponencialmente com o tamanho do problema, tornando a resolução desses problemas praticamente inviável à medida que o tamanho do problema cresce.

Diante dessa limitação, veremos nas próximas seções como os algoritmos ACO propõem abordagens heurísticas capazes de gerar soluções viáveis suficientemente boas na prática, sem necessariamente alcançar alguma solução ótima. Contudo, antes

⁴ Parecem, pois, não sabemos se $P = NP$.

de cairmos de cabeça nos formigueiros, veremos alguns exemplos de problemas de otimização bem conhecidos na literatura.

Exemplo 5.1 – Problema do caixeiro-viajante

O **problema do caixeiro-viajante** (do inglês, traveling salesman problem – TSP) remonta ao problema de determinar a rota mais curta para que um vendedor visite exatamente uma vez cada cidade em uma região e depois retorne ao ponto de partida. Formalmente, o problema consiste em determinar um ciclo hamiltoniano de custo mínimo em um digrafo simples e completo G — isto é, o conjunto dos arcos de G é^a $\{(u, v) \in V_G \times V_G \mid u \neq v\}$. No digrafo G , os vértices representam as cidades, e os arcos, ponderados por uma função $w: A_G \rightarrow \mathbb{R}_+$, conectam todo par de vértices distintos. De acordo com o modelo de problema de otimização formalizado na Seção 5.1, definimos a tripla $\text{TSP} := (A_G, \mathcal{C}, d)$, onde:

- o conjunto dos arcos A_G é o conjunto dos componentes;
- o conjunto

$$\mathcal{C} := \{c \in A_G^* \mid c \text{ é um caminho simples ou um ciclo hamiltoniano}\}$$

é o espaço de soluções; e

- a função objetivo $d: \mathcal{C} \rightarrow \mathbb{R}_+$ que atribui a cada solução $c \in \mathcal{C}$ a distância total:

$$d(c) := w(c)$$

Note que, diante deste modelo, uma busca exaustiva testando todas as permutações em \mathcal{C} requer pelo menos $O(|A_G|!)$ comparações. No entanto, calcular a vizinhança $N_{\text{TSP}}(c)$ de uma solução parcial $c = \langle a_1, a_2, \dots, a_k \rangle$ pode ser uma tarefa computacionalmente mais simples — isto é, realizável em tempo polinomial. Para calcular a vizinhança em tempo polinomial, há dois casos a considerar:

Caso 1: c é um caminho simples maximal.

Neste caso, a vizinhança é um conjunto unitário contendo o arco que conecta o último vértice de c ao primeiro vértice de c , isso é,

$$N_{\text{TSP}}(c) := \{(a_k, a_1)\}.$$

Caso 2: c não é um caminho simples maximal. Seja t o término de c . Como G é um digrafo completo, sabemos que a vizinhança de saída^b $N^+(t)$ é exatamente

$V_G - t$. Sendo assim, a vizinhança

$$N_{\text{TSP}}(c) := \{(t, u) \in \{t\} \times V_G \mid u \notin V(c)\}$$

é o conjunto dos arcos que conectam t a um vértice “não visitado” ($V(c)$ denota os vértices do caminho c).

^a Um digrafo com $2^{|V_G|}$ arcos.

^b Peço que o leitor não confunda as vizinhanças de um vértice — vide Seção 2.5 — e a vizinhança de uma solução.

Exemplo 5.2 – Problema da mochila binária

O **problema da mochila binária** (do inglês, binary knapsack problem – BKP) é um problema clássico de otimização combinatória. Dado um conjunto de itens e uma mochila com uma certa capacidade, devemos determinar quais itens serão escolhidos para colocar na mochila, respeitando sua capacidade e tentando maximizar o valor dos itens escolhidos. Formalmente, dado um conjunto I de **itens**, cujos elementos estão associados a um **peso** pela função $w: I \rightarrow \mathbb{R}_+$ e a um **valor** pela função $v: I \rightarrow \mathbb{R}_{++}$, devemos encontrar uma solução que maximize o valor total dos itens escolhidos, respeitando a **capacidade** $C \in \mathbb{R}_+$ da mochila. De acordo com o modelo formal apresentado na Seção 5.1, definimos $\text{BKP} := (I, \mathcal{M}, \nu)$, onde:

- os itens em I representam o conjunto dos componentes;
- o conjunto das sequências sem repetições cujo peso total não ultrapassa C

$$\mathcal{M} := \{m \in I^* \mid m \text{ é injetora e } w(\underline{m}) \leq C\},$$

é o espaço de soluções; e

- a função objetivo $\nu: \mathcal{M} \rightarrow \mathbb{R}_+$ que atribui a cada solução $m \in \mathcal{M}$ o oposto da soma dos valores dos itens:

$$\nu(m) := -v(\underline{m}).$$

Essa manobra^a visa adequar a função objetivo ao modelo de problema de minimização.

Note, caro leitor, que \mathcal{M} é um conjunto fechado sobre permutações, isso é, dada uma solução $m \in \mathcal{M}$, qualquer permutação m' é também uma solução. Além do mais, m e m' sempre terão o mesmo custo e valor. Em decorrência disso, chamo

a atenção novamente para o fato de que uma busca sobre o espaço de soluções é computacionalmente inviável por apresentar uma complexidade $O(|I|! \cdot 2^{|I|})$. Contudo, calcular a vizinhança $N_{\text{BKP}}(m)$ de uma solução parcial $m \in \mathcal{M}$ é uma operação de complexidade linear:

$$N_{\text{BKP}}(m) := \{i \in I \mid m \cdot i \text{ é injetora e } w(\underline{m}) + w(i) \leq C\}$$

é o conjunto dos itens que não ocorrem em m que, quando adicionados à solução parcial, não excedem a capacidade da mochila.

^a Maximizar $\nu(m)$ é equivalente a minimizar $-\nu(m)$ sobre o conjunto das soluções viáveis m .

Exemplo 5.3 – Escalonamento em máquinas idênticas

Remetendo ao tema deste trabalho, exibiremos como este problema estudado na Seção 3.1 se encaixa no formalismo de um problema de otimização. Refrescando a memória do leitor, o problema de **escalonamento em máquinas idênticas** (do inglês, identical-machines scheduling – IMS) consiste em alocar um conjunto de tarefas em um conjunto de máquinas, minimizando a ocupação da máquina mais ocupada.

Seja J um conjunto de tarefas. Cada tarefa $j \in J$ tem uma duração $p(j)$ conforme a função $p: J \rightarrow \mathbb{R}_{++}$. Seja também M um conjunto finito e não vazio de máquinas disponíveis para o processamento de qualquer tarefa em J . Devemos determinar uma e somente uma máquina $m \in M$ para cada tarefa $j \in J$.

Como de costume, codificaremos as soluções em sequências finitas. Seja s uma sequência finita sobre $(J \times M)^*$. Vamos definir dois objetos que serão úteis a seguir. Para cada $m \in M$, $J_s(m)$ denota o conjunto $\{j \in J \mid (j, m) \in \underline{s}\}$ dos trabalhos associados a m em s . Analogamente, para todo $j \in J$, $M_s(j)$ denota o conjunto $\{m \in M \mid (j, m) \in \underline{s}\}$ das máquinas associadas a j em s .

Isto posto, definimos o problema de otimização para o escalonamento em máquina única como $\text{IMS} := (J \times M, \mathcal{S}, C_{\max})$, onde:

- o conjunto dos componentes é o conjunto de pares $J \times M$;
- o espaço de soluções é o conjunto de sequências finitas de pares tarefa-máquina

$$\mathcal{S} := \{s \in (J \times M)^*: |M_s(j)| \leq 1, \text{ para todo } j \in J\},$$

isso é, das sequências tais que cada trabalho está associado a no máximo uma máquina; e

- a função objetivo $C_{\max} : \mathcal{S} \rightarrow \mathbb{R}_{++}$ que atribui a cada escalonamento $s \in \mathcal{S}$ o *makespan*:

$$C_{\max}(s) := \max\{p(J_s(m)) \mid m \in M\},$$

que é a maior ocupação (soma das durações das tarefas alocadas) de uma máquina.

Chamo a atenção uma última vez para o tamanho do espaço de soluções, que neste caso é da ordem $O(|J| \times |M|^{|M|})$. Para a nossa sorte, é possível calcular a vizinhança $N_{\text{IMS}}(s)$ para cada $s \in \mathcal{S}$ em $O(|J||M|)$, olhando para as tarefas ainda não alocadas

$$N_{\text{IMS}}(s) = \{(j, m) \in J \times M \mid \text{não existe } m' \in M, \text{ tal que } (j, m') \in s\}.$$

5.2 Seguir o rastro, você deve!

Dedicaremos as próximas seções para descrever abstratamente o arcabouço teórico do método de otimização por colônia de formigas, a fim de justificar posteriormente as especificidades que caracterizam suas variantes. Um algoritmo ACO é essencialmente um algoritmo iterativo onde cada iteração se divide em duas etapas: construção e atualização. A etapa de construção envolve a criação de um conjunto de agentes artificiais, que ao simular o comportamento de formigas, são responsáveis por construir soluções viáveis para o problema baseadas no conhecimento adquirido em iterações anteriores. Em seguida, na fase de atualização, a informação a respeito da qualidade das soluções construídas é passada adiante, para ser utilizada nas próximas iterações.

O processo de construção é guiado por escolhas probabilísticas que equilibram dois comportamentos do algoritmo. Um deles é a **exploração** do espaço de soluções, enviesada por informações heurísticas específicas do problema. O outro é o comportamento de **exploração** do espaço de soluções, que ocorre através do conhecimento sobre as iterações anteriores. Esse último se utiliza da noção abstrata de *trilha de feromônios*, que carrega informações sobre a qualidade das soluções previamente encontradas. À medida que as formigas percorrem o espaço de soluções, elas modificam a trilha, enviesando a criação de soluções que minimizam o critério de otimização estabelecido. Esse balanço entre exploração e exploração torna mais provável que o algoritmo encontre soluções de alta qualidade para problemas onde uma busca exaustiva se mostra inviável.

Seja $\mathfrak{P} := (S, \Pi, c)$ um problema de otimização. Seja também T um conjunto finito e não vazio de **pontos de feromônio**, cujos elementos, informalmente, são os

pontos da trilha onde as formigas “depositam” feromônio. Uma função **concentração de feromônio** é uma função $f: T \rightarrow \mathbb{R}_+$, que leva cada ponto de feromônio $t \in T$ a um número real $f(t)$ que representa a concentração de feromônio sobre o ponto t .

Intuitivamente, os pontos de feromônios estão associados às possíveis formas de se estender uma solução parcial π através de sua vizinhança $N(\pi)$. O papel da função concentração de feromônio, grosso modo, é induzir uma boa escolha durante a construção. Neste caso, seja $E_{\mathfrak{P}} := \{(\pi, s) \mid \pi \in \bar{\Sigma}, s \in N(\pi)\}$ o conjunto das **possibilidades de extensão** de uma solução parcial π do problema \mathfrak{P} por um componente s na sua vizinhança. Quando \mathfrak{P} for claro no contexto, escreveremos apenas E para denotar $E_{\mathfrak{P}}$.

Definiremos a função **mapeamento de feromônio** $p: E \rightarrow T$ que leva uma possibilidade de extensão (π, s) a um ponto de feromônio $p(\pi, s) \in T$. É conveniente derivarmos recursivamente uma função $p^*: \Pi \rightarrow T^*$ que, dada uma sequência $\pi \in \Pi$, devolve uma sequência finita $p^*(\pi) \in T^*$ de pontos de feromônio associados a π , chamada de **trilha de feromônios** de π :

$$\begin{aligned} p^*(\epsilon) &:= \epsilon \\ p^*(\pi \cdot s) &:= p^*(\pi) \cdot p(\pi, s) \text{ para todo } \pi \in \Pi \text{ e todo } s \in N(\pi). \end{aligned}$$

Convém, assim, coletar T e p em um par (T, p) e chamá-lo de **estrutura de feromônios** de \mathfrak{P} . Note que, como veremos adiante, o uso de uma estrutura de feromônios é um recurso didático. É importante ressaltar que os algoritmos que exibiremos *não* necessitam de uma representação explícita do conjunto T . Logo, em princípio, T pode ter tamanho exponencial em relação ao tamanho do conjunto S . O leitor atento notará que os elementos de T podem ser construídos sob demanda durante a execução dos algoritmos. É claro que, numa implementação desses algoritmos (caso o tamanho de T seja polinomial no tamanho de S), o projetista tem a liberdade de construir os elementos de T sob demanda ou gerá-los todos explicitamente antes da execução do programa.

Revisitando os exemplos 5.1, 5.2 e 5.3 vistos anteriormente, veremos possíveis estruturas de feromônios para estes problemas.

Exemplo 5.4 – Problema do caixeiro-viajante (TSP)

Seja $\text{TSP} = (A_G, \mathcal{C}, d)$ o problema definido no Exemplo 5.1. Para resolver o TSP através de um algoritmo ACO, devemos primeiramente definir uma estrutura de feromônios no contexto deste problema. Uma estrutura de feromônios para o TSP é um par $\mathcal{T}_{\text{TSP}} := (A_G, z)$, onde o conjunto de pontos é simplesmente o conjunto A_G de arcos do digrafo e o mapeamento de feromônio $z(c, a) := a$, para todo $(c, a) \in E_{\text{TSP}}$.

Gostaria, caro leitor, de justificar essa escolha recorrendo à sua intuição, se me permite. Imagine que o caixeiro-viajante que nos trouxe este problema ofereceu bons torrões de açúcar para uma colônia de formigas em troca do trabalho árduo de encontrar a melhor rota. Cada formiga da colônia, partindo de uma cidade escolhida ao acaso, visitará todas as outras cidades até completar o ciclo. Neste processo, as formigas derrubam feromônio no caminho percorrido, no melhor estilo João e Maria, para se comunicarem com suas colegas. Projetando essa analogia em nosso modelo, esta é a motivação da escolha de A_G .

Exemplo 5.5 – Problema da mochila binária (BKP)

Como vimos anteriormente, no Exemplo 5.2, a ordem dos elementos em uma solução é irrelevante no contexto do problema $\text{BKP} = (I, \mathcal{M}, \nu)$. Em virtude disso, para cada solução $m \in \mathcal{M}$, qualquer permutação m' de m é uma solução de mesmo custo e valor. Baseado nisso, vamos definir uma estrutura de feromônios $\mathcal{T}_{\text{BKP}} := (F, k)$, onde $F := \{\underline{m} \mid m \in \mathcal{M}\}$ é o conjunto de pontos de feromônio e o mapeamento de feromônio $k(m, i) := \underline{m} \cup \{i\}$ para todo $(m, i) \in E_{\text{BKP}}$.

Observe que o conjunto F tem tamanho em $O(2^{|I|})$. Uma outra alternativa, semelhante à utilizada no Exemplo 5.1, consiste em utilizar a estrutura de feromônio $\mathcal{T}'_{\text{BKP}} := (I, k')$, onde $k'(m, i) := i$ para cada $(m, i) \in E_{\text{BKP}}$. Essa abordagem considera apenas a informação a respeito de cada componente para estender a solução parcial, desconsiderando o que já foi selecionado para reduzir o consumo de recursos computacionais. Isso evidencia que existem diferentes formas de representar a estrutura de feromônios para um mesmo problema, o que destaca a importância dessa escolha na etapa de projeto do algoritmo.

Exemplo 5.6 – Problema do escalonamento (IMS)

Seja $\text{IMS} = (J \times M, \mathcal{S}, C_{\max})$ o problema definido no Exemplo 5.3. Neste caso, definiremos, de modo muito semelhante ao Exemplo 5.4, a estrutura de feromônios $\mathcal{T}_{\text{IMS}} = (J \times M, q)$, cujo conjunto de pontos de feromônio é o conjunto $J \times M$ de componentes de solução e a função mapeamento de feromônio $q(s, (j, m)) := (j, m)$ para todo $(s, (j, m)) \in E_{\text{IMS}}$.

5.3 Fui ao mercado com um problema de otimização...⁵

Nesta seção, nos aprofundaremos na construção de uma solução viável por uma formiga, destacando o papel da estrutura de feromônio nesse processo. A intuição que buscamos alcançar, ilustrada pela Figura 18, é que cada formiga escolhe componentes de solução iterativamente, de forma sistemática, até que os componentes escolhidos formem uma sequência que caracterize uma solução viável para o problema de otimização.

O formalismo desse processo depende fortemente da noção de estrutura de vizinhança, definida anteriormente. Dada uma solução $\pi \in \Pi$ para o problema de otimização (S, Π, c) , a estrutura de vizinhança N devolve um conjunto de componentes $N(\pi) \subseteq S$. Neste contexto, chamaremos o processo de escolha de um componente $s \in N(\pi)$ de **movimento viável**. Para um algoritmo ACO, a principal característica deste processo é que a escolha é realizada de maneira probabilística, enviesada pela concentração de feromônios e por heurísticas relacionadas ao problema.

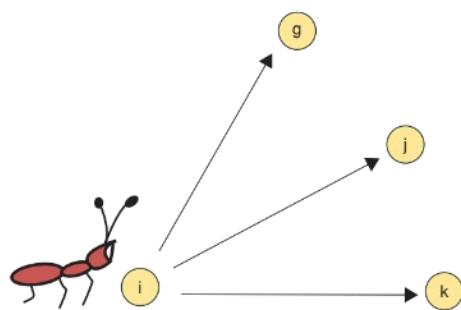


Figura 18 – Uma formiga mantendo uma solução $\pi = \langle i \rangle$ escolhendo um movimento viável sobre a vizinhança $N(\pi) = \{g, j, k\}$.

Fonte: Retirado de ([DORIGO et al., 1999](#))

Representamos essas heurísticas por meio de uma função de informação heurística apropriada para cada problema. Intuitivamente, esta função deve beneficiar movimentos

⁵ ... veio uma formiguinha e encontrou uma solução.

viáveis desejados e penalizar os movimentos custosos. Seja $\mathfrak{P} := (S, \Pi, c)$ um problema de otimização. A função **informação heurística** $\eta: E \rightarrow \mathbb{R}$ é uma função que, dada uma possibilidade de extensão $(\pi, s) \in E$, devolve um número real $\eta(\pi, s)$ que indica o quão promissor é estender π usando s . Resgatando os exemplos de problemas de otimização mencionados anteriormente, faremos uma breve menção de como η pode ser definida.

Exemplo 5.7 – Problema do caixeiro-viajante (TSP)

No TSP (Exemplo 5.1), para cada $(c, a) \in E_{\text{TSP}}$ definimos $\eta(c, a) := \frac{1}{w(a)}$ para beneficiar distâncias mais curtas e penalizar as mais longas.

Exemplo 5.8 – Problema da mochila binária (BKP)

No contexto do BKP (Exemplo 5.2), definimos $\eta(m, i) := \frac{v(i)}{w(i)}$ para cada $(m, i) \in E_{\text{BKP}}$ a fim de penalizar itens de menor custo-benefício durante a escolha e beneficiar itens de maior custo-benefício.

Exemplo 5.9 – Escalonamento em máquinas idênticas (IMS)

No IMS (Exemplo 5.3), a informação heurística $\eta(s, (j, m)) := \frac{1}{p(j) \times p(J_s(m))}$ para cada $(s, (j, m)) \in E_{\text{IMS}}$ entrega uma medida que favorece a escolha de tarefas com menor duração, enquanto penaliza máquinas com alta ocupação.

Seja $\mathfrak{P} := (S, \Pi, c)$ um problema de otimização e T um conjunto de pontos de feromônio. Uma **formiga** apropriada para \mathfrak{P}, T é uma tripla

$$(p, \text{move}, \ell),$$

cujos ingredientes são:

- uma função **mapeamento de feromônio** $p: E \rightarrow T$ que leva cada solução parcial π e cada possível extensão $s \in N(\pi)$ em um ponto de feromônio $p(\pi, s)$ em T ;
- uma função **movimento viável**

$$\text{move}: (T \rightarrow \mathbb{R}_+) \times (E \rightarrow T) \times \bar{\Sigma} \times (E \rightarrow \mathbb{R}) \rightarrow S$$

que dada uma função concentração de feromônio f , uma função mapeamento de feromônio p , uma solução parcial π e uma função informação heurística η , e devolve um componente $\text{move}(f, p, \pi, \eta) \in N(\pi)$; e

- uma função de **transformação local de feromônio** $\ell : (T \rightarrow \mathbb{R}) \times T \rightarrow (T \rightarrow \mathbb{R})$ que recebe uma função de concentração de feromônio f e um ponto de feromônio $t \in T$ para produzir uma nova função de concentração de feromônio $\ell(f, t)$; em geral, $\ell(f, t) = f[t \mapsto r]$ para algum $r \in \mathbb{R}$.

Exibimos, a seguir, o Algoritmo walk para a construção de uma solução viável por uma formiga, que pode ser encarado como uma instância do Algoritmo 9.

Algoritmo 10 – Algoritmo walk para a construção de uma solução viável

O algoritmo walk recebe

- um problema de otimização $\mathfrak{P} := (S, N, c)$,
- uma função informação heurística η de \mathfrak{P} ,
- uma formiga (p, move, ℓ) apropriada para \mathfrak{P}, T , onde T é um conjunto de pontos de feromônio;
- uma função concentração de feromônios $f_0 : T \rightarrow \mathbb{R}_+$

e devolve uma solução viável $\sigma \in \Sigma$. Cada iteração de walk mantém uma função concentração de feromônio f e uma solução $\pi \in \Pi$. A primeira iteração começa com $f = f_0$ e $\pi = \epsilon$.

O processo algorítmico de cada iteração consiste simplesmente no seguinte:

Caso 1: $N(\pi) = \emptyset$.

Devolve π e pare.

Caso 2: $N(\pi) \neq \emptyset$.

Seja $s := \text{move}(f, p, \pi, \eta)$.

Comece a próxima iteração com $\ell(f, p(\pi, s))$ no lugar de f e $\pi \cdot s$ no lugar de π .

5.4 Formigas de todas as colônias, uni-vos!

Como outros algoritmos bioinspirados, a principal heurística empregada pelo ACO se baseia no comportamento emergente que surge da interação entre muitos agentes. Veremos a seguir, de maneira abstrata, os objetos necessários para descrever um algoritmo que mimetiza esse comportamento.

Seja $\mathfrak{P} := (S, \Pi, c)$ um problema de otimização. Seja também $h : (2^\Sigma \times \Sigma) \rightarrow 2^\Sigma$

uma função **seletora de soluções viáveis** que, a partir de uma certa parte Σ' de Σ e de uma solução $\sigma \in \Sigma$, devolve um conjunto de soluções viáveis $h(\Sigma', \sigma)$. Veremos mais adiante que a função h em geral é submetida a um certo subconjunto Σ' de Σ , que corresponde a um conjunto de soluções viáveis encontradas em uma certa iteração por uma coleção de formigas, e que σ é a melhor solução encontrada pelo algoritmo até o momento.

Note, caro leitor, que, embora esse seja o caso mais frequente, a natureza abstrata desta função não impõe nenhum compromisso quanto a uma possível relação entre as soluções que entram e as que saem de h . Esse objeto propositalmente abstrato serve, como veremos mais adiante, para generalizar: (i) as diferentes abordagens de atualização global de feromônio em cada versão do algoritmo; e (ii) a possibilidade de composição do ACO com um algoritmo de busca local⁶ para aumentar a qualidade das soluções do problema.

Seja T um conjunto de pontos de feromônio. Seja também

$$g: (T \rightarrow \mathbb{R}) \times 2^\Sigma \rightarrow (T \rightarrow \mathbb{R})$$

uma função que recebe uma função concentração de feromônio f e um conjunto de soluções viáveis Σ' , para produzir uma nova função concentração de feromônio $g(f, \Sigma')$. Esse objeto, que chamaremos de função **transformação global de feromônio**, em conjunto com a função h , constituem o coração deste método. Além do mais, são nesses objetos que observaremos as maiores variações entre as diferentes versões dos algoritmos ACO.

Como o nome sugere, o ACO mimetiza o comportamento de uma *colônia* de formigas. Formalmente, definiremos uma **colônia** para \mathfrak{P} , T como sendo uma quádrupla (A, f_0, g, h) , onde

- A é um conjunto finito e não-vazio de formigas apropriadas para \mathfrak{P}, T ;
- $f_0: T \rightarrow \mathbb{R}_+$ é uma função concentração de feromônios com os valores iniciais;
- $g: (T \rightarrow \mathbb{R}) \times 2^\Sigma \rightarrow (T \rightarrow \mathbb{R})$ é uma função transformação global de feromônio; e
- $h: (2^\Sigma \times \Sigma) \rightarrow 2^\Sigma$ uma função seletora que devolve um conjunto de soluções viáveis para a atualização de feromônio.

O algoritmo aco que apresentaremos a seguir depende da noção de critério de parada. Informalmente, como o próprio nome sugere, o *critério de parada* estabelece as

⁶ Dorigo e Stutzle (2004) mencionam duas possíveis formas de compor a busca local com o ACO, inspirada nas teorias de Jean-Baptiste Lamarck e Charles Darwin. Na abordagem *lamarckista*, as concentrações de feromônio são atualizadas baseadas na solução gerada *após* a busca local. Já na *darwinista*, a atualização de feromônio é baseada na solução gerada antes da busca local. Segundo o autor, ao menos no caso do TSP, a primeira opção parece ser mais eficiente do que a segunda.

condições para que o algoritmo pare. Não me atrevo a tentar descrever todas as formas de se estabelecer um critério. Contudo, vale mencionar que critérios de parada típicos geralmente dependem do número de iterações do algoritmo ou de propriedades das soluções produzidas até a iteração corrente. Isto motiva a próxima definição. Seja X um conjunto qualquer. Dizemos que uma função $\theta : X^* \rightarrow \{0, 1\}$ é uma **função critério de parada** se $\theta(\epsilon) = 0$ e existe um $k \in \mathbb{N}_+$ tal que $\theta(\chi) = 1$ para cada $\chi \in X^*$ tal que $|\chi| \geq k$.

Algoritmo 11 – Algoritmo aco para otimização por colônia de formigas

O algoritmo aco é um algoritmo iterativo que devolve uma solução viável σ ao receber uma quádrupla

$$(\mathfrak{P}, \eta, \text{colony}, \text{stop}),$$

onde

- \mathfrak{P} é um problema de otimização;
- η é uma função informação heurística de \mathfrak{P} ;
- colony é uma colônia (A, f_0, g, h) para \mathfrak{P}, T , onde T é um conjunto de pontos de feromônio;
- stop : $\Sigma^* \rightarrow \{0, 1\}$ é uma função critério de parada.

Cada iteração do algoritmo mantém uma sequência ^a $\chi \in \Sigma^*$ e uma função concentração de feromônio f . A primeira iteração começa $\chi = \epsilon$ e $f = f_0$. Uma iteração qualquer avalia dois casos:

Caso 1: $\text{stop}(\chi) = 1$.

Devolva $\text{argmin}\{c(\sigma) \mid \sigma \in \chi\}$ e encerre o algoritmo.

Caso 2: $\text{stop}(\chi) = 0$.

Seja

$$\Sigma' := \{\text{walk}(\mathfrak{P}, \eta, \text{ant}, f) \mid \text{ant} \in A\}$$

o conjunto de soluções geradas pelas formigas e

$$\sigma := \text{argmin}\{c(\pi) \mid \pi \in \Sigma' \cup \underline{\chi}\}$$

uma melhor solução encontrada pelas formigas. Comece uma nova iteração com $g(f, h(\Sigma', \sigma))$ no lugar de f , e $\chi \cdot \sigma$ no lugar de χ .

^a É conveniente entender cada elemento χ de Σ^* como um *histórico*.

A seguir, veremos, dentre outras coisas, como cada especialização dos algoritmos define as funções f_0 , g e h .

5.5 Algoritmos ACO e suas variantes

O leitor mais ansioso pode ter-se incomodado um pouco com o nível de abstração adotado até aqui neste capítulo. Essa abordagem, que devo dizer, poderia ter sido mais extensa, se fez necessária para capturar a essência da técnica e as características de cada um dos algoritmos que vêm a seguir, onde se reúnem as particularidades de cada variante do algoritmo aco. Assim, para descrever cada uma das variantes, precisamos definir tanto a noção de uma formiga quanto a de uma colônia. Isto posto, abordaremos a seguir cinco algoritmos ACO estudados neste trabalho.

5.5.1 Sistema de formigas

O primeiro algoritmo ACO apresentado ao mundo foi o **sistema de formigas** (do inglês, *ant system* – AS) ([DORIGO et al., 1996](#)). As versões seguintes surgiram de modificações pontuais em alguns objetos da sua formulação. Uma formiga no contexto desse algoritmo não implementa a transformação local de feromônio e utiliza uma **regra de probabilidade**⁷ para a escolha dos movimentos viáveis. Sejam $\alpha, \beta \in \mathbb{R}$ parâmetros que calibram a influência da concentração de feromônio e da informação heurística nesta escolha. Esses parâmetros são heurísticos e, muitas vezes, devem ser ajustados por tentativa e erro. No Capítulo 7, dedicado à análise dos resultados, teceremos alguns comentários sobre os seus valores. Formalizaremos a seguir a implementação mais usual do algoritmo move instanciado a seguir no algoritmo probrule.

⁷ O formalismo acerca da teoria das probabilidades foge do escopo deste trabalho. Para se aprofundar nas definições e notações, vide ([ROSS, 2009](#)).

Algoritmo 12 – Algoritmo probrule: $(T \rightarrow \mathbb{R}_+) \times (E \rightarrow T) \times \bar{\Sigma} \times (E \rightarrow \mathbb{R}) \rightarrow S$ para seleção de um movimento viável

O algoritmo recebe uma função de concentração de feromônio $f: T \rightarrow \mathbb{R}_+$, uma função de mapeamento de feromônio $p: E \rightarrow T$, uma solução parcial $\pi \in \bar{\Sigma}$, e uma função informação heurística $\eta: E \rightarrow \mathbb{R}$ de um problema de otimização \mathfrak{P} , onde T é um conjunto de pontos de feromônio. Seja $P: S \rightarrow \mathbb{R}$ a função medida de probabilidade em S definida por:

$$P(s) := \begin{cases} \frac{f(p(\pi, s))^\alpha \times \eta(\pi, s)^\beta}{\sum_{u \in N(\pi)} f(p(\pi, u))^\alpha \times \eta(\pi, s)^\beta} & \text{se } s \in N(\pi), \text{ e} \\ 0 & \text{caso contrário.} \end{cases} \quad (5.2)$$

para cada $s \in S$. O algoritmo seleciona aleatoriamente um vizinho $s \in N(\pi)$ de acordo com a função P .

Seja \mathfrak{P} um problema de otimização e T um conjunto de pontos de feromônio. Como mencionado anteriormente, o algoritmo AS não implementa atualização local de feromônio. Neste caso, utilizamos a função $\ell_{AS}: (T \rightarrow \mathbb{R}_+) \times T \rightarrow (T \rightarrow \mathbb{R}_+)$ tal que $\ell_{AS}(f, t) := f$ de modo que nenhuma modificação é provocada na concentração de feromônio após a seleção de um movimento viável. Sendo assim, uma formiga apropriada para \mathfrak{P}, T no contexto do algoritmo AS é simplesmente:

$$\text{ant}_{AS} := (p, \text{probrule}, \ell_{AS}),$$

onde $p: E_{\mathfrak{P}} \rightarrow T$ é a função de mapeamento de feromônio associada a \mathfrak{P}, T .

Uma vez definida a noção de formiga, vamos agora definir a noção de colônia no AS. Uma colônia para \mathfrak{P}, T no contexto do AS é a tupla

$$\text{colony}_{AS} := (A_{AS}, f_{AS}, g_{AS}, h_{AS}),$$

cujos elementos serão definidos a seguir.

Baseado na noção de formiga, A_{AS} é um conjunto não vazio e finito de formigas apropriadas para o \mathfrak{P}, T no contexto do AS. Seja $\tau_0 \in \mathbb{R}_{++}$ um parâmetro que determina a concentração de feromônio inicial nos pontos do conjunto de feromônios. A função concentração inicial de feromônio, f_{AS} , é definida pondo-se $f_{AS}(t) := \tau_0$ para cada $t \in T$.

Vamos agora definir a função h_{AS} no contexto do AS. A cada iteração do algoritmo AS, todas as formigas participam da etapa de atualização da estrutura de feromônios. Assim, a função seletora de soluções $h_{AS}: 2^\Sigma \times \Sigma \rightarrow 2^\Sigma$ é definida pondo-se $h_{AS}(\Sigma', \sigma) := \Sigma'$ para toda parte $\Sigma' \subseteq \Sigma$ e para qualquer solução $\sigma \in \Sigma$.

Para terminar, definiremos g_{AS} no contexto do AS. Seja $\Delta: T \times \Sigma \rightarrow \mathbb{R}$ uma **função depósito de feromônio** com a seguinte propriedade: para cada $t \in T$ e cada $\sigma \in \Sigma$, Δ devolve um real positivo se $t \in \underline{p^*(\sigma)}$ e 0, caso contrário. Espera-se que o valor de $\Delta(t, \sigma)$ seja um número que mensura, por alguma heurística, a qualidade da solução σ . Obviamente, Δ também depende do problema, então façamos um breve desvio de rota para ver exemplos:

Exemplo 5.10 – Problema do caixeiro-viajante (TSP)

No contexto do TSP (Exemplo 5.1), escolhemos Δ_{TSP} inversamente proporcional à distância total percorrida em um ciclo c

$$\Delta_{\text{TSP}}(t, c) := \begin{cases} \frac{1}{d(c)} & \text{se } t \in \underline{z^*(c)} \\ 0 & \text{caso contrário.} \end{cases}$$

Exemplo 5.11 – Problema da mochila binária (BKP)

No BKP (Exemplo 5.2), Δ_{BKP} pode ser diretamente proporcional ao valor total dos itens alocados em uma mochila m (se usa $-\nu(m)$ pois este é um problema de maximização)

$$\Delta_{\text{BKP}}(t, m) := \begin{cases} -\nu(m) & \text{se } t \in \underline{k^*(m)} \\ 0 & \text{caso contrário.} \end{cases}$$

Exemplo 5.12 – Escalonamento em máquinas idênticas (IMS)

Já para o IMS (Exemplo 5.3), quanto maior o makespan, menor o depósito Δ_{IMS}

$$\Delta_{\text{IMS}}(t, s) := \begin{cases} \frac{1}{C_{\max}(s)} & \text{se } t \in \underline{q^*(s)} \\ 0 & \text{caso contrário.} \end{cases}$$

Retomando ao que interessa, seja $\rho \in \mathbb{R}_{++}$ o coeficiente de **evaporação de feromônio**, que controla a velocidade com que o feromônio evapora dos pontos de feromônio menos visitados. Eis a definição da função de atualização global de feromônio no contexto do AS:

$$g_{\text{AS}}(f, \Sigma') := f[T \ni t \mapsto (1 - \rho)f(t) + \sum_{\sigma \in \Sigma'} \Delta(t, \sigma)] \quad (5.3)$$

para cada $f: T \rightarrow \mathbb{R}_+$ e cada $\Sigma' \subseteq \Sigma$. A definição concentra a inteligência da colônia através de duas abordagens. A cada atualização, cada ponto de feromônio sofre (i) um decréscimo na concentração por um fator ρ , que emula a evaporação do feromônio neste ponto, e (ii) um acréscimo na concentração realizado por cada formiga que passou por ali. O primeiro termo garante que, à medida que o algoritmo avança, pontos de feromônio menos visitados tendem a perder feromônio e se tornam menos relevantes. Já o somatório do segundo termo garante que cada ponto receberá um depósito de feromônio de uma formiga que o utilizou, proporcional à qualidade da solução gerada.

As próximas especializações surgem de modificações pontuais nos objetos que acabamos de ver.

5.5.2 Sistema de formigas elitista

A primeira tentativa de melhorar os resultados obtidos com o algoritmo AS foi introduzida pelo **sistema de formigas elitista** (do inglês, *elitist ant system* – EAS) ([DORIGO et al., 1996](#)). Nesta versão do algoritmo, que compartilha muitas semelhanças com o AS, o conceito de elitismo foi considerado na etapa de atualização de feromônios.

O elitismo é uma estratégia que busca preservar as melhores soluções encontradas durante a etapa de atualização, buscando garantir que elas não sejam perdidas ao longo das iterações. Essa estratégia é particularmente útil para acelerar uma possível convergência⁸ do algoritmo, pois as soluções de alta qualidade se mantêm relevantes, guiando as formigas para regiões possivelmente promissoras do espaço de soluções. Por outro lado, essa estratégia pode levar a uma estagnação prematura em torno de soluções sub-ótimas, uma vez que o uso do elitismo pode enviesar o algoritmo para um cenário de maior exploração no entorno de ótimos locais.

No caso do algoritmo EAS, o elitismo é implementado reforçando o feromônio nos pontos associados à melhor solução encontrada até o momento. Isso significa que, além do feromônio depositado pelas formigas em cada iteração, uma quantidade adicional de feromônio é depositada nos pontos associados à melhor solução encontrada, aumentando a probabilidade de que as formigas sigam caminhos semelhantes.

Seja \mathfrak{P} um problema de otimização e T um conjunto de pontos de feromônio. O funcionamento do algoritmo EAS é exatamente o mesmo do algoritmo AS, com exceção de dois objetos. No que diz respeito à definição da formiga, temos que $\text{ant}_{\text{EAS}} := \text{ant}_{\text{AS}}$, isso é, assim como o AS, essa variante utiliza o algoritmo probrule e também não implementa transformações locais de feromônio. A diferença entre os algoritmos,

⁸ Possível pois, não me atrevo a afirmar que o método ACO tem convergência garantida. É bem verdade que, sob certas circunstâncias alguns algoritmos ACO convergem para uma solução ótima após um número suficientemente grande de iterações, mas não abordaremos isto neste trabalho. O leitor pode consultar as provas de convergência para os algoritmos MMAS e ACS em ([DORIGO; STUTZLE, 2004](#)).

portanto, aparece na definição de uma colônia (Equação 5.4), apenas nos objetos h_{EAS} e g_{EAS} , definidos a seguir, cuja função está associada à atualização de feromônio.

$$\text{colony}_{\text{EAS}} := (A_{\text{AS}}, f_{\text{AS}}, g_{\text{EAS}}, h_{\text{EAS}}) \quad (5.4)$$

A função g_{EAS} , descrita na Equação 5.5, difere de g_{AS} pela adição do termo $\varepsilon\Delta(t, \sigma)$, associado à solução σ , que assumimos ser a melhor encontrada. Neste termo, $\Delta(t, \sigma)$ proporciona um incremento adicional caso o ponto de feromônio t esteja na trilha desta solução. A intensidade deste reforço é controlada pelo parâmetro $\varepsilon \in \mathbb{R}_{++}$, denominado **peso do reforço elitista**.

$$g_{\text{EAS}}(f, \Sigma' \cup \{\sigma\}) := f[T \ni t \mapsto (1 - \rho)f(t) + \sum_{\sigma' \in \Sigma'} \Delta(t, \sigma') + \varepsilon\Delta(t, \sigma)] \quad (5.5)$$

para cada $f: T \rightarrow \mathbb{R}_+$ e cada $\Sigma' \subseteq \Sigma$ e $\sigma \in \Sigma$.

Para que isso funcione apropriadamente, precisamos garantir que g_{AS} sempre terá acesso à melhor solução encontrada pela colônia. Resolvemos isso definindo uma nova função seletora de soluções para o EAS, pondo $h_{\text{EAS}}(\Sigma', \sigma) := \Sigma' \cup \{\sigma\}$ para todo $\Sigma' \subseteq \Sigma$ e todo $\sigma \in \Sigma$.

5.5.3 Sistema de formigas com classificação

Outra tentativa de melhorar o algoritmo AS veio com o **sistema de formigas com classificação** (do inglês, *rank-based ant system* – RBAS) (BULLNHEIMER et al., 1999). A única mudança (em relação ao AS) introduzida por esse algoritmo aparece na definição da função de transformação global de feromônio, onde verificamos uma ligeira alteração no depósito de feromônio. Diferentemente do AS em que toda formiga no conjunto A_{AS} contribui com um depósito, no algoritmo RBAS as formigas são classificadas pelo valor da função objetivo e apenas algumas das melhores formigas participam desta etapa.

Seja \mathfrak{P} um problema de otimização e T um conjunto de pontos de feromônio. Seja também $r \in \mathbb{N}_+$ um parâmetro denominado **tamanho da classificação**. É claro que para que nada exploda, temos que $1 \leq r \leq |A_{\text{AS}}|$. Admita que $\text{rank}: (2^\Sigma - \emptyset) \times ([r+1] - 0) \rightarrow \Sigma$ é uma função que, dada uma parte finita e não vazia $\Sigma' \subseteq \Sigma$ e uma colocação $1 \leq i \leq r$, devolve a solução $\text{rank}(\Sigma', i)$ que ocupa a i -ésima colocação neste conjunto. Conforme a Equação 5.6, uma função de transformação global de feromônio para o RBAS é

$$g_{\text{RBAS}}(f, \Sigma' \cup \{\sigma\}) := f\left[T \ni t \mapsto (1 - \rho)f(t) + r\Delta(t, \sigma) + \left(\sum_{i=1}^{r-1} (r-i)\Delta(t, \text{rank}(\Sigma', i))\right)\right], \quad (5.6)$$

para todo $f: T \rightarrow \mathbb{R}_+$, todo $\Sigma' \subseteq \Sigma$ e todo $\sigma \in \Sigma$. Como de costume, o primeiro termo faz um decréscimo na concentração para emular a evaporação do feromônio. O segundo

termo faz o maior depósito sobre os pontos associados à solução σ . Já o terceiro, que envolve apenas as $r - 1$ melhores formigas em Σ' , aplica um depósito (proporcional à colocação da formiga) na concentração associada ao ponto t .

Isto posto, o algoritmo RBAS de forma muito singela, é caracterizado por:

$$\begin{aligned}\text{ant}_{\text{RBAS}} &:= \text{ant}_{\text{AS}} \\ \text{colony}_{\text{RBAS}} &:= (A_{\text{AS}}, f_{\text{AS}}, g_{\text{RBAS}}, h_{\text{EAS}}).\end{aligned}$$

5.5.4 Sistema de formigas MAX-MIN

Segundo [Dorigo e Stutzle \(2004\)](#), após a publicação do algoritmo AS em 1996, diversos pesquisadores se debruçaram sobre a técnica ACO tentando otimizar seus resultados, na expectativa de torná-lo competitivo com outros algoritmos conhecidos. Nesse contexto, como apontado por [Stutzle e Hoos \(2000\)](#), os algoritmos ACO que produziram os melhores resultados heurísticos para o TSP têm em comum uma característica de maior exploração em torno das melhores soluções. Surge então o **sistema de formigas MAX-MIN** (do inglês, *MAX-MIN ant system* – MMAS) ([STUTZLE; HOOS, 2000](#)), que traz pelo menos três modificações em relação ao AS para se aproveitar deste fato.

A primeira modificação, mais impactante, consiste em uma atualização de feromônio menos diversa comparada ao AS. No MMAS. Apenas a formiga que encontrou a solução de menor custo realiza a atualização da estrutura de feromônios. Isso pode parecer seguir o caminho oposto à ideia de exploração abordada pelo método, e de fato, somente esta modificação enviesaria a construção das soluções, levando o algoritmo a cenários de estagnação.

Para contrabalancear esse efeito, a segunda modificação no algoritmo restringe os valores da concentração de feromônio a um intervalo real $[\tau_{\min}, \tau_{\max}]$. Essa medida, inevitavelmente, limita a influência do feromônio durante a escolha do movimento viável e, consequentemente, limita a probabilidade de um componente na vizinhança ser selecionado. Desta forma, movimentos viáveis muito realizados não afetam tanto o mecanismo de exploração do algoritmo, evitando a estagnação prematura para uma solução sub-ótima.

Por fim, a concentração de feromônio no início da execução tem o valor máximo τ_{\max} em todos os pontos de feromônio. [Stutzle e Hoos \(2000\)](#) argumentam que essa escolha, quando associada a um valor pequeno para o parâmetro de evaporação de feromônio ρ , também estimula uma maior exploração do espaço de soluções no início do algoritmo, pois a diferença relativa na concentração sob os pontos de feromônio varia de forma mais suave do que nas outras versões do algoritmo.

Sejam $\tau_{\min} < \tau_{\max} \in \mathbb{R}$ parâmetros reais que correspondem respectivamente ao **valor mínimo** da função concentração de feromônio e ao **valor máximo** da função concentração de feromônio. Definimos a **função de saturação de feromônio**

$$\begin{aligned}\phi: \mathbb{R} &\rightarrow [\tau_{\min}, \tau_{\max}] \\ x &\mapsto \max(\min(x, \tau_{\max}), \tau_{\min})\end{aligned}$$

que limita um valor real ao intervalo $[\tau_{\min}, \tau_{\max}]$. Como adiantamos, o parâmetro τ_{\max} também é utilizado para definir a função concentração de feromônios inicial, isso é, $f_{\text{MMAS}}(t) := \tau_{\max}$, para todo $t \in T$.

Seja \mathfrak{P} um problema de otimização e T um conjunto qualquer de pontos de feromônio. Uma função de atualização global de feromônio no contexto do MMAS, descrita pela Equação 5.7, (i) faz uso da função ϕ para limitar os valores da função concentração de feromônios e, (ii) considera somente a melhor solução encontrada até o momento para os depósitos. Sendo assim,

$$g_{\text{MMAS}}(f, \{\sigma\}) := f[T \ni t \mapsto \phi((1 - \rho)f(t) + \Delta(t, \sigma))], \quad (5.7)$$

para todo $f: T \rightarrow \mathbb{R}_+$ e todo $\sigma \in \Sigma$. Ademais, para que isso seja possível, definimos a função seletora de soluções pondo $h_{\text{MMAS}}(\Sigma', \sigma) := \{\sigma\}$ para todo $\Sigma' \subseteq \Sigma$ e todo $\sigma \in \Sigma$.

Finalmente, o algoritmo MMAS é totalmente caracterizado por:

$$\begin{aligned}\text{ant}_{\text{MMAS}} &:= \text{ant}_{\text{AS}} \\ \text{colony}_{\text{MMAS}} &:= (A_{\text{AS}}, f_{\text{MMAS}}, g_{\text{MMAS}}, h_{\text{MMAS}}).\end{aligned}$$

5.5.5 Sistema de colônia de formigas

De todos os algoritmos ACO abordados até aqui, o **sistema de colônia de formigas** (do inglês, *ant colony system* – ACS) (DORIGO; GAMBARDELLA, 1997) apresenta as maiores diferenças. Como veremos, uma formiga no contexto do ACS desempenha suas escolhas de movimentos viáveis de maneira mais agressiva. Outro aspecto característico desta colônia é o uso de um procedimento de transformação local de feromônio.

A escolha de um movimento viável no contexto do ACS se dá por meio da **regra proporcional pseudo-aleatória**, que de acordo com um sorteio aleatório, pode apresentar um comportamento guloso ou o comportamento probabilístico já conhecido das outras versões do ACO. Essa característica busca equilibrar o grau de exploração e exploração do espaço de soluções, alternando randomicamente entre uma busca aleatória e uma busca gulosa.

Seja \mathfrak{P} um problema de otimização combinatória e seja T um conjunto de pontos de feromônio. Seja $\eta: E_{\mathfrak{P}} \rightarrow R$ a função de informação heurística e $\alpha, \beta \in \mathbb{R}$ os

parâmetros de influência já conhecidos das outras versões do algoritmo. Introduzimos agora um novo parâmetro $q_0 \in [0, 1]$, chamado **parâmetro de corte de exploração**. Considere a regra de probabilidade probrule: $(T \rightarrow \mathbb{R}_+) \times (E \rightarrow T) \times \bar{\Sigma} \times (E \rightarrow \mathbb{R}) \rightarrow S$ apresentada na Seção 5.5.1. A implementação do algoritmo move que descreve a regra proporcional pseudo-aleatória será descrita a seguir.

Algoritmo 13 – Algoritmo pseudorule: $(T \rightarrow \mathbb{R}_+) \times (E \rightarrow T) \times \bar{\Sigma} \times (E \rightarrow \mathbb{R}) \rightarrow S$
para seleção de um movimento viável

O algoritmo recebe uma função de concentração de feromônio $f: T \rightarrow \mathbb{R}_+$, uma função de mapeamento de feromônio $p: E \rightarrow T$, uma solução parcial $\pi \in \bar{\Sigma}$, e uma função informação heurística $\eta: E \rightarrow \mathbb{R}$ de um problema de otimização \mathfrak{P} , onde T é um conjunto de pontos de feromônio.

Sorteie um número aleatório $q \in [0, 1]$ de uma distribuição uniforme. Há dois casos a considerar:

Caso 1: $q \leq q_0$.

Neste caso, faremos uma escolha gulosa. Devolva

$$\operatorname{argmax}\{f(p(\pi, s))\eta(\pi, s)^\beta \mid s \in N(\pi)\}$$

e encerre o algoritmo.

Caso 2: $q > q_0$.

Devolva o componente probrule($f, p, \pi, \eta \in N(\pi)$) sorteado aleatoriamente — vide Algoritmo 12 — e encerre o algoritmo.

Visando elevar o grau de exploração do espaço de soluções, cada formiga ao realizar um movimento viável provoca uma transformação local na concentração de feromônio. Essa alteração, descrita pela Equação 5.8, se baseia em uma combinação convexa⁹ entre a concentração de feromônio $f(t)$ e a quantidade inicial de feromônio τ_0 . O parâmetro $\varphi \in [0, 1]$, denominado **coeficiente de decaimento de feromônio**, regula a importância relativa desses parâmetros na atualização.

$$\ell_{\text{ACS}}(f, t) = f[t \mapsto (1 - \varphi)f(t) + \varphi\tau_0]. \quad (5.8)$$

Com esses ingredientes, definimos a noção de uma formiga apropriada para

⁹ A combinação convexa atua como uma média ponderada. A contribuição de $f(t)$ e τ_0 é regulada por φ .

\mathfrak{P}, T no contexto do algoritmo ACS:

$$\text{ant}_{\text{ACS}} := (p, \text{pseudorule}, \ell_{\text{ACS}}),$$

onde $p: E \rightarrow T$ é a função de mapeamento de feromônio associada a \mathfrak{P}, T .

Além da transformação local na concentração de feromônios, o algoritmo ACS apresenta uma modificação importante na função de transformação global. Assim como o algoritmo MMAS, apenas uma formiga atua na transformação global de feromônio. Neste caso, diferente do que vinha sendo feito até aqui, apenas os pontos de feromônio associados à solução observada serão atualizados. Conforme a Equação 5.9, a concentração de cada ponto de feromônio na trilha de σ é atualizada através de uma combinação convexa entre $f(t)$ e $\Delta(t, \sigma)$, com o parâmetro de evaporação de feromônio ρ mediando a contribuição de cada um desses termos.

$$g_{\text{ACS}}(f, \{\sigma\}) := f[p^*(\sigma) \ni t \mapsto (1 - \rho)f(t) + \rho\Delta(t, \sigma)], \quad (5.9)$$

para cada $f: T \rightarrow \mathbb{R}_+$ e cada $\sigma \in \Sigma$.

Existem duas principais vertentes na literatura sobre qual solução deve ser entregue à função g_{ACS} . Na Equação 5.9, é possível que σ seja a **melhor solução da iteração** (do inglês, *iteration best*) ou a **melhor solução até o momento** (do inglês, *best so far*). Embora a maioria dos resultados na literatura indique que o ACS produz melhores resultados quando σ é a melhor solução até o momento (DORIGO; STUTZLE, 2004), apresentaremos aqui o uso da melhor solução da iteração. A justificativa para essa escolha é puramente didática, uma vez que se optássemos o outro caminho, teríamos $h_{\text{ACS}} := h_{\text{MMAS}}$.

Sendo assim, definimos a função seletora de soluções viáveis pondo

$$h_{\text{ACS}}(\Sigma', \sigma) := \{ \underset{\pi \in \Sigma'}{\operatorname{argmin}} \{c(\pi) \mid \pi \in \Sigma'\} \},$$

para todo $\Sigma' \subseteq \Sigma$ e todo $\sigma \in \Sigma$. É prudente refrescar a memória do leitor que c é a função custo do problema de otimização \mathfrak{P} , portanto, h_{ACS} devolve um conjunto unitário com uma solução em $\Sigma' \subseteq \Sigma$ que minimiza c .

Finalmente, uma colônia para \mathfrak{P}, T no contexto do algoritmo ACS é simplesmente

$$\text{colony}_{\text{ACS}} := (A_{\text{ACS}}, f_{\text{AS}}, g_{\text{ACS}}, h_{\text{ACS}}),$$

onde A_{ACS} é um conjunto finito e não vazio de formigas ant_{ACS} apropriadas para \mathfrak{P}, T .

5.6 Que o feromônio esteja com você

Os algoritmos que abordamos incluem, à sua maneira, diversos mecanismos para evitar estagnação prematura e equilibrar o grau de exploração e exploração

do espaço de soluções. Esses recursos, contudo, não vêm sem uma complexidade elevada, evidenciada, por exemplo, pelo número significativo de parâmetros envolvidos. Considerando o volume de informações que apresentamos, especialmente para leitores menos familiarizados com o tema, faremos agora um último esforço para sintetizar o conteúdo abordado.

Na Seção 5.1 vimos a formalização de um problema de otimização combinatória, discutindo aspectos da teoria da complexidade no processo. Vimos que, graças à noção de estrutura de vizinhança somos capazes de projetar algoritmos que exploram o espaço de soluções sem gerá-lo explicitamente, o que é especialmente útil para resolver problemas NP-difíceis. Em seguida, na Seção 5.2, começamos a discutir as bases do ACO através do conceito de feromônio. Tratamos de definir objetos abstratos chamados **pontos de feromônio** e uma função **mapeamento de feromônio** que associa o problema de otimização a esses pontos. Cada um desses pontos, possui uma certa concentração de feromônio associada, que por meio de **evaporação** e de **depósito**, varia no decorrer da execução do algoritmo.

Na Seção 5.3, falamos sobre as andarilhas que fazem o trabalho pesado neste método. Definimos a noção de movimento viável e descrevemos em alto nível, como uma formiga constrói uma solução viável. Mencionamos aqui as funções de **atualização local de feromônio** e de **informação heurística**. O comportamento da formiga foi sintetizado no Algoritmo 10.

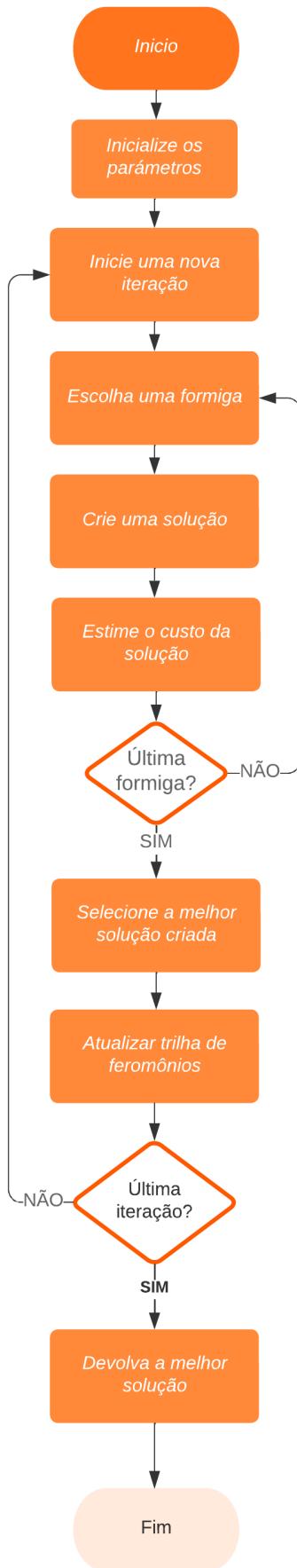


Figura 19 – Arcabouço de um algoritmo de otimização por colônia de formigas.

Uma andorinha só não faz verão, assim como uma única formiga não faz uma colônia. A Seção 5.4 vem para descrever o funcionamento do ACO através da noção de **colônia**. Nesta seção introduzimos o conceito de **transformação global de feromônio**, bem como a **função seletora de funções viáveis**, responsável por decidir, quais soluções serão consideradas na atualização. Outro objeto definido aqui é a função stop, que fundamenta a noção de **critério de parada** de um algoritmo. A Figura 19 ilustra, em um fluxograma, um resumo da implementação do método.

Por fim, uma vez formalizado, na Seção 5.5 apresentamos os cinco algoritmos ACO estudados e implementados neste trabalho. Considerando estes algoritmos, a Tabela 5 contém uma listagem dos parâmetros apresentados, detalhando sua notação e respectiva função nos algoritmos. Já a Tabela 6, ilustra a relação entre cada parâmetro da tabela anterior e os diversos algoritmos ACO apresentados.

Notação	Nome	Função
α	Fator de influência do feromônio	Determinar o peso da concentração de feromônio na seleção de um movimento
β	Fator de influência da informação heurística	Determinar o peso da informação heurística na seleção de um movimento
ρ	Coeficiente de evaporação de feromônio	Estabelece a taxa de evaporação na atualização global de feromônio
τ_0	Concentração inicial de feromônio	Determina a concentração inicial de feromônio depositado sobre a trilha
ε	Peso do reforço elitista	Controla a intensidade do reforço elitista na atualização global de feromônio
r	Tamanho do <i>ranking</i>	Controla a intensidade do reforço elitista na atualização global de feromônio
τ_{\min}	Concentração mínima de feromônio	Determina um limite inferior para a concentração de feromônio
τ_{\max}	Concentração máxima de feromônio	Determina um limite superior para a concentração de feromônio
q_0	Parâmetro de corte de exploração	Controlar o balanço entre exploração e exploração
φ	Coeficiente de decaimento de feromônio	Controla a taxa de decaimento na atualização local de feromônio

Tabela 5 – Detalhamento dos parâmetros apresentados na Seção 5.5

Notação	Nome	AS	EAS	RBAS	MMAS	ACS
α	Fator de influência do feromônio	x	x	x	x	x
β	Fator de influência da informação heurística	x	x	x	x	x
ρ	Coeficiente de evaporação de feromônio	x	x	x	x	x
τ_0	Concentração inicial de feromônio	x	x	x		x
ε	Peso do reforço elitista		x			
r	Tamanho do ranking			x		
τ_{\min}	Concentração mínima de feromônio				x	
τ_{\max}	Concentração máxima de feromônio				x	
q_0	Parâmetro de corte de exploração					x
φ	Coeficiente de decaimento de feromônio					x

Tabela 6 – Parâmetros por algoritmo

5.7 A escala e a formiga

Após toda a discussão precedente, estamos habilitados a formalizar o escalonamento flexível orientado a trabalho como um problema de otimização, a fim de submeter a algum algoritmo de otimização por colônia de formigas. Mas antes, vamos a algumas definições preliminares.

Seja $\mathfrak{F} = (J, O, M, r, n, \pi, \Phi, \varrho)$ uma instância do FJSSP, cuja definição foi devidamente apresentada no Capítulo 4. Para toda sequência finita de pares operação-máquina $\lambda \in (O \times M)^*$, diremos¹⁰ que $om \in O \times M$ **ocorre** em λ se om é um termo de λ , isto é, $om \in \underline{\lambda}$. Para dois termos distintos $om, o'm' \in \underline{\lambda}$ quaisquer, diremos que om **ocorre antes** de $o'm'$ em λ , se $o'm'$ ocorre em $\lambda_{om..}$.

Lembre-se, caro leitor, que foi estabelecido na Seção 4.2 que um par (μ, s) é uma solução viável para \mathfrak{F} se satisfaz

$$\begin{aligned} \mu_o &\in \Phi(o) \text{ para cada } o \in O; \text{ e} \\ s_{\pi_j(0)} &\geq r_j \text{ para todo } j \in J; \\ s_o + \varrho_o(\mu_o) &\leq s_{\sigma(o)} \text{ para todo } o \in \overline{O^T}; \text{ e} \\ \mu_o = \mu_{o'} &= m \implies s_o + \varrho_o(m) \leq s_{o'} \text{ ou } s_{o'} + \varrho_{o'}(m) \leq s_o \text{ para cada } (o, m, o') \in \Psi. \end{aligned}$$

Seja \mathcal{E} o conjunto das sequências λ sobre $O \times M$ tais que

- (i) para todo par $om \in \underline{\lambda}, m \in \Phi(o)$; e
- (ii) para cada operação não terminal $o \in \overline{O^T}$ e para duas máquinas $m, m' \in M$ quaisquer, se om e $\sigma_o m'$ ocorrem em λ , então om ocorre antes de $\sigma_o m'$.

A tripla

$$\text{FJSSP} := (O \times M, \mathcal{E}, C_{\max} \circ \text{seq2sol})$$

representa a formulação do problema do escalonamento flexível orientado a trabalho como um problema de otimização. O Algoritmo 14, descrito a seguir, que aparece na composição $C_{\max} \circ \text{seq2sol}$, mostra como extrair uma solução viável para o FJSSP a partir de uma sequência $\lambda \in \mathcal{E}$.

¹⁰ Como de costume, escreveremos om no lugar de (o, m) , para todo termo $(o, m) \in \underline{\lambda}$

Algoritmo 14 – Algoritmo seq2sol: $\mathcal{E} \rightarrow (O \rightarrow M) \times (O \rightarrow \mathbb{R}_+)$ para extração de uma solução viável a partir de uma sequência

O algoritmo seq2sol recebe uma sequência $\lambda \in \mathcal{E}$ e devolve uma solução viável (μ, s) para \mathfrak{F} .

Cada iteração mantém uma 4-upla (μ, s, γ, ℓ) , tal que (μ, s) é uma solução parcial para o FJSSP; γ é um sufixo de λ e $\ell: M \rightarrow O$ é uma função parcial que, dada uma máquina $m \in M$, se $m \in Dom(\ell)$ então $\ell(m)$ é a última operação alocada na máquina m em (μ, s) . A primeira iteração começa com $(\emptyset, \emptyset, \lambda, \emptyset)$. Em uma iteração arbitrária, há dois casos a considerar:

Caso 1: $\gamma = \epsilon$.

Devolve (μ, s) e encerre o algoritmo.

Caso 2: $\gamma = om \cdot \gamma'$.

Seja $j \in J$ o trabalho tal que o é uma operação de j , isso é, $o \in \underline{\pi_j}$. Seja também^a

$$\alpha := \begin{cases} s(\sigma^{-1}(o)) + \varrho(\sigma^{-1}(o), \mu(\sigma^{-1}(o))) & \text{se } o \in \overline{O_\perp} \\ r_j & \text{caso contrário,} \end{cases}$$

e^b

$$\beta := \begin{cases} s(\ell(m)) + \varrho(\ell(m), m) & \text{se } m \in Dom(\ell) \\ 0 & \text{caso contrário.} \end{cases}$$

Comece a próxima iteração com

$$(\mu[o \mapsto m], s[o \mapsto \max\{\alpha, \beta\}], \gamma', \ell[m \mapsto o])$$

mo lugar de (μ, s, γ, ℓ) .

^a O número α corresponde ao instante de liberação do trabalho que contém o , se o for uma tarefa inicial ou o instante de conclusão da tarefa predecessora de o , caso contrário.

^b O número β corresponde ao instante de conclusão da última operação alocada em m ou 0, caso não tenha nenhuma.

Como vimos nas seções anteriores, com o auxílio da estrutura de vizinhança de um problema de otimização, devemos ser capazes de criar soluções viáveis para o problema em tempo polinomial. No contexto do FJSSP, definimos uma estrutura de vizinhança como sendo

$$N_{\text{FJSSP}}(\lambda) := \begin{cases} \{om \in O_\perp \times M \mid m \in \Phi(o)\} & \text{se } \lambda = \epsilon \\ \{om \in (O \times M) \setminus \underline{\lambda} \mid \exists o'm' \in \underline{\lambda}, \sigma(o') = o, m \in \Phi(o)\} & \text{caso contrário} \end{cases}$$

para todo $\lambda \in \mathcal{E}$.

De maneira muito semelhante ao que foi formulado no Exemplo 5.6, definimos uma *estrutura de feromônios*

$$\mathcal{T}_{\text{FJSSP}} := (O \times M, f),$$

onde os pontos de feromônio são pares em $O \times M$, e o mapeamento de feromônio é uma função $f(\lambda, (o, m)) := (o, m)$ para toda possibilidade de extensão $(\lambda, (o, m)) \in E_{\text{FJSSP}}$.

Resta a definição de dois últimos componentes para que seja possível aplicar o ACO neste problema. A função *informação heurística*

$$\eta_{\text{FJSSP}}(\lambda, (o, m)) := \frac{1}{\varrho_o(m)},$$

leva toda possibilidade de extensão $(\lambda, (o, m)) \in E_{\text{FJSSP}}$ ao inverso da duração de o em m . Por fim, dado um ponto de feromônio t e uma sequência $\lambda \in \mathcal{E}$, a função *depósito de feromônio* Δ_{FJSSP}

$$\Delta_{\text{FJSSP}}(t, \lambda) := \begin{cases} C_{\max}(\text{seq2sol}(\lambda))^{-1} & \text{se } t \in f^*(\lambda) \\ 0 & \text{caso contrário,} \end{cases}$$

devolve o inverso do *makespan* da solução gerada por λ se o ponto t foi usado na construção de λ ou 0 caso contrário.

No próximo capítulo, veremos como se deu a implementação dos objetos discutidos até aqui.

6 Implementação

Neste capítulo, descreveremos a implementação dos algoritmos de escalonamento propostos. O código fonte do algoritmo foi desenvolvido em C# 12, utilizando o *framework* .NET 8. De acordo com a documentação oficial da linguagem ([Microsoft, 2024](#)), o C# é uma linguagem multiparadigma, de alto nível e propósito geral, amplamente utilizada no desenvolvimento de softwares multiplataforma. Sua escolha para o processo de implementação se deu visando a integração do algoritmo com a base de código da empresa interessada, bem como a familiaridade do autor com a linguagem.

A seguir, apresentamos o processo de desenvolvimento, detalhando as mudanças entre as versões implementadas. Todo o código-fonte escrito neste trabalho pode ser acessado [neste repositório](#) na plataforma GitHub.

6.1 Representação do problema e instâncias

O código-fonte desenvolvido está distribuído em diferentes projetos. Um **projeto** no contexto do C# é essencialmente um conjunto de arquivos que pode ser compilado em um programa executável ou em uma biblioteca de classes. Especialmente nesta seção, vamos nos ater ao projeto **Scheduling.Core**, que contém, dentre outras coisas, as classes que representam uma instância do FJSSP. As classes apresentadas a seguir se encontram no *namespace* **Scheduling.Core.FJSP**.

A partir deste ponto, considere uma instância $\mathfrak{F} = (J, O, M, r, n, \pi, \Phi, \varrho)$ do FJSSP. Sua contraparte em código é um objeto da classe **Instance**, exibida no Fragmento 6.1, cujas propriedades são coleções de objetos das classes **Job**, **Operation** e **Machine** que codificam, respectivamente, elementos dos conjuntos J , O e M .

Seja $j \in J$ uma tarefa qualquer. Sua representação em código é por meio de um objeto da classe **Job**, que, através da propriedade **ReleaseDate** representa o instante de liberação r_j e, através da propriedade **Operations**, que consiste em um vetor de objetos da classe **Operation**, codifica tanto a quantidade de operações n_j quanto a rota de processamento π_j .

Considere uma operação $o \in O$. Representamos o por meio de um objeto da classe **Operation**. Essa classe mantém em suas propriedades **ProcessingTimes** e **EligibleMachines** as informações necessárias para representar, respectivamente,¹ ϱ_o e $\Phi(o)$. Por fim, por mais óbvio que pareça, representamos uma máquina $m \in M$ através de um objeto da classe **Machine**.

¹ Lembre que ϱ_o é uma função que leva cada $m \in \Phi(o)$ na duração, $\varrho_o(m)$, da operação o na máquina m .

```

public class Instance(IEnumerable<Job> jobs, Machine[] machines)
{
    public IEnumerable<Job> Jobs { get; } = jobs;

    public Machine[] Machines { get; } = machines;

    ...
}

public class Job(int id)
{
    public int Id { get; set; } = id;

    public long ReleaseDate { get; set; }

    public Operation[] Operations { get; set; } = [];
}

public class Machine(int id)
{
    public int Id { get; set; } = id;

    public int Index { get; init; }
}

public class Operation(int id, Dictionary<Machine, long> processingTimes) :
    IEquatable<Operation>
{
    public int Id { get; set; } = id;

    public int JobId { get; set; }

    public int Index { get; set; }

    public Job Job { get; init; }

    public bool FirstOperation => Index < 1;

    public bool LastOperation => Index + 1 >= Job.Operations.Length;

    private Dictionary<Machine, long> ProcessingTimes { get; } = processingTimes;

    public HashSet<Machine> EligibleMachines { get; } = [];

    ...
}

public long GetProcessingTime(Machine m) { ... }
}

```

Fragmento 6.1 – Classes **Instance**, **Job**, **Machine** e **Operation** representantes do FJSSP

Como vimos, uma solução para uma instância \tilde{x} do FJSSP consiste em um par (μ, s) que codifica a alocação de máquina e o escalonamento para cada operação. A forma como esses objetos são construídos varia de acordo com o algoritmo utilizado, portanto,

definimos em código as *interfaces* **IFjspSolution** e **IFlexibleJobShopSchedulingSolver**, exibidas no Fragmento 6.2. A função de **IFlexibleJobShopSchedulingSolver** é estabelecer um contrato: as classes que representam algoritmos para o FJSSP devem implementar um método **Solve** que recebe um objeto da classe **Instance** e devolve um objeto de alguma classe que implementa **IFjspSolution**.

```

public interface IFjspSolution
{
    /// Start time by operation Id
    Dictionary<int, double> StartTimes { get; }

    /// Completion Times by operation Id
    Dictionary<int, double> CompletionTimes { get; }

    /// Machine Assignment by operation Id
    Dictionary<int, int> MachineAssignment { get; }

    /// Loading sequence by machine index
    Dictionary<int, List<Operation>> LoadingSequence { get; }

    public Stopwatch Watch { get; }

    double Makespan { get; }

    ...
}

public interface IFlexibleJobShopSchedulingSolver
{
    IFlexibleJobShopSchedulingSolver WithLogger(ILogger logger, bool with =
        false);

    /// <summary>
    /// Solve a flexible job shop problem instance
    /// </summary>
    /// <returns>A object representing solution with machine alocation and
    /// starting times of each operation</returns>
    IFjspSolution Solve(Instance instance);

    void Log(string message);
}

```

Fragmento 6.2 – Interfaces **IFjspSolution** e **IFlexibleJobShopSchedulingSolver**

A partir de agora, seguindo a cronologia dos fatos, vamos explorar as implementações realizadas no decorrer do desenvolvimento deste trabalho. Ao todo, foram implementadas quatro versões diferentes da heurística responsável por criar soluções viáveis no contexto do ACO (que vamos nos referir por V0, V1, V2 e V3) além de uma implementação da heurística LLM. De forma breve, na Seção 6.2 abordaremos rapidamente os aspectos de implementação das versões V0 e V1 e teceremos comentários sobre a inviabilidade destas implementações e as tentativas de otimizar sua performance. Em seguida, na Seção 6.3, apresentamos as abstrações que permitiram implementar os cinco

algoritmos ACO estudados, bem como o desenvolvimento da versão V2 (que impactou substancialmente a performance do algoritmo). Também exibimos a implementação da heurística LLM utilizada como referência para avaliar a performance e qualidade dos algoritmos. Por fim, na Seção 6.4 falamos rapidamente sobre a versão V3 que surge de uma adaptação de V2 para admitir relações de precedência não lineares. Além disso, nesta seção, falamos rapidamente sobre os testes de unidade desenvolvidos para garantir que todos os 33 algoritmos implementados construam soluções viáveis para o problema.

6.2 Com quantos digrafos disjuntivos se faz um aquecedor elétrico?

O primeiro algoritmo desenvolvido, denominado ACSV0-p, é uma implementação do ACS (vide Seção 5.5.5) com a heurística construtiva V0 em uma abordagem de programação paralela. A versão V0 ataca o problema por meio de um digrafo disjuntivo, seguindo a formulação apresentada em (ROSSI; DINI, 2007) (veja também o Exemplo 4.7). A estratégia consiste em partitionar o conjunto de formigas em segmentos e executar a construção das soluções em paralelo, utilizando múltiplas *threads* de trabalho distribuídas entre os processadores. Esperava-se, com essa abordagem, tempos de execução condizentes com o observado na literatura.

No entanto, testes preliminares evidenciaram a ineficiência dessa primeira versão diante de instâncias de médio porte, como as propostas por Brandimarte (1993). No ACS, cada movimento viável realizado por uma formiga é imediatamente seguido por uma transformação local na estrutura de feromônios e, em uma abordagem paralela, é necessário tomar certos cuidados com o acesso concorrente às estruturas de dados. A primeira hipótese para os resultados obtidos foi um possível *overhead* na execução, decorrente da sincronização no acesso à estrutura de feromônio. Isso motivou a implementação de uma nova versão, denominada ACSV0-i, seguindo uma abordagem iterativa, onde as formigas constroem uma solução viável de cada vez.

Os resultados que serão apresentados na Seção 7.3.1 invalidam essa hipótese. Veremos que o algoritmo ACSV0-i não só apresentou resultados piores que ACSV0-p, como, em alguns casos, se mostrou três vezes mais lento. Contudo, o palpite não foi de todo ruim. A Figura 20 apresenta uma captura do perfilador de desempenho presente na IDE Visual Studio, que ajudou a constatar que o problema de fato estava associado aos movimentos viáveis: quase 90% do tempo de CPU de ACSV0-p e ACSV0-i é gasto enumerando os movimentos viáveis.

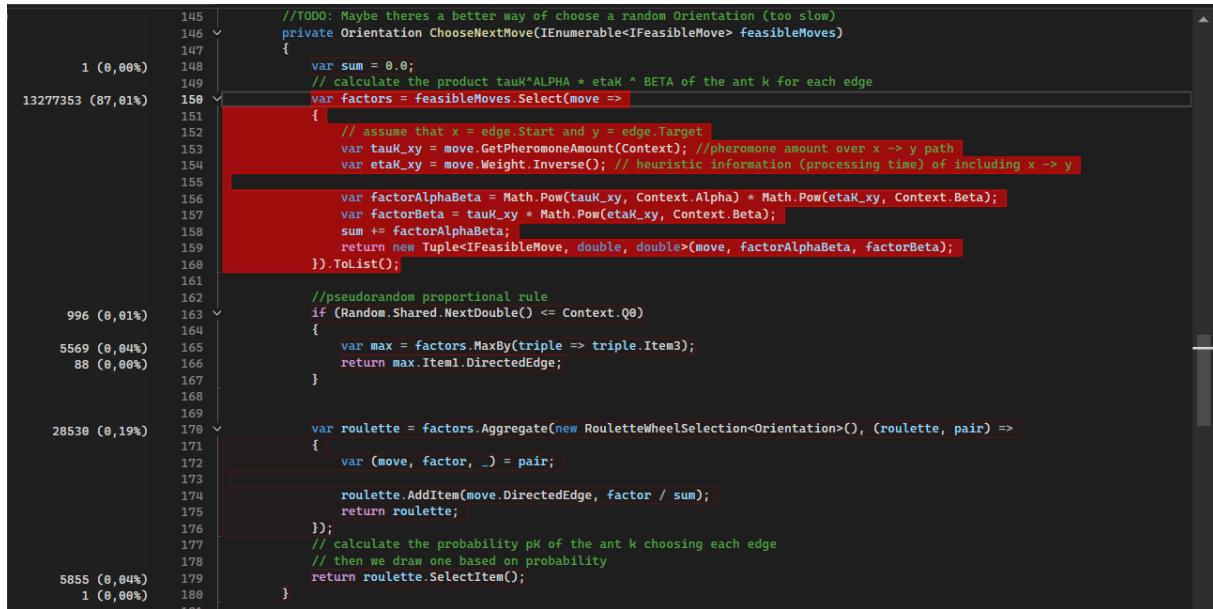


Figura 20 – Relatório do perfilador de desempenho indicando o gargalo de execução em ACSV0-p

Há duas causas para este problema. A primeira delas, mais simples de perceber, foi o fato de que a cada passo de construção, como bem mostra o Fragmento 6.3, uma busca acontece sobre o conjunto das disjunções. Isso se agrava, pois, neste modelo de digrafo disjuntivo, o conjunto das disjunções cresce muito rápido em relação ao tamanho da instância, o que nos leva à segunda causa do problema: o modelo teórico adotado se mostrou excessivamente complicado para os nossos propósitos, prejudicando a implementação. A solução deste problema, no entanto, será abordada mais adiante.

```

private IEnumerable<IFeasibleMove<Orientation>>
    GetFeasibleMoves(IEnumerable<Node> candidateNodes)
{
    var lastScheduledNodes = LoadingSequence.Values.Select(sequence =>
        sequence.Peek());
    var disjunctiveMoves = candidateNodes.SelectMany(candidateNode =>
    {
        return lastScheduledNodes.SelectMany(lastScheduledNode =>
        {
            var intersection = lastScheduledNode.IncidentDisjunctions
                .Intersect(candidateNode.IncidentDisjunctions);
            return intersection.Select(disjunction =>
            {
                var direction = disjunction.Target == candidateNode
                    ? Direction.SourceToTarget
                    : Direction.TargetToSource;
                return new FeasibleMove(disjunction, direction);
            });
        });
    });
    return disjunctiveMoves;
}

```

Fragmento 6.3 – Cálculo dos movimentos viáveis nos algoritmos ACSV0-p e ACSV0-i

A fim de resolver o desperdício de recursos computacionais, uma nova versão da heurística construtiva, denominada V1, foi desenvolvida e aplicada no ACS. O Fragmento 6.4 exibe a principal modificação envolvendo a rotina de cálculo de movimentos viáveis. As coleções **unscheduledNodes** e **scheduledNodes** são atualizadas pelo algoritmo a cada iteração, para evitar que sejam calculadas a cada movimento viável.

```
public IEnumerable<I可行Move<Orientation>> GetFeasibleMoves(HashSet<Node>
    unscheduledNodes, HashSet<Node> scheduledNodes)
{
    return unscheduledNodes.SelectMany(candidateNode =>
    {
        return candidateNode.IncidentDisjunctions
            .Where(disjunction =>
                scheduledNodes.Contains(disjunction.Other(candidateNode)))
            .Select(disjunction => new FeasibleMove(disjunction,
                disjunction.Target == candidateNode ?
                    Direction.SourceToTarget :
                    Direction.TargetToSource));
    });
}
```

Fragmento 6.4 – Implementação do cálculo dos movimentos viáveis na versão V1

A essa altura, o trabalho havia ganho um nível de maturidade e abstração que permitiu trabalhar com mais esmero na implementação. Por um lado, o desenvolvimento do modelo abstrato para algoritmos de colônia de formigas (Capítulo 5) impulsionou a implementação de novas versões do ACO. Por outro, o domínio sobre o problema permitiu uma nova implementação da heurística de construção de soluções, que veremos em detalhes na próxima seção.

Também veremos detalhadamente como os cinco algoritmos ACO foram implementados na próxima seção. Por hora, o que nos interessa é o desempenho deles junto à versão V1. Os resultados apresentados no final da Seção 7.3.1 evidenciam que, dentre as 10 implementações baseadas na versão V1 (resultantes das abordagens iterativas e paralelas dos 5 algoritmos ACO), apesar de algumas se mostrarem mais rápidas que as implementações com V0, no geral, todas continuaram performando abaixo do esperado em termos de tempo de execução. Por outro lado, veremos que as soluções obtidas por V0 eram tão boas quanto as geradas por V1, o que também não correspondeu ao esperado.

Não desperdiçaremos mais o tempo do leitor falando sobre os detalhes dessas implementações neste capítulo (retomaremos posteriormente para uma análise mais aprofundada dos resultados). Sua menção aqui é meramente um registro do processo de desenvolvimento, de modo que os detalhes valem a pena falar e que aqui foram negligenciados, serão devidamente resgatados na próxima seção.

6.3 De volta ao básico

Sem mais delongas, o Fragmento 6.5 apresenta a implementação da heurística LLM-FJSSP (Algoritmo 7), que abriu as portas para que a versão V2 da heurística construtiva do ACO fosse desenvolvida. Os resultados obtidos, disponíveis na Seção 7.3.2, foram animadores, pois essa implementação se mostrou extremamente econômica tanto no consumo de recursos quanto no tempo de execução para as instâncias testadas. Além do mais, a estrutura do algoritmo lembra muito os algoritmos 9 e 10 vistos no Capítulo 5, o que incentivou a abandonar o modelo baseado em digrafos disjuntivos.

```

        solution.CriticalPredecessors[operation.Id] =
            criticalPredecessor;
        startTime = Math.Max(startTime,
            criticalPredecessorCompletionTime);
    }

    var completionTime = startTime +
        operation.GetProcessingTime(machine);
    solution.StartTimes.TryAdd(operation.Id, startTime);
    solution.CompletionTimes.TryAdd(operation.Id, completionTime);

    // updating data structures
    solution.LoadingSequence[machine.Index].Add(operation);
    solution.MachineOccupancy[machine] = completionTime;
    solution.MachineAssignment.Add(operation.Id, machine.Index);

    if (operation.LastOperation)
        unscheduledJobOperations.Remove(operation.Job);
    else
        unscheduledJobOperations[operation.Job] =
            operation.Job.Operations[operation.Index + 1];
}

solution.Watch.Stop();
Log($"Solution found after {solution.Watch.Elapsed}.");
return solution;
}

private (Operation, Machine) GetGreedyMachineAllocation(Dictionary<Job,
    Operation> unscheduledJobOperations, GreedySolution solution)
{
    var candidateAllocations = unscheduledJobOperations.Values
        .SelectMany(operation => operation.EligibleMachines
            .Select(machine => (operation, machine)));
    return candidateAllocations.MinBy(om =>
        solution.MachineOccupancy[om.machine] +
        om.operation.GetProcessingTime(om.machine)
    );
}
}

```

Fragmento 6.5 – Implementação da heurística LLM-FJSSP

6.3.1 Ant against the machine

Ao todo, foram realizadas neste trabalho 32 implementações de algoritmos ACO para resolver o FJSSP, além da implementação da heurística LLM-FJSSP². O leitor com alguma experiência em desenvolvimento de *software* pode imaginar que o gerenciamento e a manutenção disso pode vir a se tornar um verdadeiro desafio, tendo em vista que boa parte do código se repete. Isso motivou um trabalho de abstração de código, seguindo

² Com exceção de V0 que foi implementada apenas para o ACS, contabilizamos as implementações iterativas e paralelas de cada um dos 5 algoritmos ACO estudados para com as versões V1, V2 e V3.

princípios de desenvolvimento e padrões de projeto que serão comentados nesta seção. A Figura 21 ilustra em um diagrama as classes e interfaces que tornaram possível o desenvolvimento de forma escalável e organizada. Por uma questão de brevidade, abordaremos apenas as especializações no contexto de V2, deixando o leitor livre para consultar o código desenvolvido.

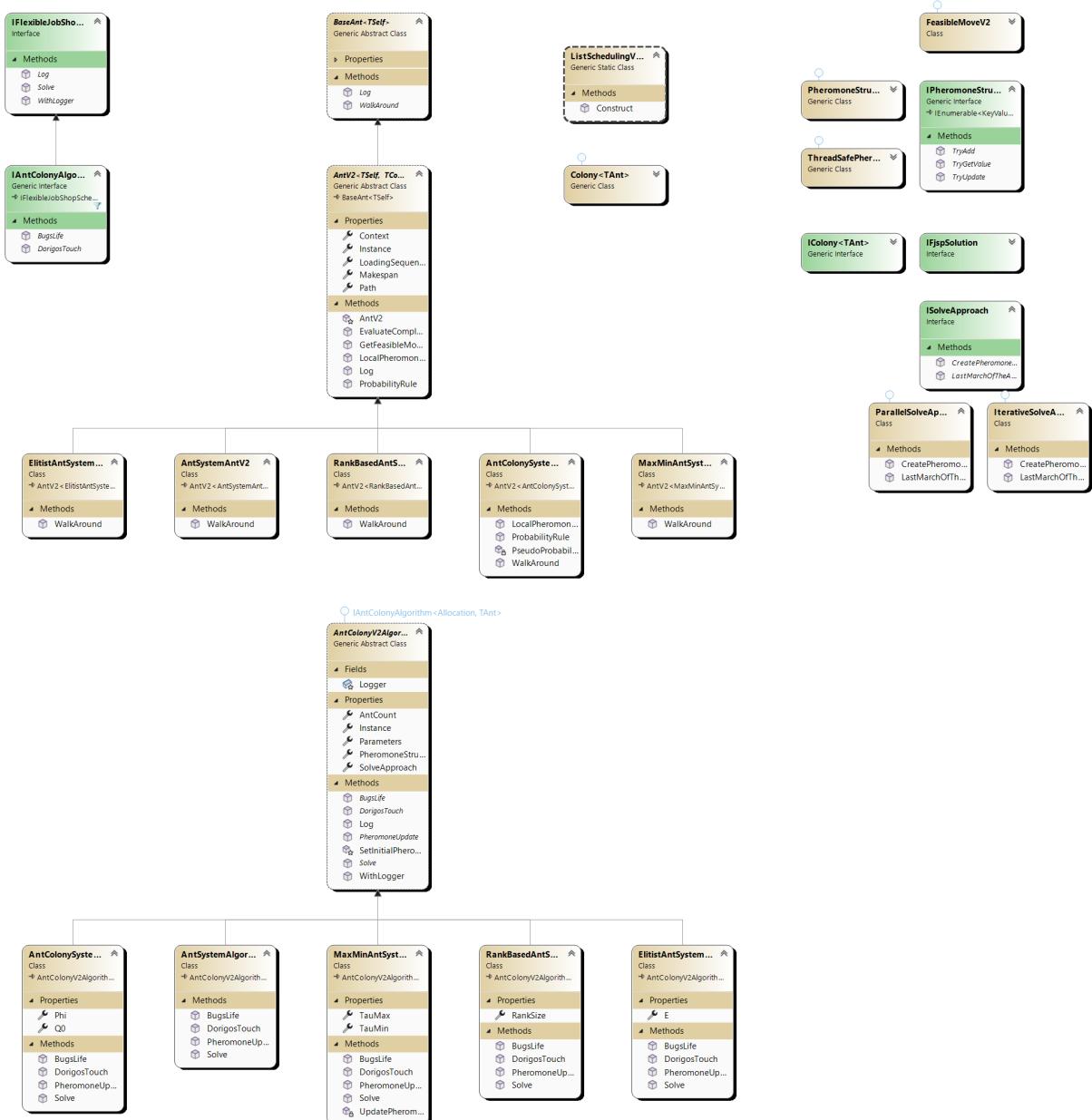


Figura 21 – Diagrama de classes do `namespace Scheduling.Solver`

Todo código que vem a seguir será descrito considerando aspectos do modelo teórico apresentado na Seção 5.7. Começando pela estrutura de feromônios $\mathcal{F}_{JSSP} := (O \times M, f)$, nos deparamos com um dilema de ordem prática. Uma vez que a estrutura de feromônios é uma estrutura de dados compartilhada entre as formigas, as implementações que seguem uma abordagem paralela (são elas ASV2-p, EASV2-p, RBASV2-p,

MMASV2-p e ACSV2-p) precisam que esta seja uma estrutura de dados *thread-safe*³. Isso motivou a seguinte abstração (exibida no Fragmento 6.6): a interface **IPheromoneStructure** estabelece o contrato das operações básicas sobre a estrutura de feromônios. A classe **ThreadSafePheromoneStructure** é uma implementação de **IPheromoneStructure** que utiliza internamente um **ConcurrentDictionary<TPheromonePoint, double>** para garantir a consistência das operações realizadas em abordagens paralelas. Já a classe **PheromoneStructure** é a implementação para abordagens iterativas, que dispensa a necessidade de sincronização e, portanto, utiliza como estrutura de dados um simples **Dictionary<TPheromonePoint, double>**. A escolha de dicionários para representar a estrutura de feromônios se deu pela possibilidade de codificar tanto o conjunto de pontos de feromônios quanto a função concentração de feromônio em um mesmo objeto. O uso do tipo genérico **TPheromonePoint** permite que essa estrutura de dados trabalhe com qualquer tipo de dado, refletindo assim a abstração também presente no modelo teórico.

```

public interface IPheromoneStructure<TPheromonePoint>:
    IEnumerable<KeyValuePair<TPheromonePoint, double>>
{
    bool TryAdd(TPheromonePoint key, double value);

    bool TryGetValue(TPheromonePoint key, out double value);

    bool TryUpdate(TPheromonePoint key, double newValue, double comparisonValue);
}

public class PheromoneStructure<TPheromonePoint> :
    IPheromoneStructure<TPheromonePoint> where TPheromonePoint : notnull
{
    private readonly Dictionary<TPheromonePoint, double> _trail = [];
    ...
}

public class ThreadSafePheromoneStructure<TPheromonePoint> :
    IPheromoneStructure<TPheromonePoint> where TPheromonePoint : notnull
{
    private readonly ConcurrentDictionary<TPheromonePoint, double> _trail =
        new();
    ...
}

```

Fragmento 6.6 – Implementação da estrutura de feromônio

Seguindo com as boas práticas e padrões de desenvolvimento, aplicamos o princípio da inversão de dependência⁴ em tudo que tange o gerenciamento da abordagem de execução. A ideia é que as implementações dos algoritmos sejam agnósticas à abordagem de execução, isto é, não tenham que se preocupar se as formigas estão sendo lançadas

³ Grosso modo, uma estrutura de dados é dita *thread-safe* se seu uso compartilhado por múltiplas *threads* é feito de uma forma que garanta consistência nas operações realizadas.

⁴ Por motivos óbvios, não nos aprofundaremos nas bases teóricas deste e de nenhum outro postulado SOLID. Para isso, consulte (MARTIN, 2017).

sequencialmente ou em paralelo. Para que isso seja possível, os algoritmos dependem de uma interface chamada **ISolveApproach** (recebida por meio de injeção de dependência), que selo o contrato para implementação das classes concretas que gerenciam as diferentes abordagens de execução. São elas: **IterativeSolveApproach** e **ParallelSolveApproach**, exibidas no Fragmento 6.7. O método **CreatePheromoneTrail** cria a estrutura de feromônios adequada para a abordagem (uma instância de **PheromoneStructure** ou de **ThreadSafePheromoneStructure**) e o método **LastMarchOfTheAnts** orquestra (sequencialmente ou paralelamente) a atuação das formigas na rotina de construção de soluções.

```

public interface ISolveApproach
{
    IPheromoneStructure<TPheromonePoint> CreatePheromoneTrail<TPheromonePoint>()
        where TPheromonePoint : notnull;

    TAnt[] LastMarchOfTheAnts<TPheromonePoint, TAnt>(int currentIteration,
        IAntColonyAlgorithm<TPheromonePoint, TAnt> solverContext,
        Func<int, int, TAnt> bugSpawner) where TAnt : BaseAnt<TAnt>;
}

public class IterativeSolveApproach : ISolveApproach
{
    public IPheromoneStructure<TPheromonePoint>
        CreatePheromoneTrail<TPheromonePoint>() where TPheromonePoint : notnull
    => new PheromoneStructure<TPheromonePoint>();

    public TAnt[] LastMarchOfTheAnts<TPheromonePoint, TAnt>(int
        currentIteration, IAntColonyAlgorithm<TPheromonePoint, TAnt>
        solverContext, Func<int, int, TAnt> bugSpawner) where TAnt :
        BaseAnt<TAnt>
    {
        var ants = new TAnt[solverContext.AntCount];
        solverContext.Log($"#{currentIteration}th wave ants start to walk...");
        for (int j = 0; j < solverContext.AntCount; j++)
        {
            var antId = j + 1;
            ants[j] = bugSpawner(antId, currentIteration);
            ants[j].WalkAround();
        }
        return ants;
    }
}

public class ParallelSolveApproach : ISolveApproach
{
    public IPheromoneStructure<TPheromonePoint>
        CreatePheromoneTrail<TPheromonePoint>() where TPheromonePoint : notnull
    => new ThreadSafePheromoneStructure<TPheromonePoint>();

    public TAnt[] LastMarchOfTheAnts<TPheromonePoint, TAnt>(
        int currentIteration,
        IAntColonyAlgorithm<TPheromonePoint, TAnt> solverContext,
        Func<int, int, TAnt> bugSpawner) where TAnt : BaseAnt<TAnt>
    {
        var ants = Enumerable.Range(0, solverContext.AntCount)
            .Select(i => bugSpawner(i + 1,

```

```

        currentIteration)).ToArray();
solverContext.Log(${currentIteration}th wave ants start to walk...");
return ants.AsParallel()
    .WithDegreeOfParallelism(Environment.ProcessorCount)
    .Select(a =>
{
    a.WalkAround();
    return a;
})
.ToArray();
}

```

Fragmento 6.7 – Implementação do gerenciador de abordagem de execução

Vimos no Capítulo 5 que o modelo teórico do ACO gira em torno das noções de formiga e de colônia. Recapitulando, esses conceitos, considere um problema de otimização \mathfrak{P} e um conjunto de pontos de feromônio T . Uma formiga apropriada para \mathfrak{P}, T é uma tripla formada por uma função de *mapeamento de feromônio* p , uma função *movimento viável* $move$ e uma função de *transformação local de feromônio* ℓ . Já uma colônia para \mathfrak{P}, T é a 4-upla formada por um conjunto A de *formigas* apropriadas para \mathfrak{P}, T , uma função *concentração de feromônios* inicial f_0 , uma função *transformação global de feromônio* g e uma função *seletora de soluções* h .

Na implementação, esses objetos teóricos foram codificados através da classe abstrata **BaseAnt** e da interface⁵ **IAntColonyAlgorithm<TPheromonePoint, out TAnt>**, ambas exibidas no Fragmento 6.8. No caso de **BaseAnt**, vale a pena mencionar o método abstrato **WalkAround()**, que, como veremos mais adiante, força uma implementação de walk (Algoritmo 10). Já a interface **IAntColonyAlgorithm**, que é uma extensão da interface **IFlexibleJobShopSchedulingSolver**, força a implementação dos métodos **DorigosTouch** (onde ocorre a parametrização de cada algoritmo conforme discutido no Capítulo 7) e **BugsLife**, que invocará o método **LastMarchOfTheAnts** do gerenciador de abordagem de execução (recebido por injeção de dependência), passando como argumento um método construtor de formigas (através de um *delegate*⁶).

```

public abstract class BaseAnt<TSelf> where TSelf : BaseAnt<TSelf>
{
    public int Id { get; init; }
    public int Generation { get; init; }
    public IFjspSolution Solution { get; set; } = new AntSolution();
    public abstract void WalkAround();
    ...
}

```

⁵ A palavra reservada **out** aparece no contexto de covariância de tipos genéricos. Consulte [este link](#) para mais detalhes

⁶ Um delegate é um elemento da linguagem C# que permite fazer referência a um método. De forma simplória, podemos encará-lo o equivalente a ponteiros para funções nesta linguagem.

```

public interface IAntColonyAlgorithm<TPheromonePoint, out TAnt> :
    IFlexibleJobShopSchedulingSolver
{
    int AntCount { get; }

    /// <summary>
    /// Run (DORIGO; STUTZLE, 2004) parameter settings for ACO algorithms
    /// without local search
    /// </summary>
    /// <param name="instance"></param>
    void DorigosTouch(Instance instance);

    /// <summary>
    /// Pheromone trail data structure
    /// </summary>
    IPheromoneStructure<TPheromonePoint> PheromoneStructure { get; }

    TAnt[] BugsLife(int currentIteration);
}

```

Fragmento 6.8 – Abstrações **BaseAnt** e **IAntColonyAlgorithm**

A classe **Colony** é responsável por manter algumas informações importantes ao longo da execução do algoritmo. Como podemos ver no Fragmento 6.9, além de guardar métricas de execução, um objeto desta classe mantém uma referência para uma melhor formiga de cada iteração no dicionário **IterationBests**, bem como uma referência, na propriedade **EmployeeOfTheMonth**, para uma formiga que encontrou uma melhor solução dentre todas as avaliadas. O método **UpdateBestPath** é invocado a cada iteração para atualizar o estado deste objeto e ajudar a controlar o critério de parada do algoritmo, com a propriedade **LastProductiveWave**.

```

public class Colony<TAnt> : IColony<TAnt> where TAnt : BaseAnt<TAnt>
{
    /// Ant who found the best path
    public TAnt? EmployeeOfTheMonth { get; private set; }

    public Dictionary<int, TAnt> IterationBests { get; } = [];

    public TAnt BestSoFar => IterationBests.MinBy(a =>
        a.Value.Solution.Makespan).Value;

    public int LastProductiveWave { get; private set; } = 0;

    public Stopwatch Watch { get; } = new();

    public void UpdateBestPath(TAnt[] ants)
    {
        foreach (var ant in ants)
        {
            IterationBests.TryAdd(ant.Generation, ant);

            if (ant.Solution.Makespan <
                IterationBests[ant.Generation].Solution.Makespan)

```

```

        IterationBests[ant.Generation] = ant;

        var hasBestSolution = EmployeeOfTheMonth is not null;
        var antFoundBetterPath = ant.Solution.Makespan <
            EmployeeOfTheMonth?.Solution.Makespan;
        if (hasBestSolution && !antFoundBetterPath) continue;

        EmployeeOfTheMonth = ant;
        LastProductiveGeneration = ant.Generation;
    }
}

```

Fragmento 6.9 – Classe **Colony** responsável por manter o estado do algoritmo

Toda essa estrutura é utilizada pelas 32 implementações de algoritmos ACO. Pensando somente nas 10 sobre a versão V2, encontramos um nível intermediário de abstração. As estruturas de dados e métodos reutilizáveis se encontram nas classes abstratas **AntV2<TSelf, TContext>** e **AntColonyV2AlgorithmSolver<TSelf, TAnt>**, possibilitando, através de herança e sobrescrita⁷, que cada versão implemente somente o necessário. O Fragmento 6.10 nos permite observar algumas coisas:

- O método **SetInitialPheromoneAmount** representa f_0 , uma vez que, usa o gerenciador da abordagem de execução para criar a estrutura de feromônios e definir seu valor inicial. No caso da versão V2, os pontos de feromônio são instâncias do *record Allocation* (que associa uma instância de **Operation** a uma de **Machine**). Isso condiz com a escolha do conjunto de pares $O \times M$ como pontos de feromônio na Seção 5.7);
- O método **PheromoneUpdate** é definido como um método abstrato para forçar a implementação por cada versão concreta do algoritmo. Esse método contempla a função seletora de soluções h e a função transformação global de feromônio g no modelo teórico;
- O método **LocalPheromoneUpdate**, que contempla a função de transformação local de feromônio ℓ no modelo teórico, é definido como um método virtual com uma implementação padrão vazia. Isso permitirá, como veremos na Subseção 6.3.1.5, que seja sobreescrito somente quando necessário (na Seção 5.5 vemos que somente o algoritmo ACS faz uso da atualização local);
- O método **ProbabilityRule** é um método virtual que fornece uma implementação padrão com o código referente ao algoritmo probrule (Algoritmo 12). O fato dele ser

⁷ Para amarrar as abstrações e evitar *type castings*, foi utilizado um *pattern* chamado **padrão de templates curiosamente recorrente** (do inglês, *curiously recurring template pattern* - CRTP). Esse padrão possibilita o uso de polimorfismo em tempo de compilação, de modo que uma classe base pode invocar membros de classes derivadas sem a necessidade de conhecê-las.

virtual permitirá que seja sobreescrito pela classe **AntColonySystemAntV2** (também na Subseção 6.3.1.5). Essa classe faz parte de uma implementação do algoritmo ACS, que, como vimos, depende do Algoritmo 13. Lembre que, tanto o Algoritmo 12 quanto o Algoritmo 13 são instâncias de move;

- O método **GetFeasibleMoves** é essencialmente uma codificação da estrutura de vizinhança N_{FJSSP} associada ao problema (Seção 5.7). A função N_{FJSSP} recebe uma solução parcial λ e devolve o conjunto dos componentes que podem estender λ . O método **GetFeasibleMoves** opera de forma diferente. Ele recebe uma coleção de operações não escalonadas e devolve uma coleção de pares da forma (o, m) , onde o é uma operação não escalonada e m é uma máquina compatível com o , de tal forma que (o, m) pode estender a solução parcial construída até o momento (veja o Caso 2 do Algoritmo 7);
- O método abstrato **BugsLife** equivale ao início do Caso 2 no algoritmo aco (Algoritmo 11) do modelo teórico, onde o conjunto

$$\Sigma' = \{\text{walk}(\mathfrak{P}, \eta, \text{ant}, f) \mid \text{ant} \in A\}$$

de soluções viáveis é construído. O conjunto A de formigas é construído sob demanda com auxílio do delegate **BugSpawner**.

```

public abstract class AntColonyV2AlgorithmSolver<TSelf, TAnt>(Parameters
    parameters, ISolveApproach solveApproach) :
    IAntColonyAlgorithm<Allocation, TAnt> where TSelf :
        AntColonyV2AlgorithmSolver<TSelf, TAnt> where TAnt : BaseAnt<TAnt>
{
    ...
    public abstract void DorigosTouch(Instance instance);

    public IPheromoneStructure<Allocation> PheromoneStructure { get; protected
        set; }

    public abstract IFjspSolution Solve(Instance instance);

    public abstract TAnt[] BugsLife(int currentIteration);

    protected void SetInitialPheromoneAmount(double amount)
    {
        PheromoneStructure = solveApproach.CreatePheromoneTrail<Allocation>();

        foreach (var job in Instance.Jobs)
            foreach (var operation in job.Operations)
                foreach (var machine in operation.EligibleMachines)
                    if (!PheromoneStructure.TryAdd(new Allocation(operation,
                        machine), amount))
                        Log($"Error on adding pheromone over
                            O{operation.Id}M{machine.Id}");
    }
}

```

```
public abstract void PheromoneUpdate(IColony<TAnt> colony, TAnt[] ants, int
    currentIteration);

}

public abstract class AntV2<TSelf, TContext> : BaseAnt<TSelf>
    where TSelf : AntV2<TSelf, TContext>
    where TContext : AntColonyV2AlgorithmSolver<TContext, TSelf>
{
    protected AntV2(int id, int generation, TContext context)
    {
        Id = id;
        Generation = generation;
        Context = context;
    }

    public TContext Context { get; }

    public Instance Instance => Context.Instance;

    public HashSet<Allocation> Path { get; } = [];

    public double Makespan => Solution.Makespan;

    ...

    public virtual void LocalPheromoneUpdate(Allocation selectedMove) { }

    public virtual IFeasibleMove<Allocation>
        ProbabilityRule(IEnumerable<IFeasibleMove<Allocation>> feasibleMoves)
    {
        var sum = 0.0;
        var rouletteWheel = new List<(IFeasibleMove<Allocation> Move, double
            Probability)>();

        // create roulette wheel
        foreach (var move in feasibleMoves)
        {
            var tauXY = move.GetPheromoneAmount(Context.PheromoneStructure);
            var etaXY = move.Weight.Inverse(); // heuristic information
            var tauXYAlpha = Math.Pow(tauXY, Context.Parameters.Alpha);
            var etaXYBeta = Math.Pow(etaXY, Context.Parameters.Beta);

            double probFactor = tauXYAlpha * etaXYBeta;
            rouletteWheel.Add((move, probFactor));
            sum += probFactor;
        }

        // roulette wheel
        var cumulative = 0.0;
        var randomValue = Random.Shared.NextDouble() * sum;
        foreach (var (move, probability) in rouletteWheel)
        {
            cumulative += probability;
            if (randomValue <= cumulative)
                return move;
        }
    }
}
```

```

        throw new InvalidOperationException("FATAL ERROR: No move was
            selected.");
    }

    public IEnumerable<I可行Move<Allocation>>
    GetFeasibleMoves(HashSet<Operation> unscheduledNodes)
    {
        return unscheduledNodes.SelectMany(candidateNode =>
            candidateNode.EligibleMachines.Select(m => new
                FeasibleMoveV2(candidateNode, m)));
    }

    public void EvaluateCompletionTime(AntSolution antSolution, Allocation
        selectedMove)
    {
        // evaluate start and completion times
        List<Operation> predecessors = [];
        // job predecessor (non initial operations)
        if (!selectedMove.Operation.FirstOperation)
            predecessors.Add(selectedMove.Operation.Job
                .Operations[selectedMove.Operation.Index - 1]);
        // machine predecessor
        if (antSolution.LoadingSequence[selectedMove.Machine.Index].Any())
            predecessors.Add(antSolution
                .LoadingSequence[selectedMove.Machine.Index].Last());
        // s(o) \leq r_j \forall o \in \underline{\pi_j}
        var startTime = Convert.ToDouble(selectedMove.Operation.Job.ReleaseDate);
        if (predecessors.Any())
        {
            var criticalPredecessor = predecessors.MaxBy(p =>
                antSolution.CompletionTimes[p.Id]);
            var criticalPredecessorCompletionTime = criticalPredecessor != null
                ? antSolution.CompletionTimes[criticalPredecessor.Id]
                : 0;
            antSolution.CriticalPredecessors[selectedMove.Operation.Id] =
                criticalPredecessor;
            startTime = Math.Max(startTime, criticalPredecessorCompletionTime);
        }

        Path.Add(selectedMove);
        antSolution.StartTimes.TryAdd(selectedMove.Operation.Id, startTime);
        antSolution.CompletionTimes.TryAdd(selectedMove.Operation.Id, startTime
            + selectedMove.Operation.GetProcessingTime(selectedMove.Machine));
        antSolution.LoadingSequence[selectedMove.Machine.Index]
            .Add(selectedMove.Operation);
    }
}

```

Fragmento 6.10 – Implementações de **AntV2** e **AntColonyV2AlgorithmSolver** que concentram o código compartilhado entre as versões que usam a implementação V2

Isso tudo permite que qualquer um dos algoritmos ACO estudados seja implementado em código de forma objetiva. As subseções 6.3.1.1, 6.3.1.2, 6.3.1.3, 6.3.1.4 e 6.3.1.5 exibem, respectivamente, o código referente às versões paralela e iterativa dos

algoritmos AS, EAS, RBAS, MMAS e ACS baseados na heurística construtiva V2. Note que cada implementação do método **Solve**, que codifica grande parte do algoritmo aco (Algoritmo 11), apenas invoca os métodos mencionados anteriormente, controlando apenas o critério de parada. Note também que cada classe concreta fornece apenas as implementações necessárias, mantendo o código desacoplado e abstraído, como veremos a seguir.

6.3.1.1 Implementação do sistema de formigas

Todo o esforço teórico e prático em abstrações facilitou uma codificação minimalista dos algoritmos, como pode ser observado na implementação do sistema de formigas, introduzido na Seção 5.5.1.

```

public class AntSystemAntV2(int id, int generation, AntSystemAlgorithmV2
    context) : AntV2<AntSystemAntV2, AntSystemAlgorithmV2>(id, generation,
    context)
{
    public override void WalkAround()
    {
        ListSchedulingV2Heuristic<AntSystemAlgorithmV2,
            AntSystemAntV2>.Construct(this);
    }
}

public class AntSystemAlgorithmV2(Parameters parameters, ISolveApproach
    solveApproach) :
    AntColonyV2AlgorithmSolver<AntSystemAlgorithmV2, AntSystemAntV2>(parameters,
        solveApproach)
{
    public override void DorigosTouch(Instance instance)
    {
        Parameters.Alpha = 1;
        Parameters.Rho = 0.5;
        AntCount = instance.Jobs.Count();
        Parameters.Tau0 = AntCount.DividedBy(instance.UpperBound);
    }
}

public override IFjspSolution Solve(Instance instance)
{
    Instance = instance;
    DorigosTouch(instance);
    Stopwatch iSw = new();
    Colony<AntSystemAntV2> colony = new();
    colony.Watch.Start();
    SetInitialPheromoneAmount(Parameters.Tau0);
    Log($"Depositing {Parameters.Tau0} pheromone units over
        {PheromoneStructure.Count()} machine-operation pairs...");
    for (int i = 0; i < Parameters.Iterations; i++)
    {
        var currentIteration = i + 1;
        Log($"Generating {AntCount} artificial ants from
            #{currentIteration}th wave...");
        iSw.Restart();
        var ants = BugsLife(currentIteration);
    }
}

```

```

    iSw.Stop();
    colony.UpdateBestPath(ants);
    PheromoneUpdate(colony, ants, currentIteration);

    var generationsSinceLastImprovement = i - colony.LastProductiveWave;
    if (generationsSinceLastImprovement >
        Parameters.StagnantGenerationsAllowed)
    {
        Log($"\\n\\nDeath by stagnation...");
        break;
    }
}
colony.Watch.Stop();

Log($"Every ant has stopped after {colony.Watch.Elapsed}.");
Log("\\nFinishing execution...");

if (colony.EmployeeOfTheMonth is not null)
    Log($"Better solution found by ant {colony.EmployeeOfTheMonth?.Id}
        on #{colony.EmployeeOfTheMonth?.Generation}th wave!");

AntColonyOptimizationSolution<AntSystemAntV2> solution = new(colony);
Log($"Makespan: {solution.Makespan}");

return solution;
}

public override void PheromoneUpdate(IColony<AntSystemAntV2> colony,
    AntSystemAntV2[] ants, int currentIteration)
{
    foreach (var (allocation, currentPheromoneAmount) in PheromoneStructure)
    {
        var antsUsingAllocation = ants.Where(ant =>
            ant.Path.Contains(allocation));

        // if the ant is not using this allocation, then its contribution to
        // delta is 0
        var delta = antsUsingAllocation.Sum(ant => ant.Makespan.Inverse());
        var updatedAmount = (1 - Parameters.Rho) * currentPheromoneAmount + delta;

        if (!PheromoneStructure.TryUpdate(allocation, updatedAmount,
            currentPheromoneAmount))
            Log($"Offline Update pheromone failed on {allocation}");
    }
}

public override AntSystemAntV2[] BugsLife(int currentIteration)
{
    AntSystemAntV2 BugSpawner(int id, int generation) => new(id, generation,
        this);

    return SolveApproach.LastMarchOfTheAnts(currentIteration, this,
        BugSpawner);
}

```

6.3.1.2 Implementação do sistema de formigas elitista

O sistema de formigas elitista traz diferenças na implementação do método **PheromoneUpdate**, conforme as justificativas apresentadas na Seção 5.5.2.

```

public class ElitistAntSystemAntV2(int id, int generation,
    ElitistAntSystemAlgorithmV2 context) : AntV2<ElitistAntSystemAntV2,
    ElitistAntSystemAlgorithmV2>(id, generation, context)
{
    public override void WalkAround()
    {
        ListSchedulingV2Heuristic<ElitistAntSystemAlgorithmV2,
            ElitistAntSystemAntV2>.Construct(this);
    }
}

public class ElitistAntSystemAlgorithmV2(Parameters parameters, double e,
    ISolveApproach solveApproach) :
    AntColonyV2AlgorithmSolver<ElitistAntSystemAlgorithmV2,
        ElitistAntSystemAntV2>(parameters, solveApproach)
{
    /// Elitist weight
    public double E { get; protected set; } = e;

    public override void DorigosTouch(Instance instance)
    {
        Parameters.Alpha = 1;
        Parameters.Rho = 0.5;
        AntCount = instance.Jobs.Count();
        E = AntCount;
        Parameters.Tau0 = (E + AntCount).DividedBy(Parameters.Rho *
            instance.UpperBound);
    }

    public override void PheromoneUpdate(IColony<ElitistAntSystemAntV2> colony,
        ElitistAntSystemAntV2[] ants, int currentIteration)
    {
        var bestSoFarSolution = colony.BestSoFar.Path;
        var bestSoFarDelta = colony.BestSoFar.Makespan.Inverse();
        foreach (var (allocation, currentPheromoneAmount) in PheromoneStructure)
        {
            var antsUsingAllocation = ants.Where(ant =>
                ant.Path.Contains(allocation));

            // if the ant is not using this allocation, then its contribution to
            // delta is 0
            var delta = antsUsingAllocation.Sum(ant => ant.Makespan.Inverse());
            var elitistReinforcement = bestSoFarSolution.Contains(allocation) ?
                bestSoFarDelta : 0;
            var updatedAmount = (1 - Parameters.Rho) * currentPheromoneAmount +
                delta + E * elitistReinforcement;

            if (!PheromoneStructure.TryUpdate(allocation, updatedAmount,
                currentPheromoneAmount))
                Log($"Offline Update pheromone failed on {allocation}");
        }
    }
}

```

```

public override IFjspSolution Solve(Instance instance) { [...] }

public override ElitistAntSystemAntV2[] BugsLife(int currentIteration)
{ [...] }

```

Fragmento 6.12 – Especializações que dão origem a EASV2-p e EASV2-i

6.3.1.3 Implementação do sistema de formigas com classificação

Conforme a Seção 5.5.3, o sistema de formigas com classificação é outra versão com poucas modificações em relação ao sistema de formigas.

```

public class RankBasedAntSystemAntV2(int id, int generation,
    RankBasedAntSystemAlgorithmV2 context) : AntV2<RankBasedAntSystemAntV2,
    RankBasedAntSystemAlgorithmV2>(id, generation, context)
{
    public override void WalkAround()
    {
        ListSchedulingV2Heuristic<RankBasedAntSystemAlgorithmV2,
            RankBasedAntSystemAntV2>.Construct(this);
    }
}

public class RankBasedAntSystemAlgorithmV2(Parameters parameters, int
    rankSize, ISolveApproach solveApproach) :
    AntColonyV2AlgorithmSolver<RankBasedAntSystemAlgorithmV2,
    RankBasedAntSystemAntV2>(parameters, solveApproach)
{
    public int RankSize { get; protected set; } = rankSize;
    public override void DorigosTouch(Instance instance)
    {
        AntCount = instance.Jobs.Count();
        Parameters.Rho = 0.1;
        RankSize = 6;
        Parameters.Tau0 = 1.DividedBy(Parameters.Rho * instance.UpperBound);
    }

    public override void PheromoneUpdate(IColony<RankBasedAntSystemAntV2>
        colony, RankBasedAntSystemAntV2[] ants, int currentIteration)
    {
        // ensures that size is an int between 1 and ants.Length
        var size = Math.Max(1, Math.Min(RankSize, ants.Length));
        var topAnts = ants.OrderBy(a => a.Makespan).Take(size - 1).ToArray();
        var bestSolutionPath = colony.BestSoFar.Path;
        foreach (var (allocation, currentPheromoneAmount) in PheromoneStructure)
        {
            // if using allocation, increase is proportional rank position and
            // quality
            var delta = topAnts.Select((ant, rank) =>
                ant.Path.Contains(allocation) ? (size - rank - 1) *
                    ant.Makespan.Inverse() : 0
            ).Sum();

            var allocationBelongsToBestScheduling =
                bestSolutionPath.Contains(allocation);
        }
    }
}

```

```

    // pheromone deposited only by best so far ant
    var deltaOpt = allocationBelongsToBestScheduling ?
        colony.BestSoFar.Makespan.Inverse() : 0;
    var updatedAmount = (1 - Parameters.Rho) * currentPheromoneAmount +
        delta + RankSize * deltaOpt;

    if (!PheromoneStructure.TryUpdate(allocation, updatedAmount,
        currentPheromoneAmount))
        Log($"Offline Update pheromone failed on {allocation}");
}
}

public override IFjspSolution Solve(Instance instance) { [...] }

public override RankBasedAntSystemAntV2[] BugsLife(int currentIteration)
{ [...] }
}

```

Fragmento 6.13 – Especializações que dão origem a RBASV2-p e RBASV2-i

6.3.1.4 Implementação do sistema de formigas MAX-MIN

O sistema de formigas MAX-MIN, apresentado na Seção 5.5.4, apresenta uma atualização de feromônio muito mais simples do que as anteriores. Um aspecto que vale a pena mencionar desta implementação foi o uso do método **UpdatePheromoneTrailLimits**, invocado após cada atualização de feromônio, para atualizar os limites da saturação à medida que o algoritmo avança, conforme sugerido por [Dorigo e Stutzle \(2004\)](#).

```

public class MaxMinAntSystemAntV2(int id, int generation,
    MaxMinAntSystemAlgorithmV2 context)
    : AntV2<MaxMinAntSystemAntV2, MaxMinAntSystemAlgorithmV2>(id, generation,
        context)
{
    public override void WalkAround()
    {
        ListSchedulingV2Heuristic<MaxMinAntSystemAlgorithmV2,
            MaxMinAntSystemAntV2>.Construct(this);
    }
}

public class MaxMinAntSystemAlgorithmV2(Parameters parameters, double tauMin,
    double tauMax, ISolveApproach solveApproach)
    : AntColonyV2AlgorithmSolver<MaxMinAntSystemAlgorithmV2,
        MaxMinAntSystemAntV2>(parameters, solveApproach)
{
    /// Max pheromone amount accepted over pheromone points
    public double TauMax { get; private set; } = tauMax;

    /// Min pheromone amount accepted over pheromone points
    public double TauMin { get; private set; } = tauMin;

    public override void DorigosTouch(Instance instance)
    {
        Parameters.Alpha = 1;
        Parameters.Rho = 0.02;
    }
}

```

```

        AntCount = instance.Jobs.Count();
        TauMax = 1.DividedBy(Parameters.Rho * instance.UpperBound);
        TauMin = TauMax.DividedBy(instance.MachinesPerOperation);
    }

    private void UpdatePheromoneTrailLimits(IColony<MaxMinAntSystemAntV2> colony)
    {
        TauMax = 1.DividedBy(Parameters.Rho * colony.BestSoFar.Makespan);
        TauMin =
            TauMax.DividedBy(colony.BestSoFar.Instance.MachinesPerOperation);
    }

    public override IFjspSolution Solve(Instance instance)
    {
        ...
        PheromoneUpdate(colony, ants, currentIteration);
        UpdatePheromoneTrailLimits(colony);
        ...
    }

    public override MaxMinAntSystemAntV2[] BugsLife(int currentIteration)
    { ... }

    public override void PheromoneUpdate(IColony<MaxMinAntSystemAntV2> colony,
        MaxMinAntSystemAntV2[] ants, int currentIteration)
    {
        var bestSolutionPath = colony.BestSoFar.Path;

        foreach (var (allocation, currentPheromoneAmount) in PheromoneStructure)
        {
            var allocationBelongsToBestScheduling =
                bestSolutionPath.Contains(allocation);
            // pheromone deposited only by best so far ant
            var delta = allocationBelongsToBestScheduling ?
                colony.BestSoFar.Makespan.Inverse() : 0;

            var updatedAmount = Math.Max(Math.Min((1 - Parameters.Rho) *
                currentPheromoneAmount + delta, TauMax), TauMin);

            if (!PheromoneStructure.TryUpdate(allocation, updatedAmount,
                currentPheromoneAmount))
                Log($"Offline Update pheromone failed on {allocation}");
        }
    }
}

```

Fragmento 6.14 – Especializações que dão origem a MMASV2-p e MMASV2-i

6.3.1.5 Implementação do sistema de colônia de formigas

Finalmente, temos o sistema de colônia de formigas. Essa implementação realiza mais sobrescritas de métodos do que as anteriores, pois esse algoritmo se destaca como a versão do ACO com as maiores diferenças em relação ao original. O método **PseudoProbabilityRule** implementa pseudoprob (Algoritmo 13) e o método

LocalPheromoneUpdate realiza a atualização local de feromônio após cada movimento viável realizado. O parâmetro **Q0** sofre uma atualização no início de cada iteração, de acordo com a implementação de [Rossi e Dini \(2007\)](#).

```
public class AntColonySystemAntV2(int id, int generation,
    AntColonySystemAlgorithmV2 context) : AntV2<AntColonySystemAntV2>,
    AntColonySystemAlgorithmV2>(id, generation, context)
{
    public override void WalkAround()
    {
        ListSchedulingV2Heuristic<AntColonySystemAlgorithmV2,
            AntColonySystemAntV2>.Construct(this);
    }

    /// Create feasible moves set and use pseudo probability rule to choose one
    public override IFeasibleMove<Allocation>
        ProbabilityRule(IEnumerable<IFeasibleMove<Allocation>> feasibleMoves)
        => PseudoProbabilityRule(feasibleMoves);

    private IFeasibleMove<Allocation>
        PseudoProbabilityRule(IEnumerable<IFeasibleMove<Allocation>>
            feasibleMoves)
    {
        var sum = 0.0;
        var rouletteWheel = new List<(IFeasibleMove<Allocation> Move, double
            Probability)>();
        IFeasibleMove<Allocation> greedyMove = null;
        var greedyFactor = double.MinValue;
        // create roulette wheel and evaluate greedy move for pseudorandom
        // proportional rule at same time
        foreach (var move in feasibleMoves)
        {
            var tauXy = move.GetPheromoneAmount(context.PheromoneStructure);
            var etaXy = move.Weight.Inverse(); // heuristic information
            var tauXyAlpha = Math.Pow(tauXy, context.Parameters.Alpha);
            var etaXyBeta = Math.Pow(etaXy, context.Parameters.Beta);

            double probFactor = tauXyAlpha * etaXyBeta, pseudoProbFactor = tauXy
                * etaXyBeta;
            rouletteWheel.Add((move, probFactor));
            sum += probFactor;

            if (greedyFactor >= pseudoProbFactor) continue;
            greedyFactor = pseudoProbFactor;
            greedyMove = move;
        }

        // pseudo random proportional rule
        if (Random.Shared.NextDouble() <= context.Q0)
            return greedyMove;

        // roulette wheel
        var cumulative = 0.0;
        var randomValue = Random.Shared.NextDouble() * sum;

        foreach (var (move, probability) in rouletteWheel)
```

```
        {
            cumulative += probability;
            if (randomValue <= cumulative)
                return move;
        }

        throw new InvalidOperationException("FATAL ERROR: No move was
            selected.");
    }

    public override void LocalPheromoneUpdate(Allocation selectedMove)
    {
        if (!context.PheromoneStructure.TryGetValue(selectedMove, out double
            currentPheromoneValue) ||
            !context.PheromoneStructure.TryUpdate(selectedMove, (1 -
                context.Phi) * currentPheromoneValue + context.Phi *
                context.Parameters.Tau0, currentPheromoneValue))
            Console.WriteLine("Unable to decay pheromone after construction
                step...");
    }
}

public class AntColonySystemAlgorithmV2(Parameters parameters, double phi,
    ISolveApproach solveApproach)
: AntColonyV2AlgorithmSolver<AntColonySystemAlgorithmV2,
    AntColonySystemAntV2>(parameters, solveApproach)
{
    /// Pheromone decay coefficient
    public double Phi { get; protected set; } = phi;

    /// Pseudorandom proportional rule parameter (between 0 and 1)
    public double Q0 { get; internal set; }

    public override void DorigosTouch(Instance instance)
    {
        Parameters.Alpha = 1;
        AntCount = 10;
        Phi = 0.1;
        Parameters.Rho = 0.1;
        Parameters.Tau0 = 1.0.DividedBy(instance.OperationCount *
            instance.UpperBound);
    }

    public override AntColonySystemAntV2[] BugsLife(int currentIteration)
    { [...] }

    public override IFjspSolution Solve(Instance instance)
    {
        ...
        for (int i = 0; i < Parameters.Iterations; i++)
        {
            var currentIteration = i + 1;
            Q0 = Math.Log(currentIteration) / Math.Log(Parameters.Iterations);
            iSw.Restart();
            var ants = BugsLife(currentIteration);
            ...
        }
    }
}
```

```

    ...
    return solution;
}

public override void PheromoneUpdate(IColony<AntColonySystemAntV2> colony,
    AntColonySystemAntV2[] ants, int currentIteration)
{
    var iterationBestAnt = colony.IterationBests[currentIteration];
    var bestSolutionPath = iterationBestAnt.Path;

    var delta = iterationBestAnt.Makespan.Inverse();
    foreach (var allocation in bestSolutionPath)
    {
        if (allocation is not null &&
            PheromoneStructure.TryGetValue(allocation, out double
            currentPheromoneAmount))
        {
            // new pheromone amount it's a convex combination between
            // currentPheromoneAmount and delta
            var updatedAmount = (1 - Parameters.Rho) *
                currentPheromoneAmount + Parameters.Rho * delta;

            if (!PheromoneStructure.TryUpdate(allocation, updatedAmount,
                currentPheromoneAmount))
                Log($"Local Update pheromone failed on {allocation} (race
                    condition guard)");
        }
    }
}

```

Fragmento 6.15 – Especializações que dão origem a ACSV2-p e ACSV2-i

O leitor deve ter notado que ambas as implementações de **WalkAround** nos fragmentos de código apresentados, invocam o mesmo método **Construct** da classe estática **ListSchedulingV2Heuristic**. Isso acontece porque o algoritmo de construção de soluções viáveis está desacoplado para evitar repetição de código. Caracterizando a versão V2 da heurística construtiva, o Fragmento 6.16 exibe a adaptação do algoritmo **LeastLoadedMachineHeuristicAlgorithmSolver** (Fragmento 6.5) para o contexto do ACO. Note que, comparado ao Fragmento 6.5, a escolha gulosa do movimento viável foi substituída pela regra de probabilidade característica do ACO.

```

public static class ListSchedulingV2Heuristic<TContext, TAnt>
    where TContext : AntColonyV2AlgorithmSolver<TContext, TAnt>
    where TAnt : AntV2<TAnt, TContext>
{
    public static class ListSchedulingV2Heuristic<TContext, TAnt>
        where TContext : AntColonyV2AlgorithmSolver<TContext, TAnt>
        where TAnt : AntV2<TAnt, TContext>
    {
        public static void Construct(TAnt ant)
        {
            // creating data structures
            var unscheduledOperations = new HashSet<Operation>();
            ant.Instance.Jobs.ForEach(job =>
            {
                job.Operations.ForEach(o =>
                    ant.Solution.CriticalPredecessors.Add(o.Id, null));
                unscheduledOperations.Add(job.Operations[0]);
            });
            ant.Instance.Machines.ForEach(m =>
                ant.Solution.LoadingSequence.Add(m.Index, []));

            while (unscheduledOperations.Any())
            {
                var feasibleMoves = ant.GetFeasibleMoves(unscheduledOperations);
                var nextMove = ant.ProbabilityRule(feasibleMoves) as
                    FeasibleMoveV2;

                ant.EvaluateCompletionTime(ant.Solution, nextMove.Allocation);
                ant.LocalPheromoneUpdate(nextMove.Allocation);
                ant.Solution.MachineAssignment
                    .Add(nextMove.Allocation.Operation.Id,
                        nextMove.Machine.Index);

                unscheduledOperations.Remove(nextMove.Operation);
                if (!nextMove.Operation.LastOperation)
                    unscheduledOperations.Add(nextMove.Operation.Job
                        .Operations[nextMove.Operation.Index + 1]);
            }

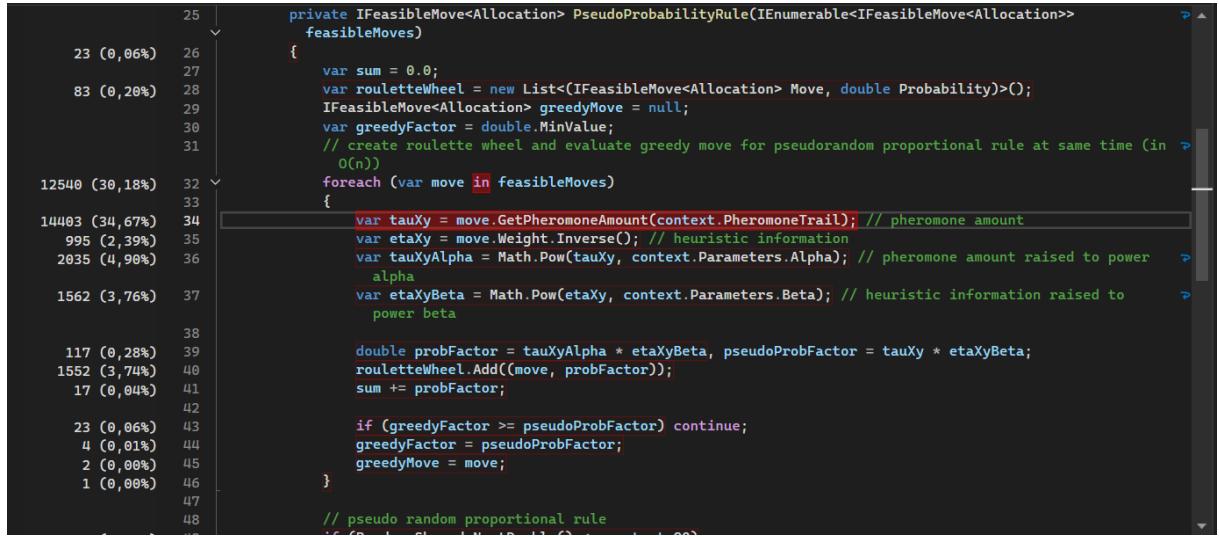
            if (ant.Context.Parameters.DisableLocalSearch)
                return;
            ant.NonImprovedSolution = ant.Solution;
            ant.Solution = FlexibleJobShopLocalSearchAlgorithm
                <AntSolution>.Run(ant.Instance, ant.Solution);
        }
    }
}

```

Fragmento 6.16 – Implementação da versão V2 para construção de soluções viáveis nos algoritmos ACO

A diferença no consumo de recursos e no tempo de execução em relação às implementações V0 e V1 foi gritante. A Figura 22 apresenta uma captura do perfilador de desempenho após uma execução sobre essa nova implementação. Comparando com a Figura 20, podemos observar que não há mais gargalos de execução enumerando

movimentos viáveis e a instrução que consome mais tempo de CPU (cerca de 34%) está associada ao acesso à concentração de feromônio.



The screenshot shows a performance profiler interface with assembly code and call stack information. The assembly code is in C#:

```

25     private IFeasibleMove<Allocation> PseudoProbabilityRule(IEnumerable<IFeasibleMove<Allocation>>
26         feasibleMoves)
27     {
28         var sum = 0.0;
29         var rouletteWheel = new List<(IFeasibleMove<Allocation> Move, double Probability)>();
30         IFeasibleMove<Allocation> greedyMove = null;
31         var greedyFactor = double.MinValue;
32         // create roulette wheel and evaluate greedy move for pseudorandom proportional rule (in O(n))
33         foreach (var move in feasibleMoves)
34         {
35             var tauXy = move.GetPheromoneAmount(context.PheromoneTrail); // pheromone amount
36             var etaXy = move.Weight.Inverse(); // heuristic information
37             var tauXyAlpha = Math.Pow(tauXy, context.Parameters.Alpha); // pheromone amount raised to power alpha
38             var etaXyBeta = Math.Pow(etaXy, context.Parameters.Beta); // heuristic information raised to power beta
39             double probFactor = tauXyAlpha * etaXyBeta, pseudoProbFactor = tauXy * etaXyBeta;
40             rouletteWheel.Add((move, probFactor));
41             sum += probFactor;
42             if (greedyFactor >= pseudoProbFactor) continue;
43             greedyFactor = pseudoProbFactor;
44             greedyMove = move;
45         }
46         // pseudo random proportional rule
47         // ...
48     }

```

Call stack statistics are listed on the left:

- 23 (0,06%)
- 83 (0,20%)
- 12540 (30,18%)
- 14403 (34,67%)
- 995 (2,39%)
- 2035 (4,90%)
- 1562 (3,76%)
- 117 (0,28%)
- 1552 (3,74%)
- 17 (0,04%)
- 23 (0,06%)
- 4 (0,01%)
- 2 (0,00%)
- 1 (0,00%)

Figura 22 – Relatório do perfilador de desempenho que sugere a eficiência da nova implementação

6.4 Rejeite a linearidade. Abrace a generalização.

A vitória conquistada com a nova versão tinha um gosto amargo. O modelo preliminar baseado em digrafo disjuntivo contemplaria uma extensão natural do problema: relações de precedência não lineares. A versão V2 assume a linearidade das precedências para simplificar a implementação da estrutura de vizinhança e realizar o cômputo dos movimentos viáveis em tempo polinomial.

Contudo, como vimos no Algoritmo 8, é possível adaptar a heurística LLM-FJSSP para admitir relações de precedência não lineares. O Fragmento 6.17 apresenta a versão V3 da heurística construtiva, que deu origem a mais 10 implementações de algoritmos ACO. Cada formiga mantém o digrafo de precedência que compõe a instância do problema, além de um dicionário de contadores, inicializado com o grau de entrada de cada vértice. O método **ReleaseVertexSuccessors**, inspirado no algoritmo de Kahn, é invocado a cada movimento viável, decrementando os contadores. Sempre que um contador de um vértice vai a 0, todas as operações predecessoras da operação correspondente foram devidamente escalonadas, de modo que o vértice pode ser adicionado na lista de operações não escalonadas.

```

public static class ListSchedulingV3Heuristic<TContext, TAnt>
    where TContext : AntColonyV3AlgorithmSolver<TContext, TAnt>
    where TAnt : AntV3<TAnt, TContext>
{
    public static void Construct(TAnt ant)
    {
        // creating data structures
        ant.InitializeDataStructures();
        var scheduledOperations = new HashSet<OperationVertex>();
        var unscheduledOperations = new HashSet<OperationVertex>();
        var inDegreeCounters = new Dictionary<OperationVertex, int>();
        ant.PrecedenceDigraph.VertexSet.ForEach(o =>
            inDegreeCounters.Add(o,
                ant.PrecedenceDigraph.NeighbourhoodIn(o).Count)
        );

        inDegreeCounters.Where(o => o.Value == 0)
            .ForEach(o => unscheduledOperations.Add(o.Key));

        while (unscheduledOperations.Any())
        {
            var feasibleMoves = ant.GetFeasibleMoves(unscheduledOperations,
                scheduledOperations);
            var nextMove = ant.ProbabilityRule(feasibleMoves);
            ant.EvaluateCompletionTime(nextMove);
            ant.LocalPheromoneUpdate(nextMove);
            unscheduledOperations.Remove(nextMove.Vertex);
            ReleaseVertexSuccessors(ant, nextMove.Vertex, inDegreeCounters,
                scheduledOperations, unscheduledOperations);
        }

        // creating mu function
        foreach (var (m, operations) in ant.LoadingSequence)
            foreach (var o in operations)
                ant.Solution.MachineAssignment.Add(o.Id, m.Index);
    }

    private static void ReleaseVertexSuccessors(TAnt ant, OperationVertex u,
        Dictionary<OperationVertex, int> vertexCounters,
        HashSet<OperationVertex> scheduledNodes, HashSet<OperationVertex>
        unscheduledNodes)
    {
        ant.PrecedenceDigraph.NeighbourhoodOut(u)
            .ForEach(v =>
        {
            vertexCounters[v] -= 1;
            if (vertexCounters[v] == 0 &&
                scheduledNodes.DoesNotContain(v))
                unscheduledNodes.Add(v);
        });
    }
}

```

Fragmento 6.17 – Implementação da versão V3 para construção de soluções viáveis nos algoritmos ACO

Por fim, para garantir a viabilidade das soluções geradas por cada um dos 33 algoritmos implementados, foram desenvolvidos testes de unidade para validar cada restrição do problema, conforme o Fragmento 6.18. A Figura 23 exibe uma captura de tela com o resultado da sua execução.

Test	Duration	Traits
✓ Scheduling.Tests (4)	2,7 sec	
✓ Scheduling.Tests (4)	2,7 sec	
✓ FJSPSolutionTests (4)	2,7 sec	
✓ FjspSolution_ConjunctiveRestriction_MustBeSatisfied	948 ms	
✓ FjspSolution_DisjunctiveRestriction_MustBeSatisfied	583 ms	
✓ FjspSolution_EligibleMachineRestriction_MustBeSatisfied	655 ms	
✓ FjspSolution_ReleaseDateRestriction_MustBeSatisfied	544 ms	

Figura 23 – Gerenciador de testes da IDE Visual Studio

```
public class FJSPSolutionTests
{
    private IBenchmarkReaderService _readerService = new
        BenchmarkReaderService(null);
    private const string BENCHMARK_FILE = "//workspaces//SchedulingAlgorithms//"
        Scheduling.Benchmarks//Data//6_Fattahi//Fattahi12.fjs";

    #region Solvers batch
    public static IEnumerable<object[]> GetSolvers()
    {
        Parameters parameters = new(alpha: 1.0, beta: 1.0, rho: 0.5, tau0: 100,
            ants: 10, iterations: 2, stagnantGenerationsAllowed: 1);

        # region AS
        yield return
        [
            new AntSystemAlgorithmV1(parameters, new IterativeSolveApproach())
        ];
        yield return
        [
            new AntSystemAlgorithmV1(parameters, new ParallelSolveApproach())
        ];

        ...
    }

    yield return
    [
        new AntColonySystemAlgorithmV3(parameters, phi: 0.5, new
            ParallelSolveApproach())
    ];
    #endregion

    yield return
    [

```

```
        new LeastLoadedMachineHeuristicAlgorithmSolver()
    ];
}

#endregion

[Theory]
[MemberData(nameof(GetSolvers))]
public void FjspSolution_EligibleMachineRestriction_MustBeSatisfied
(IFlexibleJobShopSchedulingSolver solver)
{
    var instance = _readerService.ReadInstance(BENCHMARK_FILE);
    var solution = solver.Solve(instance);

    foreach (var job in instance.Jobs)
    {
        foreach (var operation in job.Operations)
        {
            Assert.True(solution.MachineAssignment.ContainsKey(operation.Id),
                $"Restriction 0 was violated: No machine allocation for
                    operation {operation.Id}");
            var allocation = solution.MachineAssignment[operation.Id];
            Assert.True(operation.EligibleMachines.Any(m => m.Index ==
                allocation), $"Restriction 0 was violated: Forbidden
                    allocation for {operation.Id}");
        }
    }
}

[Theory]
[MemberData(nameof(GetSolvers))]
public void FjspSolution_ReleaseDateRestriction_MustBeSatisfied
(IFlexibleJobShopSchedulingSolver solver)
{
    var instance = _readerService.ReadInstance(BENCHMARK_FILE);
    instance.Jobs.ForEach(j => j.ReleaseDate = Random.Shared.Next(10, 100));
    var solution = solver.Solve(instance);

    foreach (var job in instance.Jobs)
    {
        var firstOperation = job.Operations.First();
        var jobStartTime = solution.StartTimes[firstOperation.Id];
        Assert.True(jobStartTime >= job.ReleaseDate, "Restriction 1 was
            violated");
    }
}

[Theory]
[MemberData(nameof(GetSolvers))]
public void FjspSolution_ConjunctiveRestriction_MustBeSatisfied
(IFlexibleJobShopSchedulingSolver solver)
{
    var instance = _readerService.ReadInstance(BENCHMARK_FILE);
    var solution = solver.Solve(instance);

    foreach (var job in instance.Jobs)
    {
```

```

        Operation previousOperation = null;
        foreach (var operation in job.Operations)
        {
            if (previousOperation is not null)
                Assert.True(solution.StartTimes[operation.Id] >=
                    solution.CompletionTimes[previousOperation.Id],
                    $"Restriction 2 was violated: operation {operation.Id}
                     starting ({solution.StartTimes[operation.Id]}) 
                     before {previousOperation.Id} finish
                     ({solution.CompletionTimes[previousOperation.Id]})");
            previousOperation = operation;
        }
    }

    [Theory]
    [MemberData(nameof(GetSolvers))]
    public void FjspSolution_DisjunctiveRestriction_MustBeSatisfied
    (IFlexibleJobShopSchedulingSolver solver)
    {
        var instance = _readerService.ReadInstance(BENCHMARK_FILE);
        var solution = solver.Solve(instance);

        foreach (var machine in instance.Machines)
        {
            var operations = solution.MachineAssignment.Where(mu => mu.Value == machine.Index).Select(mu => mu.Key);

            foreach (var o1 in operations)
            {
                foreach (var o2 in operations)
                {
                    if(o1 != o2)
                        Assert.True(
                            solution.StartTimes[o2] >=
                                solution.CompletionTimes[o1] ||
                            solution.StartTimes[o1] >=
                                solution.CompletionTimes[o2],
                            "Restriction 3 was violated");
                }
            }
        }
    }
}

```

Fragmento 6.18 – Testes de unidade desenvolvidos para validar as soluções geradas pelos algoritmos implementados

7 Resultados e Discussões

Dedicaremos este capítulo para tratar dos experimentos computacionais realizados no decorrer deste trabalho e a discussão dos seus resultados. Como vimos no Capítulo 6, foram implementados 33 algoritmos. São eles:

- ACSV0-p
- ACSV0-i
- ASV1-i
- ASV1-p
- EASV1-i
- EASV1-p
- RBASV1-i
- RBASV1-p,
- MMASV1-i
- MMASV1-p
- ACSV1-i
- ACSV1-p
- ASV2-i
- ASV2-p
- EASV2-i
- EASV2-p
- RBASV2-i
- RBASV2-p
- MMASV2-i
- MMASV2-p
- ACSV2-i
- ACSV2-p
- ASV3-i
- ASV3-p
- EASV3-i
- EASV3-p
- RBASV3-i
- RBASV3-p
- RBASV3-i
- ACSV3-i
- ACSV3-p.
- LLM-FJSSP

Primeiramente, abordaremos a metodologia acerca dos experimentos, falando de detalhes que pautaram os testes, como a parametrização e as métricas utilizadas. Em seguida, reservamos uma seção para as instâncias utilizadas e seus respectivos limitantes. Logo após, discutimos os experimentos computacionais, passando pelo delineamento experimental e uma extensa lista de resultados. Por fim, encerramos o capítulo com um novo conjunto de instâncias desenvolvido com base em cenários reais da indústria gráfica e os resultados obtidos a partir dele.

7.1 Metodologia

Nesta seção, veremos a metodologia empregada para a avaliação dos algoritmos desenvolvidos neste trabalho. Os principais objetivos destes experimentos são

- (i) investigar o desempenho destes algoritmos na resolução do FJSSP, considerando critérios de qualidade da solução e tempo de execução;

- (ii) investigar se o desempenho dos algoritmos desenvolvidos é competitivo frente a outros métodos encontrados na literatura;
- (iii) comparar a qualidade das soluções obtidas por esses algoritmos com as soluções geradas pela solução *ad-hoc* do sistema da empresa interessada.

Uma aplicação C# de linha de comando, denominada **Scheduling.Console**, foi desenvolvida para expor uma API para a execução dos algoritmos. Cada algoritmo é capaz de construir soluções e coletar métricas de sua própria execução (as instruções para utilização foram disponibilizadas no [README deste repositório](#)). Um *script* foi escrito em Python para automatizar a execução dos algoritmos, orquestrando a execução e submetendo [neste outro repositório](#) os arquivos com os resultados obtidos nos experimentos.

Os experimentos foram conduzidos majoritariamente sobre conjuntos clássicos de instâncias do FJSSP utilizados na literatura, que foram compilados e disponibilizados publicamente [neste repositório](#) na plataforma GitHub por [Lei et al. \(2022\)](#). As instâncias, que abrangem diferentes características em termos de número de operações, máquinas disponíveis e complexidade estrutural, estão organizadas em arquivos, seguindo o seguinte formato:

- Primeira linha: número de trabalhos $|J|$, número de máquinas $|M|$ e o número médio de máquinas por operação ξ ;
- Para todo $0 < j \leq |J|$, na $(j + 1)$ -ésima linha:
 - número de operações do j -ésimo trabalho;
 - para cada operação o do trabalho:
 - * número de máquinas compatíveis com o ;
 - * para cada máquina m compatível com o : um par de números $m \varrho_o(m)$ representando a máquina m e a respectiva duração de o em m .

Além das instâncias da literatura, um conjunto de instâncias (cujos arquivos estão disponibilizados [neste link](#)) foi desenvolvido com base nos dados de clientes da empresa interessada, a fim de comparar com as soluções geradas pelo seu *software* proprietário. O Fragmento 7.1 exibe o conteúdo do arquivo correspondente à instância SJSF2 do conjunto de instâncias de [Fattahi et al. \(2007\)](#), utilizada no Exemplo 4.5.

```
2 2 1.5
2 1 1 43 2 1 64 2 71
2 2 1 21 2 35 1 2 43
```

Fragmento 7.1 – Arquivo correspondente a instância SJSF2 de [Fattahi et al. \(2007\)](#)

Para cada algoritmo ACO, os parâmetros foram definidos conforme recomendações da literatura. A Tabela 7 destaca os valores recomendados (para implementações sem busca local)¹ por Dorigo e Stutzle (2004), com exceção do parâmetro de corte de exploração

$$q_0 := \frac{\log(i)}{\log(n)}$$

do algoritmo ACS. Aqui, a parametrização segue a sugestão de Rossi e Dini (2007), com $n > 0$ sendo a quantidade total de iterações e $1 < i \leq n$ o número de iterações executadas². Cada instância foi executada 30 vezes. Os critérios de parada adotados foram: número de iterações (calibrado para 50) e o número de iterações sem melhora (calibrado para 20% do número de iterações).

Parâmetro	AS	EAS	RBAS	MMAS	ACS
α : influência da concentração de feromônio	1	1	1	1	1
β : influência da informação heurística	1,4	1,4	1,4	1,4	1,4
ρ : coeficiente de evaporação de feromônio	0,5	0,5	0,1	0,02	0,1
τ_0 : quantidade inicial de feromônio	$\frac{1}{C_{LLM}}$	$\frac{\varepsilon A }{\rho C_{LLM}}$	$\frac{1}{\rho C_{LLM}}$	τ_{\max}	$\frac{1}{ J C_{LLM}}$
$ A $: número de formigas	$ J $	$ J $	$ J $	$ J $	10
ε : peso do reforço elitista	-	$ J $	-	-	-
r : tamanho da classificação	-	-	6	-	-
τ_{\min} : valor mínimo da concentração de feromônio	-	-	-	$\frac{\tau_{\max}}{\xi}$	-
τ_{\max} : valor máximo da concentração de feromônio	-	-	-	$\frac{1}{\rho C_{LLM}}$	-
φ : coeficiente de decaimento de feromônio	-	-	-	-	0,1

Tabela 7 – Valores dos parâmetros sugeridos por Dorigo e Stutzle (2004). O símbolo $|J|$ representa a quantidade de trabalhos, ξ o número médio de máquinas por operação e C_{LLM} o *makespan* obtido pela execução da heurística LLM-FJSSP.

As soluções obtidas foram avaliadas de acordo com o *makespan* e o tempo de execução. Para análise comparativa, foram computadas estatísticas como o melhor *makespan* encontrado, *makespan* médio, desvio padrão e tempo médio de CPU. Quando pertinente, o *gap* percentual em relação aos melhores valores conhecidos na literatura foi utilizado nas análises. Veremos agora no que consiste essa métrica.

¹ A parametrização sugerida é baseada em implementações para o TSP, de modo que alguns parâmetros são definidos com base no resultado de um algoritmo de aproximação para este problema. No nosso contexto, foi utilizado o resultado da heurística LLM-FJSSP em seu lugar. Ademais, nos lugares onde a literatura sugeriu usar o número de cidades do TSP, foi utilizado o número de trabalhos.

² Note caro leitor, que esse número tende a 1 a medida que o algoritmo avança. Resgatando o papel de q_0 na regra de probabilidade pseudo-aleatória, isso implica que o algoritmo ACS terá um comportamento mais exploratório no início de sua execução e ao final, um comportamento mais explotatório.

Seja $i \in I$ uma instância em um conjunto I finito e não vazio de instâncias do FJSSP. Considere um algoritmo A capaz de resolver i . Denotaremos por $A(i)$ o *makespan* encontrado em uma execução de A sobre i . Considere o melhor *makespan* conhecido na literatura para a instância i , denotado por $\alpha(i)$. O **gap percentual** (também conhecido como **erro relativo**) de A para a instância i é

$$\text{gap}_i(A) := \left(\frac{\overline{A(i)} - \alpha(i)}{\alpha(i)} \right) \times 100,$$

onde $\overline{A(i)}$ é o *makespan* médio obtido nas execuções de A sobre i . Por mais sem sentido que pareça num primeiro momento, veremos o *gap* percentual para uma instância considerando um conjunto \mathcal{A} finito e não vazio de algoritmos. Neste caso, calculamos

$$\text{gap}_i(\mathcal{A}) := \left(\frac{\overline{\mathcal{A}(i)} - \alpha(i)}{\alpha(i)} \right) \times 100,$$

onde $\overline{\mathcal{A}(i)} := \frac{1}{|\mathcal{A}|} \sum_{A \in \mathcal{A}} \overline{A(i)}$ é a média do *makespan* médio dos algoritmos avaliados.

Finalmente, o **gap percentual médio** (ou, **erro relativo médio**) de um conjunto \mathcal{A} de algoritmos para um conjunto de instâncias I é

$$\overline{\text{gap}}(\mathcal{A}, I) := \frac{1}{|I|} \sum_{i \in I} \text{gap}_i(\mathcal{A}).$$

Essa métrica (que por economia será referida simplesmente como *gap* médio) será de grande valia para comparar a qualidade das heurísticas construtivas implementadas. Por exemplo, se tomarmos os conjuntos

$$V0 := \{\text{ACSV0-i}, \text{ACSV0-p}\} \quad \text{e} \quad V2 := \{\text{ASV2-i}, \text{ASV2-p}, \text{EASV2-i}, \dots, \text{ACSV2-i}, \text{ACSV2-p}\}$$

de algoritmos que implementam as heurísticas $V0$ e $V2$, podemos comparar os *gaps* médios $\overline{\text{gap}}(V0, I)$ e $\overline{\text{gap}}(V2, I)$ sobre o conjunto de instâncias I e tirar conclusões sobre a qualidade das soluções geradas. Quanto mais próximo de zero for o valor desta métrica, melhor deve ser o desempenho geral do conjunto de algoritmos.

7.2 Instâncias

Abordaremos alguns conjuntos de instâncias que figuram dentre os mais utilizados na literatura sobre o FJSSP e, em razão disso, foram utilizados nos experimentos. Para cada conjunto apresentado a seguir, considerando os resultados da literatura para minimização de *makespan*, uma relação com o tamanho da instância (caracterizado pelo número de trabalhos $|J|$ e de máquinas $|M|$) e os melhores limitantes inferiores (LI) e superiores (LS) encontrados é apresentada. Nos casos em que o resultado é conhecidamente ótimo, o valor do limitante aparece destacado.

7.2.1 Instâncias de Fattahi et al. (2007)

O conjunto de instâncias de Fattahi et al. (2007), apresentado na Tabela 8, consiste em 20 instâncias de pequeno e médio porte geradas de forma aleatória. Das 20 instâncias, 16 têm seu ótimo conhecido: SFJS1 a SFJS10 encontrados por Fattahi et al. (2007); MFJS1, MFJS2, MFJS3 e MFJS5 por Özgüven et al. (2010); e MFJS4 e MFJS7 por Birgin et al. (2013). Ademais, Birgin et al. (2013) encontraram os melhores limitantes para MFJS6, MFJS8, MFJS9 e MFJS10.

Instância	$ J $	$ M $	LI	LS
SFJS1	2	2	66	66
SFJS2	2	2	107	107
SFJS3	3	2	221	221
SFJS4	3	2	355	355
SFJS5	3	2	119	119
SFJS6	3	2	320	320
SFJS7	3	5	397	397
SFJS8	3	4	253	253
SFJS9	3	3	210	210
SFJS10	4	5	516	516
MFJS1	5	6	468	468
MFJS2	5	7	446	446
MFJS3	6	7	466	466
MFJS4	7	7	554	554
MFJS5	7	7	514	514
MFJS6	8	7	614	634
MFJS7	8	7	879	879
MFJS8	9	8	775	884
MFJS9	11	8	845,26	1088
MFJS10	12	8	944,8	1251

Tabela 8 – Instâncias de Fattahi et al. (2007).

7.2.2 Instâncias de Brandimarte (1993)

Brandimarte (1993) introduziu 15 instâncias de médio porte, exibidas na Tabela 9. Dentre elas, sete têm seu valor ótimo conhecido, tendo Behnke e Geiger (2012) encontrado para MK1, MK9, MK12 e MK14 através da técnica de programação por restrições e Mastrolilli e Gambardella (2000) para MK3, MK4 e MK8. Para as instâncias MK2 e MK6, o melhor limitante inferior foi encontrado por Birgin et al. (2013) e o melhor limitante superior por Mastrolilli e Gambardella (2000). O próprio Brandimarte (1993) foi responsável pelos melhores limitantes superiores para MK11, MK13 e MK15. Os demais limitantes foram retirados do trabalho de Behnke e Geiger (2012).

Instância	$ J $	$ M $	LI	LS
MK1	10	6	40	40
MK2	10	6	25	26
MK3	15	8	204	204
MK4	15	8	48	60
MK5	15	4	168	172
MK6	10	15	37	58
MK7	20	5	133	139
MK8	20	10	523	523
MK9	20	10	307	307
MK10	20	15	165	197
MK11	30	5	594	649
MK12	30	10	508	508
MK13	30	10	353	478
MK14	30	15	694	694
MK15	30	15	283	383

Tabela 9 – Instâncias de Brandimarte (1993).

7.2.3 Instâncias de Dauzère-Pérès e Paulli (1997)

Dauzère-Pérès e Paulli (1997) introduziram 18 instâncias de médio porte, conforme exibido na Tabela 10. Os limitantes inferiores são todos provenientes do trabalho Dauzère-Pérès e Paulli (1997) e os limitantes superiores do trabalho de Behnke e Geiger (2012).

Instância	$ J $	$ M $	LI	LS
DP1a	10	5	2505	2518
DP2a	10	5	2228	2231
DP3a	10	5	2228	2229
DP4a	10	5	2503	2503
DP5a	10	5	2189	2216
DP6a	10	5	2162	2196
DP7a	15	8	2187	2283
DP8a	15	8	2061	2069
DP9a	15	8	2061	2066
DP10a	15	8	2178	2291
DP11a	15	8	2017	2063
DP12a	15	8	1969	2030
DP13a	20	10	2161	2257
DP14a	20	10	2161	2167
DP15a	20	10	2161	2165
DP16a	20	10	2148	2255
DP17a	20	10	2088	2140
DP18a	20	10	2057	2127

Tabela 10 – Instâncias de Dauzère-Pérès e Paulli (1997).

7.2.4 Instâncias de Hurink et al. (1994)

[Hurink et al. \(1994\)](#) adaptaram uma série de conjuntos de instâncias clássicos para o JSSP para o contexto do FJSSP. De acordo com [Behnke e Geiger \(2012\)](#), [Hurink et al. \(1994\)](#) modificaram 66 instâncias da literatura, sendo 3 delas provenientes de [Fisher e Thompson \(1963\)](#), 40 de [Lawrence \(1984\)](#), 5 de [Adams et al. \(1988\)](#), 8 de [Carlier e Pinson \(1989\)](#) e 10 de [Applegate e Cook \(1991\)](#). Esse conjunto de instâncias do JSSP foi denominado por sdata e utilizado como base para mais 3 conjuntos: edata, rdata e vdata. A principal diferença entre eles é o nível de flexibilidade de máquinas adotado, variando de nenhuma flexibilidade (sdata) a uma flexibilidade³ mais alta (vdata). Para o bem do leitor, neste trabalho adotaremos somente as 40 instâncias vindas de [Lawrence \(1984\)](#), em suas versões nos conjuntos sdata e vdata, conforme exibido na Tabela 11. Os valores dos limitantes inferiores e superiores das instâncias de sdata foram retirados deste repositório e os das instâncias de vdata do levantamento realizado por [Behnke e Geiger \(2012\)](#).

³ Nesse contexto, “flexibilidade” quer dizer o número médio de máquinas por operação.

Instância	$ J $	$ M $	sdata		vdata	
			LI	LS	LI	LS
la01	10	5	666	666	570	570
la02	10	5	655	655	529	529
la03	10	5	597	597	477	477
la04	10	5	590	590	502	502
la05	10	5	593	593	457	457
la06	15	5	926	926	799	799
la07	15	5	890	890	749	749
la08	15	5	863	863	765	765
la09	15	5	951	951	853	853
la10	15	5	958	958	804	804
la11	20	5	1222	1222	1071	1071
la12	20	5	1039	1039	936	936
la13	20	5	1150	1150	1038	1038
la14	20	5	1292	1292	1070	1070
la15	20	5	1207	1207	1089	1089
la16	10	10	945	945	717	717
la17	10	10	784	784	646	646
la18	10	10	848	848	663	663
la19	10	10	842	842	617	617
la20	10	10	902	902	756	756
la21	15	10	1046	1046	800	806
la22	15	10	927	927	733	739
la23	15	10	1032	1032	809	815
la24	15	10	935	935	773	777
la25	15	10	977	977	751	756
la26	20	10	1218	1218	1052	1054
la27	20	10	1235	1235	1084	1085
la28	20	10	1216	1216	1069	1070
la29	20	10	1152	1152	993	994
la30	20	10	1355	1355	1068	1069
la31	30	10	1784	1784	1520	1520
la32	30	10	1850	1850	1657	1658
la33	30	10	1719	1719	1497	1497
la34	30	10	1721	1721	1535	1535
la35	30	10	1888	1888	1549	1549
la36	15	15	1268	1268	948	948
la37	15	15	1397	1397	986	986
la38	15	15	1196	1196	943	943
la39	15	15	1233	1233	922	922
la40	15	15	1222	1222	955	955

Tabela 11 – Instâncias de Hurink et al. (1994).

7.3 Experimentos computacionais

Nesta seção, detalharemos os experimentos computacionais realizados. Todos os experimentos foram realizados em um processador Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz com 16 GB de RAM, rodando o sistema operacional Kubuntu 24.04. As versões que seguem uma abordagem paralela usam a biblioteca C# nativa **PLINQ** para gerenciar o processamento em múltiplos núcleos através de *threads* nativas.

Os 32 algoritmos ACO avaliados foram agrupados conforme a implementação da heurística construtiva utilizada. Vamos nos referir a esses conjuntos de algoritmos por:

- V0: ACSV0-p e ACSV0-i;
- V1: ASV1-p, ASV1-i, EASV1-p, EASV1-i, RBASV1-p, RBASV1-i, MMASV1-p, MMASV1-i, ACSV1-p e ACSV1-i;
- V2: ASV2-p, ASV2-i, EASV2-p, EASV2-i, RBASV2-p, RBASV2-i, MMASV2-p, MMASV2-i, ACSV2-p e ACSV2-i; e
- V3: ASV3-p, ASV3-i, EASV3-p, EASV3-i, RBASV3-p, RBASV3-i, MMASV3-p, MMASV3-i, ACSV3-p e ACSV3-i.

O delineamento experimental adotado consiste em submeter instâncias do FJSSP aos algoritmos de cada um desses conjuntos e comparar os *gaps* médios com o *gap* médio da heurística gulosa LLM-FJSSP para avaliar a qualidade das soluções geradas. Também foram comparados os maiores tempos médios de CPU de cada conjunto, para uma análise de viabilidade dos algoritmos.

7.3.1 Testes preliminares

Começaremos a discussão com resultados das versões preliminares que justificaram o desenvolvimento das versões posteriores. Mais especificamente, analisaremos os algoritmos desenvolvidos sobre as heurísticas construtivas V0 e V1, que utilizam um modelo baseado em digrafo disjuntivo flexível. Como mencionado no capítulo anterior, o primeiro algoritmo desenvolvido foi ACSV0-p e da primeira tentativa de melhorá-lo, surgiu ACSV0-i. Ambos os algoritmos foram submetidos a testes com os conjuntos de instâncias de [Fattahi et al. \(2007\)](#) e [Brandimarte \(1993\)](#) para testar a hipótese de um *overhead* na sincronização do acesso à estrutura de feromônio durante a execução.

Os gráficos nas figuras 24 e 25 derrubam a hipótese. A curva azul associada a ACSV0-i sempre limita superiormente a curva vermelha associada a ACSV0-p. É possível observar que, nos piores casos, ACSV0-p foi cerca de três vezes mais rápido que ACSV0-i. Contudo, comparado ao tempo de execução da heurística LLM-FJSSP e de

outros casos na literatura, a performance se mostrou muito abaixo do esperado para instâncias deste tamanho. Ademais, os gráficos nas figuras 26 e 27 sugerem que ambos os algoritmos deixam a desejar na exploração do espaço de busca, gerando em média soluções parecidas com as soluções gulosas do algoritmo LLM-FJSSP (que se mostrou extremamente mais rápido). De fato, para as instâncias de Fattahi et al. (2007), o conjunto V0 apresentou um *gap* médio de 59,26%, que é comparável ao *gap* de 67,96% registrado pelo algoritmo LLM-FJSSP. Já para as instâncias de Brandimarte (1993), o *gap* médio foi de 164,87%, que é próximo ao *gap* de 215,11% observado para LLM-FJSSP.

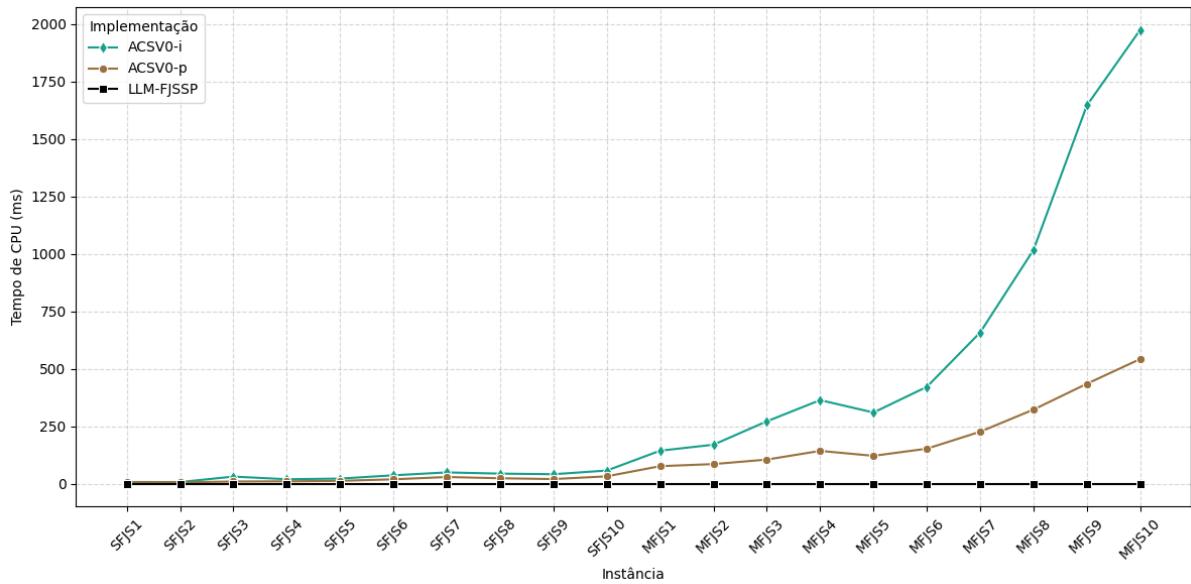


Figura 24 – Tempo médio de CPU do conjunto V0 para as instâncias de Fattahi et al. (2007)

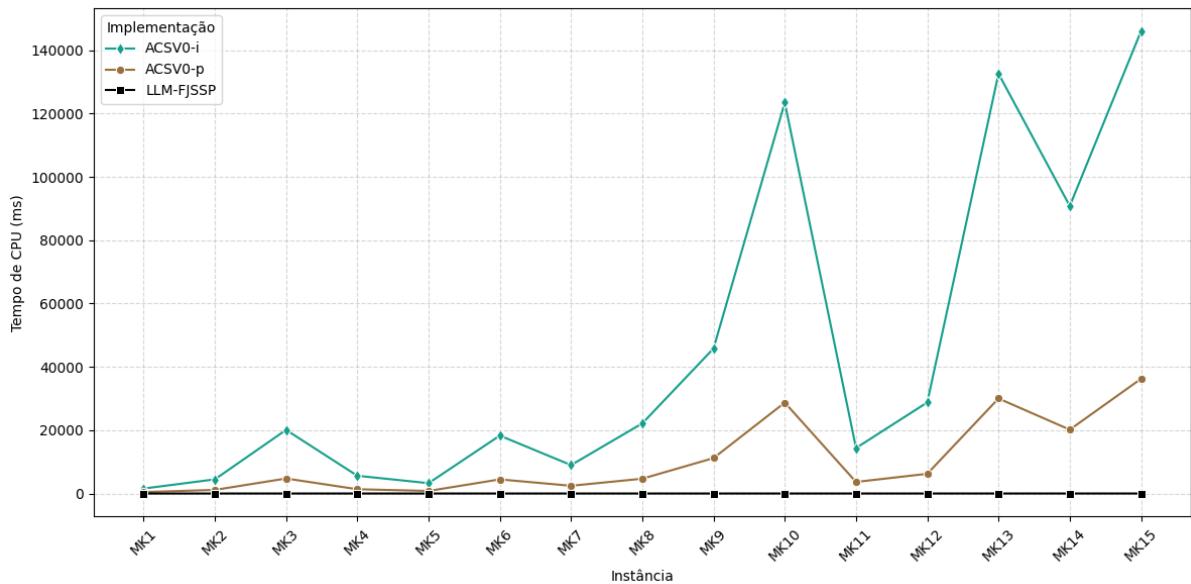


Figura 25 – Tempo médio de CPU do conjunto V0 para as instâncias de Brandimarte (1993)

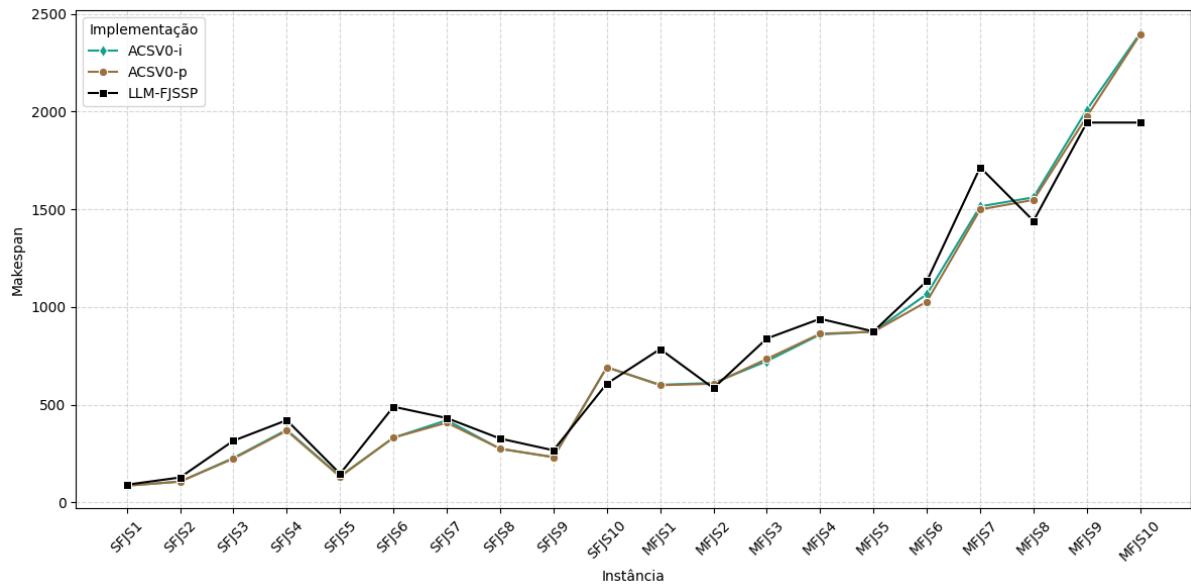


Figura 26 – Makespan médio do conjunto V0 para as instâncias de Fattahi et al. (2007)

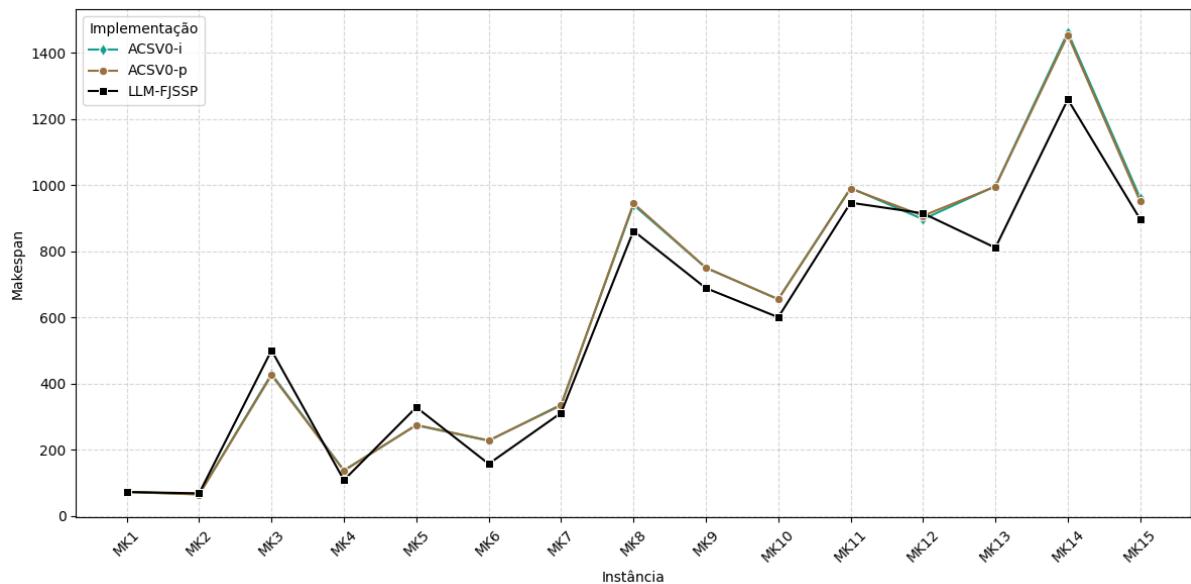


Figura 27 – Makespan médio do conjunto V0 para as instâncias de Brandimarte (1993)

Vimos na Seção 6.2 que o grande problema da heurística V0 é o gargalo de execução enumerando movimentos viáveis. Também vimos que, neste ponto, outros algoritmos ACO foram implementados. Visando otimizar o cômputo dos movimentos viáveis, um pequeno ajuste realizado na heurística V0 deu origem à heurística V1. Surgiram então implementações de todos os algoritmos ACO estudados neste trabalho baseados na heurística V1, totalizando 10 algoritmos implementados (5 seguindo uma abordagem iterativa e 5 seguindo uma abordagem paralela).

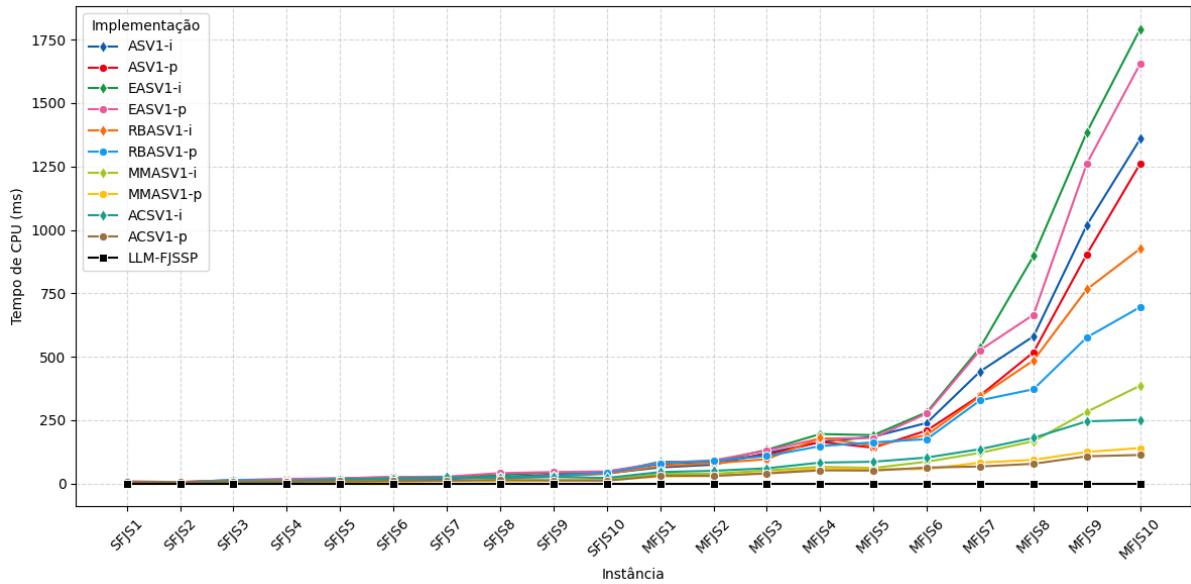


Figura 28 – Tempo médio de CPU do conjunto V1 para as instâncias de Fattahi et al. (2007)

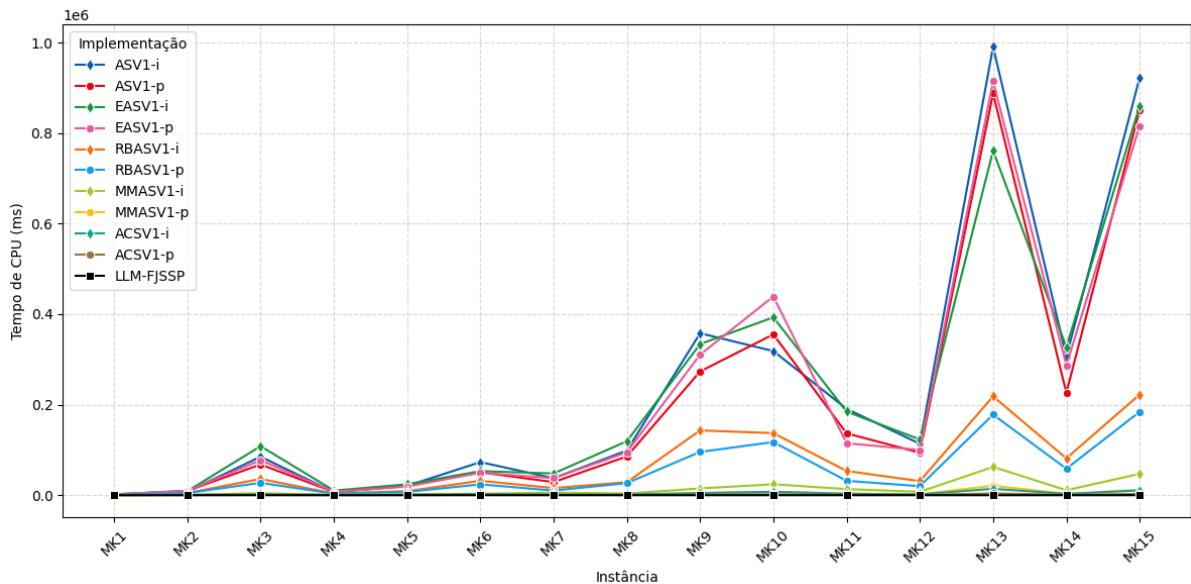


Figura 29 – Tempo médio de CPU do conjunto V1 para as instâncias de de Brandimarte (1993)

Os gráficos ilustrados nas figuras 28 e 29 confirmam a superioridade da abordagem paralela, principalmente nas maiores instâncias. Seguindo o que foi observado anteriormente, as curvas que correspondem aos algoritmos que seguem uma abordagem iterativa (marcadores no formato de diamante) quase sempre limitam superiormente as curvas das suas contrapartes paralelas (marcadores no formato circular).

Além do mais, os gráficos nas figuras 30 e 31 prenunciam um comportamento curioso que veremos se repetir: os 5 algoritmos ACO estudados, ao menos no contexto do

FJSSP, empatam no quesito “qualidade das soluções”, uma vez que o *makespan* médio dos 10 algoritmos é muito semelhante em todas as instâncias observadas⁴. Contudo, assim como no caso anterior, o *gap* médio do conjunto de algoritmos V1 é muito próximo do *gap* de LLM-FJSSP (53,07% contra 67,96% para as instâncias de Fattahi et al. (2007) e 160,74% contra 215,11% para as Brandimarte (1993)). Isso atesta a ineficiência dos 12 algoritmos ACO baseados no digrafo disjuntivo flexível de Rossi e Dini (2007), que se mostram computacionalmente caros frente a um algoritmo muito mais simples, econômico e indiscutivelmente mais rápido.

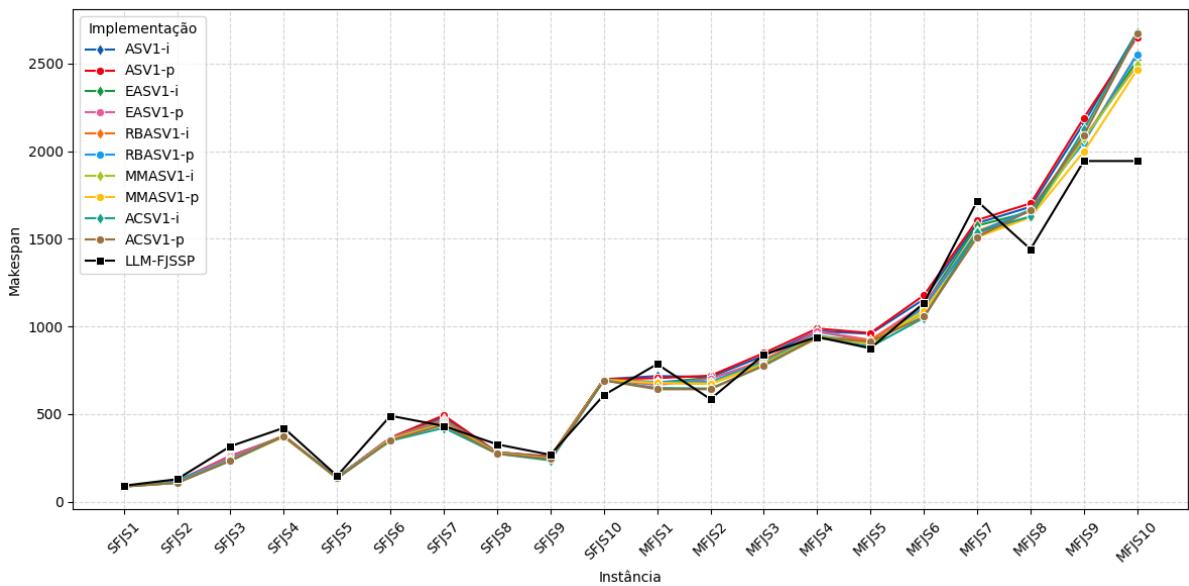


Figura 30 – *Makespan* médio do conjunto V1 para as instâncias de Fattahi et al. (2007)

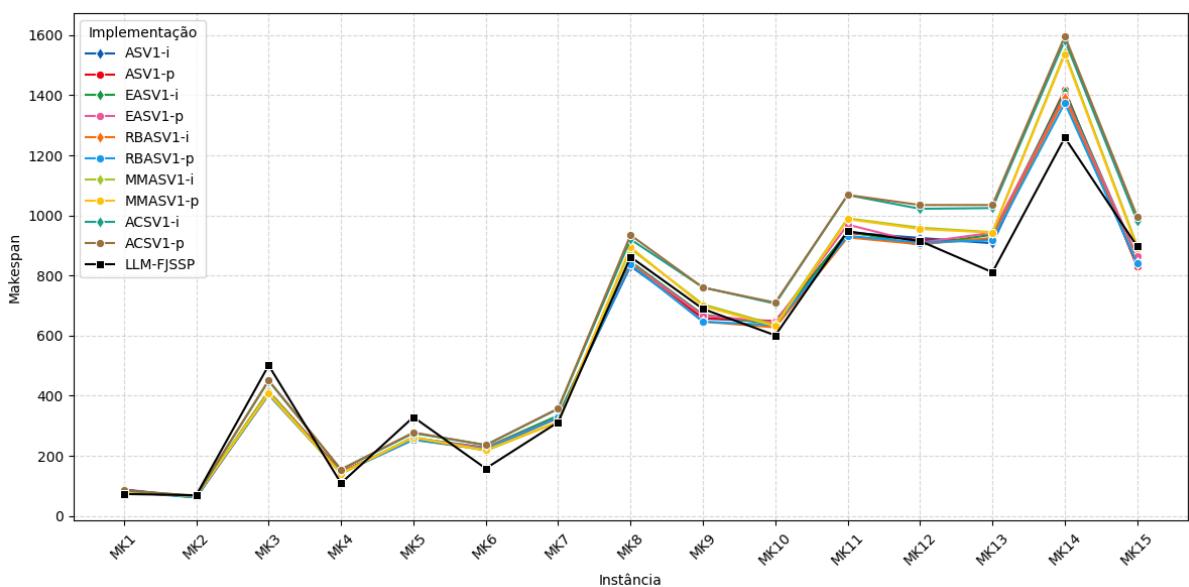


Figura 31 – *Makespan* médio do conjunto V1 para as instâncias de de Brandimarte (1993)

⁴ Isso pode ser explicado pelo fato de que a heurística de construção das soluções é exatamente a mesma, e as diferenças entre os algoritmos pouco interferem no processo de construção.

Para além da ineficiência dessas versões, começamos a observar algumas coisas interessantes sobre os diferentes algoritmos estudados. Como esperado, a etapa de atualização de feromônio dos algoritmos é o ponto que mais impacta seu tempo de execução. Em razão disso, os algoritmos AS e EAS despontam com os piores tempos médios de CPU justamente por terem a atualização de feromônio menos econômica. Por outro lado, os algoritmos RBAS, MMAS e ACS se beneficiam de suas políticas de atualização que restringem a quantidade de formigas atuando nesta etapa. Pelo mesmo motivo, o algoritmo ACS é consistentemente o mais rápido, uma vez que a quantidade de formigas adotada nos testes é constante em todas as execuções — vide Seção 7.1. Muitas vezes, o tempo médio de CPU desse algoritmo ACO é comparável ao de LLM-FJSSP.

Outro aspecto que vale a pena mencionar começou a ficar perceptível em instâncias maiores. Dentre os algoritmos ACO estudados, o ACS é o único que implementa um procedimento de transformação local de feromônio. Como mencionamos no capítulo anterior, nas abordagens paralelas a estrutura de feromônios foi implementada através de uma estrutura de dados *thread safe* para garantir a sincronização da estrutura. Isso quer dizer que, em condições de corrida, a operação de atualização deve falhar. É exatamente isso que acontece na sobrescrita do método **LocalPheromoneUpdate**, exibido no Fragmento 6.15: a atualização local só ocorre se o método **TryUpdate** retornar um valor verdadeiro. Foi identificado que cerca de 2% das atualizações locais de feromônio falham, o que pode explicar uma ligeira superioridade no *makespan* médio dos algoritmos ACS que seguem uma abordagem iterativa.

7.3.2 That's why he's the GOAT

Abandonar o modelo de digrafo disjuntivo foi extremamente benéfico para a performance dos algoritmos, ao ponto de possibilitar testes com mais instâncias da literatura. As figuras 32, 33 e 34 ilustram em seus gráficos as curvas do tempo médio de CPU dos 10 algoritmos que implementam a heurística V2. Podemos observar que, comparado às figuras 28 e 29, o tempo médio de CPU abaixou consideravelmente. De fato, olhando para as instâncias de Fattahi et al. (2007), mesmo considerando o pior tempo médio de CPU registrado entre todos os algoritmos do conjunto V2, houve uma redução de 94,1% em comparação com o pior tempo médio de CPU observado no conjunto V1 (de 1,83 s para 108,1 ms). No caso das instâncias de Brandimarte (1993), o mesmo foi observado, com uma redução de 99,6% (de 16,5 min para 4,01 s).

Como observado nos testes preliminares, os algoritmos que seguem abordagens iterativas tendem a performar um pouco pior que os que seguem uma abordagem paralela. Outra característica que se manteve foi a vantagem que as variantes RBAS, MMAS e ACS têm sobre seus concorrentes devido às suas políticas de atualização de feromônio. De fato, olhando para os gráficos, essa vantagem pode parecer supérflua

frente aos resultados obtidos com a redução do tempo médio de CPU. Contudo, observaremos seu verdadeiro impacto na Seção 7.4 quando abordarmos instâncias bem maiores.

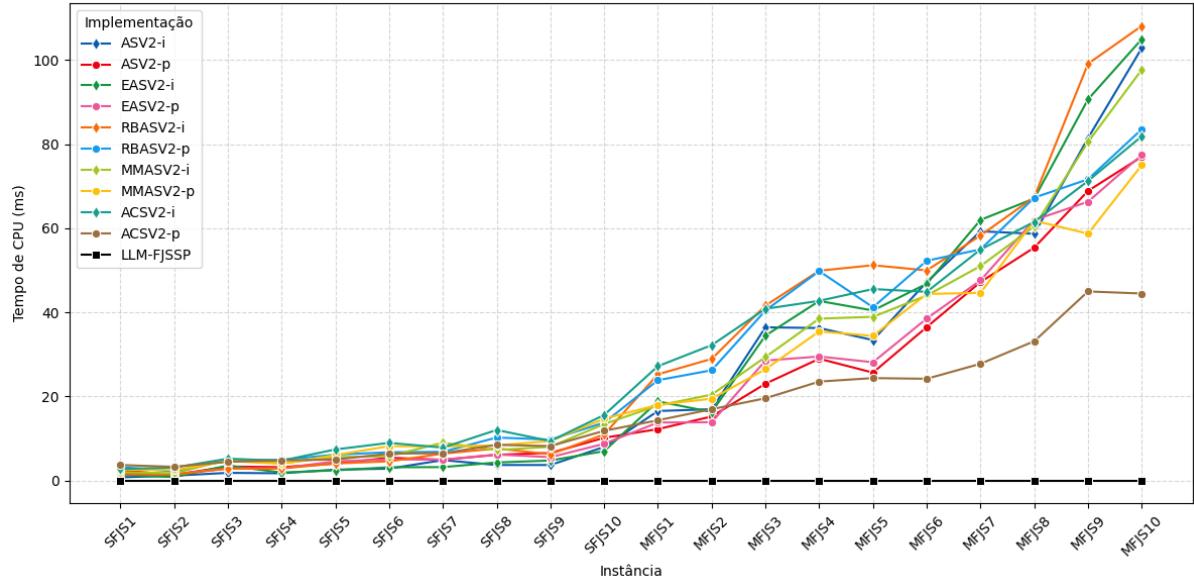


Figura 32 – Tempo médio de CPU do conjunto V2 para as instâncias de Fattahi et al. (2007)

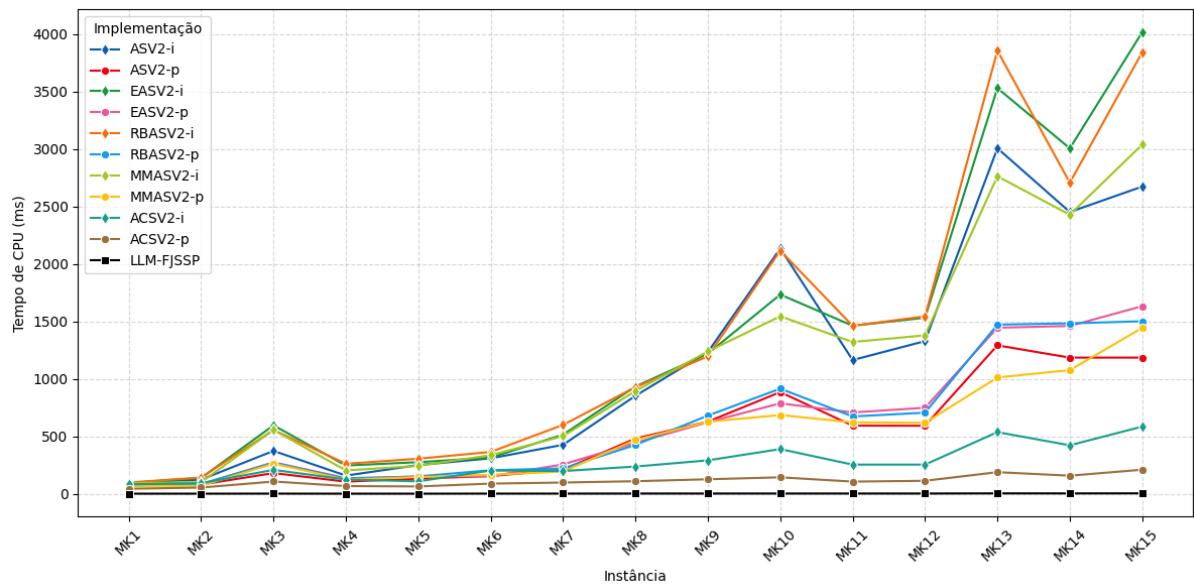


Figura 33 – Tempo médio de CPU do conjunto V2 para as instâncias de Brandimarte (1993)

As instâncias de Dauzère-Pérès e Paulli (1997) são comparáveis às de Brandimarte (1993) em termos de tamanho. Em razão disso, esperava-se que o tempo médio de CPU observado para essas instâncias seguisse próximo aos de Brandimarte (1993). De fato,

ambos os conjuntos de instâncias atingiram seus piores tempos médios em instâncias cujo tempo médio de CPU foi cerca de 4 segundos.

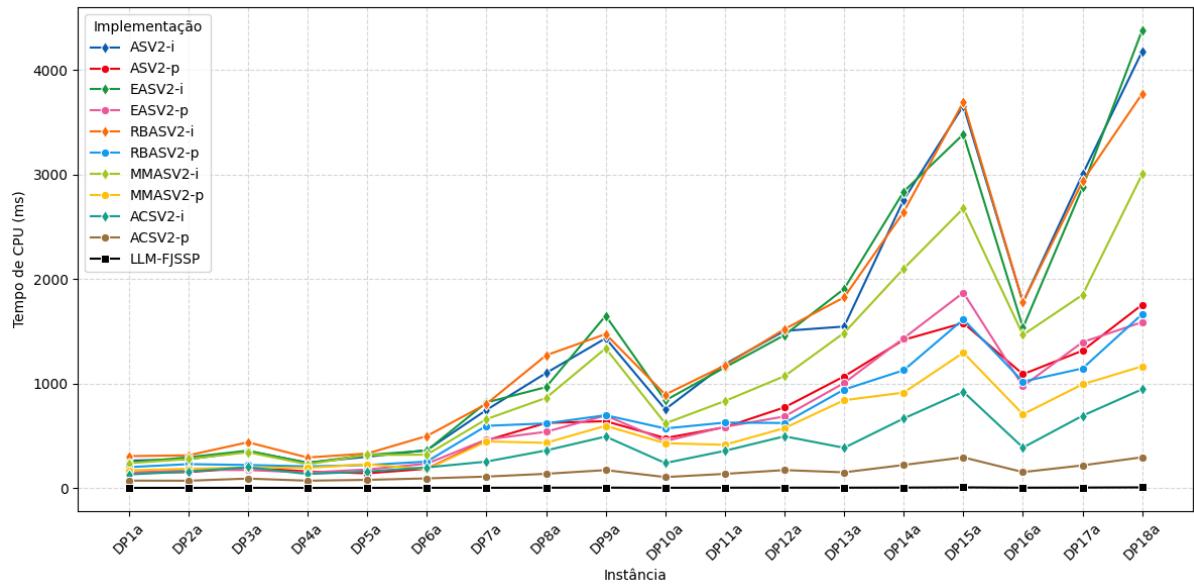


Figura 34 – Tempo médio de CPU do conjunto V2 para as instâncias de Dauzère-Pérès e Paulli (1997)

O êxito da heurística V2 não se restringe ao tempo médio de CPU mais baixo. A nova heurística implementada promoveu modificações na estrutura de vizinhança do problema, diminuindo a quantidade de movimentos viáveis enumerados a cada passo de construção. Por tabela, isso permitiu uma maior exploração do espaço de soluções e, consequentemente, resultados melhores. As figuras 35, 36 e 37 exibem nas curvas de seus gráficos evidências de que essa nova heurística possibilitou um valor de *makespan* médio mais baixo do que suas predecessoras. De fato, nas instâncias de Fattahi et al. (2007) o *gap* médio baixou para 28,51% (contra 67,96% de LLM-FJSSP) e para 93,08% (contra 215,11% de LLM-FJSSP) nas instâncias de Brandimarte (1993). No caso das instâncias de Dauzère-Pérès e Paulli (1997) os resultados foram mais modestos, embora muito melhores que os da heurística gulosa: um *gap* médio de 101,43% contra 235,26% de LLM-FJSSP.

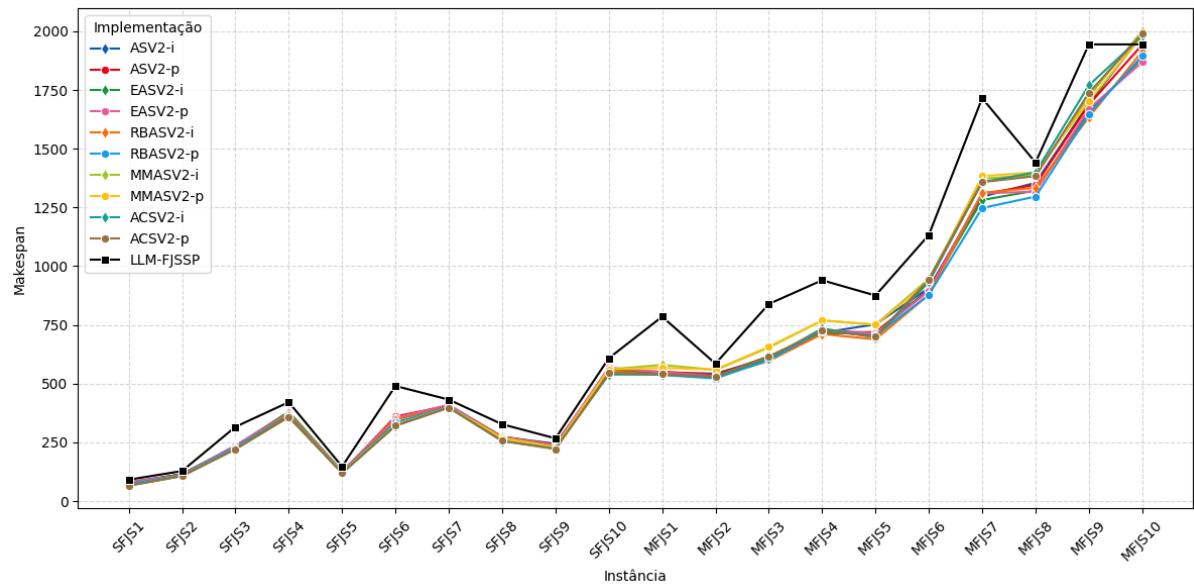


Figura 35 – Makespan médio do conjunto V2 para as instâncias de Fattahi et al. (2007)

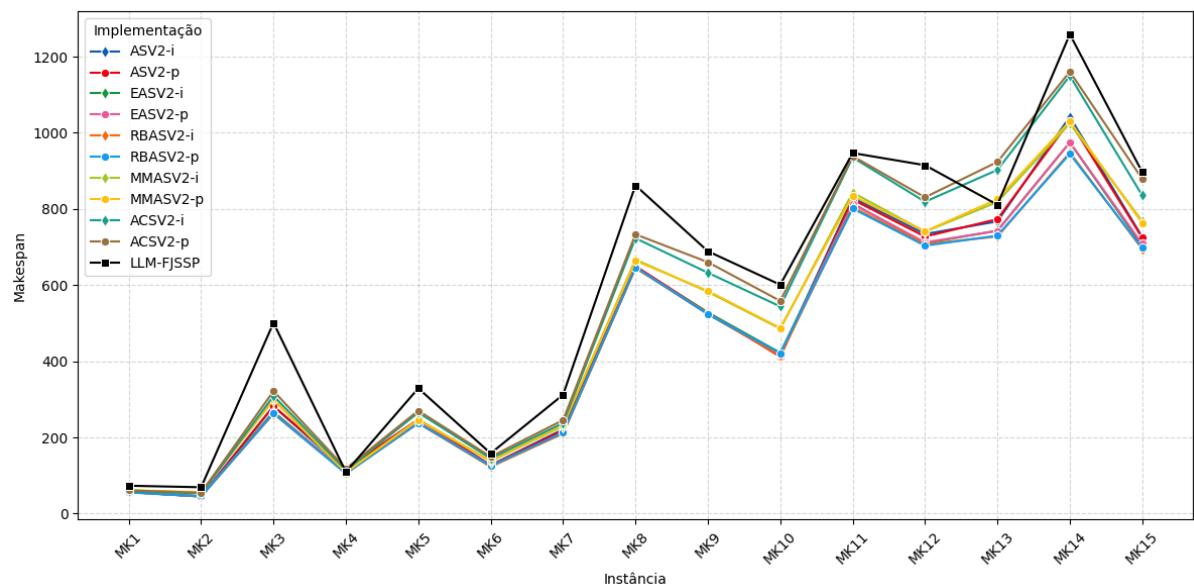


Figura 36 – Makespan médio do conjunto V2 para as instâncias de Brandimarte (1993)

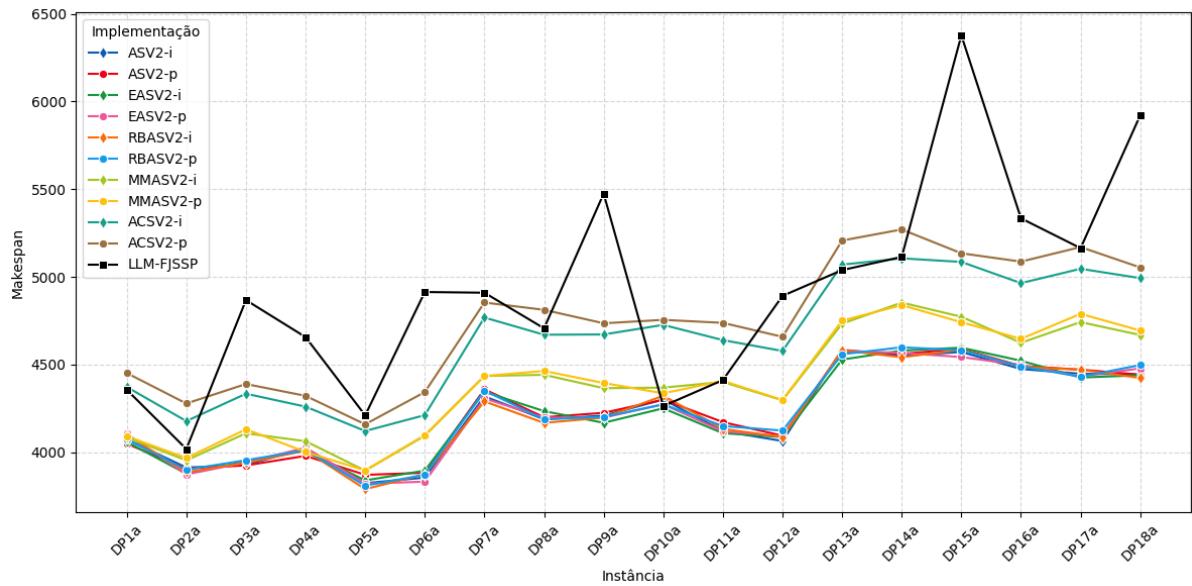


Figura 37 – *Makespan* médio do conjunto V2 para as instâncias de Dauzère-Pérès e Paulli (1997)

A essa altura já estava claro que a abordagem paralela é muito mais vantajosa do que a abordagem iterativa. Além disso, as variantes AS e EAS amargaram na última posição em quase todos os testes de duração realizados. Como observado anteriormente, o *makespan* médio de todos os algoritmos ACO em todas as instâncias é muito próximo, de modo que as curvas de *makespan* médio dos algoritmos ACO em todos os gráficos seguem a mesma tendência. Isso sugere que restringir a atuação das formigas em troca de maior velocidade na execução do algoritmo parece uma troca justa. Em razão disso, mais testes foram realizados, submetendo outras instâncias da literatura para os algoritmos mais bem-sucedidos até aqui.

O conjunto composto pelos algoritmos RBASV2-p, MMASV2-p, ACSV2-i e ACSV2-p, denominado TOPV2, foi escolhido para mais testes com as instâncias de Hurink et al. (1994). Primeiramente, os algoritmos foram executados sobre as instâncias sdata, originadas na literatura sobre o JSSP. Os gráficos nas figuras 38 e 39 exibem o comportamento desses algoritmos diante de um cenário sem flexibilidade de máquinas. O tempo médio de CPU de todos os algoritmos se manteve na casa dos décimos de segundo, mesmo para as instâncias maiores (o pior tempo médio de CPU foi de 732,83ms). Isso pode ser explicado pela falta de flexibilidade de máquina nesse conjunto de instâncias, de modo que o espaço de soluções é menor do que seria em casos de flexibilidade. No caso do *makespan* médio, o conjunto de algoritmos TOPV2 traz consigo um *gap* médio de 64,38%, contra 171,38% de LLM-FJSSP para sdata.

Contudo, comparando o *makespan* médio com os resultados de algoritmos desenvolvidos para o JSSP, um sinal de alerta se acendeu. Mahmud et al. (2022) realizou uma implementação de uma heurística evolutiva híbrida mais sofisticada, integrada

a um algoritmo de busca local. Essa implementação deu origem a três algoritmos, denominados sHEA, rHEA e bHEA, cujos *gaps* médios nestas instâncias são absurdamente menores: 0,72% de sHEA, 0,82% para rHEA e 1,45% para bHEA. Isso pode ser explicado por algumas coisas, como o fato de que esses algoritmos implementam um procedimento de busca local, ou ainda, pelo fato de que esses algoritmos foram projetados diretamente para o JSSP, diferentemente dos algoritmos deste trabalho. Infelizmente, Mahmud et al. (2022) não disponibilizou informações sobre o tempo de CPU de seus algoritmos, de modo que não é possível avaliar o custo-benefício das nossas implementações neste cenário.

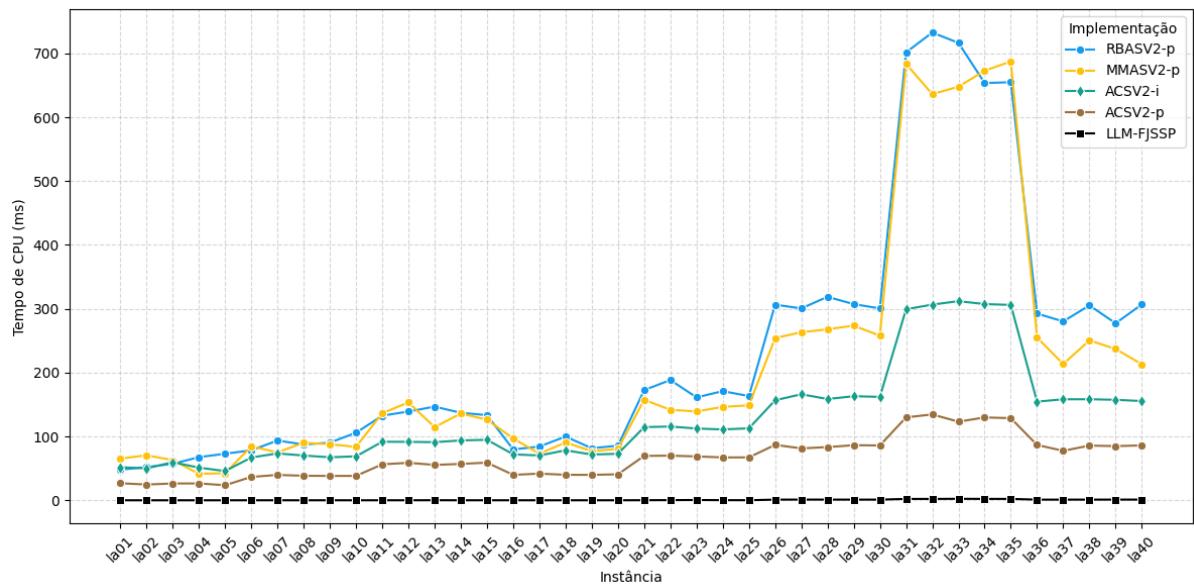


Figura 38 – Tempo médio de CPU do conjunto TOPV2 para as instâncias sdata de Hurink et al. (1994)

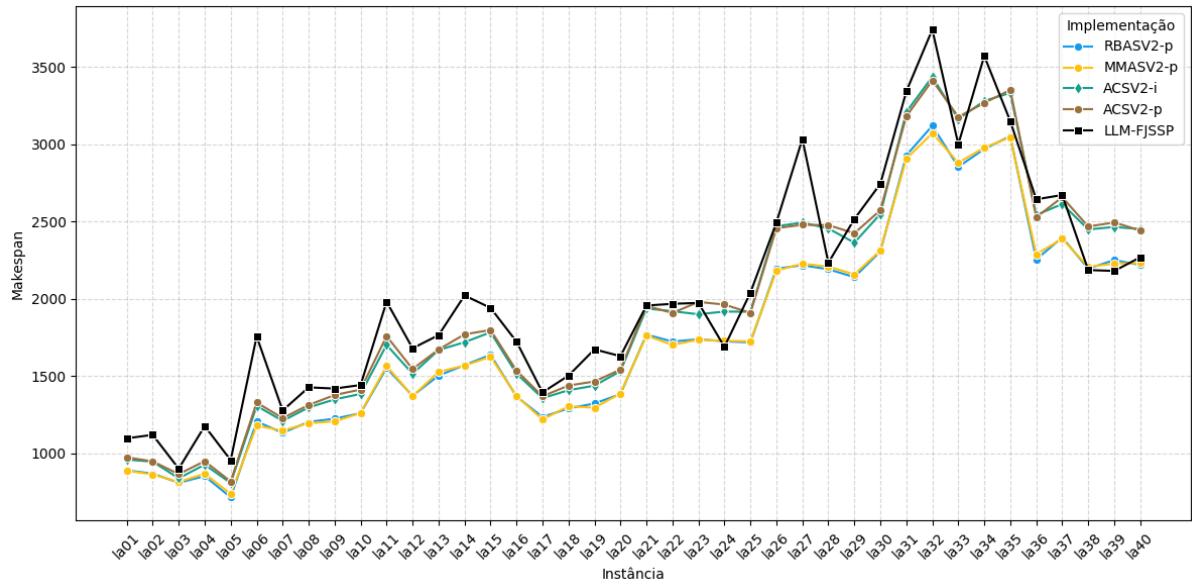


Figura 39 – *Makespan* médio do conjunto TOPV2 para as instâncias sdata de Hurink et al. (1994)

Olhando para as instâncias vdata, observamos nos gráficos 40 e 41 um salto no tempo médio de CPU e no *makespan* médio. Isso era esperado, uma vez que essas instâncias possuem um grau maior de flexibilidade de máquina. O maior tempo médio de CPU passou de 732,83ms em sdata para 1,9s (cerca de 2,6x vezes maior) em vdata. Já o *makespan* médio passou para um *gap* médio de 96,29%, contra 218,99% de LLM-FJSSP.

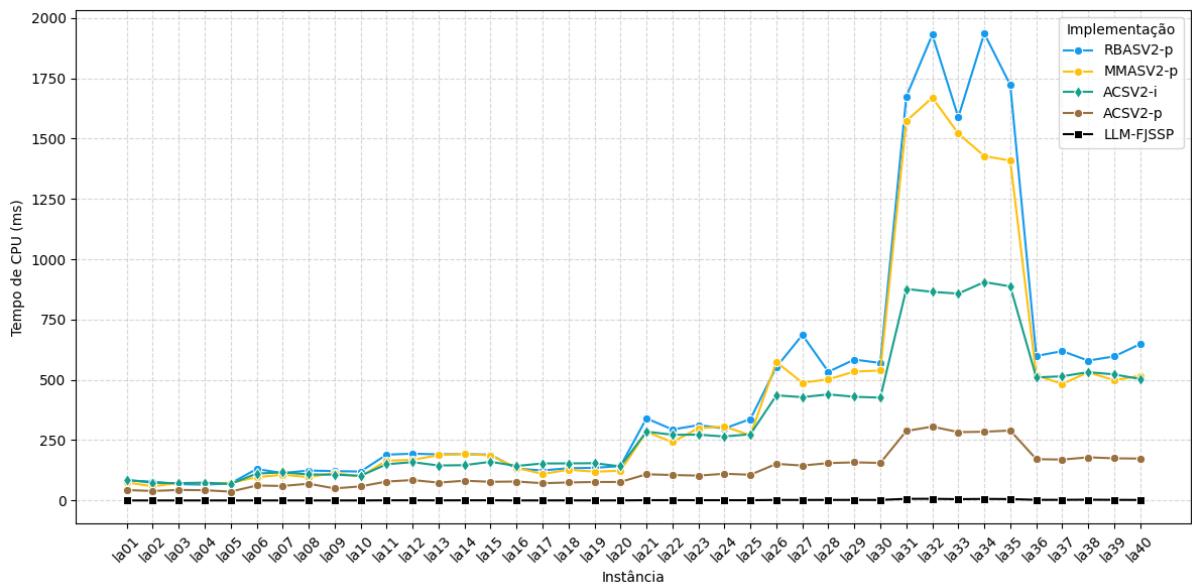


Figura 40 – Tempo médio de CPU do conjunto TOPV2 para as instâncias vdata de Hurink et al. (1994)

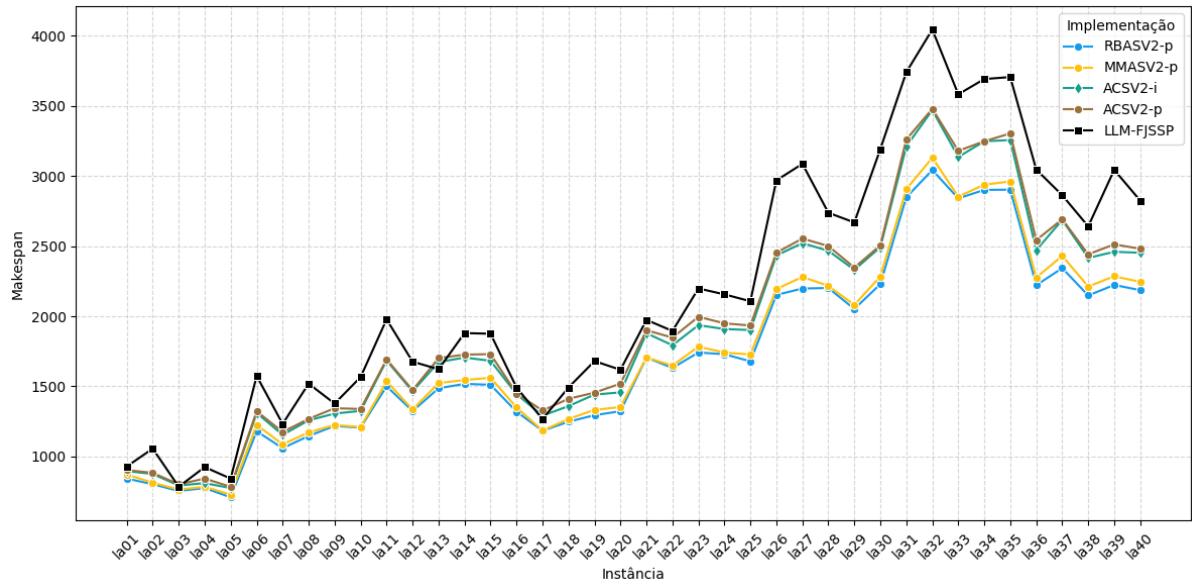


Figura 41 – *Makespan* médio do conjunto TOPV2 para as instâncias vdata de Hurink et al. (1994)

Os resultados obtidos com os algoritmos do conjunto TOPV2 se somam ao que foi observado nos testes anteriores. O RBAS e o MMAS disputam pelo pódio de melhor *makespan* médio, com uma leve vantagem do primeiro. Por outro lado, o ACS se apresenta como uma alternativa competitiva aos outros dois, conseguindo resultados não muito discrepantes, apesar da quantidade adotada de formigas ser reduzida. Isso pode ser explicado pela sua política de atualização local de feromônio e à regra de probabilidade pseudo-aleatória que permitem uma maior exploração do espaço de soluções. O número de formigas adotado, inclusive, coloca o ACS numa posição de altíssimo custo-benefício, uma vez que o *makespan* médio ligeiramente mais alto pode ser compensado em alguns cenários pelo seu tempo médio de CPU, que é muitas vezes comparável ao de LLM-FJSSP.

7.3.3 Quem generaliza os generalizadores?

Como mencionado na Seção 6.4, abandonar o modelo de digrafo disjuntivo reduz a aplicabilidade do método de solução para casos mais gerais. Com isso em mente, uma nova implementação da heurística de construção (denominada V3) foi desenvolvida para admitir relações de precedências não lineares, e com ela mais 10 variações de algoritmos ACO foram implementados.

Por falta de tempo hábil, esses algoritmos não puderam ser submetidos a instâncias mais gerais do problema. No entanto, para as instâncias usadas nos testes anteriores, esperava-se que os algoritmos V3 gerassem soluções tão boas quanto as geradas por V2 e que seu tempo médio de CPU fosse pelo menos superior ao de V1.

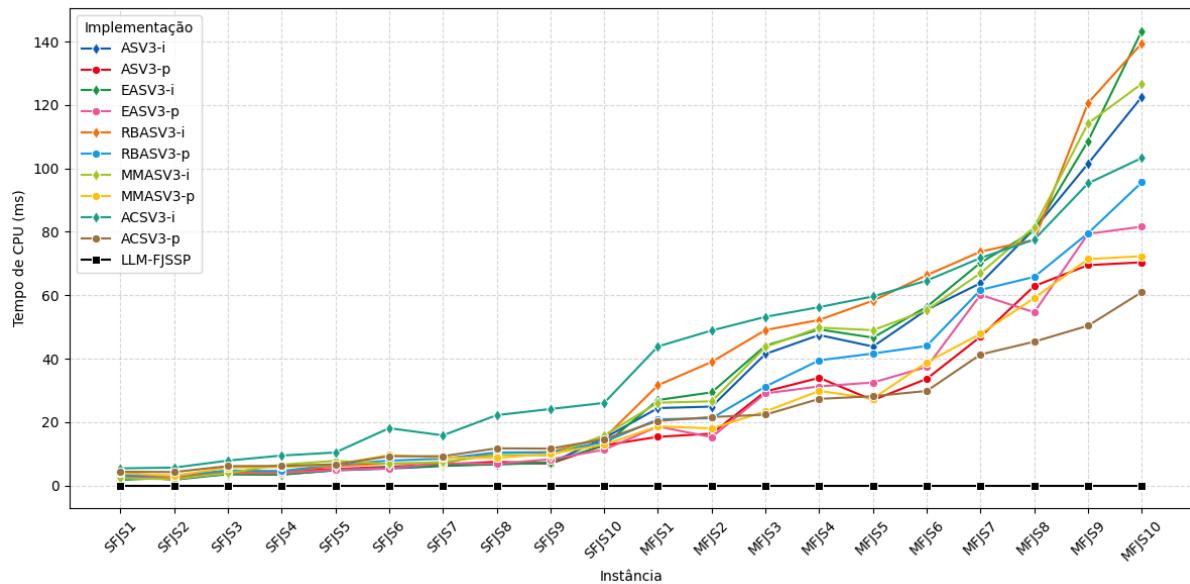


Figura 42 – Tempo médio de CPU do conjunto V3 para as instâncias de Fattahi et al. (2007)

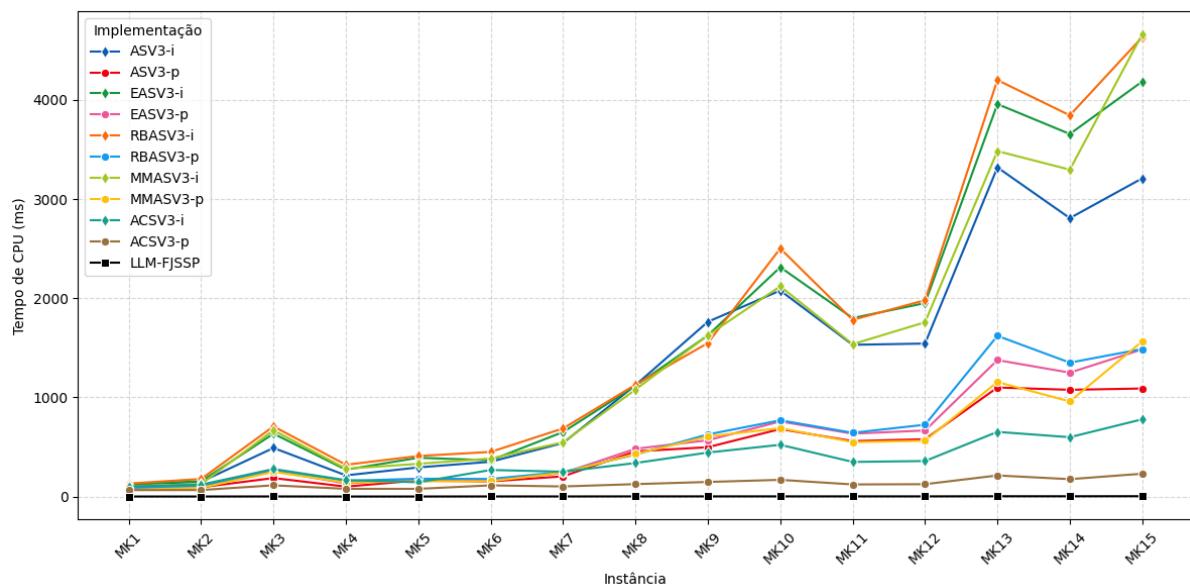


Figura 43 – Tempo médio de CPU do conjunto V3 para as instâncias de Brandimarte (1993)

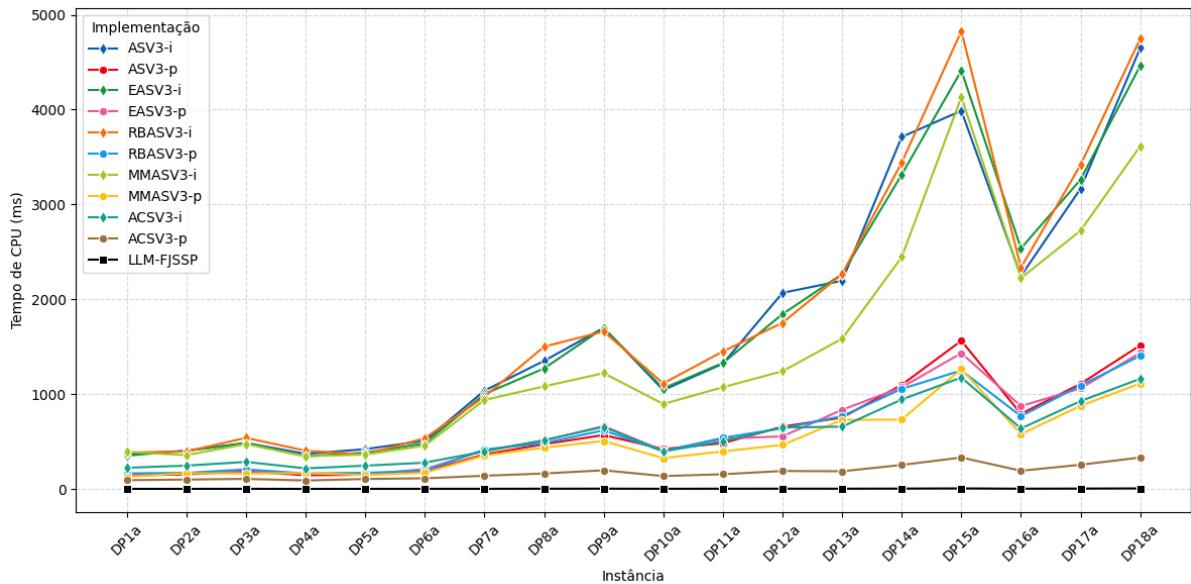


Figura 44 – Tempo médio de CPU do conjunto V3 para as instâncias de Dauzère-Pérès e Paulli (1997)

Os resultados foram animadores. Flexibilizar as relações de precedência não causou prejuízo no *makespan* médio comparado aos resultados de V2. O *gap* médio se manteve: 28,78% (contra 67,96% de LLM-FJSSP) nas instâncias de Fattahi et al. (2007), 92,89% (contra 215,11%) nas de Brandimarte (1993) e 101,58% (contra 235,26%) nas de Dauzère-Pérès e Paulli (1997). Mais do que isso, o tempo médio de CPU foi muito mais próximo de V2 do que de V1. Isso pode ser explicado pelo mesmo motivo que tornou V2 tão mais rápida que V1: o algoritmo é muito mais econômico ao computar os movimentos viáveis, uma vez que não há uma explosão combinatória como no digrafo disjuntivo de Rossi e Dini (2007).

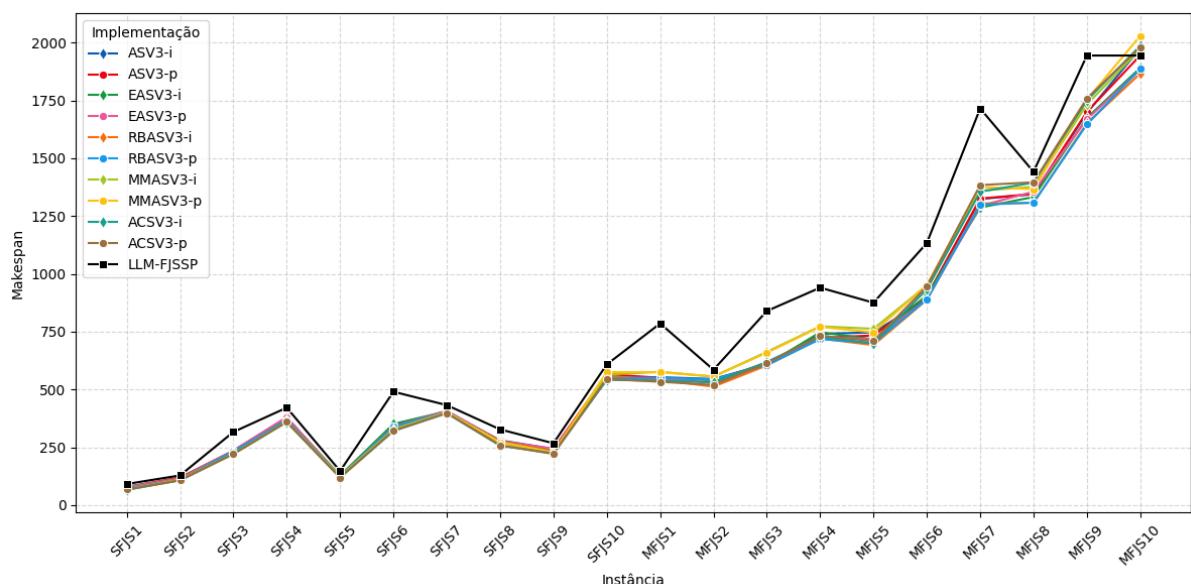


Figura 45 – Makespan médio do conjunto V3 para as instâncias de Fattahi et al. (2007)

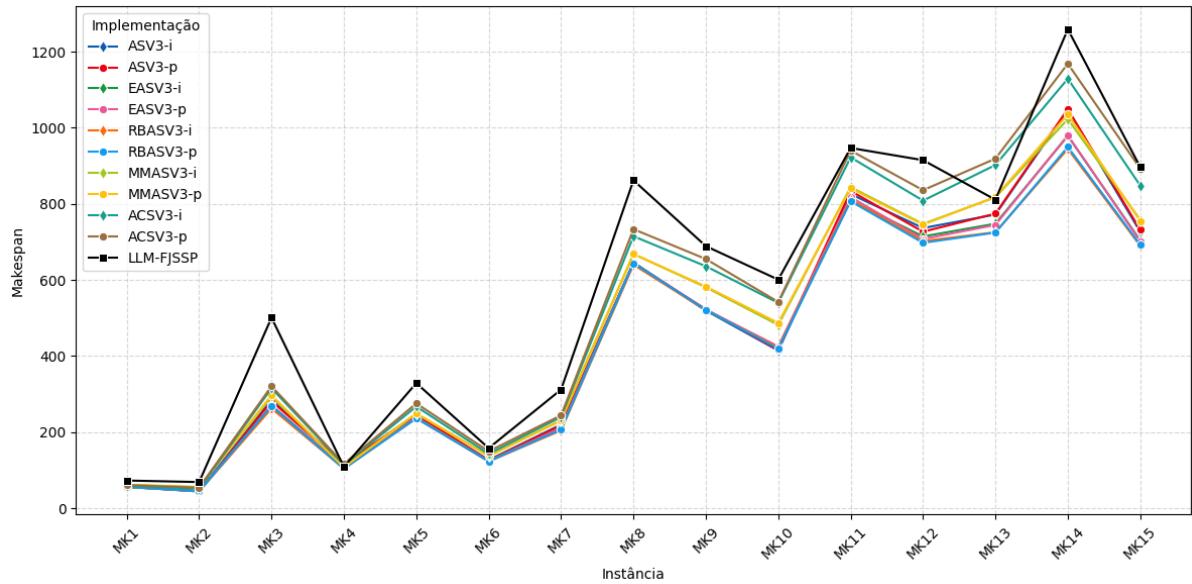


Figura 46 – Makespan médio do conjunto V3 para as instâncias de Brandimarte (1993)

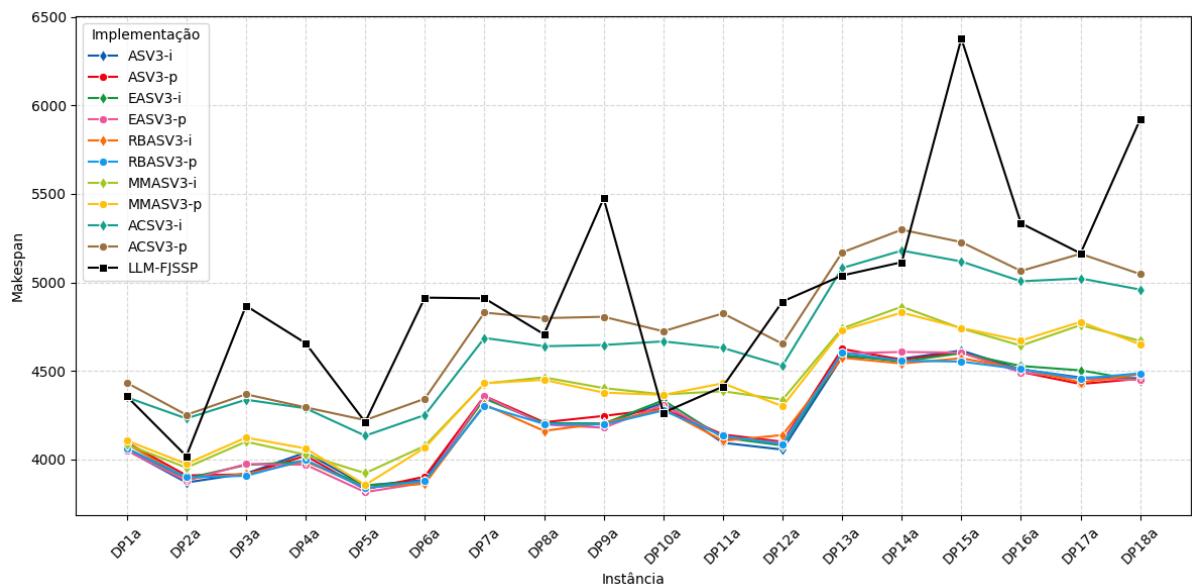


Figura 47 – Makespan médio de V3 para as instâncias de Dauzère-Pérès e Paulli (1997)

Análogo ao que foi realizado com o conjunto de algoritmos V2, um conjunto denominado TOPV3 foi utilizado para testes adicionais sobre as instâncias de Hurink et al. (1994). O gráfico na Figura 49 indica, como esperado, que o *makespan* médio não diverge muito do obtido pelos algoritmos de TOPV2. De fato, o *gap* médio para as instâncias de sdata foi de 64,16%, contra 171,38% obtido pela heurística LLM-FJSSP. Já o tempo médio de CPU observado, ilustrado na Figura 48, foi ligeiramente maior que o anterior: o pior tempo médio de CPU vem de MMASV3-p que gastou em média 859,9ms na instância la34.

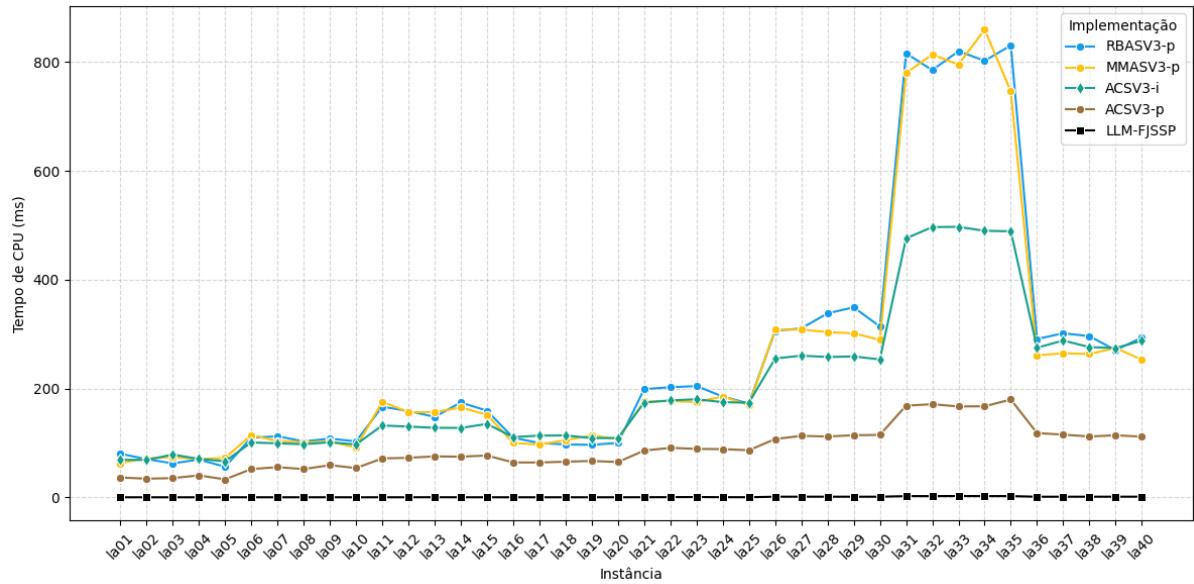


Figura 48 – Tempo médio de CPU do conjunto TOPV3 para as instâncias sdata de Hurink et al. (1994)

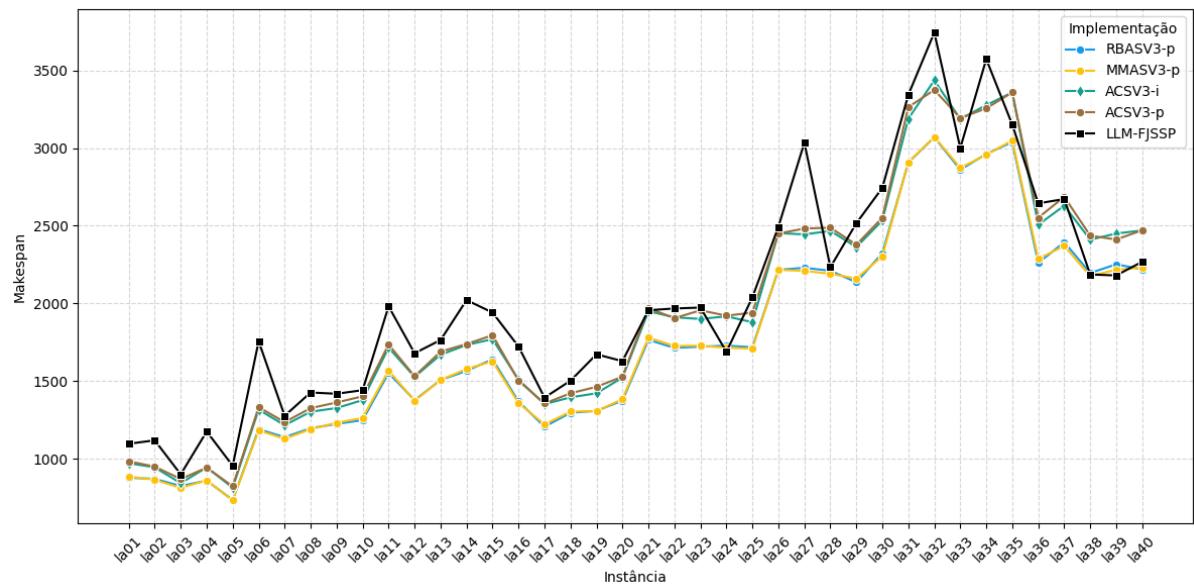


Figura 49 – Makespan médio do conjunto TOPV3 para as instâncias sdata de Hurink et al. (1994)

Por fim, os testes sobre as instâncias vdata são protocolares, pois não trazem nenhuma informação nova. As figuras 51 e 51 apenas endossam o padrão observado até aqui, com um gap médio de 96,31% (contra 218,99%) e um tempo médio de CPU ligeiramente mais alto (um pior tempo médio de 2,09s pelo algoritmo RBASV3-p na instância la35).

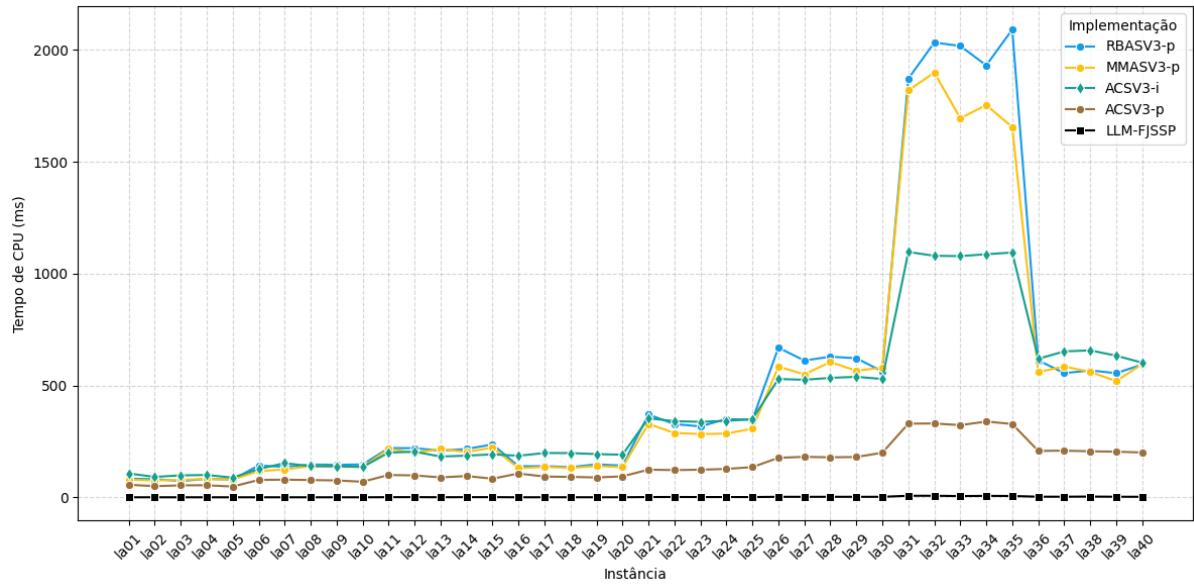


Figura 50 – Tempo médio de CPU do conjunto TOPV3 para as instâncias vdata de Hurink et al. (1994)

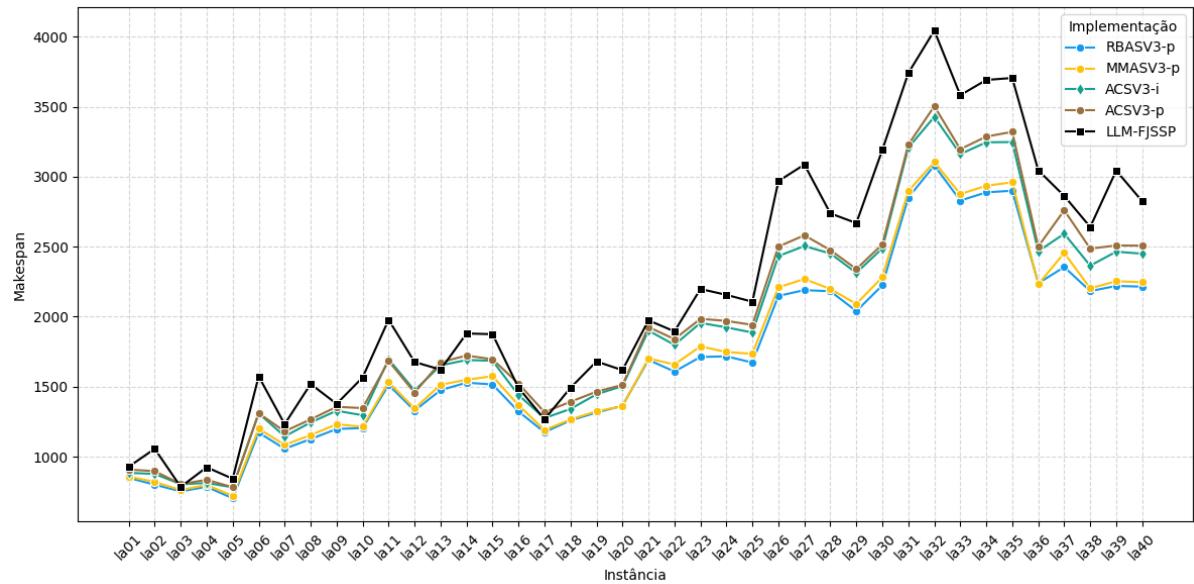


Figura 51 – Makespan médio do conjunto TOPV3 para as instâncias vdata de Hurink et al. (1994)

7.3.4 Síntese

Reservaremos esse momento para uma pequena síntese dos resultados obtidos nos experimentos sobre as instâncias clássicas da literatura. Dentre as 32 variações de algoritmos ACO implementadas, observamos diferentes variantes do ACO, diferentes abordagens de execução e variações na heurística construtiva empregada. Em razão disso, para facilitar a análise, os algoritmos foram divididos em 4 conjuntos, de acordo

com a heurística construtiva utilizada. O gráfico na Figura 52 ilustra uma representação do *gap* médio por conjunto de algoritmos e conjunto de instâncias.

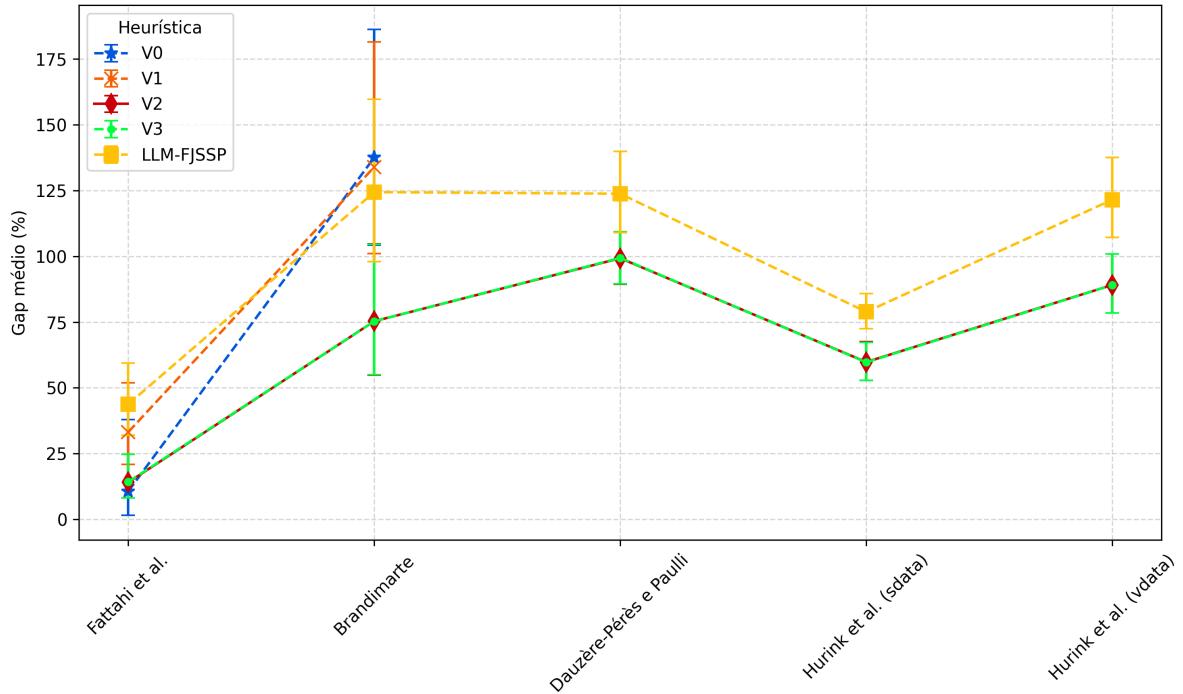


Figura 52 – *Gap* percentual médio por conjunto de algoritmos e conjunto de instâncias.

Na Subseção 7.3.1 vemos que os testes com V0 e V1 se restringiram às instâncias de Fattahi et al. (2007) e Brandimarte (1993). Essas instâncias foram suficientes para concluir que os algoritmos implementados baseados no modelo de digrafo disjuntivo flexível performaram abaixo do esperado, tanto em qualidade quanto em velocidade.

A Subseção 7.3.2 traz à tona a discussão sobre os resultados da execução dos algoritmos do conjunto V2, cuja heurística construtiva foca na linearidade das relações de precedência do problema para simplificar a estrutura de vizinhança. Testes com as instâncias de Fattahi et al. (2007), Brandimarte (1993) e Dauzère-Pérès e Paulli (1997) mostram que os melhores *gaps* médios e tempos médios de CPU foram obtidos através dessa heurística. Ademais, os algoritmos com melhores resultados nestes testes foram escolhidos para testes adicionais sobre as instâncias de Hurink et al. (1994) que consistem em adaptações de instâncias clássicas da literatura para o JSSP. Observamos nestes testes indícios de que a qualidade das soluções obtidas está aquém do que o estado da arte pode proporcionar. Uma possível explicação para isso é a falta de um procedimento de busca local para refinar as soluções geradas.

Em seguida, na Subseção 7.3.3 avaliamos os algoritmos V3 obtidos após uma adaptação em V2 para admitir relações de precedência não lineares. Como esperado, não houve impacto no *makespan* médio, uma vez que as instâncias testadas foram

essencialmente as mesmas. Para afirmações mais concretas a respeito dessa heurística, os testes carecem de instâncias com relações de precedência mais gerais. O que se pode dizer objetivamente é que o ajuste veio com tempos médios de CPU mais altos em relação a V2, ainda que bem próximos na grande maioria dos casos.

Em todas as implementações, todos os algoritmos paralelos foram mais rápidos do que suas contrapartes iterativas, sem exceção. Contudo, durante a fase de testes, observou-se que, principalmente em casos onde o número de formigas excede o número de processadores, o potencial das CPUs não é completamente utilizado. A Figura 53 exibe uma captura de tela do monitor de recursos do sistema operacional que atesta esse fenômeno.

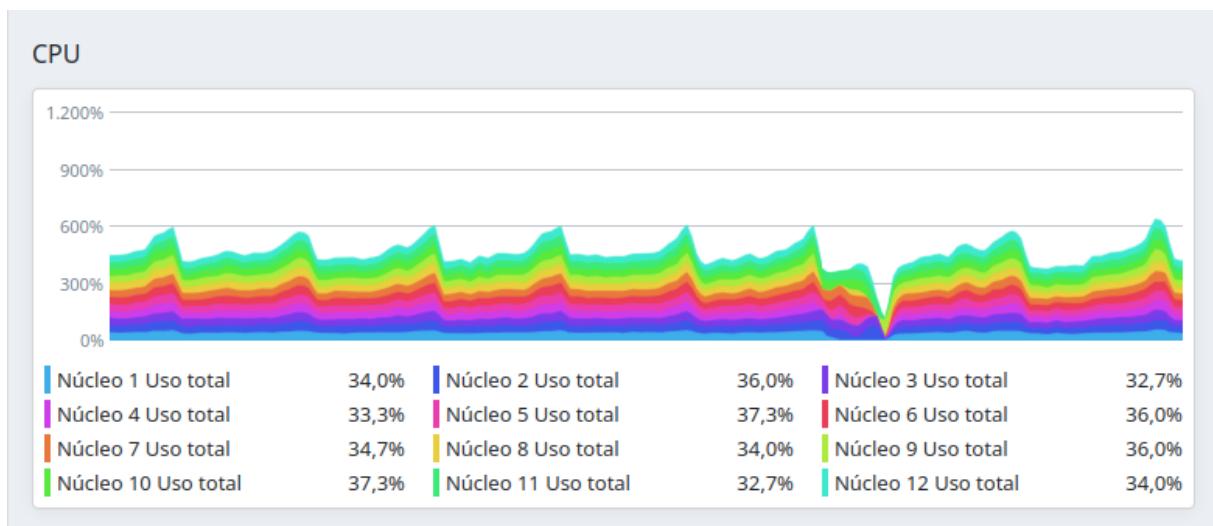


Figura 53 – Captura de tela do monitor de recursos do sistema operacional.

Por fim, no tocante aos algoritmos ACO estudados, os experimentos confirmam a intuição proporcionada pela teoria. Alguns aspectos que valem a pena ressaltar:

- Os algoritmos AS e EAS são consistentemente mais lentos que os demais, uma vez que todas as formigas participam da atualização das estruturas de feromônio; Em compensação, frequentemente apresentam os melhores resultados de *makespan* médio;
- O algoritmo ACS consegue com pouquíssimas formigas resultados competitivos com os demais. Isso torna este algoritmo a alternativa de maior custo-benefício;
- Os algoritmos RBAS e MMAS se colocam como um meio termo entre um tempo de CPU razoável e qualidade das soluções próximas as de AS e EAS.

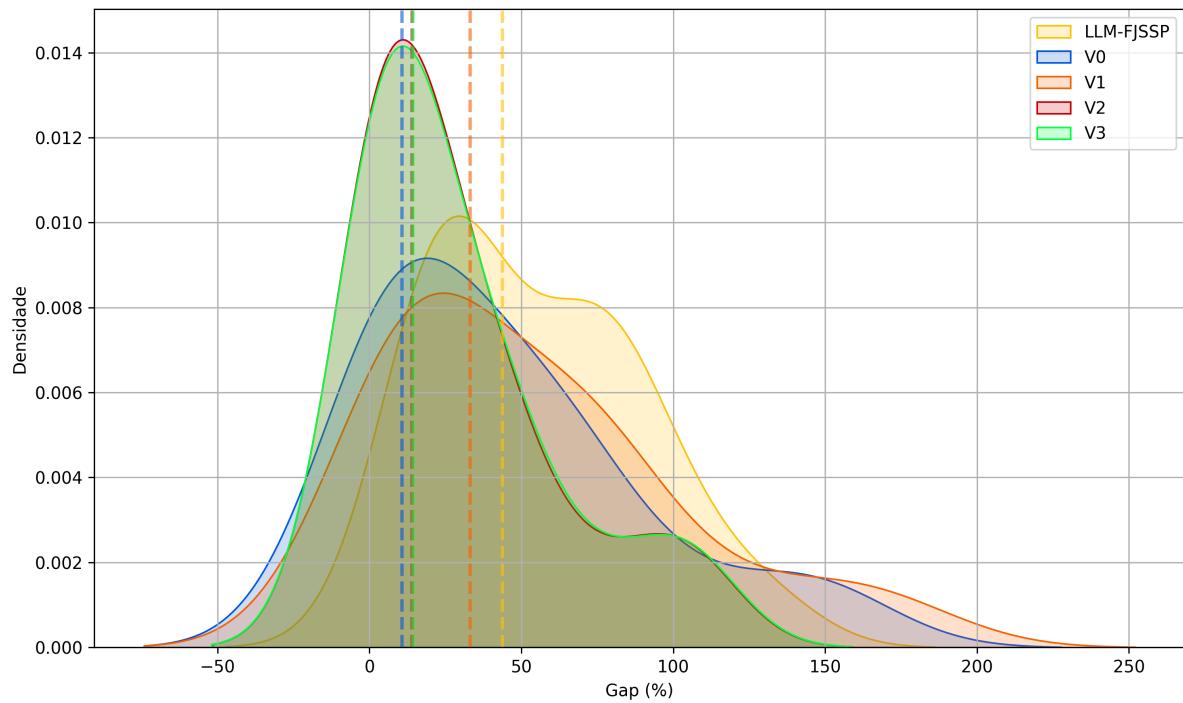


Figura 54 – Estimativas de densidade por *kernel* dos gaps médios para as instâncias de Fattahi et al. (2007)

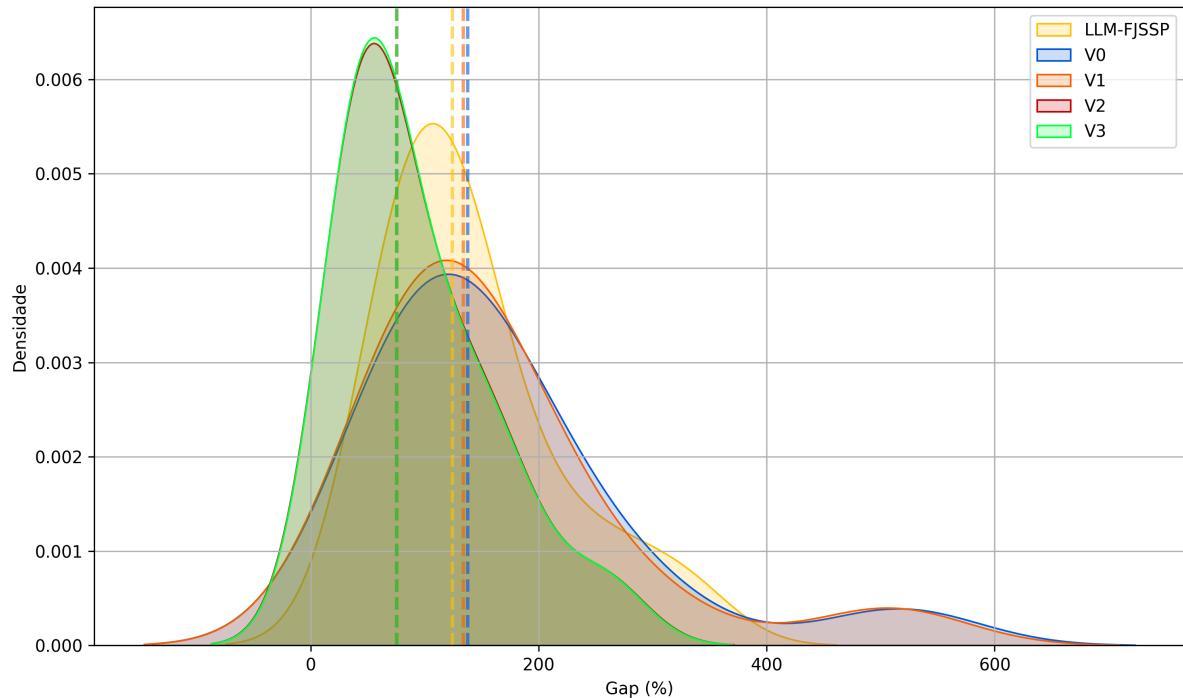


Figura 55 – Estimativas de densidade por *kernel* dos gaps médios para as instâncias de Brandimarte (1993)

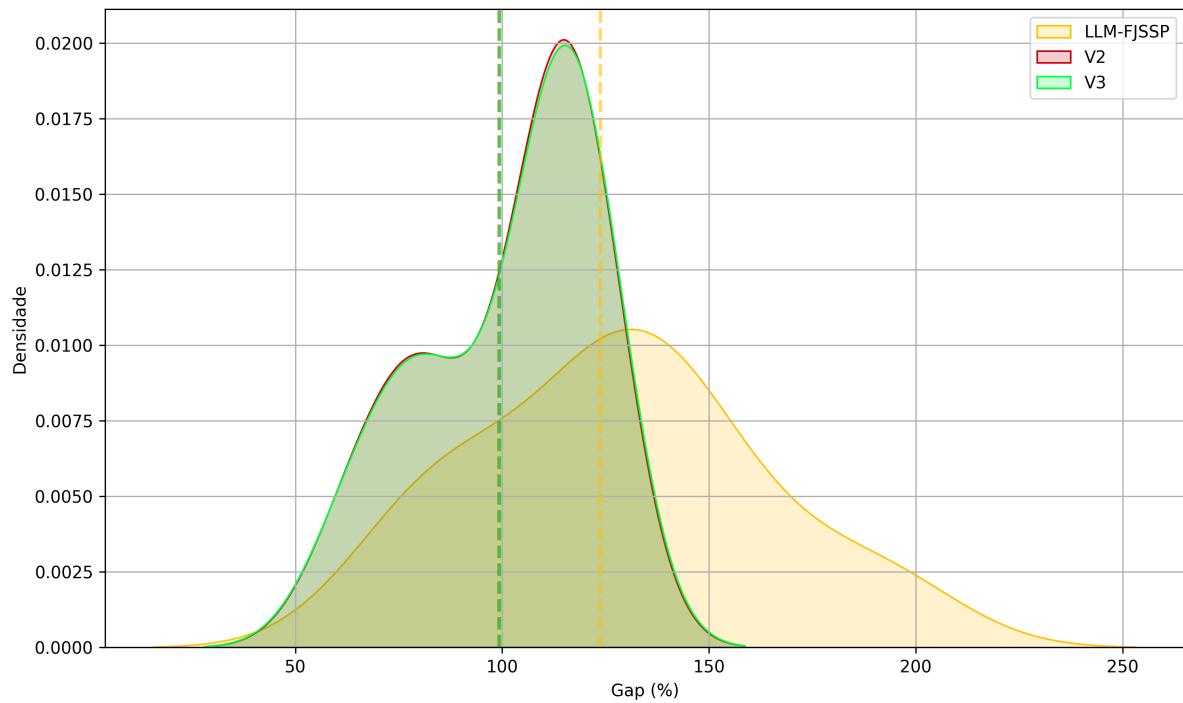


Figura 56 – Estimativas de densidade por *kernel* dos *gaps* médios para as instâncias de Dauzère-Pérès e Paulli (1997)

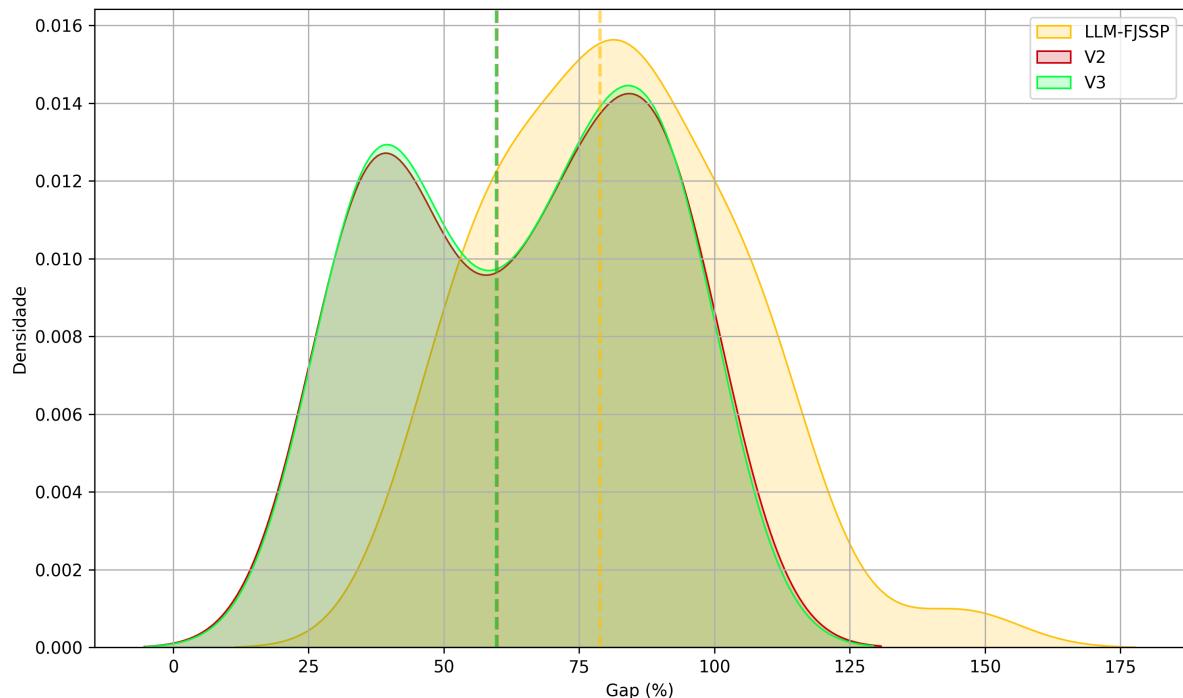


Figura 57 – Estimativas de densidade por *kernel* dos *gaps* médios para as instâncias sdata de Hurink et al. (1994)

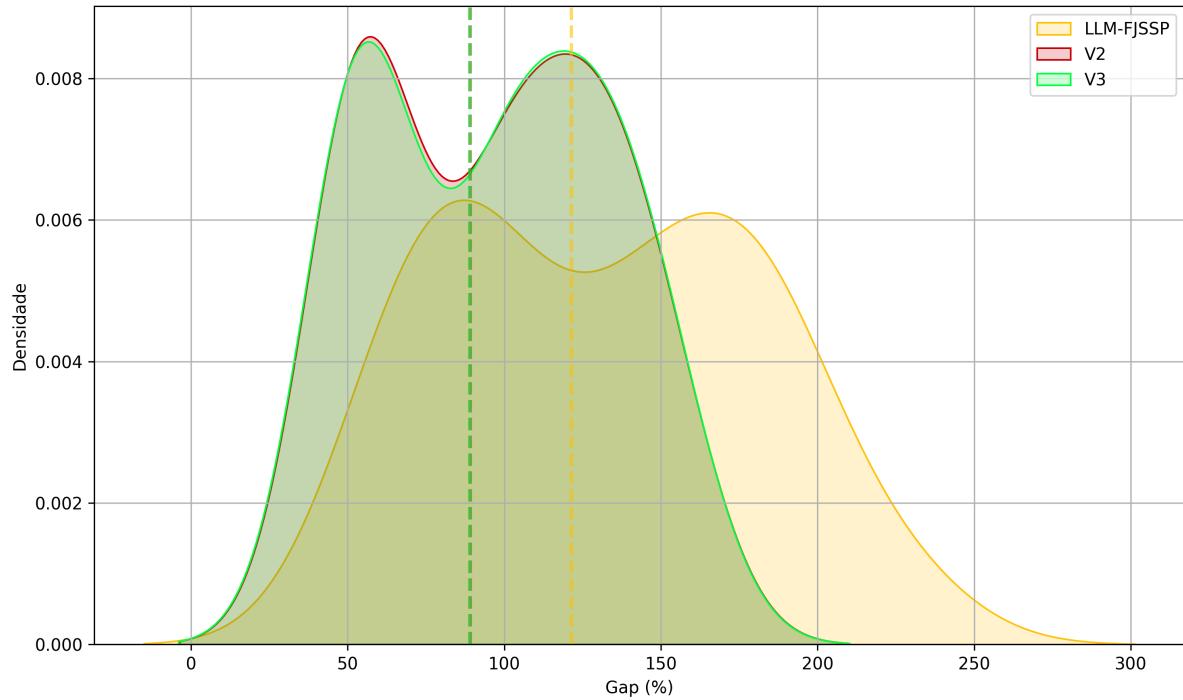


Figura 58 – Estimativas de densidade por *kernel* dos *gaps* médios para as instâncias vdata de Hurink et al. (1994)

7.4 Novo conjunto de instâncias

Após avaliar a capacidade dos conjuntos de algoritmos V2 e V3 de gerar soluções para os principais conjuntos de instâncias encontrados na literatura, chegou a hora de avaliar sua performance em situações mais próximas da realidade da empresa interessada. Para tal, um conjunto com 3 instâncias foi desenvolvido baseado nos dados de diferentes clientes da empresa.

Neste ponto, é importante mencionar a metodologia empregada na montagem desse pequeno conjunto de instâncias, disponível [neste link](#). Os dados foram extraídos do sistema Eplan, uma solução *ad-hoc* da empresa que se propõe a auxiliar no processo de planejamento e controle de produção. Os dados foram extraídos da tela de programação de OSs no formato JSON e, através de um *script* em JavaScript, foram criados arquivos compatíveis com o formato de entrada dos algoritmos desenvolvidos — vide Seção 7.1.

Os dados extraídos envolvem a duração e precedência de cada processo produtivo, bem como as máquinas compatíveis com sua execução. Além disso, o menor instante de início programado e o maior instante de conclusão da grade de programação foram extraídos a fim de calcular o tempo total da produção programada. Esse valor será útil como referência para avaliar a qualidade das soluções. Um aspecto importante na extração dos dados é que o sistema Eplan admite relações de precedência não lineares

em alguns casos específicos. A Tabela 12 descreve as 3 instâncias que serão abordadas abaixo. Para evitar preocupações envolvendo precisão numérica, as durações foram armazenadas nas instâncias com seus valores em segundos, embora estejam convertidas para horas nos resultados a seguir.

Instância	$ J $	$ M $	ξ	\widehat{C}_{\max}
RibeiroSuzarte1	110	27	2	1370,81
RibeiroSuzarte2	155	40	1	14310,42
RibeiroSuzarte3	180	39	3	522,62
RibeiroSuzarte4	1435	29	1	1536,37

Tabela 12 – Instâncias desenvolvidas com dados de clientes da empresa interessada. $|J|$ e $|M|$ representam respectivamente o número de trabalhos e de máquinas. O número médio de máquinas por operação é representado por ξ e \widehat{C}_{\max} representa o *makespan* de referência calculado a partir do sistema Eplan (maior término programado - menor instante de início)

Podemos afirmar que, em razão dos resultados precedentes com as instâncias da literatura, o conjunto de algoritmos V2 é o que tem mais a oferecer para essa nova bateria de testes. Considerando o tamanho das instâncias da Tabela 12, os conjuntos V0 e V1 estão fora de cogitação, e o conjunto V3 proporcionaria os mesmos resultados, mas com um tempo médio de CPU ligeiramente mais alto. Com os testes que veremos a seguir, espera-se avaliar o comportamento destes algoritmos em instâncias de grande porte. Isto posto, as instâncias desenvolvidas foram submetidas aos 10 algoritmos do conjunto V2 e os resultados serão abordados a seguir.

7.4.1 RibeiroSuzarte1

Já na primeira instância, somos capazes de observar de maneira mais contundente o que já foi discutido anteriormente. A começar pela discrepância entre as abordagens iterativa e paralela, característica que se repete em todas as execuções destes algoritmos. Além disso, toda a discussão precedente sobre as características dos algoritmos ACO estudados se faz presente nesse conjunto de instâncias. Novamente, chamo a atenção para o alto custo-benefício do algoritmo ACS que apresenta o menor tempo médio de CPU sem impactar na qualidade do *makespan* gerado.

Os resultados obtidos para essa instância agradaram. O *makespan* médio de cada algoritmo do conjunto V2 é categoricamente menor que o *makespan* de referência desta instância: o pior *makespan* médio (1278,54 h por ACSV2-i) é 6,73% menor que o *makespan* de referência (1370,81 h). O menor tempo médio de CPU, obtido por ACSV2-p, foi de 2,83 segundos. O pior, que levou 1,85 minutos, foi alcançado por ASV2-i.

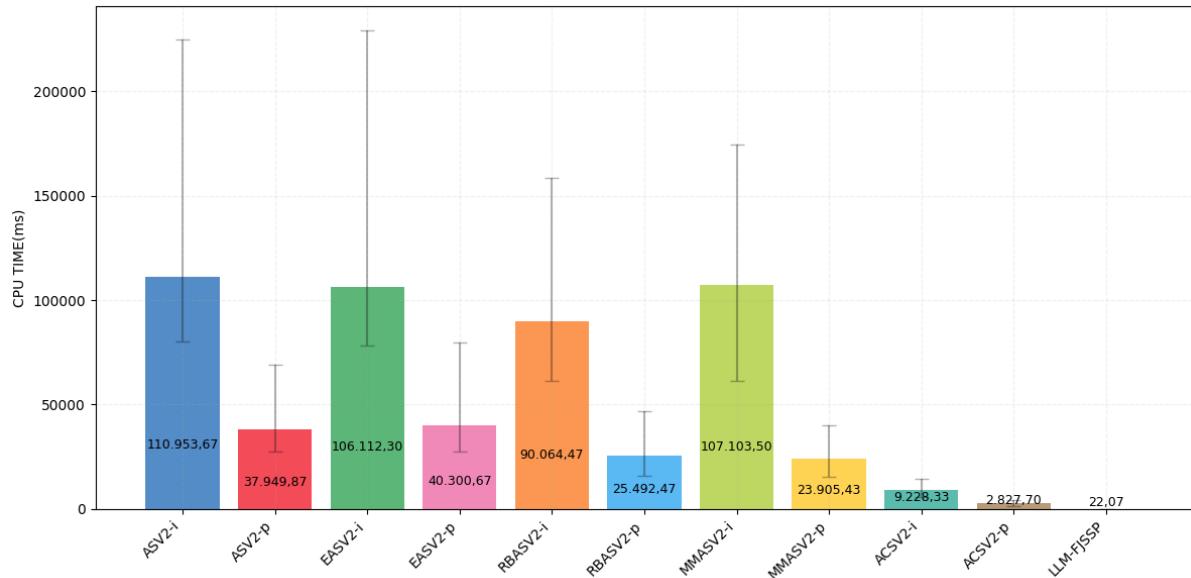


Figura 59 – Tempo médio de CPU de V2 para a instância RibeiroSuzarte1

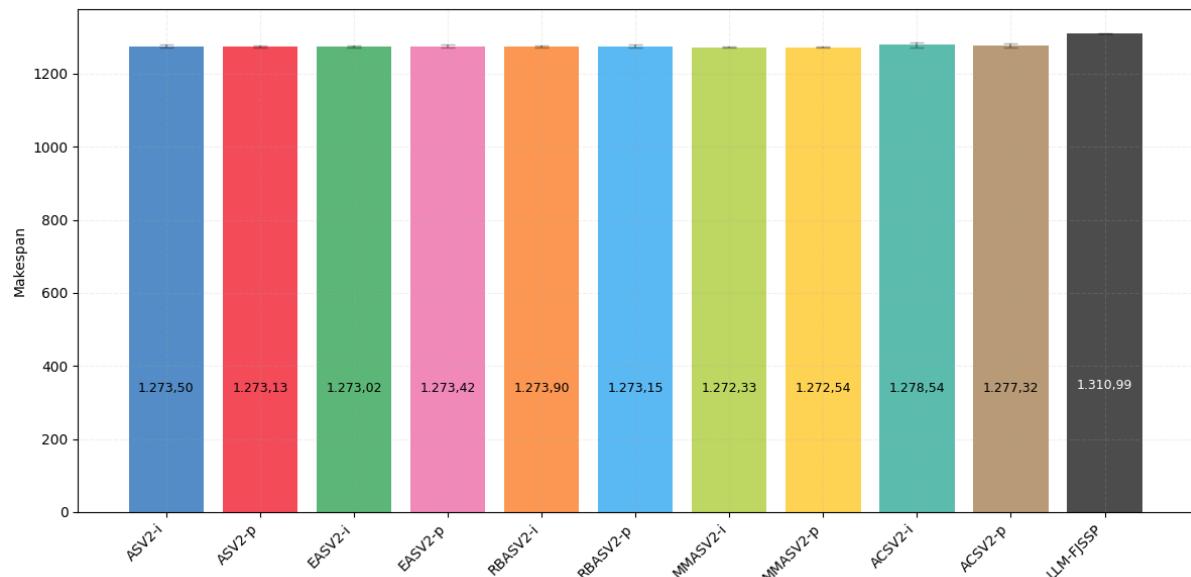


Figura 60 – Makespan médio de V2 para a instância RibeiroSuzarte1

As barras de erro ilustradas nos gráficos a seguir trazem à tona novas perspectivas sobre os algoritmos. Elas serem compridas no gráfico da Figura 59 indica uma alta dispersão no tempo de CPU registrado durante as execuções. Isso pode ser explicado pelo critério de parada adotado, que interrompe a execução prematuramente quando um número específico de iterações sem melhora é ultrapassado. Por outro lado, as barras de erro no gráfico da Figura 60 mostram baixíssima dispersão nos valores de *makespan*, indicando que os algoritmos, apesar do fator probabilístico, são extremamente consistentes em seus resultados.

7.4.2 RibeiroSuzarte2

Os padrões se repetem, portanto vamos aos resultados. Curiosamente, para essa instância, o *makespan* médio coincidiu em todos os algoritmos ACO: 15663,11 h. Infelizmente, desta vez, este valor foi cerca de 9,45% maior que o valor de referência (14310,43 h). O maior tempo médio de CPU, obtido por EASV2-i, foi de 1,56 min. O algoritmo mais rápido foi o ACSV2-p, que levou em média 1,4 s para concluir sua execução.

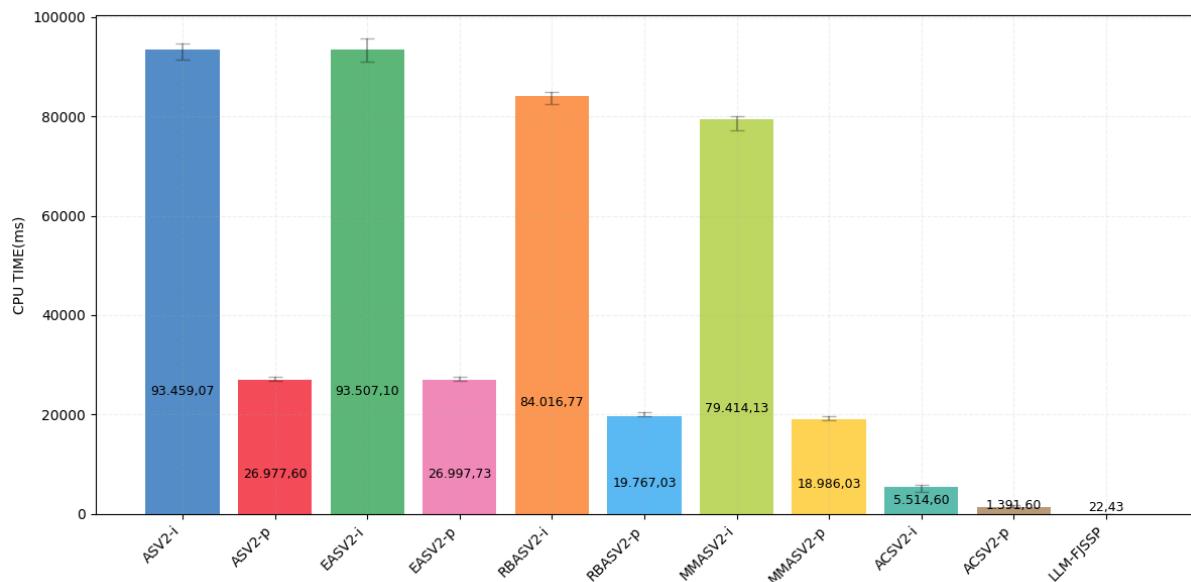


Figura 61 – Tempo médio de CPU de V2 para a instância RibeiroSuzarte2

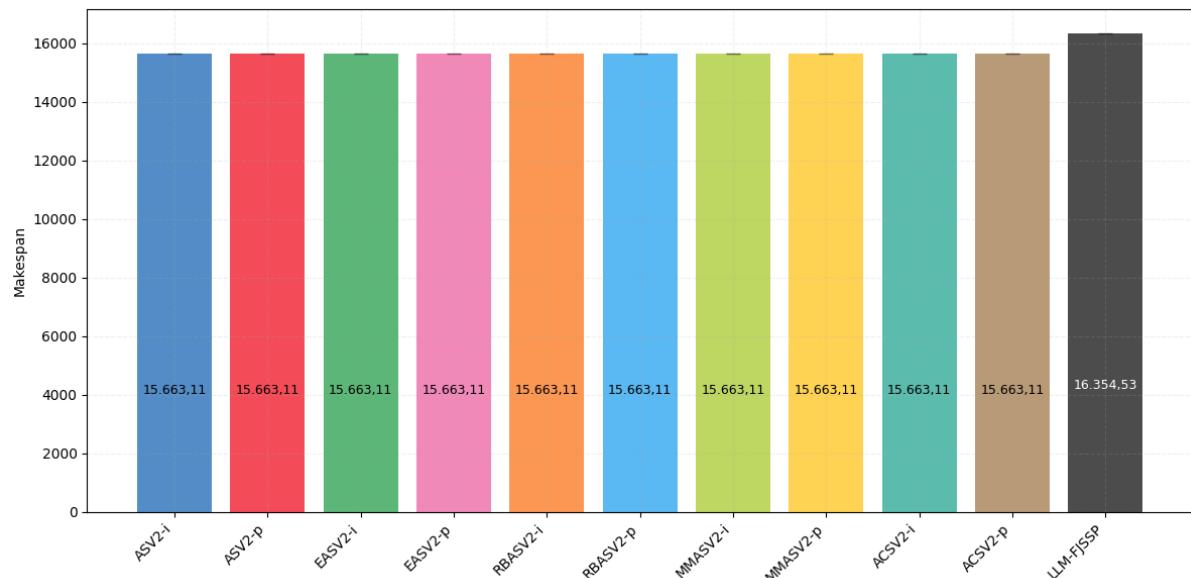


Figura 62 – *Makespan* médio de V2 para a instância RibeiroSuzarte2

7.4.3 RibeiroSuzarte3

Novamente, os algoritmos ACO apresentam o mesmo comportamento na execução. O pior *makespan* médio, obtido pelo algoritmo ACSV2-i foi 294,16 h, que corresponde a um valor 43,72% menor que o *makespan* de referência. O tempo médio de CPU nesta instância foi mais alto que os anteriores, e isso pode ser explicado pelo grau de flexibilidade de máquina levemente mais alto (o número médio de máquinas por operação nesta instância é 3). O pior tempo médio foi na casa dos 3,23 min, obtido pelo algoritmo ASV2-i. O melhor, obtido pelo algoritmo ACSV2-p, foi cerca de 1,95 s.

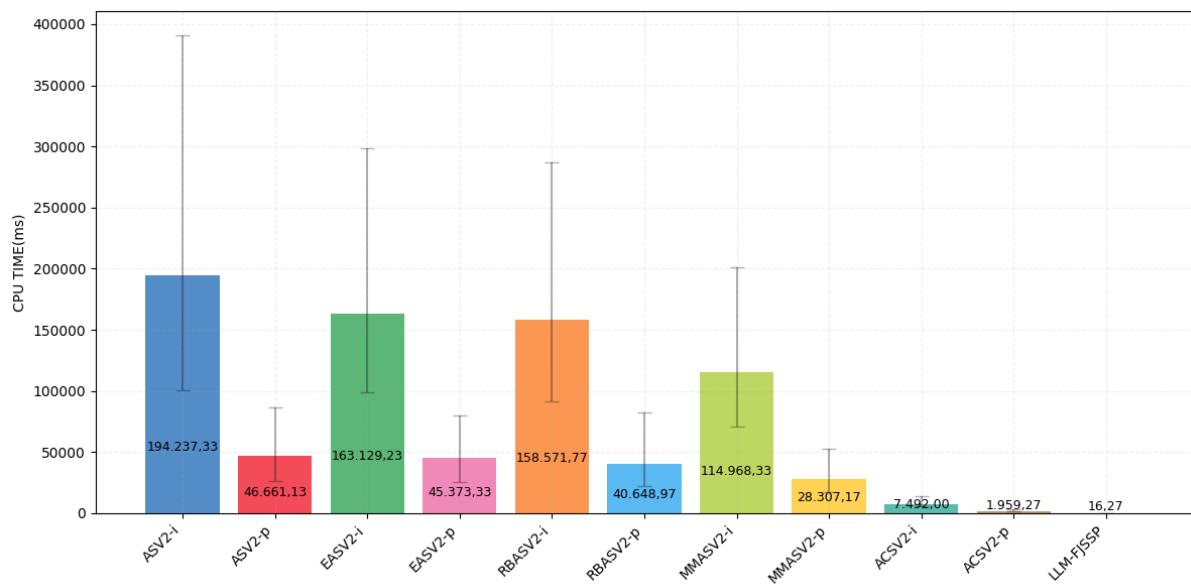


Figura 63 – Tempo médio de CPU de V2 para a instância RibeiroSuzarte3

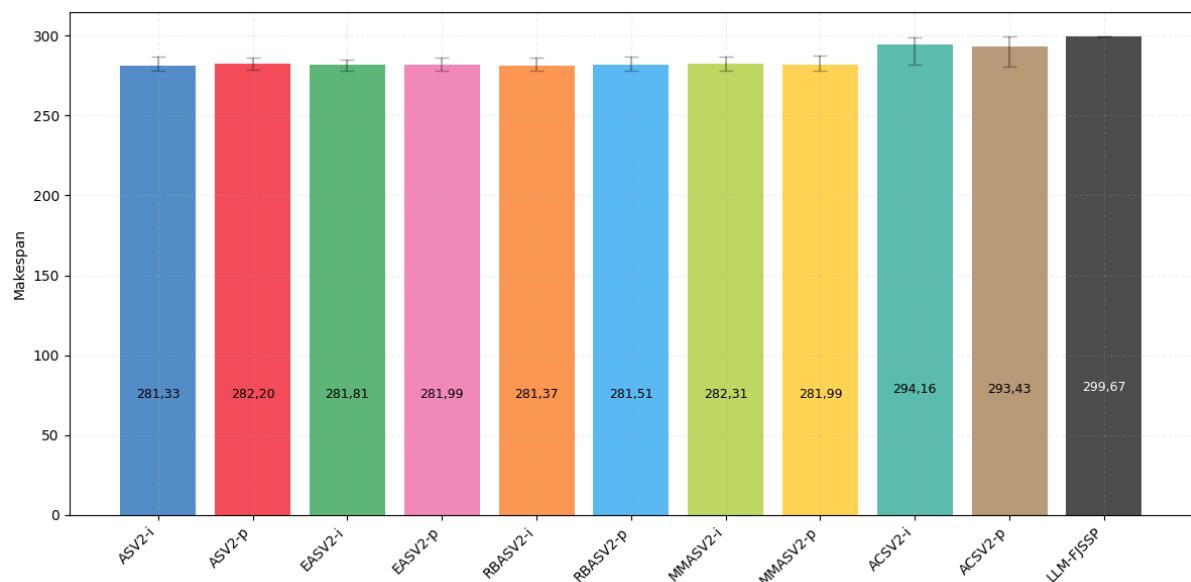


Figura 64 – *Makespan* médio de V2 para a instância RibeiroSuzarte3

7.4.4 RibeiroSuzarte4

De uma forma quase que poética, terminaremos este capítulo da mesma forma que começamos: analisando resultados do algoritmo ACS. Desta vez, em razão do tamanho colossal desta instância, mudamos levemente o delineamento experimental. Considerando que o ACS é o nosso algoritmo ACO mais rápido, e que ele se mostrou tão bom quanto todos os outros para encontrar soluções para esse problema, faremos uma avaliação do impacto do número de iterações no comportamento do algoritmo. Em virtude do tempo necessário para concluir uma execução de RibeiroSuzarte4, após uma análise cuidadosa, concluiu-se que apenas 10 execuções seriam suficientes para obter um *makespan* médio condizente. Os algoritmos ACSV2-i e ACSV2-p foram executados por 10, 100, 1000 e 10000 iterações.

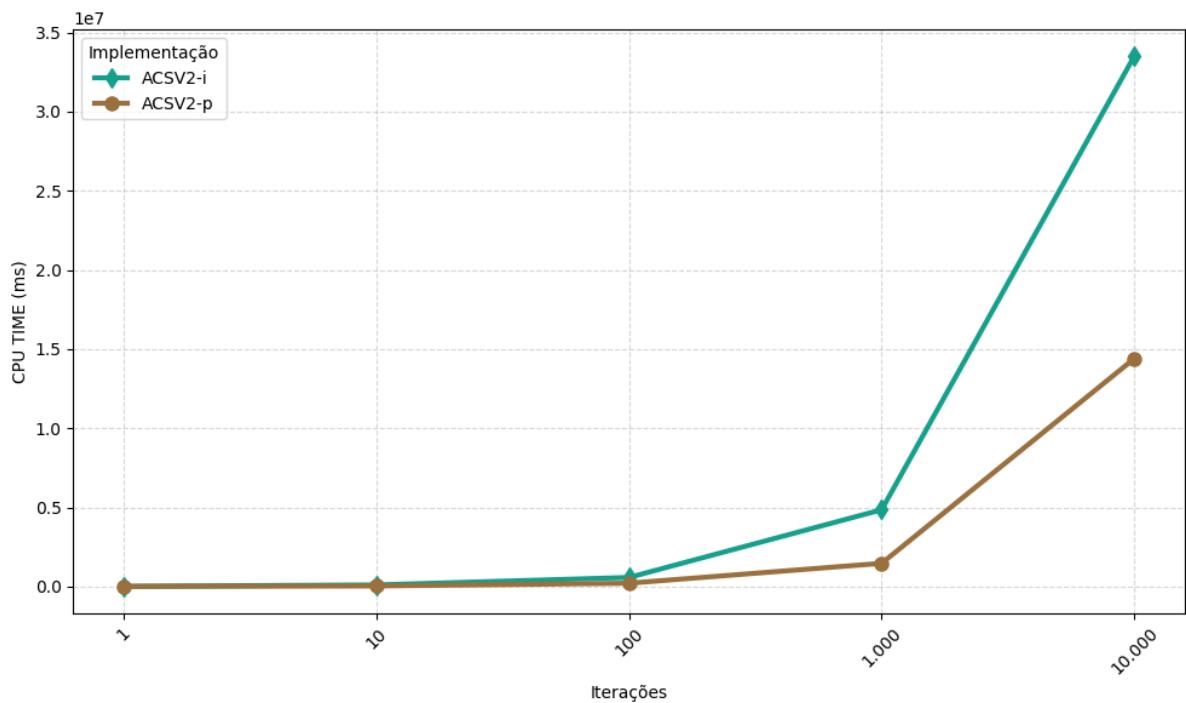


Figura 65 – Tempo médio de CPU de ACSV2 por número de iterações

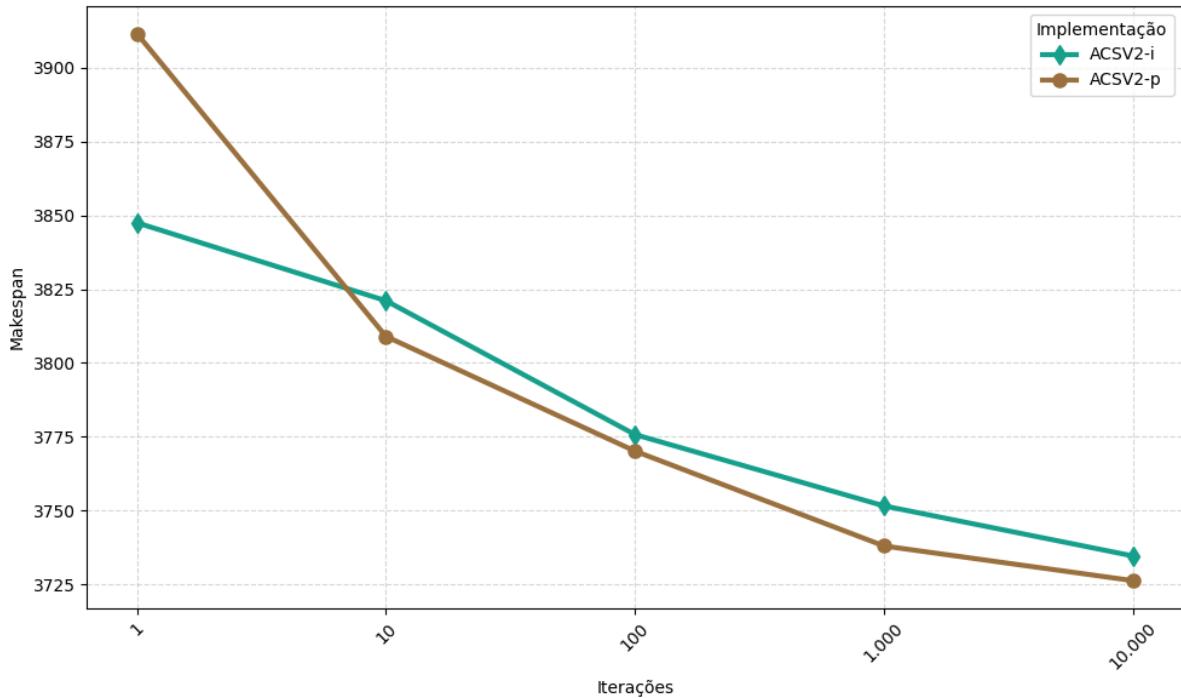


Figura 66 – *Makespan* médio de CPU de ACSV2 por número de iterações

Os gráficos nas figuras 65 e 66 evidenciam como os algoritmos se comportam à medida que o número de iterações aumenta. Como esperado, o tempo médio de CPU aumenta com o número de iterações, de forma muito mais rápida na versão iterativa do que na paralela. Já o *makespan* médio tende a diminuir à medida que o número de iterações aumenta. Isso também era esperado, uma vez que com mais iterações, maior deve ser o grau de exploração do espaço de soluções. Ademais, o ACSV2-p parece performar melhor que o ACSV2-i, apesar de cerca de 2% das tentativas de transformação local de feromônio falharem por condições de corrida.

Contudo, apesar da tendência observada na Figura 66, a inclinação das curvas de *makespan* indica uma variação mais lenta do que o desejado. Isso sugere que, apesar do baixo tempo de execução deixar certa margem para melhorar os resultados (tanto para instâncias da literatura, quanto para estas instâncias), estamos no limite do que esta técnica é capaz de fazer sem o auxílio de procedimentos de busca local.

8 Conclusões

Foi uma longa jornada até aqui. Sente-se perto de uma fogueira e descanse um pouco enquanto relembrar nosso percurso. Após um capítulo protocolar de preliminares, realizamos um breve passeio por alguns problemas de escalonamento da literatura que valiam a pena mencionar. Vimos, no Capítulo 3, o problema de escalonamento em máquinas idênticas. Abordamos o algoritmo de Graham, capaz de construir soluções viáveis em tempo polinomial para este problema, as quais provamos se tratar de 2-approximações. Em seguida, vimos o problema de escalonamento com restrições de precedência em máquina única e o algoritmo de Lawler que constrói soluções ótimas em tempo polinomial. Provamos essa afirmação e vimos que isso acontece porque este problema está associado a uma estrutura combinatória chamada antimatróide.

No Capítulo 4, começamos a abordar problemas de escalonamento no contexto da indústria gráfica. Começamos com o escalonamento orientado a trabalho, apresentando o modelo, discutindo métodos exatos e heurísticas construtivas de solução, para no final apresentarmos o modelo de digrafo disjuntivo. Após o leitor adquirir familiaridade com o problema, abordamos a versão de interesse, chamada escalonamento flexível orientado a trabalho. Essa versão do problema consiste em determinar um escalonamento e uma alocação de máquina adequada diante das restrições do problema. Nesta seção, mostramos brevemente o digrafo disjuntivo flexível e apresentamos a heurística LLM-FJSSP, bem como uma adaptação dela para admitir relações de precedência não lineares.

Após esgotarmos o assunto sobre problemas de escalonamento, no Capítulo 5, apresentamos ao leitor o conceito de otimização por colônia de formigas. Este capítulo teve como objetivo construir aos poucos um modelo formal abstrato que capturasse as especificidades dos diferentes algoritmos ACO estudados neste trabalho. Começamos formalizando a noção de problema de otimização combinatória, para em seguida começar a lidar com o método de solução. Vimos seções dedicadas à estrutura de feromônio, colônias e formigas. Isso posto, apresentamos o algoritmo que captura o conceito de algoritmo ACO.

Uma vez definidos todos os objetos, nos aprofundamos, na Seção 5.5, na modelagem dos cinco algoritmos ACO estudados neste trabalho. São eles: sistema de formigas (AS), sistema de formigas elitistas (EAS), sistema de formigas com classificação (RBAS), sistema de formigas MAX-MIN (MMAS) e sistema de colônia de formigas (ACS). Após uma checagem minuciosa do que faz cada um desses algoritmos, vamos para a Seção 5.7 onde modelamos o FJSSP conforme o modelo de problema de otimização combinatória

que foi apresentado neste capítulo. Do ponto de vista teórico, isso amarra o uso do método de otimização por colônia de formigas para encontrar soluções viáveis para o FJSSP.

No Capítulo 6, descrevemos de forma cronológica todo o processo de implementação dos 32 algoritmos desenvolvidos. Dentre elas, temos combinações de 4 implementações da heurística construtiva com 2 abordagens de execução diferentes e com as implementações dos 5 algoritmos ACO estudados. Também temos uma implementação pura do algoritmo LLM-FJSSP que viria a ser utilizada como base de comparação.

Por fim, no Capítulo 7, temos uma discussão profunda sobre os resultados obtidos. No início, apresentamos a metodologia e delineamento experimental do trabalho. Logo após, também em ordem cronológica, apresentamos evidências de que as primeiras heurísticas construtivas que se baseiam em um digrafo disjuntivo flexível (V0 e V1) não corresponderam às expectativas. Em seguida, os resultados da heurística construtiva inspirada em LLM-FJSSP (V2) são apresentados e, por fim, os resultados da adaptação para admitir relações de precedência não lineares (V3).

Os dados desta seção indicam que as heurísticas V2 e V3 são as melhores que possuímos. Quanto à abordagem de execução, vimos que a execução paralela supera a iterativa e, do ponto de vista do ACO, concluímos que os algoritmos RBAS, MMAS e ACS se destacam, por diferentes motivos, na missão de construir soluções viáveis para o FJSSP. Ao final do capítulo, avançamos para discutir o desenvolvimento do novo conjunto de instâncias e os resultados que as envolvem.

8.1 Contribuições

As contribuições desse trabalho são de ordem teórica e prática. Do ponto de vista teórico, esse trabalho se propôs a apresentar descrições alternativas dos modelos e algoritmos envolvidos nos problemas de escalonamento e no método de otimização por colônia de formigas. Os algoritmos foram descritos de uma perspectiva funcional, focando nas transformações e invariantes. O modelo alternativo desenvolvido para o método de otimização por colônia de formigas é baseado em sequências, diferente do modelo apresentado por [Dorigo e Stutzle \(2004\)](#). Em razão dessas mudanças, esse trabalho busca contribuir para o entendimento da técnica e abrir caminho para novas aplicações.

Ainda do ponto de vista teórico, esse trabalho procura contribuir com a literatura dos problemas de escalonamento, adicionando novas instâncias de grande porte e seus respectivos limitantes superiores para o problema de escalonamento flexível orientado a trabalho.

No aspecto prático, as contribuições deste trabalho envolvem a implementação de algoritmos eficientes para a resolução do problema estudado. Ao final deste trabalho, obtivemos um *solver* para problemas de escalonamento com 33 algoritmos disponíveis para execução. É bem verdade que cerca de 12 desses algoritmos não são lá muito eficientes, como observado nos resultados, mas o trabalho de engenharia de *software* empregado para que a coexistência e manutenção deles fosse escalável é digno de nota.

8.2 Limitações

Esse trabalho esteve sujeito a algumas limitações. A principal delas, sem dúvidas, foi seu próprio escopo e o tempo disponível para realizá-lo. Inúmeras coisas poderiam facilmente ter entrado, não fosse a linha delimitada para que sua conclusão fosse possível. É evidente que muitas coisas ficariam de fora, uma vez que abordamos assuntos para os quais existem áreas inteiras de pesquisa, e a direção escolhida valorizou a qualidade e cuidado frente à quantidade. Dentre as coisas que ficaram de fora, podemos mencionar: a inclusão na revisão de literatura da clássica notação de três campos desenvolvida por [Graham \(1969\)](#) e o estudo de mais variantes dos algoritmos ACO, bem como um trabalho experimental mais minucioso.

Outra limitação importante que definiu os rumos deste trabalho foi a impossibilidade, tanto prática quanto financeira, de empregar um *solver* para resolver instâncias deste problema. Como vimos, ([DAUZÈRE-PÉRÈS et al., 2024](#)) nos alertou da inviabilidade das abordagens exatas na resolução de instâncias grandes, que era o caso deste trabalho. Além do mais, todos os interessados na aplicação destes algoritmos estavam dispostos a aceitar um algoritmo que não oferecesse soluções necessariamente ótimas, desde que o *trade-off* envolvesse um cenário de alta performance. Por esse motivo, a abordagem metaheurística foi adotada.

8.3 Trabalhos Futuros

Os objetivos do trabalho foram plenamente alcançados, contudo, no decorrer de seu desenvolvimento surgiram pontos para possíveis trabalhos futuros. Começando pelo mais simples, do ponto de vista experimental, testes com diferentes parametrizações ficaram de fora deste trabalho e poderiam ser completados futuramente. Outro aspecto que definitivamente vale a pena desenvolver é a composição do algoritmo com algum método de busca local. Esta etapa está prevista na teoria e seria de grande valia analisar experimentalmente a performance e qualidade das soluções obtidas.

Do ponto de vista prático, alguns detalhes de implementação podem agregar positivamente na performance do algoritmo, que por falta de tempo ou conveniência

não foram realizados. Podemos, por exemplo, realizar algumas modificações buscando um melhor aproveitamento das CPUs nas abordagens paralelas. Outra modificação benéfica seria a substituição de algumas listas por *arrays*, visando obter mais localidade no acesso à memória.

Em relação aos algoritmos e modelos, duas linhas de abordagem seriam significativas em trabalhos futuros. A primeira delas, puramente teórica, seria tentar encaixar outros algoritmos ACO no modelo e eventualmente modificá-lo. Alguns exemplos de algoritmos são o *Best-worst ant system* (BWAS), *Fast ant system* (FAS), *Ant-Q*, *A.N.T.S*, e *Hypercube framework ant system* (HFAS). A segunda linha diz respeito ao problema de escalonamento orientado a trabalho e contempla a variação das heurísticas construtivas. Neste trabalho, utilizamos exclusivamente a heurística LLM, mas outras poderiam ter sido exploradas, como a *Shortest processing time first* (SPT) e a *Earliest start time first* (EST), ou até mesmo outros métodos como procedimentos de busca gulosos, aleatórios e adaptativos (GRASP).

Por fim, do ponto de vista da empresa interessada, trabalhos futuros se farão necessários para maior adaptação do modelo à realidade do setor gráfico. Aspectos importantes que ficaram de fora, por uma questão de escopo, são a inclusão de intervalos não produtivos nas máquinas, tempos de *setup* dependentes da sequência, datas de entrega dos trabalhos e até mesmo outros tipos de função objetivo, como a minimização do atraso total.

Referências

- ADAMS, J.; BALAS, E.; ZAWACK, D. The shifting bottleneck procedure for job shop scheduling. *Management Science*, INFORMS, v. 34, n. 3, p. 391–401, 1988. ISSN 00251909, 15265501. Disponível em: <<http://www.jstor.org/stable/2632051>>. Citado na página 124.
- APPLEGATE, D.; COOK, W. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, Institute for Operations Research and the Management Sciences (INFORMS), v. 3, n. 2, p. 149–156, maio 1991. ISSN 2326-3245. Disponível em: <<http://dx.doi.org/10.1287/ijoc.3.2.149>>. Citado na página 124.
- AZZOUZ, A.; ENNIGROU, M.; SAID, L. B. Solving flexible job-shop problem with sequence dependent setup time and learning effects using an adaptive genetic algorithm. *International Journal of Computational Intelligence Studies*, Inderscience Publishers, v. 9, n. 1/2, p. 18, 2020. ISSN 1755-4985. Disponível em: <<http://dx.doi.org/10.1504/IJCISTUDIES.2020.106486>>. Nenhuma citação no texto.
- BEHNKE, D.; GEIGER, M. J. Test instances for the flexible job shop scheduling problem with work centers. Helmut-Schmidt-Universität, 2012. Disponível em: <<https://openhsu.ub.hsu-hh.de/handle/10.24405/436>>. Citado 3 vezes nas páginas 122, 123 e 124.
- BERTEROTTIèRE, L.; DAUZÈRE-PÉRÈS, S.; YUGMA, C. Flexible job-shop scheduling with transportation resources. *European Journal of Operational Research*, Elsevier BV, v. 312, n. 3, p. 890–909, fev. 2024. ISSN 0377-2217. Disponível em: <<http://dx.doi.org/10.1016/j.ejor.2023.07.036>>. Nenhuma citação no texto.
- BIRGIN, E. G. et al. A milp model for an extended version of the flexible job shop problem. *Optimization Letters*, Springer Science and Business Media LLC, v. 8, n. 4, p. 1417–1431, jul. 2013. ISSN 1862-4480. Disponível em: <<http://dx.doi.org/10.1007/s11590-013-0669-7>>. Citado na página 122.
- BONDY, A.; MURTY, U. *Graph Theory*. [S.l.]: Springer, 2008. v. 244. (Graduate Texts in Mathematics, v. 244). ISBN 978-1-84628-969-9. Citado na página 4.
- BOYD, E.; FAIGLE, U. An algorithmic characterization of antimatroids. *Discrete Applied Mathematics*, Elsevier BV, v. 28, n. 3, p. 197–205, set. 1990. ISSN 0166-218X. Disponível em: <[http://dx.doi.org/10.1016/0166-218X\(90\)90002-T](http://dx.doi.org/10.1016/0166-218X(90)90002-T)>. Citado na página 31.
- BRANDIMARTE, P. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, Springer Science and Business Media LLC, v. 41, n. 3, p. 157–183, set. 1993. ISSN 1572-9338. Disponível em: <<http://dx.doi.org/10.1007/BF02023073>>. Citado 11 vezes nas páginas 89, 122, 123, 126, 127, 130, 131, 132, 133, 140 e 144.
- BRUCKER, P.; KNUST, S. *Complex Scheduling*. 2. ed. Berlin, Germany: Springer, 2011. (GOR-Publications). Citado 5 vezes nas páginas 35, 37, 42, 43 e 51.

BULLNHEIMER, B.; HARTL, R.; STRAUSS, C. A new rank based version of the ant system - a computational study. *Central European Journal of Operations Research*, v. 7, p. 25–38, 01 1999. Citado 2 vezes nas páginas 56 e 74.

CAPUTI, A.; MIRANDA, D. *Bases Matemáticas*.

<https://danielmiranda.prof.ufabc.edu.br/livros/basesmatematicas/bases.pdf>: Universidade Federal do ABC, 2023. Citado na página 4.

CARLIER, J.; PINSON, E. An algorithm for solving the job-shop problem. *Management Science*, INFORMS, v. 35, n. 2, p. 164–176, 1989. ISSN 00251909, 15265501. Disponível em: <<http://www.jstor.org/stable/2631909>>. Citado na página 124.

CORMEN, T. H.; LEISERSON, C. E. *Introduction to Algorithms, fourth edition*. London, England: MIT Press, 2022. Citado na página 9.

DASGUPTA, S.; PAPADIMITRIOU, C. H.; VAZIRANI, U. V. *Algorithms*. New York, NY: McGraw-Hill Professional, 2006. Citado na página 8.

DAUZÈRE-PÉRÈS, S. et al. The flexible job shop scheduling problem: A review. *European Journal of Operational Research*, Elsevier BV, v. 314, n. 2, p. 409–432, abr. 2024. ISSN 0377-2217. Disponível em: <<http://dx.doi.org/10.1016/j.ejor.2023.05.017>>. Citado 2 vezes nas páginas 38 e 157.

DAUZÈRE-PÉRÈS, S.; PAULLI, J. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*, Springer Science and Business Media LLC, v. 70, n. 0, p. 281–306, abr. 1997. ISSN 1572-9338. Disponível em: <<http://dx.doi.org/10.1023/A:1018930406487>>. Citado 5 vezes nas páginas 123, 132, 133, 140 e 144.

DORIGO, M.; BIRATTARI, M.; STUTZLE, T. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, v. 1, p. 28–39, 12 2006. Citado na página 55.

DORIGO, M.; CARO, G. D.; GAMBARDELLA, L. M. Ant Algorithms for Discrete Optimization. *Artificial Life*, v. 5, n. 2, p. 137–172, 04 1999. ISSN 1064-5462. Disponível em: <<https://doi.org/10.1162/106454699568728>>. Citado 2 vezes nas páginas 56 e 65.

DORIGO, M.; GAMBARDELLA, L. M. Ant colonies for the travelling salesman problem. *Biosystems*, Elsevier BV, v. 43, n. 2, p. 73–81, jul. 1997. ISSN 0303-2647. Disponível em: <[http://dx.doi.org/10.1016/S0303-2647\(97\)01708-5](http://dx.doi.org/10.1016/S0303-2647(97)01708-5)>. Citado 2 vezes nas páginas 56 e 76.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, Institute of Electrical and Electronics Engineers (IEEE), v. 26, n. 1, p. 29–41, fev. 1996. ISSN 1941-0492. Disponível em: <<http://dx.doi.org/10.1109/3477.484436>>. Citado 3 vezes nas páginas 56, 70 e 73.

DORIGO, M.; STUTZLE, T. *Ant colony optimization*. Cambridge, MA: Bradford Books, 2004. (A Bradford Book). Citado 7 vezes nas páginas 68, 73, 75, 78, 107, 120 e 156.

FATTAHI, P.; MEHRABAD, M. S.; JOLAI, F. Mathematical modeling and heuristic approaches to flexible job shop scheduling problems. *Journal of Intelligent Manufacturing*, Springer Science and Business Media LLC, v. 18, n. 3, p. 331–342, jul. 2007. ISSN

1572-8145. Disponível em: <<http://dx.doi.org/10.1007/s10845-007-0026-8>>. Citado 10 vezes nas páginas 49, 119, 122, 126, 127, 130, 131, 133, 140 e 144.

FISHER, H.; THOMPSON, G. Probabilistic learning combinations of local job-shop scheduling rules. *Industrial Scheduling*, Prentice Hall, 1963. Disponível em: <<https://cir.nii.ac.jp/crid/1572824501187192960>>. Citado na página 124.

GAREY, M. R.; JOHNSON, D. S. “ strong ” np-completeness results: Motivation, examples, and implications. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 25, n. 3, p. 499–508, jul. 1978. ISSN 0004-5411. Disponível em: <<https://doi.org/10.1145/322077.322090>>. Citado 2 vezes nas páginas 22 e 24.

GENOVA, K.; KIRILOV, L.; GULIASHKI, V. A survey of solving approaches for multiple objective flexible job shop scheduling problems. *Cybernetics and Information Technologies*, v. 15, n. 2, p. 3–22, 2015. Disponível em: <<https://doi.org/10.1515/cait-2015-0025>>. Citado na página 52.

GRAHAM, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, Society for Industrial & Applied Mathematics (SIAM), v. 17, n. 2, p. 416–429, mar. 1969. ISSN 1095-712X. Disponível em: <<http://dx.doi.org/10.1137/0117039>>. Citado 2 vezes nas páginas 26 e 157.

HILLIER, F. S.; LIEBERMAN, G. J. *Introduction to operations research*. 10. ed. Maidenhead, England: McGraw Hill Higher Education, 2014. ISBN 0073523453 , 9780073523453. Citado na página 1.

HURINK, J.; JURISCH, B.; THOLE, M. Tabu search for the job-shop scheduling problem with multi-purpose machines. *OR Spektrum*, Springer Science and Business Media LLC, v. 15, n. 4, p. 205–215, dez. 1994. ISSN 1436-6304. Disponível em: <<http://dx.doi.org/10.1007/BF01719451>>. Citado 5 vezes nas páginas 124, 125, 135, 141 e 144.

LAWRENCE, S. Resource constrained project scheduling : an experimental investigation of heuristic scheduling techniques (supplement). *Graduate School of Industrial Administration, Carnegie-Mellon University*, 1984. Disponível em: <<https://cir.nii.ac.jp/crid/1571980073974705920>>. Citado na página 124.

LEI, K. et al. A multi-action deep reinforcement learning framework for flexible job-shop scheduling problem. *Expert Systems with Applications*, Elsevier BV, v. 205, p. 117796, nov. 2022. ISSN 0957-4174. Disponível em: <<http://dx.doi.org/10.1016/j.eswa.2022.117796>>. Citado na página 119.

MAHMUD, S. et al. Switching strategy-based hybrid evolutionary algorithms for job shop scheduling problems. *Journal of Intelligent Manufacturing*, Springer Science and Business Media LLC, v. 33, n. 7, p. 1939–1966, abr. 2022. ISSN 1572-8145. Disponível em: <<http://dx.doi.org/10.1007/s10845-022-01940-1>>. Citado 2 vezes nas páginas 135 e 136.

MARTIN, R. C. *Clean architecture*. Philadelphia, PA: Prentice Hall, 2017. Citado na página 95.

- MASTROLILLI, M.; GAMBARDELLA, L. M. Effective neighbourhood functions for the flexible job shop problem. *Journal of Scheduling*, n. 1, p. 3–20, 2000. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1425%28200001/02%293%3A1%3C3%3A%3AAID-JOS32%3E3.0.CO%3B2-Y>>. Citado na página 122.
- Microsoft. *C# Language Documentation*. [S.I.], 2024. Disponível em: <<https://learn.microsoft.com/en-us/dotnet/csharp/>>. Citado na página 86.
- ÖZGÜVEN, C.; ÖZBAKİR, L.; YAVUZ, Y. Mathematical models for job-shop scheduling problems with routing and process plan flexibility. *Applied Mathematical Modelling*, Elsevier BV, v. 34, n. 6, p. 1539–1548, jun. 2010. ISSN 0307-904X. Disponível em: <<http://dx.doi.org/10.1016/j.apm.2009.09.002>>. Citado na página 122.
- PINEDO, M. L. *Scheduling*. 5. ed. Cham, Switzerland: Springer International Publishing, 2016. Citado na página 1.
- ROSS, S. *Probabilidade: Um Curso Moderno com Aplicações*. [S.I.]: Bookman Editora, 2009. Citado na página 70.
- ROSSI, A.; DINI, G. Flexible job-shop scheduling with routing flexibility and separable setup times using ant colony optimisation method. *Robotics and Computer-Integrated Manufacturing*, Elsevier BV, v. 23, n. 5, p. 503–516, out. 2007. ISSN 0736-5845. Disponível em: <<http://dx.doi.org/10.1016/j.rcim.2006.06.004>>. Citado 6 vezes nas páginas 51, 89, 109, 120, 130 e 140.
- SHEN, L.; DAUZÈRE-PÉRÈS, S.; NEUFELD, J. S. Solving the flexible job shop scheduling problem with sequence-dependent setup times. *European Journal of Operational Research*, Elsevier BV, v. 265, n. 2, p. 503–516, mar. 2018. ISSN 0377-2217. Disponível em: <<http://dx.doi.org/10.1016/j.ejor.2017.08.021>>. Nenhuma citação no texto.
- SOTSKOV, Y.; SHAKHLEVICH, N. Np-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*, v. 59, n. 3, p. 237–266, 1995. ISSN 0166-218X. Disponível em: <<https://www.sciencedirect.com/science/article/pii/0166218X9580004N>>. Citado na página 38.
- STUTZLE, T.; HOOS, H. H. Max-min ant system. *Future Generation Computer Systems*, Elsevier BV, v. 16, n. 8, p. 889–914, jun. 2000. ISSN 0167-739X. Disponível em: <[http://dx.doi.org/10.1016/S0167-739X\(00\)00043-1](http://dx.doi.org/10.1016/S0167-739X(00)00043-1)>. Citado 2 vezes nas páginas 56 e 75.
- VAZIRANI, V. V. *Approximation algorithms*. 1. ed. Berlin, Germany: Springer, 2002. Citado na página 25.

Índice

A

- ACO, 55
- ACS, 76
 - coeficiente de decaimento de feromônio, 77
 - parâmetro de corte de exploração, 77
 - regra proporcional pseudo-aleatória, 76
- AS, 70
 - coeficiente de evaporação, 72
 - depósito de feromônio, 72
 - parâmetros de influência, 70
 - regra de probabilidade, 70
- colônia, 68
- concentração de feromônio, 63
- EAS, 73
 - peso do reforço elitista, 74
- estrutura de feromônios, 63
- exploração, 62
- exploração, 62
- feromônio, 55
- formiga, 66
- função critério de parada, 69
- função seletora de soluções viáveis, 68
- informação heurística, 66
- mapeamento de feromônio, 63
- MMAS, 75
 - concentração máxima de feromônio, 76
 - concentração mínima de feromônio, 76
 - saturação de feromônio, 76
- movimento viável, 65

pontos de feromônios, 62

possibilidade de extensão, 63

RBAS, 74

tamanho da classificação, 74

transformação global de feromônios, 68

transformação local de feromônio, 67

Algoritmo, 8

algoritmos de aproximação, 25

algoritmos gulosos, 24

assinatura, 8

metaheurística, 55

Antimatróide, 31

C

C#, 86

namespace, 86

PLINQ, 126

projeto, 86

Complexidade, 9

algoritmo polinomial, 9, 25

Big-O, 9

classes de complexidade, 9

NP-difícil, 58

Conjuntos, 4

conjunto das partes, 5

conjunto maximal, 5

conjunto vazio, 4

conjuntos disjuntos, 6

diferença, 6

interseção, 6

pertinência, 4

produto cartesiano, 6

subconjunto, 4

subconjunto próprio, 5

união, 5

D

Digrafo, 12
acíclico, 26
algoritmo de Kahn, 113
arco, 12
arcos antiparalelos, 13
arcos paralelos, 13
cabeça, 12
caminho, 14
comprimento, 14
custo, 17
início, 15
termino, 15
caminho hamiltoniano, 15
caminho não trivial, 15
caminho simples, 15
caminhos críticos, 19
cauda, 12
ciclo, 15
concatenação de caminhos, 15
conjunto dos caminhos, 15
digrafo acíclico, 16
digrafo ponderado, 17
digrafo simples, 13
função δ , 18
função de incidência, 12
ordenação acíclica, 16
pontas, 12
predecessor crítico, 19
ramificação, 20
ramificação w -ótima, 20
ramificação geradora, 20
remoção de arcos, 14
remoção de vértices, 14
subdigrafo, 13
subdigrafo gerador, 14
subdigrafo próprio, 14
vizinhança de entrada, 13
vizinhança de saída, 13

vértice, 12

vértice fonte, 13
vértice sorvedouro, 13
vértices predecessores, 13
vértices sucessores, 13

E

Escalonamento, 1, 22
escalonamento parcial, 22
algoritmo de Graham, 24
algoritmo de Lawler, 29
alocação de máquina, 34
carga de trabalho, 22, 39
custo, 27
diagrama Gantt, 23
digrafo conjuntivo, 43
digrafo disjuntivo, 41
custo da seleção, 43
seleção, 42
seleção completa, 43
seleção consistente, 43
digrafo disjuntivo flexível, 50
duração mais longo primeiro, 26
escalonamento com restrições de precedência em máquina única, 26
escalonamento em máquinas idênticas, 22
escalonamento flexível orientado a trabalho, 46
escalonamento orientado a trabalho, 34
conjunto das máquinas compatíveis, 48
escalonamento parcial, 27
escalonamento por lista, 26
heurística da máquina menos carregada, 39
makespan, 23, 36
máquina, 22, 34
ocupação de máquina, 22, 39

- penalidade, 27
relação de precedência, 27
solução viável, 36
trabalho, 22, 27, 34
 duração, 22, 27
 instante de conclusão, 36
 instante de liberação, 34
 operações, 34
 operações iniciais, 35
 operações não iniciais, 35
 operações não terminais, 35
 operações predecessoras, 35
 operações sucessoras, 35
 operações terminais, 35
- F**
- Função, 7
 bijetora, 8
 domínio, 7
 estritamente crescente, 8
 função parcial, 7
 gráfico, 7
 imagem, 7
 imagem inversa, 8
 injetora, 8
 monotonicidade, 8
 relação, 6
 relação inversa, 6
 restrição, 7
 sobrejetora, 8
- G**
- Gap percentual, 121
Gap percentual médio, 121
- P**
- Problema de otimização, 56
 componentes, 57
 custo, 57
 espaço de soluções, 57
 estrutura de vizinhança, 57
mochila binária, 60
solução parcial, 57
solução viável, 57, 65
solução viável ótima, 58
TSP, 55, 59
- S**
- Sequência, 9
 comprimento, 9
 concatenação, 10
 fecho de Kleene, 10
 permutação, 11, 64
 prefixo, 11
 sequência vazia, 9
 subsequência, 11
 sufixo, 11
 termo, 9