Sequenciamento da produção em sistemas *flow shop*

Hélio Fuchigami

Sequenciamento da produção em sistemas *flow shop*

COLEÇÃO LAVOR

Dedico este meu primeiro livro de *scheduling*
aos meus pais Lourdes e Yochio
e ao meu mestre Daisaku Ikeda.

Agradeço profundamente
aos professores doutores
Rubén Ruiz García
e João Vitor Moccellin,
ao meu único irmão, Rafael,
e aos queridos Jorge e Zilda Maciel.

Sumário

Apresentação

Conheço o Hélio há muitos anos. Nos une a paixão pelos problemas de sequenciamento e, em concreto, os problemas do tipo *flow shop* que, além disso, são os mais habituais na prática. A primeira vez que trabalhei com o Hélio foi durante um longo período de pesquisa que ele empreendeu em Valência, se não lembro mal, entre setembro de 2006 e março de 2007, quando então estava terminando seu doutorado. Agora vejo que seu trabalho tem dado bons frutos. Um autor que se preze deve ser entusiasta e conhecedor do problema em questão. Essas qualidades são reconhecidas no Hélio de forma excepcional. Todos nós, membros da minha equipe de pesquisa, sempre nos surpreendíamos que Hélio nunca parava de sorrir. Apesar das dificuldades do trabalho de doutorado, Hélio sempre sorria. Para mim, é um prazer introduzir ao leitor o livro que tem agora em suas mãos.

Os problemas de produção são inerentes à atividade manufatureira. Dentro da gestão de operações aparece a programação da produção, que consiste na alocação, sequenciamento e estabelecimento do cronograma das tarefas nos recursos produtivos. Trata-se de uma atividade enormemente complexa em que geralmente existem milhões de combinações possíveis, às

quais o ser humano somente pode aspirar dar soluções aproximadas. Para ir além e otimizar a produção, é necessário um conhecimento avançado de algoritmos e técnicas de programação da produção.

Obviamente, dada a importância desse problema, existem muitos livros na literatura sobre sequenciamento, talvez até demais. Não obstante, são pouquíssimas as obras que abordam os sistemas do tipo fábrica em fluxo de forma específica. São menos frequentes ainda os textos que tratam os assuntos de um ponto de vista completo e ao mesmo tempo didático. Muitas vezes, a literatura concreta sobre sistemas de sequenciamento é excessivamente teórica e árdua para nossos estudantes. As revisões do estado da arte requerem com frequência um conhecimento prévio do problema. As publicações e artigos científicos também consideram supostos e conhecidos muitos aspectos dos problemas. O resultado que observamos é que os estudantes têm um início complicado no aprendizado dos sistemas de fábrica em fluxo e demoram vários meses para começar a conhecer de forma precisa esses problemas, prorrogando de forma desnecessária a finalização de seus estudos doutorais.

O presente livro vem mitigar essa situação. Primeiro e sobretudo, está escrito em português, o que rompe uma das principais barreiras pedagógicas, que é a língua inglesa. Segundo, tem uma estrutura muito simples, que apresenta inicialmente uma introdução geral à gestão de operações e sequenciamento, seguida de uma definição descomplicada dos principais problemas de produção, para então centrar-se na fábrica em fluxo de forma incremental, delineando primeiro o problema e tratando depois das técnicas exatas, heurísticas e meta-heurísticas. Terceiro, cada aspecto se mostra de forma fácil, sem exigir conhecimentos prévios, e se conduz a um nível suficiente de detalhamento, de forma que o leitor, ao querer conhecer mais, não precise consultar além da nutrida seção de referências. Exemplos completos e ao mesmo tempo sucintos acompanham as explicações distintas, permitindo ao estudante nivelar-se no problema de fábrica em fluxo com a máxima eficácia e em um tempo reduzido.

O livro também abrange os principais e mais consolidados avanços no problema de fábrica em fluxo, desde as heurísticas e os modelos matemáticos

clássicos até algumas das mais modernas meta-heurísticas. Portanto, a obra é também indicada para pesquisadores que vierem a ter uma ampla experiência em outras áreas e que não tenham trabalhado previamente com o problema específico de fábrica em fluxo.

Após ter lido o livro em profundidade, não posso deixar de recomendar ao leitor a sua esmerada leitura, para honrar a esmerada escrita do autor. Espero que este livro seja o primeiro de uma longa lista de contribuições do meu amigo e companheiro Hélio, do qual sempre recordo com um grande sorriso cruzando seu rosto de orelha a orelha.

Rubén Ruiz García*
Valência, 30 de setembro de 2014

* Professor catedrático e pesquisador do Departamento de Estadística e Investigación Operativa Aplicadas y Calidad, da Universitat Politècnica de València (UPV), na Espanha, e coordenador do Grupo de Sistemas de Otimização Aplicada (SOA) e do Instituto Tecnológico de Informática (ITI).

Prefácio

A programação da produção ou *scheduling* é uma interseção de três áreas do conhecimento: Pesquisa Operacional (PO), Planejamento e Controle da Produção (PCP) e Otimização Combinatória (OC). Isso porque consiste em métodos para tomada de decisão presente nos diversos processos produtivos, visando sempre a melhorar alguma medida de desempenho.

Sob a ótica da PO, na programação da produção utilizam-se modelos, algoritmos e ferramentas computacionais para a tomada de decisão de problemas reais. Para o PCP, a programação da produção consiste no melhor emprego dos recursos produtivos, assegurando a execução do que foi previsto no momento, com a quantidade e os recursos adequados. E na Matemática Aplicada, especificamente na área de OC, os problemas de programação da produção consistem no escalonamento de um conjunto de tarefas em um conjunto de máquinas de forma a minimizar ou maximizar uma função ou medida de desempenho.

Nesse contexto, uma parcela significativa dos sistemas de produção, o que inclui bens tangíveis e serviços, pode ser modelada como o problema denominado *flow shop*. Um sistema *flow shop* é identificado quando se tem

um processo que passa por etapas sucessivas. O termo *flow* significa "fluxo" e *shop*, "fábrica ou ambiente de produção". Por exemplo, na área industrial, o fluxo de produção em indústrias automotivas, de maquinários agrícolas, de produtos eletrônicos, têxteis, químicos, farmacêuticos e alimentícios caracteriza o problema de *flow shop*. Na área de serviços, a editoração de publicações e os projetos de consultoria que passam por várias etapas também podem ser modelados como *flow shop*.

Nesses casos, o problema reside em determinar a ordem em que as tarefas, produtos ou clientes passarão pelo sistema, ou seja, existe um **problema de sequenciamento**. Encontrar uma sequência apropriada para determinado ambiente, com suas especificidades e restrições, conduz a uma economia em diversas medidas de desempenho possíveis de serem consideradas, tanto em termos de tempo (duração total da programação, tempo de fluxo das tarefas, atrasos, adiantamentos) como diretamente de custos (multas por atraso, custos de estoque, de transporte, deterioração de material).

Na prática, esses problemas são fáceis de compreender e difíceis de resolver, não tanto por exigirem ferramental teórico ou capacidade de abstração, mas pela sua natureza combinatorial. O número de soluções possíveis cresce muito rapidamente com o aumento do número de tarefas e de máquinas do problema, por exemplo. Encontrar qualquer sequência (viável) de tarefas é muito fácil. Identificar uma boa solução não é tão difícil. Obter uma solução próxima da ótima é o desafio. Eis o papel dos chamados métodos heurísticos – resolver o problema em tempo de execução computacional aceitável, fornecendo uma solução pelo menos próxima da ótima.

Não é uma prática comum nas empresas utilizar métodos muito sofisticados no sequenciamento de produtos. É mais fácil encontrar indústrias que sequenciam a produção de modo empírico ou com base na *experiência* do programador. Também não existem no mercado muitos *softwares* comerciais que ofereçam soluções satisfatórias na área de programação da produção. Isso indica o grande potencial de pesquisa prática dessa área.

Além disso, como observaram Gupta e Stafford Jr. (2006), a teoria matemática de programação em *flow shop* sofre com muita abstração e pouquíssimas aplicações. O uso prático de técnicas de programação em *flow*

shop ainda é raro. Apesar dos quase sessenta anos de publicações nessa área, sabe-se pouco sobre o *flow shop* prático, exceto que é um problema muito frequente nos diversos tipos de indústrias. E os autores são ainda mais enfáticos ao afirmarem que, sem compreender de fato o *flow shop* real e prático, podemos passar mais 25 anos tentando resolver um problema que talvez não precise de solução, por ser o problema errado, sob a perspectiva prática.

Portanto, é preciso diversificar os esforços em pesquisa nessa área, incluindo os problemas emergentes, fazendo um levantamento das práticas industriais para ajudar na identificação dos problemas de programação reais. Segundo Gupta e Stafford Jr. (2006), isso irá preencher a lacuna entre os desenvolvimentos teóricos e as práticas de produção industrial.

Assim, esta obra objetiva disponibilizar o conhecimento de programação da produção em língua portuguesa, desde os métodos clássicos até os procedimentos de solução mais recentes, uma vez que não há um livro com essas características publicado em nosso idioma. Em inglês, existem muitas obras de referência, como as de Conway, Maxwell e Miller (1967), Baker (1974, 1992), French (1982), Morton e Pentico (1993), Błażewicz et al. (2001), Baker e Trietsch (2009) e Pinedo (2012).

Destas, muitas definições clássicas e suas demonstrações são apenas encontradas na obra consagrada de Baker (1974), que, apesar de nada recente em termos de ano de publicação, mantém-se válida e aplicável até hoje. Por esse motivo, neste livro também optou-se por apresentar muitas citações de Baker (1974), ainda que a fonte pareça equivocadamente desatualizada, e de Pinedo (2012), por se tratar da obra de referência mais recente que se conhece.

Esta obra resulta em parte de muitos estudos e investigações desenvolvidos por mais de dez anos, ao mesmo tempo que constitui um ponto de partida para diversas outras pesquisas, tanto aquelas apoiadas em simulação computacional como as práticas e aplicadas. Espero que seja de frequente utilização e consulta àqueles que já pertencem a esta área de estudos e que muitos novos pesquisadores, a partir desta leitura, possam também despertar interesse neste campo tão rico, inesgotável e apaixonante.

1 Introdução

Este capítulo contextualiza a área de programação da produção, muito conhecida também no Brasil por *scheduling*, como parte de outros grandes campos de estudo, como PO e PCP. Além disso, exemplifica os sistemas produtivos em que há tipicamente problemas de programação e sequenciamento da produção e enfatiza a importância estratégica da atividade de *scheduling* para a empresa.

Pesquisa Operacional aplicada aos sistemas produtivos

A área de conhecimento denominada Pesquisa Operacional (PO) fornece métodos objetivos para a tomada de decisão, principalmente em problemas grandes e complexos, substituindo os critérios intuitivos e subjetivos.

Sua origem remonta à época da Segunda Guerra Mundial, em que havia a necessidade de utilizar os recursos escassos de forma eficiente nas diversas operações militares. Os comandantes ingleses e norte-americanos convocaram um grande número de cientistas para desenvolver uma abordagem científica que lidasse com os problemas estratégicos e táticos de guerra. Na prática, eles foram contratados para realizar *pesquisa* sobre *operações* militares (Hillier; Lieberman, 2006, p. 1).

Com o fim da guerra, os resultados bem-sucedidos da PO despertaram interesse na sua aplicação fora da área militar, como é o caso dos setores industriais, comerciais e governamentais. O desenvolvimento industrial causou um rápido aumento da complexidade e da especialização das organizações. Os profissionais então perceberam que se deparavam basicamente com os mesmos problemas militares, porém num contexto diferente. Dessa forma, foram surgindo métodos de solução para problemas nas diversas áreas do conhecimento em que há necessidade de tomada de decisão, como os sistemas produtivos.[1]

Segundo Harding (1981, p. 24), um **sistema de produção** é um conjunto de partes inter-relacionadas que, quando ligadas, atuam de acordo com padrões estabelecidos sobre *inputs* (entradas) no sentido de produzir *outputs* (saídas).

É um processo composto por entradas, transformação e saídas. As entradas podem ser matérias-primas, informações, consumidores; as transformações são as atividades que modificam o estado das entradas, por exemplo, operações de corte, perfuração e moldagem, ou então, cozimento e congelamento de alimentos; as saídas constituem-se, em geral, de bens e serviços, especificamente de produtos fabricados, cargas transportadas, materiais impressos, pacientes examinados.

O sistema de produção está inter-relacionado com outros sistemas, por exemplo, o sistema financeiro, de *marketing*, de recursos humanos, entre outros. E ainda faz parte de um sistema maior, a empresa, que por sua vez integra o sistema econômico do país, e assim por diante.

É importante salientar que essa definição não se restringe apenas a indústrias, em que há produção de bens tangíveis, mas também a empresas de serviços, como pode ser observado nos exemplos citados. Agências bancárias, hospitais e empresas de consultoria são casos de sistemas de produção de serviços.

Podem-se classificar os sistemas de produção de várias formas. Uma das mais importantes e a mais relacionada ao contexto deste livro é a **classificação quanto ao fluxo dos processos**, proposta por Lustosa et al. (2008, p. 18):

[1] Mais detalhes sobre a teoria e os métodos de PO podem ser encontrados em Hillier e Lieberman (2006), Arenales et al. (2007) e Taha (2008).

- **Processo em linha ou contínuo:** fabricação em larga escala de produtos padronizados e com pouca diversificação, como no caso do aço;
- **Processo em lote ou intermitente:** produção caracterizada pela flexibilidade, ou seja, a capacidade de produzir uma grande variedade de produtos; as linhas de produção devem se ajustar a cada caso, o que representa maior dificuldade de operação e controle. Exemplos claros são as indústrias têxteis e automobilísticas;
- **Projeto:** produção de itens complexos ou de grande porte, geralmente únicos, tais como navios, edifícios, plataformas de petróleo e gasodutos.

Embora os problemas de programação e sequenciamento da produção possam ser encontrados em qualquer tipo de empresa, inclusive nas de serviços, surgem com mais frequência em processos de produção intermitente.

O mercado consumidor competitivo tem forçado as empresas a oferecer seus produtos e serviços com alta qualidade, baixo preço e reduzidos prazos de entrega. Além disso, os clientes frequentemente solicitam mudanças nas especificações dos produtos e dos pedidos, exigindo maior flexibilidade por parte das organizações.

Essa situação contrasta com muitos sistemas produtivos, em que existem linhas de produção dimensionadas para fabricar grandes lotes de produtos a baixo custo. Entretanto, há o inconveniente de que as mudanças exigem ajustes nas linhas, o que requer tempo e investimento.

Nesse contexto, as empresas devem oferecer um amplo catálogo de produtos inovadores e diferenciados e possuir um sistema de produção flexível que permita ao mesmo tempo rapidez na entrega e custos baixos para assegurar a completa satisfação do cliente. É evidente que, com essas premissas, faz-se necessário um uso racional e eficiente dos recursos produtivos.

Planejamento e controle da produção

Nas empresas, independentemente do tipo de sistema de produção, é necessário tomar uma série de decisões que dependem do horizonte de

tempo considerado. As funções dos sistemas de produção são desenvolvidas pelo Planejamento e Controle da Produção (PCP). As atividades do PCP são exercidas nos três níveis hierárquicos de uma empresa: estratégico, tático e operacional (Tubino, 2006, p. 24; Arenales et al., 2007, p. 205; Corrêa; Gianesi; Caon, 2011, p. 21-28).

O nível mais alto é o **estratégico**, em que são definidas as políticas estratégicas de longo prazo da companhia e que envolve altos investimentos. O PCP participa da formulação do planejamento estratégico da produção, gerando um plano de produção. Esse nível trata da escolha e do projeto do processo, relacionados ao arranjo de máquinas e outros equipamentos e com a determinação da sua capacidade, em função da demanda futura.

Um dos parâmetros estabelecidos nesse nível é a quantidade de produção unitária, ou seja, a menor quantidade em que um produto é fabricado. Tamanhos de lotes determinados no nível tático são múltiplos inteiros da quantidade de produção unitária. Por exemplo, uma unidade de produção de pisos de cerâmica é uma caixa com várias peças, e um lote de produção corresponde a todas as caixas produzidas com o mesmo modelo.

O nível **tático**, em que são preparados os planos de médio prazo para a produção, trata do planejamento das atividades, que consiste de dois subníveis: planejamento agregado da produção e planejamento de quantidades de produção. O primeiro envolve decisões como níveis de mão de obra, hora extra e subcontratação, que duram tipicamente de vários meses a um ano. As necessidades são determinadas por uma medida agregada de demanda, por exemplo, horas de produção. Já o planejamento das quantidades de produção envolve a determinação, para cada produto, de quanto e quando produzir em um horizonte que se estende geralmente de algumas semanas a seis meses; nesse período os níveis de mão de obra e tempo disponível de máquina são considerados fixos.

Dessa forma, o PCP desenvolve o Plano Mestre de Produção (*master production schedule* – MPS) e o Planejamento de Recursos de Produção (*material resource planning*, conhecido como MRP II). A partir das demandas externas ou independentes, o MPS determina quanto e quando produzir de

cada produto final e, a partir disso, o MPR II planeja de forma sincronizada as necessidades dos componentes e a matéria-prima dos produtos finais.

No nível **operacional** são preparados os programas de curto prazo de produção e é realizado o acompanhamento do processo, controlando-se as atividades diárias das ordens de produção provenientes do nível tático.

Nesse nível, o PCP é responsável por administrar estoques, elaborar sequenciamento e emitir e liberar ordens de compras, fabricação e montagem. As principais decisões envolvidas nesse nível são: designação de tarefas a máquinas e programação das tarefas em cada máquina, ou seja, definição da sequência de processamento das tarefas e do instante de início e término da sua execução. Essa atividade constitui a **programação da produção**.

Morton e Pentico (1993, p. 5) definem a programação da produção de forma mais ampla, defendendo a sua importância estratégica para toda a empresa, não apenas no nível operacional. Segundo os autores, a programação permeia toda a atividade econômica da organização, pois consiste no processo de escolher e agendar a utilização dos recursos para realizar todas as atividades necessárias a fim de produzir os resultados almejados, no momento desejado, enquanto satisfaz um grande número de restrições referentes ao tempo, à relação entre as atividades e aos recursos.

Todas as definições feitas para *scheduling* conduzem a um resultado. *Scheduling* é um dos mais importantes processos de tomada de decisão no gerenciamento das empresas, uma vez que constitui uma base importante para o planejamento de atividades. Além disso, tem uma ampla gama de aplicações, abrangendo planejamento de projetos, gestão de fábricas, escala de horários, transporte e roteamento de veículos, entre outras (Yagmahan; Yenisey, 2009).

Fernandes e Godinho Filho (2010) citam exemplos de problemas de programação da produção no setor de serviços e no setor industrial, respectivamente: em um grande aeroporto, os aviões devem ser sequenciados para que se possa controlar com segurança as decolagens e aterrissagens nas pistas disponíveis; em uma fundição, são produzidas peças de compressores herméticos, cujos lotes são programados visando a minimizar o estoque em processo e o estoque de itens acabados.

Lustosa et al. (2008, p. 12) defendem a eficácia operacional como elemento estratégico da empresa para obter desempenho superior nos negócios. A eficácia operacional é capaz de conferir vantagem competitiva sustentável.

Esse contexto reforça o papel fundamental do PCP na integração dos sistemas. As decisões do PCP utilizam o conhecimento de fatores econômicos, de mercado, do projeto de engenharia, das limitações de capacidade de produção, entre outros fatores, consistindo na compreensão sistêmica do processo de produção para que a empresa atinja os seus objetivos.

Estrutura do livro

Após este capítulo introdutório, que aborda a origem da PO e dos sistemas produtivos e também o PCP, o restante da obra está estruturado da seguinte forma. O segundo capítulo traz importantes definições sobre programação da produção e tempos de preparação (*setup*), a classificação dos problemas nos diversos ambientes produtivos, as variáveis e a conhecida notação de três campos dos problemas.

O detalhamento do problema-foco desta obra, o sistema *flow shop*, consta do terceiro capítulo. As características do ambiente de produção, alguns tipos particulares, representações gráficas, propriedades estruturais e as medidas de desempenho também são apresentados. O quarto capítulo descreve os métodos de solução exata clássicos da literatura, como o algoritmo de Johnson, o algoritmo *branch-and-bound* e um modelo de programação linear inteira mista.

O quinto capítulo descreve e exemplifica os principais métodos heurísticos de solução, iniciando pelas regras de prioridade e passando pelos famosos algoritmos CDS, NEH e N&M, além dos critérios para avaliação de desempenho das heurísticas. Por fim, o sexto capítulo apresenta as meta-heurísticas mais aplicadas aos problemas de *flow shop*, como algoritmo genético, *simulated annealing*, busca tabu, colônia de formigas, otimização por nuvem de partículas, algoritmo imunológico e GRASP, entre outras.

2 Programação da produção

É importante apresentar aqui as definições clássicas e aquelas em torno das quais há divergência dependendo do autor, visando a uniformizar a linguagem nesta obra e especificar o significado utilizado nos conceitos. Este capítulo apresenta as definições fundamentais para o estudo dos problemas de programação e sequenciamento, tanto do tema central deste livro, o *flow shop*, como de outros ambientes de produção.

Definições

A programação da produção ou *scheduling* pode ser definida como a alocação de recursos escassos para a execução de tarefas em uma base de tempo (Baker, 1974, p. 2; Baker; Trietsch, 2009, p. 2). Os recursos podem ser máquinas em uma fábrica, pistas em um aeroporto, trabalhadores em construções e unidades de processamento em ambiente computacional. As tarefas podem ser os pedidos de produção, as decolagens e aterrissagens de aviões, os estágios em um projeto de construção, as execuções de programas de computador, entre outras (Pinedo, 2012, p. 1).

É muito comum utilizar o termo original em inglês, pois o termo "programação da produção" nem sempre atende ao seu significado abrangente. Ou então, para se evitar confusão ao dizer que a programação da produção é composta pelas etapas de alocação, sequenciamento e programação, como será definido ainda nesta seção.

A teoria de *scheduling* inclui uma variedade de técnicas úteis para resolver problemas de programação da produção. Além disso, esse campo de pesquisa tem foco no desenvolvimento, aplicação e avaliação de procedimentos combinatórios, técnicas de simulação, problemas de rede e métodos de solução heurística. A seleção de uma técnica apropriada depende da complexidade do problema, da natureza do modelo, do critério de otimização, entre outros fatores. Em muitos casos, é mais adequado utilizar várias técnicas alternativas. Por essa razão, a teoria de *scheduling* compreende tanto o estudo de métodos como o de modelos (Baker, 1974, p. 6).

As principais obras de referência na área são as de Conway, Maxwell e Miller (1967), Baker (1974, 1992), French (1982), Morton e Pentico (1993), Błażewicz et al. (2001), Baker e Trietsch (2009) e Pinedo (2012).

Neste livro serão padronizadas algumas das definições clássicas. Uma *operação* é uma atividade elementar a ser executada. O tempo requerido pela operação é chamado de *tempo de processamento*. Uma *tarefa* é um conjunto de operações inter-relacionadas por restrições de precedência derivadas de limitações tecnológicas. As restrições de precedência definem a *rota* das operações, ou seja, a ordem em que são processadas. Uma *máquina* é um equipamento, dispositivo ou instalação capaz de executar uma operação (Hax; Candea, 1984, p. 258-259).

A atividade de *scheduling* envolve a **alocação** (associação das tarefas às máquinas), o **sequenciamento** (ordenação das tarefas em cada máquina) e a posterior **programação** (obtenção dos tempos de início e término de cada operação). Segundo Pinedo (2012, p. 1), *scheduling* é um processo de tomada de decisão presente tanto em sistemas de produção como em ambientes de processamento de informações, além das empresas de transporte e distribuição e outros tipos de serviços industriais.

Em alguns ambientes, a programação é dada por uma simples permutação de tarefas, ou seja, não é necessário fazer a alocação às máquinas. É o caso de sistemas com apenas uma máquina (problema de máquina única) ou com máquinas dispostas em série por onde todas as tarefas passam na mesma ordem (*flow shop* permutacional). Assim, a programação da produção consiste em um problema de sequenciamento. Por isso, é comum referir-se ao problema simplesmente como **sequenciamento da produção**.

Segundo Ruiz (2003, p. 5), a programação da produção pressupõe que o planejamento da produção já tenha sido realizado. Portanto, conhecem-se os produtos a fabricar e suas correspondentes quantidades, assim como a configuração, disposição e quantidade de recursos disponíveis. Um sequenciamento adequado da produção é vital para o funcionamento da empresa, garantindo os resultados almejados. Como afirma Pinedo (2012, p. 1), os objetivos do sequenciamento podem ser diversos, como a minimização da data de término da última tarefa ou do número de pedidos que se finalizam após a data de entrega prometida ao cliente.

De acordo com Slack et al. (1999, p. 245), a programação é uma das mais complexas atividades no gerenciamento da produção. Os programadores precisam lidar com tipos diferentes de recursos ao mesmo tempo. Além disso, as máquinas terão capacidades variáveis e o pessoal terá diferentes habilidades. Ainda mais importante, o número de programações possíveis cresce rapidamente à medida que o número de tarefas e operações aumenta. Ou seja, para um conjunto de n tarefas a serem processadas em uma máquina, há $n!$ (n fatorial) sequências possíveis. Considerando-se m máquinas, o número de programações possíveis passa para $(n!)^m$. Isso é válido quando as tarefas têm a mesma data de liberação para serem executadas.

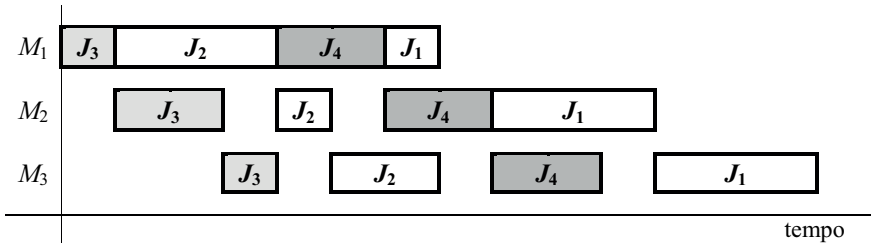
Assim, a primeira característica que torna os problemas de programação difíceis de serem resolvidos é sua natureza combinatorial, o que significa que o número de soluções possíveis cresce exponencialmente em várias dimensões, de acordo com a quantidade de tarefas, operações ou máquinas.

Uma ferramenta comumente usada na programação é o **gráfico de Gantt**, inventado em 1917 por Henry Laurence Gantt (1861-1919), que

representa por meio de barras o tempo de processamento das operações. Os momentos de início e fim de atividades podem ser indicados no gráfico, bem como, algumas vezes, o progresso real do trabalho. A vantagem do gráfico de Gantt é que ele proporciona uma representação visual simples do que deve ocorrer em cada operação (Slack et al., 1999, p. 244-245). Gantt foi um dos primeiros a desenvolver um sistema de PCP baseado em restrições de capacidade e tempo, cujos cálculos eram elaborados manualmente (Lustosa et al., 2008, p. 1).

A Figura 1 mostra um exemplo de gráfico de Gantt representando a execução das operações de quatro tarefas, J_1, J_2, J_3 e J_4, em um sistema com três máquinas consecutivas.

Figura 1 - Exemplo de gráfico de Gantt



Fonte: Elaborada pelo autor.

Classificação de problemas

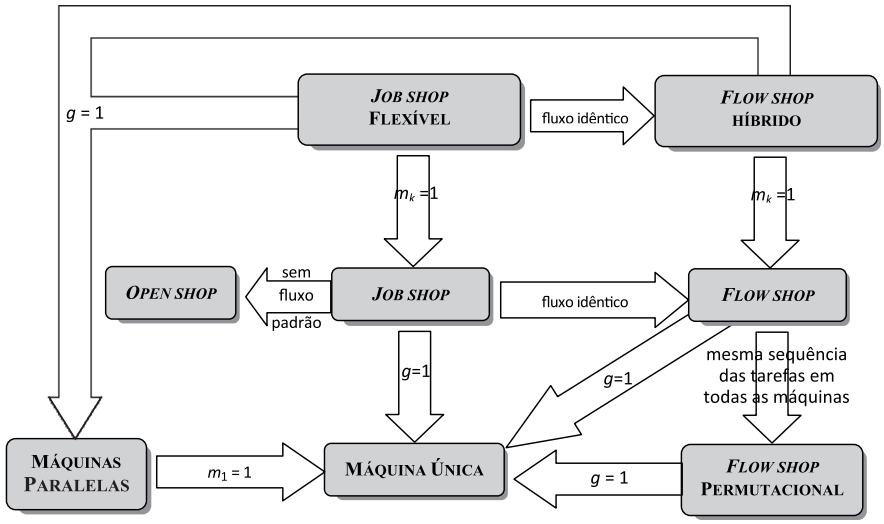
Um problema de programação da produção é determinado pelo número de tarefas e operações a serem processadas, pelo número e tipo de máquinas disponíveis, pelo padrão de fluxo das operações nas máquinas e pelo critério de otimização com que se avalia uma solução (Conway; Maxwell; Miller, 1967, p. 6). Normalmente, nos problemas de programação, assume-se que o número de tarefas e de máquinas seja finito e determinado.

Os problemas tornam-se mais complexos com a presença de restrições, como, por exemplo, relacionando-se uma tarefa a outra, recursos a tarefas, um recurso a outro e um recurso ou tarefa a eventos externos ao sistema.

Pode haver também uma restrição de precedência entre tarefas ou não ser possível usar dois recursos simultaneamente durante certo período de tempo. Ou então um recurso pode não estar disponível durante um intervalo de tempo específico em razão da manutenção. Uma vez que esses complexos inter-relacionamentos podem tornar muito difícil a busca da solução exata ou mesmo aproximada de um grande problema, é natural resolver primeiro versões mais simples. Então, a sensibilidade da solução pode ser testada quanto à sua complexidade e soluções aproximadas podem ser encontradas para problemas difíceis (Morton; Pentico, 1993, p. 6).

Para classificar os principais modelos de programação, é necessário caracterizar a configuração dos recursos e o comportamento das tarefas. Segundo MacCarthy e Liu (1993) e as adaptações de Moccellin e Nagano (2003), as restrições tecnológicas são determinadas principalmente pelo padrão do fluxo das tarefas nas máquinas, levando à classificação apresentada na Figura 2, que esquematiza até mesmo a relação entre os problemas:

FIGURA 2 - Relação entre as classes de problemas



g: número de estágios de produção
m_k: número de máquinas do estágio *k* (para *k* = 1, 2, ..., *g*)

Fonte: Elaborada pelo autor.

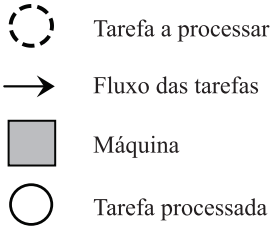
Como ilustra a Figura 2, o problema mais simples é o de máquina única, que se diferencia do problema de máquinas paralelas pelo número de máquinas ($m_1 = 1$), pois ambos possuem um único estágio de produção ($g = 1$). Ao se reduzir problemas de fluxo como *flow shop*, *flow shop* permutacional e *job shop* para um único estágio ($g = 1$), chega-se também ao problema de máquina única.

A Figura 2 também mostra a relação entre os problemas de *flow shop* e *flow shop* permutacional, que se diferenciam pela ordem em que as tarefas são processadas nas máquinas, como será detalhado mais adiante na seção "*Flow shop*". A figura apresenta também a relação entre os problemas de *job shop* e *open shop*, que se distinguem pela existência ou não do fluxo padrão das tarefas nas máquinas (as características desses problemas são detalhadas nas seções "*Job shop*" e "*Open shop*").

Ainda pode ser observado na Figura 2 que, assim como o problema de *job shop* flexível se reduz ao *job shop*, o problema de *flow shop* híbrido transforma-se num *flow shop* ao se limitar a apenas um o número de máquinas em cada estágio k ($m_k = 1$ para $k = 1, ..., g$). Os problemas de *job shop* e *job shop* flexível não possuem fluxo idêntico nos estágios, o que passa a ocorrer no *flow shop* e no *flow shop* híbrido. Tanto o *job shop* flexível como o *flow shop* híbrido se transformam no problema de máquinas paralelas ao se reduzir a apenas um o número de estágios de produção ($g = 1$).

Na área de programação da produção, muitos termos são utilizados em sua forma original inglesa. Nos casos em que o termo em português tenha significado equivalente, serão introduzidas aqui ambas as formas.

As próximas seções caracterizam cada um dos problemas citados e utilizam a seguinte legenda para ilustrar o fluxo das tarefas nas máquinas.



Máquina única

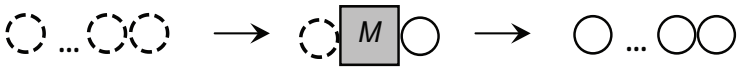
O problema de **máquina única** é caracterizado pela existência de apenas uma etapa de produção. É o ambiente em que há apenas uma máquina ou equipamento disponível para o processamento das tarefas, ou então se trata de uma situação modelada como um único estágio de produção.

Para um problema com *n* tarefas, existem *n*! sequências possíveis. Em problemas em que todas as tarefas estão disponíveis na data zero da programação, não há tempos de preparação (*setup*) explícitos nem tempo ocioso de máquina, e a duração total da programação (medida denominada *makespan*) tem sempre o mesmo valor, independentemente da sequência das tarefas.

A solução ótima de muitos problemas de máquina única pode ser encontrada por meio de simples regras de prioridade (que serão definidas no quinto capítulo).

O problema de máquina única é ilustrado na Figura 3.

FIGURA 3 - Problema de máquina única



Fonte: Elaborada pelo autor.

Alguns exemplos de aplicações incluem:

- oficinas com uma única máquina;
- conjunto de máquinas e equipamentos que operam como se fossem uma única máquina, por exemplo, em indústrias químicas;
- máquina dominante em um processo, que exige atenção especial na atividade de sequenciamento, por exemplo, em indústrias de papel;
- máquina-gargalo no sistema (que aqui representa aquela com maior carga de trabalho);
- pista em um aeroporto;
- equipamentos de uma sala de cirurgia.

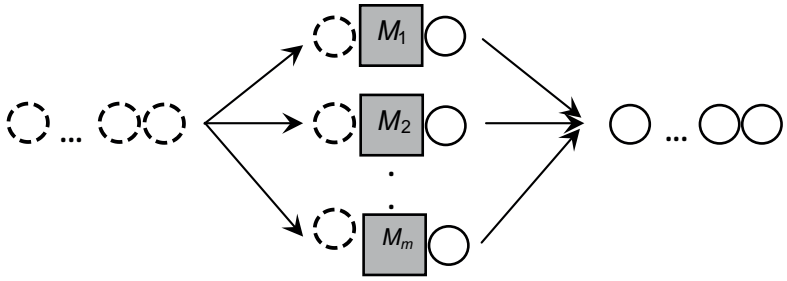
Máquinas paralelas

Os ambientes que possuem um único estágio de produção contendo duas ou mais máquinas disponíveis para executar qualquer tarefa são conhecidos como **máquinas paralelas**. Nesse problema, cada tarefa pode ser executada por qualquer máquina, porém em apenas uma delas.

Tais ambientes podem representar um incremento de capacidade em relação a um antigo problema de máquina, quando um ou mais equipamentos novos foram agregados a outro já existente, passando todos a operar em paralelo. Para um problema com n tarefas e m máquinas, existem $(n!)^m$ programações possíveis. Nesse caso, a duração total da programação depende da programação das tarefas nas máquinas.

O problema de máquinas paralelas é esquematizado na Figura 4.

FIGURA 4 - Problema de máquinas paralelas



Fonte: Elaborada pelo autor.

O problema pode ser classificado de acordo com o tipo de máquina que contém:

- **Máquinas idênticas:** todas as máquinas têm as mesmas especificações e, portanto, o tempo de processamento das tarefas é o mesmo em todas elas;
- **Máquinas uniformes ou proporcionais:** as máquinas possuem diferentes velocidades, e as tarefas têm diferentes tempos de processamento em cada uma delas, porém com um fator de proporcionalidade;

- **Máquinas não relacionadas:** cada tarefa tem tempos de processamento diferentes em cada máquina, sem nenhuma correlação entre os valores.

Os exemplos de aplicações são:

- várias injetoras operando em paralelo;
- caixas de um banco;
- computadores com processamento paralelo;
- pontos de atracação de navios.

Flow shop

Neste sistema, as máquinas são distintas e dispostas em série, ambiente também conhecido como "fábrica em fluxo" em razão da rota unidirecional que as tarefas seguem. Todas as tarefas possuem o mesmo fluxo de processamento nas máquinas. Esse problema será mais bem detalhado no terceiro capítulo.

A Figura 5 apresenta o esquema de um problema de *flow shop*.

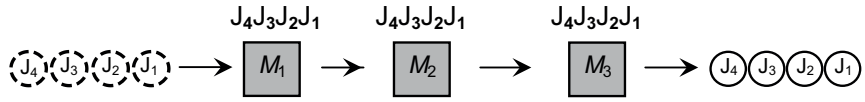
FIGURA 5 - Problema de ***flow shop***

Fonte: Elaborada pelo autor.

Quando as tarefas puderem saltar máquinas, ou seja, não serem processadas em uma ou mais máquinas, o problema pode também ser denominado ***flow line***. Além disso, quando a ordem de processamento das tarefas em cada máquina é estritamente a mesma, o problema é denominado *flow shop* permutacional. As Figuras 6 e 7 exemplificam, respectivamente, um *flow shop* permutacional e um *flow shop* não permutacional. Embora o fluxo nas máquinas seja o mesmo, repare-se na ordem de execução das tarefas em cada máquina individualmente.

Para um problema com n tarefas e m máquinas, haverá $(n!)^m$ programações possíveis em um *flow shop* não permutacional e apenas $n!$ sequências possíveis em um *flow shop* permutacional.

FIGURA 6 - *Flow shop* permutacional



Fonte: Elaborada pelo autor.

FIGURA 7 - *Flow shop* não permutacional

Fonte: Elaborada pelo autor.

As aplicações mais comuns são:

- máquinas e equipamentos organizados em série;
- sequência de células de produção especializadas em determinados tipos de produtos;
- operações de fundição e laminação com fluxo unidirecional;
- fluxo de produção em indústrias automotivas e de eletrônicos.

Flow shop híbrido

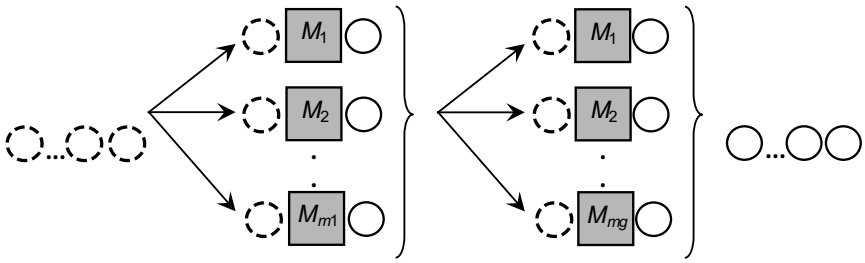
No ***flow shop* híbrido** ou **flexível**, as tarefas são processadas em múltiplos estágios de produção na mesma ordem e em cada um deles há máquinas paralelas, podendo-se variar a quantidade por estágio. Cada tarefa é processada por apenas uma máquina em cada estágio. Pode representar um incremento de capacidade em relação a um *flow shop* tradicional, quando novas máquinas foram agregadas aos estágios.

Na literatura, esse problema também é encontrado com as denominações *flow shop* com múltiplas máquinas, *flow shop* com múltiplos processadores, *flexible flow shop* e *flexible flow line*.

Em um problema com n tarefas, g estágios e m máquinas em cada estágio, haverá $(n!)^{gm}$ programações possíveis. Ou então, quando há diferentes quantidades de máquinas em cada estágio, representadas por m^k, haverá $(n!)^{\prod_{k=1}^{g} m_k} = (n!)^{m_1 m_2 \ldots m_g}$ programações possíveis.

A Figura 8 mostra um *flow shop* híbrido.

FIGURA 8 - *Flow shop* híbrido



Fonte: Elaborada pelo autor.

Algumas aplicações são:

- várias prensas operando em paralelo;
- estágios de produção com capacidade incrementada.

Job shop

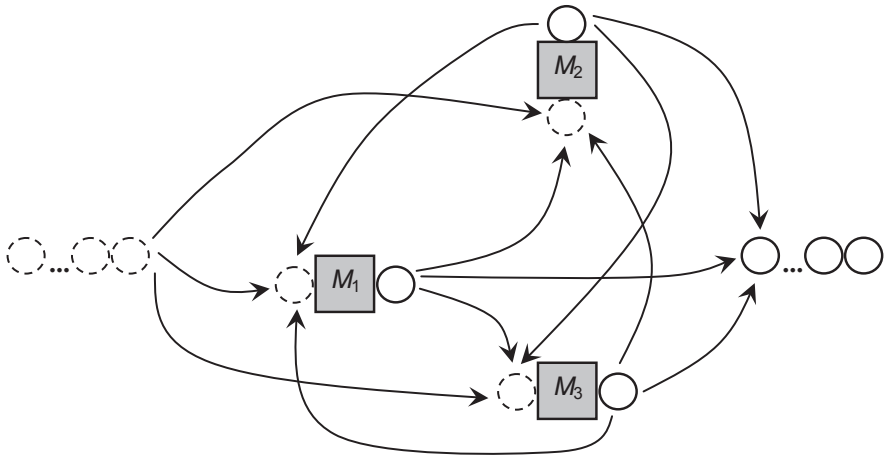
No problema de ***job shop***, as tarefas possuem rotas distintas e predefinidas nas máquinas. Ou seja, cada tarefa tem seu fluxo individual (*job* = tarefa, *shop* = fábrica). Todas as máquinas podem iniciar e terminar uma rota. Esse ambiente também é chamado de fábrica por rota ou oficina de máquinas.

As aplicações de *job shop* mais comuns são:

- oficina de manutenção;
- oficina de ferramentaria;
- funilaria de carros.

A Figura 9 esquematiza o problema de um *job shop* com três máquinas.

Figura 9 - *Job shop*



Fonte: Elaborada pelo autor.

Job shop flexível

O ***job shop*** flexível é um problema de *job shop* em que existem estágios com duas ou mais máquinas paralelas, sendo cada tarefa processada por apenas uma máquina em cada um dos estágios.

Open shop

O problema de ***open shop*** é semelhante ao *job shop*, com a diferença de que as tarefas não possuem rotas específicas, ou seja, não seguem um fluxo padrão preestabelecido, podendo até mesmo não passar por uma ou mais máquinas. As tarefas podem ser processadas em qualquer ordem.

As principais aplicações de *open shop* são:

- oficina de reparos específicos, em aviões, por exemplo;
- centros de controle de qualidade, com atividades simultâneas.

Variáveis de um problema

Nos problemas de programação da produção, a função objetivo pode ser definida em termos de várias medidas de desempenho, que podem ser a duração da programação (*makespan*), o tempo de fluxo das tarefas (*flow time*), adiantamentos, atrasos, número de tarefas atrasadas etc.

Em geral, os problemas exibem um conjunto de n tarefas, definido por $J = \{J_1, J_2, ..., J_j, ..., J_n\}$, e um conjunto de m máquinas, definido por $M = \{M_1, M_2, ..., M_k, ..., M_m\}$.

Os seguintes parâmetros são usados como entrada de um problema:

- Tempo de processamento (*processing time*) – $\boldsymbol{p_j}$: tempo requerido para a execução da tarefa J_j;
- Data de liberação (*release date*) – $\boldsymbol{r_j}$: instante em que a tarefa J_j está pronta para ser processada;
- Prazo ou data de entrega (*due date*) – $\boldsymbol{d_j}$: instante em que a tarefa J_j deve ser concluída.

Os parâmetros a seguir são encontrados após a determinação da programação:

- Data de término (*completion date*) – $\boldsymbol{C_j}$: instante em que a tarefa J_j é completada;
- Tempo de fluxo (*flow time*) – $\boldsymbol{F_j}$: tempo gasto pela tarefa J_j no sistema;
- Desvio de pontualidade (*lateness*) – $\boldsymbol{L_j}$: desvio do término da tarefa J_j em relação ao seu prazo;
- Atraso (*tardiness*) – $\boldsymbol{T_j}$: atraso da tarefa J_j em relação ao prazo;
- Adiantamento (*earliness*) – $\boldsymbol{E_j}$: adiantamento da tarefa J_j em relação ao prazo.

O desvio de pontualidade pode ser negativo, positivo ou zero. Valores negativos indicam adiantamento, positivos representam atraso e zero significa pontualidade. Adiantamentos na execução das tarefas podem ocasionar

custos de estoque ou de deterioração de material, e atrasos acarretam multas e redução da qualidade do serviço.

O cálculo desses parâmetros é feito da seguinte forma:

$$F_j = C_j - r_j \tag{2.1}$$

$$L_j = C_j - d_j \tag{2.2}$$

$$T_j = \max\{C_j - d_j, 0\} \tag{2.3}$$

$$E_j = \max\{d_j - C_j, 0\} \tag{2.4}$$

As medidas de desempenho dos problemas geralmente se referem às datas de término das tarefas. Considerando que haja *n* tarefas a serem programadas, as principais medidas de desempenho podem ser definidas como:

- **Duração total da programação ou *makespan*:**

$$C_{\max} = \max_{1 \le j \le n} C_j - \min_{1 \le j \le n} r_j \tag{2.5}$$

- **Tempo médio de fluxo:** $\overline{F} = \dfrac{\sum_{j=1}^{n} F_j}{n}$ (2.6)

- **Atraso médio:** $\overline{T} = \dfrac{\sum_{j=1}^{n} T_j}{n}$ (2.7)

- **Atraso máximo:** $T_{\max} = \max_{1 \le j \le n} T_j$ (2.8)

- **Máximo desvio de pontualidade:** $L_{\max} = \max_{1 \le j \le n} L_j$ (2.9)

- **Número de tarefas atrasadas:**

$$n_T = \sum_{j=1}^{n} U_j, onde \begin{cases} U_j = 1, se\, T_j > 0 \\ U_j = 0, caso\, contrário \end{cases} \tag{2.10}$$

- **Adiantamentos e atrasos:** $E + T = \sum_{j=1}^{n} (E_j + T_j)$ (2.11)

As medidas de desempenho podem ser divididas em duas categorias: medidas baseadas em datas de término e medidas baseadas em prazos.

O *makespan* e o tempo médio de fluxo são do primeiro tipo, enquanto as demais pertencem à segunda categoria. Além disso, as tarefas podem ter pesos (w_j) de acordo com sua importância ou prioridade, e esses valores podem compor também a medida de desempenho do problema.

A minimização do *makespan* equivale à maximização da utilização dos recursos, enquanto a minimização do tempo médio de fluxo corresponde à redução do estoque em processo (conhecido por *work-in-process* – WIP).

A otimização conjunta dos adiantamentos e atrasos é parte da filosofia *just-in-time*, que envida esforços para entregar os produtos o mais próximo possível da data prometida, reduzindo assim estoques, tempos de espera e multas por atraso. Essa medida de desempenho também pode ser considerada com pesos globais para adiantamentos e atrasos ($\alpha\Sigma E_j + \beta\Sigma T_j$) ou pesos individuais para os valores de cada tarefa ($\Sigma\alpha_j E_j + \Sigma\beta_j T_j$).

Além disso, os problemas podem ter apenas uma medida de desempenho ou então considerar mais de uma medida em conjunto, sendo assim bicritérios ou multicritérios. Uma revisão de programação da produção multiobjetivo foi apresentada por Lei (2009).

Tempos de *setup*

Muitas pesquisas em programação da produção desconsideram os tempos de preparação das máquinas ou então os incluem nos tempos de processamento das tarefas. Isso simplifica a análise das aplicações, porém afeta a qualidade da solução quando tais tempos têm uma variabilidade relevante em função da ordenação das tarefas nas máquinas. Por se tratar de uma restrição que tem se tornado cada vez mais comum nas pesquisas, este assunto merece especial atenção.

O **tempo de *setup*** ou **tempo de preparação** inclui o trabalho de preparar a máquina, o processo ou a oficina para a fabricação de produtos. Isso abrange o tempo para obtenção das ferramentas, posicionamento dos materiais a serem usados no trabalho, processos de limpeza, preparação e ajuste das ferramentas e inspeção de materiais (Allahverdi; Gupta; Aldowaisan, 1999).

Muitas pesquisas têm enfocado a importância de se considerar explicitamente os tempos de *setup* no problema. Foram publicados vários trabalhos de revisão da literatura envolvendo tempos de *setup*, tais como os de Liaee e Emmons (1997), Allahverdi, Gupta e Aldowaisan (1999), Zhu e Wilhelm (2006) e Allahverdi et al. (2008).

Existem duas classificações para os tipos de *setup* presentes nos problemas, que podem ou não coexistir em situações reais, em função das características da tarefa e da máquina envolvidas.

A primeira classificação refere-se à dependência da sequência de tarefas:

- **Tempos de *setup* dependentes da sequência:** a duração do *setup* depende tanto da tarefa a ser processada quanto daquela processada imediatamente antes na mesma máquina. Essa abordagem é necessária em indústrias químicas, por exemplo, onde o processo de limpeza de componentes variados é diferenciado para assegurar os baixos níveis toleráveis de impureza. Situações semelhantes podem ser encontradas na produção de diferentes cores de tinta, concentrações de detergente e misturas de combustível. Além disso, existem processos que requerem temperaturas específicas; assim, o ajuste da temperatura (aquecimento ou resfriamento) também é um exemplo de *setup* dependente da sequência.
- **Tempos de *setup* independentes da sequência:** a duração do *setup* depende somente da tarefa a ser processada. Ocorre nos casos em que é suficiente considerar o tempo de *setup* explicitamente, ou seja, separado dos tempos de processamento das tarefas – por exemplo, em indústrias moveleiras e de cerâmica com produtos que não envolvem diferentes cores de tinta.

As Tabelas 1 e 2 exemplificam os dois tipos de *setup* descritos. Quando os tempos de *setup* são independentes da sequência e representados por s_j, há apenas um valor para cada tarefa. Quando são dependentes da sequência e denotados por s_{ij}, é preciso uma matriz de tempos para informar a tarefa anterior e a que será produzida. A matriz pode ser quadrada (*nxn*) ou ter

(n+1) linhas e n colunas, se for considerado o *setup* da primeira tarefa da sequência; nesse caso, é representada uma tarefa inicial fictícia J_0.

Tabela 1 - Exemplo de *setup* independente da sequência

	J_1	J_2	J_3	J_4	J_5	J_6
s_j	6	4	2	8	1	9

Fonte: Elaborada pelo autor.

Tabela 2 - Exemplo de *setup* dependente da sequência

s_{ij}	J_1	J_2	J_3	J_4	J_5	J_6
J_0	5	7	8	3	4	11
J_1	–	2	1	2	6	8
J_2	9	–	3	9	7	5
J_3	8	12	–	13	1	6
J_4	4	2	7	–	5	1
J_5	3	1	4	5	–	2
J_6	6	3	9	8	4	–

Fonte: Elaborada pelo autor.

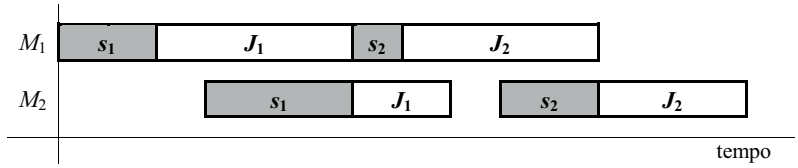
A segunda classificação dos tempos de *setup* refere-se à possibilidade de antecipação em relação à liberação da tarefa:

- ***Setup* antecipado:** pode ser realizado antes da liberação da tarefa, ou seja, antes do término da operação no estágio anterior. Uma importante implicação dos tempos de *setup* antecipados é que eles podem ser iniciados na máquina ou estágio subsequente enquanto a tarefa ainda está sendo executada. E como é comum haver tempo ocioso na segunda máquina em diante, antecipar o *setup* é uma vantagem para medidas de desempenho regulares, como é o caso do *makespan* e do tempo de fluxo.

- ***Setup* não antecipado:** pode ser realizado somente após a liberação da tarefa na máquina. São os casos que requerem que a peça ou o produto a ser processado esteja presente na máquina para que o *setup* seja realizado, como, por exemplo, nas operações de ajuste e posicionamento.

As Figuras 10 e 11 ilustram, respectivamente, um *setup* antecipado e um não antecipado, perceptíveis na máquina M_2 de cada caso.

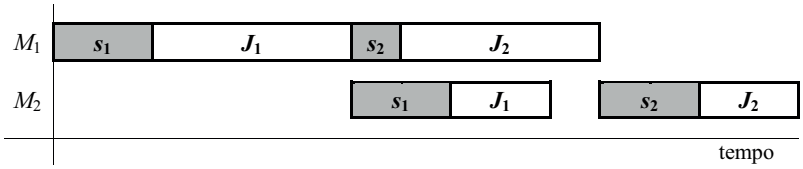
FIGURA 10 - Exemplo de *setup* antecipado



Fonte: Elaborada pelo autor.

Observe-se que, quando o *setup* é antecipado, ele é feito antes do término da operação na máquina anterior (M_1) e da chegada da operação à máquina M_2.

FIGURA 11 - Exemplo de *setup* não antecipado



Fonte: Elaborada pelo autor.

Quando o *setup* não pode ser antecipado, é preciso concluir a operação na máquina M_1 para então iniciar o *setup* da operação na máquina M_2.

É importante observar que podem existir em um mesmo problema todos os tipos de *setup*; isso depende da natureza da atividade de *setup* e das características de cada tarefa.

Como o *setup* é uma atividade que não agrega valor, muitas pesquisas nas últimas décadas têm se voltado para técnicas de sequenciamento que minimizam o tempo total de *setup*. Burbidge (1975), Robinson (1990) e Shingo (1996) descrevem as várias vantagens de reduzir o *setup* no processo produtivo. Allahverdi e Soroush (2008) discorrem sobre a importância de reduzir tanto o tempo como o custo de *setup*.

A produção em grandes lotes é empregada para reduzir os efeitos prejudiciais dos longos tempos de *setup*. Shingo (1996, p. 254) propõe a utilização da troca rápida de ferramenta (TRF) ou *single minute exchange of die* (SMED) para não haver a necessidade de produzir em grandes lotes. Segundo ele, os *setups* rápidos fizeram com que o cálculo de lotes econômicos ficasse sem sentido, porque não existe mais a necessidade de equilibrar os méritos da produção com lotes grandes às desvantagens do aumento de estoque.

Shingo (2000) define o conceito de "troca de ferramentas em um tempo inferior a dez minutos" como a melhoria do *setup* envolvendo três estágios: separação entre *setup* interno e externo, conversão do *setup* interno em externo e racionalização dos *setups* para que atinjam menos de dez minutos.

Vale ressaltar que a proposta de Shingo que classifica o *setup* em interno e externo não está diretamente relacionada às técnicas de programação da produção, pois foca os esforços em reduzir a duração das operações de *setup*. Os métodos de programação da produção, por outro lado, visam a minimizar diversas medidas, além do tempo total de *setup*, partindo da premissa de que os tempos de *setup* são dados de entrada fixos do problema, de que não é possível reduzi-los de forma pontual e de que sua redução não necessariamente acarretaria economias em escala global (há situações em que reduzir o tempo de *setup* requer mais operadores e equipamentos, o que acaba aumentando o custo final).

Notação de três campos

Pinedo (2012, p. 589) define problema como uma "descrição genérica" do ambiente de produção, a exemplo de um problema de programação em máquinas paralelas. Uma **instância** ou **problema-teste**, por sua vez, indica

um *problema* com seu respectivo conjunto de dados numéricos. Por exemplo, o conjunto "duas máquinas, cinco tarefas com tempos de processamento 2, 3, 5, 5, 8 e minimização do *makespan*" é uma instância do problema de programação em máquinas paralelas.

Na literatura, os problemas de programação da produção são representados pela conhecida notação de três campos: $\alpha \mid \beta \mid \gamma$. As definições básicas dessa notação podem ser encontradas em Pinedo (2012, p. 8-14). Vignier, Billaut e Proust (1999) adaptaram a notação proposta por Rinnooy Kan (1976) especificamente para *flow shop* híbridos, e Kurz (2001, p. 21-22) fez algumas adequações para o ambiente *flexible flow line*. Allahverdi, Gupta e Aldowaisan (1999) também propuseram algumas representações adicionais, e Lin e Cheng (2005) utilizaram uma notação para *setups* antecipados e não antecipados.

O campo α representa o ambiente do sistema e é composto por quatro parâmetros: $\alpha = \alpha_1 \alpha_2, (\alpha_3 \alpha_4^{(k)})_{k=1}^{\alpha_2})$. O primeiro indica a disposição das máquinas, com $\alpha_1 \in \{1,P,Q,R,J,O,F,HF,FFS\}$, em que

1: máquina única (também representada por $\alpha_1 = \varnothing$)

P: máquinas paralelas idênticas

Q: máquinas paralelas uniformes

R: máquinas paralelas não relacionadas

J: *job shop*

O: *open shop*

F: *flow shop*

FL: *flow line*

HF: *flow shop* híbrido

FFS: *flexible flow shop* ou *flexible flow line*

O segundo parâmetro α_2 informa o número de estágios quando superior a um. O par $\alpha_3 \alpha_4$ é repetido de acordo com o número de estágios, com $\alpha_3 \in \{1,P,Q,R\}$ e $\alpha_4 \in \{1, M^{(k)}, M^{(k)}(t)\}$, em que

1: uma máquina em cada estágio (também representada por $\alpha_3\alpha_4 = \varnothing$)

$M^{(k)}$: número de máquinas do estágio *k*

$M^{(k)}(t)$: número de máquinas do estágio *k* variável no tempo

Pode-se ainda utilizar a notação $(\alpha_3\alpha_4)^k$ para indicar os *k* estágios consecutivos de uma configuração com α_4 máquinas paralelas do tipo α_3.

Assim, por exemplo, para um *flow shop* híbrido com cinco estágios compostos por três máquinas idênticas nos dois primeiros e duas máquinas idênticas nos seguintes, a notação do campo α será $HFS5,(P3^{(1)},P3^{(2)},P2^{(3)},P2^{(4)},P2^{(5)})$, $HFS5,((P3^{(k)})_{k=1}^{2},(P2^{(k)})_{k=3}^{5})$ ou então $HFS5,((P3)^2,(P2)^3)$. Se os detalhes forem irrelevantes, pode-se representar α por $HFS5,((PM^{(k)})_{k=1}^{g})$.

É importante ressaltar que é muito comum utilizar apenas o parâmetro α_1 seguido do número de máquinas do problema. Por exemplo, representa-se o problema de *flow shop* por *Fm* e o de máquinas paralelas idênticas por *Pm*.

O campo β descreve as propriedades dos recursos e das tarefas, podendo ter vários componentes. Alguns dos mais comuns são:

r_j: presença de datas de liberação (*release dates* ou *ready times*)

d_j: presença de prazos de entrega (*due dates*)

s_{ijk}: presença de tempos de *setup* dependentes da sequência (no problema de máquina única, como $k = 1$, representa-se apenas s_{ij})

s_{jk}: presença de tempos de *setup* independentes da sequência (no problema de máquina única, como $k = 1$, representa-se apenas s_j)

as: tempos de *setup* antecipados

ns: tempos de *setup* não antecipados

prmp: as operações podem ser interrompidas (*preemption*)

prmu: a solução deve ser uma programação permutacional

brkdwn: as máquinas podem não estar continuamente disponíveis (*breakdown*)

split: as operações podem ser subdivididas (*splitting*)

block: bloqueio (*blocking*); indica que existe uma capacidade finita de armazenamento temporário (*buffer*) entre duas máquinas. Se o armazenamento atingir a capacidade total, a máquina que o antecede ficará bloqueada.

M_j: elegibilidade de máquina; conjunto de máquinas que podem processar a operação j

nwt: as tarefas não podem esperar entre duas máquinas sucessivas (*no-wait*)

recrc: as tarefas podem visitar uma máquina mais de uma vez (*recirculation*)

prec: presença de restrições de precedência geral; outros tipos de restrições de precedência são *chain* (cada tarefa pode ter somente uma antecessora e uma sucessora), *in tree* (cada tarefa pode ter somente uma sucessora) e *out tree* (cada tarefa pode ter somente uma antecessora).

O campo γ indica a medida de desempenho do problema. As medidas mais usuais, denominadas medidas de desempenho regulares, são:

C_{max}: duração total da programação (*makespan*)

$\overline{F}$: tempo médio de fluxo das tarefas (*flow time* médio)

F: tempo total de fluxo das tarefas (*flow time* total)

F_w: tempo de fluxo ponderado das tarefas (com pesos w_j)

L_{max}: maior desvio de pontualidade das tarefas (*lateness*)

T_{max}: maior atraso das tarefas (*maximum tardiness*)

T: atraso total das tarefas (*total tardiness*)

T_w: atraso ponderado das tarefas (com pesos w_j)

E_{max}: maior adiantamento das tarefas (*earliness*)

n_T (ou $\sum U_j$): número de tarefas em atraso (*tardy jobs*)

I: tempo ocioso total das máquinas (*idle time*)

Essa notação evolui constantemente, pode acomodar quaisquer elementos desejados e exibir variação dependendo do autor.

Como exemplos da notação de três campos, podem-se citar: $Fm|s_{ijk}|C_{\max}$, para o problema de *flow shop* com m máquinas, tempos de *setup* dependentes da sequência e minimização do *makespan*; $Pm|r_j|\ \overline{F}$, para máquinas paralelas idênticas com datas de liberação das tarefas e minimização do tempo médio de fluxo; $1|d_j|T$, para o problema de máquina única com datas de entrega e minimização da soma dos atrasos.

3 Sistema *flow shop*

Este capítulo apresenta as especificidades e restrições dos ambientes *flow shop*, bem como as empresas típicas onde esse sistema se configura. São enunciados os pressupostos gerais e aqueles referentes às tarefas, às máquinas e às políticas operacionais. Além disso, são demonstradas propriedades estruturais do problema, algumas formas de representação e as principais medidas de desempenho utilizadas na prática. Constam também a complexidade computacional dos problemas de *flow shop* e alguns casos especiais.

Características do ambiente de produção

O problema de programação em *flow shop* tem sido extensivamente estudado por quase sessenta anos, desde o trabalho pioneiro de Johnson (1954). A razão é que esse ambiente é encontrado em um grande número de aplicações industriais. Bagchi, Gupta e Sriskandarajah (2006) afirmam que cerca de um quarto das empresas de manufatura, montagem, serviços ou processamento de informações pode ser modelado como *flow shop*.

O *flow shop* é um caso de sistema de produção intermitente caracterizado por um fluxo unidirecional de tarefas, realizadas por múltiplas máquinas

dispostas em série. Ou seja, todas as tarefas "fluem" na mesma ordem de processamento nas máquinas.

Em 2006, o *European Journal of Operational Research* lançou uma edição especial em comemoração aos cinquenta anos de publicação do primeiro artigo sobre programação em *flow shop* (o de Johnson, de 1954). No artigo editorial dessa edição, Gupta e Stafford Jr. (2006) mostraram a evolução dos problemas e métodos de solução para problemas de *flow shop* nas últimas cinco décadas.

O estado da arte para os respectivos momentos de *flow shop* com minimização do *makespan* foi apresentado por Framinan, Gupta e Leisten (2004), Reza Hejazi e Saghafian (2005) e Ruiz e Maroto (2005). Para o problema de *flow shop* permutacional com minimização do tempo médio de fluxo, extensas revisões da literatura e comparações entre os métodos existentes foram conduzidas por Framinan, Leisten e Ruiz-Usano (2005) e Pan e Ruiz (2013). Para o problema de minimização do atraso total, uma revisão e avaliação foram propostas por Vallada, Ruiz e Minella (2008).

Também foram feitas revisões e classificações específicas para os problemas de *flow shop* multiobjetivo, como as de Sun et al. (2011) e Yenisey e Yagmahan (2014). Cheng, Gupta e Wang (2000) realizaram uma revisão da literatura dos problemas de programação em *flow shop* estáticos e determinísticos com tempos de *setup* explícitos.

Bagchi, Gupta e Sriskandarajah (2006) apresentaram uma revisão dos trabalhos em que o *flow shop* é modelado como o conhecido Problema do Caixeiro Viajante (*traveling salesman problem* – TSP). Essa técnica pode ser adotada até mesmo em problemas em que não há *setup* dependente da sequência, considerando-se "distâncias entre duas cidades", por exemplo, como o intervalo entre o instante de início de uma tarefa em uma máquina e o instante de início da tarefa imediatamente seguinte na mesma máquina. Os autores também mostraram que essa abordagem é eficaz na programação de *flow shop* com restrições *no-wait* (em que não é permitida espera entre as operações de cada tarefa) e *blocking* (quando não pode haver estoques intermediários entre as máquinas) e na programação de movimentos de robôs em células de produção automatizadas.

Genericamente, o *flow shop* contém m máquinas diferentes e cada tarefa consiste de m operações, cada uma requerendo uma máquina específica. Em outras palavras, é um ambiente que possui uma ordem natural de máquinas: é possível numerar as máquinas de forma que se a i-ésima operação de qualquer tarefa precede sua j-ésima operação, então a máquina requerida pela i-ésima operação tem um número menor do que a máquina requerida pela j-ésima operação. As máquinas são numeradas por 1, 2, ..., m, e as operações da tarefa J_j são correspondentemente denominadas o_{j1}, o_{j2}, ..., o_{jm}.

Nos casos em que as tarefas possuem um número de operações menor que m, o tempo de processamento correspondente às operações inexistentes pode ser considerado igual a zero. Isso significa que uma tarefa tem a possibilidade de saltar uma ou mais máquinas.

O problema de programação pode ser descrito desta forma: dadas n tarefas a serem processadas em m máquinas na mesma ordem tecnológica e seus respectivos tempos de processamento p_{jk} da tarefa J_j na máquina M_k (j = 1, 2, ..., n; k = 1, 2, ..., m), deseja-se encontrar uma programação dessas tarefas nas máquinas tal que minimize uma medida de desempenho da produção.

O problema de programação em *flow shop* tem os seguintes pressupostos gerais:

- Um conjunto de n tarefas com múltiplas operações está disponível para processamento na data zero.
- Os tempos de preparação (*setup*) para as operações são independentes da sequência de tarefas e estão incluídos nos tempos de processamento.
- As informações das tarefas são conhecidas previamente.
- Existem m máquinas diferentes continuamente disponíveis para processamento.
- O processamento das operações não pode ser interrompido.

Evidentemente, tais suposições são mais gerais que uma linha de montagem real e muito restritivas. Em virtude de limitações ainda existentes na

teoria de programação da produção, várias outras hipóteses simplificadoras devem ser consideradas. Embora elas possam se distanciar das situações reais, aumentar a generalidade dos modelos pode conduzir à solução ótima. Esses pressupostos adicionais incluem características das tarefas, das máquinas e das políticas operacionais da fábrica (Gupta; Stafford Jr., 2006).

Seguem os pressupostos sobre as tarefas:

- Todas as tarefas estão liberadas para processamento no início da programação.
- Cada tarefa tem sua própria data de entrega (prazo) que é conhecida e não sujeita à mudança.
- As tarefas são independentes.
- Cada tarefa consiste de operações específicas, cada qual executada por uma única máquina.
- As operações de cada tarefa têm uma precedência tecnológica previamente conhecida.
- Cada tarefa requer um tempo de processamento finito e conhecido a ser executado por várias máquinas. Esse tempo inclui os tempos de transporte e de preparação (*setup*) e é independente da sequência de tarefas.
- Cada tarefa é processada por uma única máquina de cada vez.
- As tarefas (operações) podem ter que esperar entre as máquinas, e o estoque intermediário é, portanto, permitido.

Estes são os pressupostos sobre as máquinas:

- Cada estágio consiste de uma única máquina, ou seja, o sistema possui uma única máquina de cada tipo.
- As máquinas estão ociosas no início da programação.
- Cada máquina opera independentemente das demais e assim funciona segundo sua própria capacidade máxima.

- Cada máquina pode processar no máximo uma tarefa por vez.
- As máquinas estão continuamente disponíveis para processar as tarefas durante o período da programação e não há interrupções por quebras, manutenções ou qualquer outra causa.

Estes são os pressupostos sobre as políticas operacionais:

- Cada tarefa é processada o mais cedo possível. Assim, não há inserção intencional de tempo de espera de tarefa ou ociosidade de máquina.
- Cada tarefa é considerada uma entidade indivisível, embora possa ser composta por um número de unidades individuais.
- Cada tarefa, uma vez iniciada, é processada até o fim; ou seja, não é permitido o cancelamento de tarefas.
- Cada tarefa (operação), uma vez iniciada em uma máquina, é executada completamente antes que outra possa ser iniciada nessa máquina; ou seja, não é permitida a interrupção.
- Cada tarefa é processada em não mais que uma máquina.
- Cada máquina tem o espaço adequado que permite às tarefas esperar seu processamento.
- Cada máquina é completamente alocada às tarefas consideradas durante todo o período da programação; ou seja, as máquinas não podem ser usadas para qualquer outro propósito durante esse período.
- As máquinas processam as tarefas na ordem em que chegam; ou seja, não é permitido ultrapassagem.

Essa lista indica o quanto situações práticas devem ser analisadas de modo explícito. A relaxação de um ou mais desses pressupostos ocasiona outros modelos, tanto para *flow shop* como para sistemas mais gerais. Ao longo

dos anos, os pesquisadores modificaram uma ou mais dessas hipóteses para tornar os problemas mais realistas.

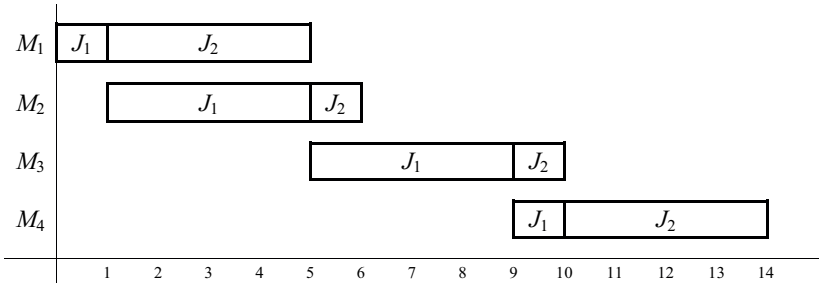
Programações permutacionais

O problema definido, quando considerado em seu caso geral, resulta em $(n!)^m$ programações possíveis. Isso porque cada máquina possui $n!$ sequências possíveis. Mesmo para problemas pequenos, com poucas tarefas e máquinas, o número de programações possíveis é tão grande que inviabiliza a enumeração direta na busca pela melhor solução.

Uma versão simplificada do problema, aplicável em muitas situações práticas, é o caso do ***flow shop* permutacional**, em que as tarefas são processadas na mesma ordem em cada uma das máquinas. Nesse caso, o número de programações possíveis se reduz a $n!$. Quando a mesma sequência de tarefas é mantida em todas as máquinas, ou seja, quando a programação é caracterizada unicamente por uma permutação de tarefas, denomina-se **programação permutacional**.

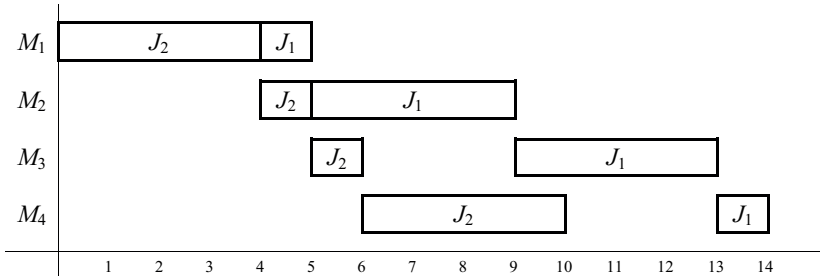
Entretanto, como pode ser visto nas Figuras 12 a 14, considerando-se, por exemplo, o *makespan* como medida de desempenho, a melhor programação pode não ser permutacional.

Figura 12 - Programação permutacional – sequência 1



Fonte: Baker (1974, p. 138).

FIGURA 13 - Programação permutacional – sequência 2

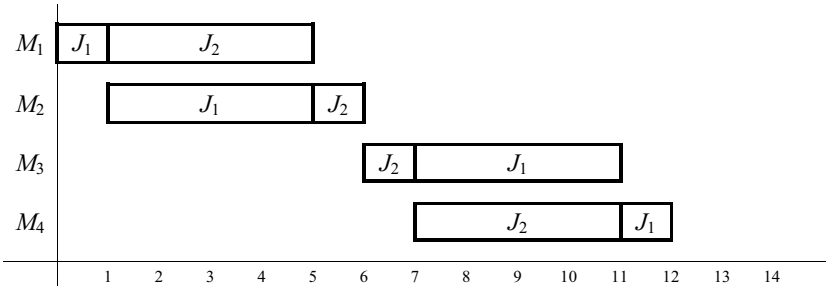


Fonte: Baker (1974, p. 138).

Como pode ser visto nas Figuras 12 e 13, ambas as programações permutacionais acarretam um *makespan* $C_{\max}$ = 14. Se for considerado também como medida de desempenho o tempo médio de fluxo, seu valor é de $\overline{F}$ =12.

Observe-se agora a programação não permutacional das mesmas tarefas na Figura 14.

FIGURA 14 - Programação não permutacional



Fonte: Baker (1974, p. 138).

Essa programação fornece tanto o *makespan* como o tempo médio de fluxo ótimos: $C^*_{\max}$ =12 e $\overline{F}^*$ =11,5, respectivamente.

Contudo, não é necessário considerar sempre as $(n!)^m$ programações para determinar a solução ótima. As duas propriedades de dominância

descritas abaixo, que constam em Baker (1974, p. 139-140) e em Gupta e Stafford Jr. (2006), proporcionam uma redução na busca pela solução do problema de *flow shop*.

Teorema 1. *Para o problema de programação em* flow shop *com* m *máquinas, todas as tarefas simultaneamente disponíveis e o objetivo de minimizar uma função não decrescente das datas de término das tarefas, é suficiente considerar somente as programações em que a mesma sequência de tarefas ocorre na primeira e na segunda máquinas.*

A implicação desse teorema é que, para o problema considerado, existe um conjunto dominante constituído por $(n!)^{m-1}$ programações.

Teorema 2. *Para o problema de programação em* flow shop *com* m *máquinas e o objetivo de minimizar o* makespan*, é suficiente considerar apenas as programações em que a mesma sequência de tarefas ocorre nas duas últimas máquinas.*

E a implicação do Teorema 2 é que, com $m > 2$, existe um conjunto dominante composto por $(n!)^{m-2}$ programações.

As demonstrações de ambos os teoremas podem ser encontradas em Baker (1974, p. 139-141).

Principais tipos de *flow shop*

Além do *flow shop* tradicional e do *flow shop* permutacional já descritos, existem algumas restrições que definem outros principais tipos de problemas:

- ***No-wait flow shop***: as tarefas devem ser processadas do início ao fim, ou seja, desde a primeira até a última máquina, sem interrupção. Exemplos: $F2|no\text{-}wait|\overline{F}$ e $Fm|no\text{-}wait|C_{\max}$.
- ***Blocking flow shop***: não há estoque intermediário entre as máquinas. A operação só pode deixar uma máquina se a seguinte estiver livre. Exemplos: $F2|block|\Sigma F$ e $Fm|block|C_{\max}$.

- ***Limited-buffer flow shop*:** existe um limite para o número de tarefas que podem permanecer no estoque intermediário (*buffer*) entre as máquinas. Exemplos: $F2|buffer|\Sigma F$ e $Fm|buffer=1|C_{max}$.

Caso particular: *flow shop* proporcional

Um importante caso especial dos problemas de máquinas em série é o chamado ***flow shop* permutacional proporcional**, representado por $Fm|prmu, p_{jk}=p_j|C_{max}$. Nesse ambiente, os tempos de processamento das tarefas em cada máquina são iguais a p_j, ou seja, $p_{j1} = p_{j2} = ... = p_{jm} = p_j$. Esse problema possui uma estrutura muito peculiar e, segundo Pinedo (2012, p. 168), acerca dele pode ser definido o seguinte teorema.

Teorema 3. *Para o problema* $Fm|prmu, p_{jk}=p_j|C_{max}$, *o* makespan *independe da programação e é igual a*

$$C_{max} = \sum_{j=1}^{n} p_j + (m-1)\max_j p_j.$$

Esse teorema mostra que o *flow shop* proporcional é semelhante ao problema de máquina única, uma vez que em ambos o *makespan* não depende da sequência. Segundo Pinedo (2012, p. 168), as afirmações a seguir mostram outras similaridades:

- A regra SPT é ótima para $1||F$ e também para $Fm \mid prmu, p_{jk} = p_j \mid F$.
- O algoritmo de Hodgson, que fornece a solução ótima para $1||n_T$, também resulta na solução ótima para $Fm|prmu, p_{jk} = p_j|n_T$.
- A regra EDD é ótima para $1||T_{max}$ e $1||L_{max}$ e também para $Fm|prmu, p_{jk} = p_j|T_{max}$ e $Fm|prmu, p_{jk} = p_j|L_{max}$.
- A propriedade de dominância que pode ser aplicada a $1||T_w$ também pode ser aplicada a $Fm|prmu, p_{jk} = p_j|T_w$.

Evidentemente, não são todos os resultados dos problemas de máquina única que podem ser aplicados ao *flow shop* proporcional. Por exemplo,

a regra WSPT não necessariamente minimiza o tempo de fluxo ponderado nos *flow shops* proporcionais.

Esse problema pode ser generalizado para incluir máquinas com diferentes velocidades. Se a velocidade da máquina M_k é v_k, então o tempo de processamento da tarefa J_j na máquina M_k é $p_{jk} = p_j/v_k$. A máquina com a menor velocidade é uma máquina-gargalo. E agora o *makespan* não é mais independente da sequência.

Teorema 4. *Em um* flow shop *proporcional com diferentes velocidades, se a primeira (última) máquina é gargalo, então a regra LPT (SPT) minimiza o* makespan.

A prova desse teorema pode ser encontrada em Pinedo (2012, p. 169).

Representação por grafo orientado

Dada uma programação permutacional de tarefas para um *flow shop* com m máquinas, o instante de término $C_{[j]k}$ da j-ésima tarefa na máquina M_k pode ser calculado facilmente por meio da equação recursiva:

$$C_{[j]k} = \max\left(C_{[j]k-1}, C_{[j-1]k}\right) + p_{[j]k} \qquad k = 1, ..., m; j = 1, ..., n. \tag{3.1}$$

O valor do *makespan* torna-se: $C_{\max} = C_{[n]m}$. (3.2)

De forma mais explícita, a data de término da primeira tarefa e a das tarefas da primeira máquina podem ser calculadas da seguinte forma, respectivamente:

$$C_{[1]k} = \sum_{u=1}^{k} p_{[1]u} \qquad k = 1, ..., m \tag{3.3}$$

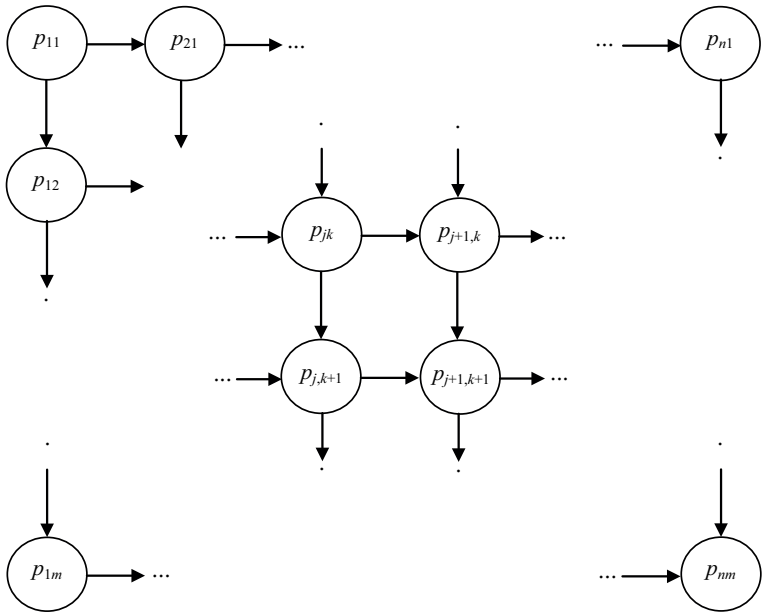
$$C_{[j]1} = \sum_{v=1}^{j} p_{[j]1} \qquad i = 1, ..., n. \tag{3.4}$$

O valor do *makespan* de dada programação permutacional pode também ser calculado determinando-se o **caminho crítico** em um grafo orientado

que corresponde à programação. O caminho crítico em um grafo é o maior caminho percorrido do nó origem ao nó destino.

Para dada sequência de tarefas, um grafo orientado é construído da seguinte forma: para cada operação, ou seja, para o processamento da tarefa $J_{[j]}$ na máquina M_k, tem-se o nó (k,j), cujo valor ou peso é igual ao tempo de processamento p_{jk}. De cada nó (k,j), $k = 1, ..., m$ e $j = 1, ..., n–1$, partem arcos que chegam aos nós $(k+1,j)$ e $(k,j+1)$. E dos nós correspondentes à máquina M_m e à tarefa $J_{[n]}$ partem apenas um arco, exceto do nó (m,n), do qual não partem arcos, como pode ser observado na Figura 15.

FIGURA 15 - Grafo orientado representando o problema de *flow shop*



Fonte: Pinedo (2012, p. 153).

O comprimento do maior caminho (caminho crítico) desde o nó (1,1) até o nó (m,n) corresponde ao *makespan*. O sequenciamento por meio do grafo consiste na seleção dos arcos de forma que o grafo resultante seja acíclico e que o comprimento do caminho crítico seja o menor possível.

Em inglês, esse grafo orientado (*directed graph*) também é chamado de *grid graph* ("grafo em grade"). Nos problemas de *job shop*, esse grafo orientado pode também ser denominado de **grafo disjuntivo** (*disjunctive graph*). Na programação do *job shop*, a construção de uma solução determinará qual operação precederá a outra em uma máquina. Em termos de rede, as possibilidades de ordenação entre os nós representam **arcos disjuntivos**, ou seja, relações de precedência que não podem ser determinadas antes da construção da programação (diferentemente dos problemas de *flow shop*, por exemplo, em que as relações de precedência entre as operações já são conhecidas previamente). Entretanto, uma vez que a sequência entre as operações é determinada para uma máquina, em vez dos arcos disjuntivos ligando os nós, são considerados os arcos de precedência orientados, ou seja, **arcos conjuntivos**, conforme o exemplo que será apresentado logo adiante.

Em outras palavras, entre as operações de um grafo que representa um problema de *job shop* pode haver arcos conjuntivos (simples, em uma única direção), representando restrições de precedência, e arcos disjuntivos (arcos duplos, em direções opostas), indicando o fato de que cada máquina só pode processar uma única operação de cada vez.

Exemplo 3.1 (Pinedo, 2012, p. 153)

Considere o seguinte exemplo de um problema de programação em um *flow shop* com cinco tarefas e quatro máquinas, com os tempos de processamento dados na Tabela 3.

Tabela 3 - Dados do Exemplo 3.1

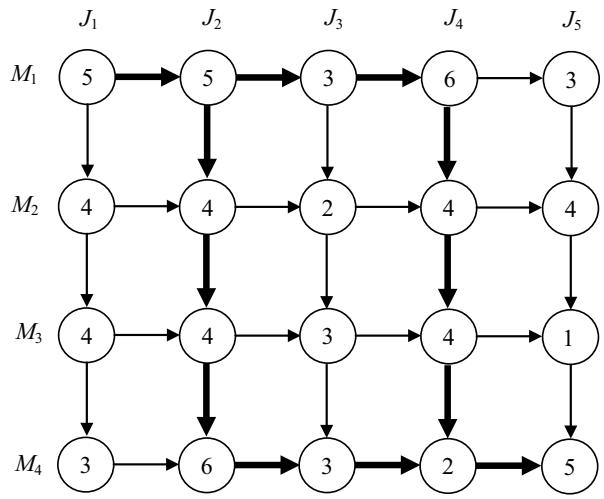
Tarefa J_j	J_1	J_2	J_3	J_4	J_5
p_{j1}	5	5	3	6	3
p_{j2}	4	4	3	4	4
p_{j3}	4	4	3	4	1
p_{j4}	3	6	3	2	5

Fonte: Elaborada pelo autor.

Diferentemente de um *job shop*, que é mais flexível por permitir rotas em ambas as direções entre dois nós, o *flow shop* impõe apenas a existência de arcos orientados (conjuntivos).

Para o problema dado, o grafo orientado correspondente à sequência $J_1 - J_2 - J_3 - J_4 - J_5$ é apresentado na Figura 16.

FIGURA 16 - Grafo orientado correspondente ao problema-exemplo



Fonte: Elaborada pelo autor.

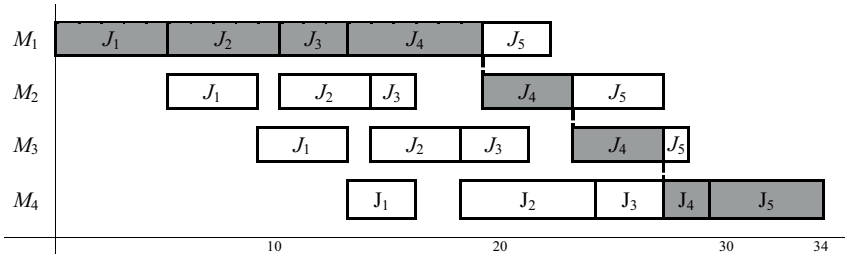
Dentre os vários caminhos possíveis do nó origem (1,1) ao nó destino (5,4), os dois maiores possuem comprimento 34, dado pela soma dos tempos de processamento que constam dos nós pertencentes ao caminho. Estes são caminhos críticos e estão destacados no grafo da Figura 16 com arcos mais largos. É comum haver mais de um caminho crítico nesse tipo de problema.

Um deles, que será aqui denominado de "caminho crítico 1", percorre os nós (1,1), (1,2), (1,3), (1,4), (2,4), (3,4), (4,4) e (4,5) e perfaz a soma dos valores 5 + 5 + 3 + 6 + 4 + 4 + 2 + 5 = 34. O segundo, chamado de "caminho crítico 2", percorre os nós (1,1), (1,2), (2,2), (3,2), (4,2), (4,3), (4,4) e (4,5) e soma os seguintes valores: 5 + 5 + 4 + 4 + 6 + 3 + 2 + 5 = 34.

Repare-se que outras possibilidades de caminhos indo da origem (1,1) ao destino (4,5) somam valores inferiores a 34 e, portanto, não constituem caminho crítico. E, como foi dito, o *makespan* é definido pelo caminho crítico.

A Figura 17 apresenta o gráfico de Gantt da sequência definida para o problema-exemplo, identificando o caminho crítico 1 descrito.

FIGURA 17 - Gráfico de Gantt do problema-exemplo – caminho crítico 1

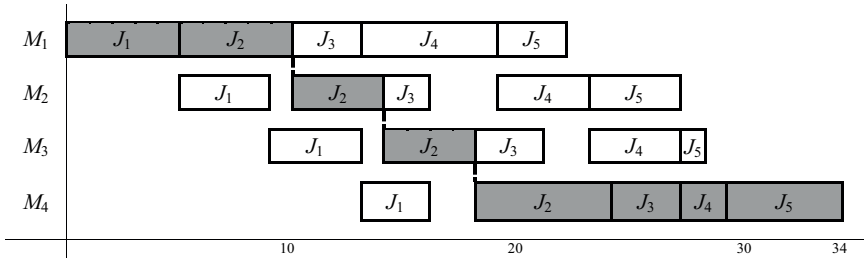


Fonte: Elaborada pelo autor.

Os subcaminhos horizontais no grafo da Figura 16 indicam "blocos de tarefas" no caminho crítico. Todas as operações que fazem parte do caminho crítico foram destacadas em sombreado no gráfico de Gantt da Figura 17.

O caminho crítico 2, também descrito anteriormente, é ilustrado no gráfico de Gantt da Figura 18, com as suas operações em destaque.

FIGURA 18 - Gráfico de Gantt do problema-exemplo – caminho crítico 2



Fonte: Elaborada pelo autor.

Limitantes inferiores

Taillard (1993) propôs limitantes inferiores (*lower bounds*) para o problema de minimização do *makespan* em *flow shop*.

Limitantes inferiores são valores calculados independentemente da sequência, de forma que se garanta que sejam menores ou iguais ao valor ótimo da função objetivo. No caso do *makespan*, o limitante inferior deve ser $LB \leq C^*_{\max}$. Isso significa que, quanto maior o seu valor, mais "exato" será em relação ao problema e melhor será como base de comparação.

Primeiramente, de acordo com o conceito *machine-based bound*, ou seja, limitante baseado em máquinas, e considerando-se a existência de uma máquina-gargalo no sistema, podem ser definidos os seguintes limitantes, por exemplo, para um problema com três máquinas:

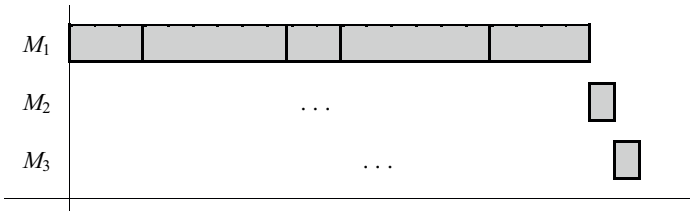
$$LB_1 = \sum_{j=1}^{n} p_{j1} + \min_j \sum_{k=2}^{3} p_{jk} \quad \text{, caso a máquina } M_1 \text{ seja o gargalo} \tag{3.5}$$

$$LB_2 = \sum_{j=1}^{n} p_{j2} + \min_j p_{j1} + \min_j p_{j3} \quad \text{, caso a máquina } M_2 \text{ seja o gargalo} \tag{3.6}$$

$$LB_3 = \sum_{j=1}^{n} p_{j3} + \min_j \sum_{k=1}^{2} p_{jk} \quad \text{, caso a máquina } M_3 \text{ seja o gargalo} \tag{3.7}$$

Nesse exemplo com três máquinas, o LB_1 considera a soma da carga da primeira máquina – que, já se sabe, não terá tempo ocioso por não haver operações precedentes – mais os tempos da menor tarefa nas máquinas seguintes, conforme ilustrado na Figura 19.

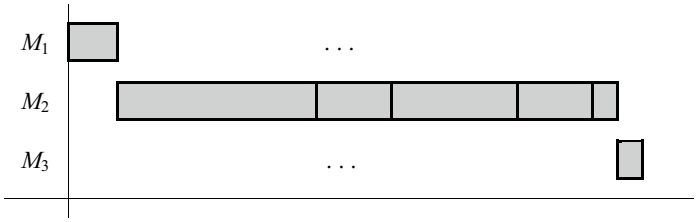
Figura 19 - Limitante inferior LB_1



Fonte: Elaborada pelo autor.

Já o LB_2 utiliza a soma da carga da segunda máquina mais o tempo da menor tarefa da máquina anterior e o tempo da menor tarefa da máquina seguinte, como mostra a Figura 20.

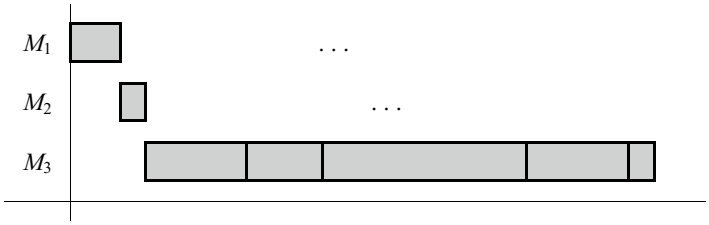
FIGURA 20 - Limitante inferior LB_2



Fonte: Elaborada pelo autor.

Finalmente, o limitante LB_3 considera a soma da carga da terceira máquina mais os tempos de processamento da menor tarefa nas máquinas anteriores, tal como apresenta a Figura 21.

FIGURA 21 - Limitante inferior LB_3



Fonte: Elaborada pelo autor.

Evidentemente, quanto maior o limitante inferior, mais acurado o seu valor. Portanto, considera-se:

$$LB_{machine} = \max\{LB_1,\ LB_2,\ LB_3\}. \tag{3.8}$$

Em problemas com qualquer quantidade de máquinas, pode-se verificar qual delas possui a maior carga de trabalho (máquina-gargalo) e considerar

o tempo mínimo necessário antes de tal máquina iniciar a sua execução (segundo somatório da expressão abaixo) e o tempo mínimo em que permanecerá inativa após o processamento do seu trabalho (terceiro somatório da expressão abaixo). Isso pode ser expresso genericamente como:

$$LB_{machine} = \max_{1 \le k \le m} \left\{ \sum_{j=1}^{n} p_{jk} + \min_{j} \sum_{q=1}^{k-1} p_{jq} + \min_{j} \sum_{q=k+1}^{m} p_{jq} \right\}. \quad (3.9)$$

A outra abordagem é o *job-based bound* ou o limitante baseado em tarefas, que considera o pior caso, isto é, a tarefa com o maior tempo total de processamento em todas as máquinas:

$$LB_{job} = \max_{j} \sum_{k=1}^{m} p_{jk}. \quad (3.10)$$

Assim, considerando-se simultaneamente as duas situações numa mesma expressão, tem-se o limitante inferior geral para o *makespan* em problema com m máquinas:

$$LB = \max \left\{ \max_{1 \le k \le m} \left\{ \sum_{j=1}^{n} p_{jk} + \min_{j} \sum_{q=1}^{k-1} p_{jq} + \min_{j} \sum_{q=k+1}^{m} p_{jq} \right\}, \max_{j} \sum_{k=1}^{m} p_{jk} \right\}. \quad (3.11)$$

As ideias desses limitantes inferiores foram utilizadas no algoritmo *branch-and-bound*, que será descrito no quarto capítulo.

Propriedades estruturais

Relações de viabilidade entre tarefas adjacentes

Para o problema de programação em *flow shop* tradicional, são conhecidas relações de viabilidade entre os tempos de máquina parada (*idle time*) e os tempos de espera entre operações sucessivas de uma mesma tarefa (*waiting time*). Se a tarefa J_u precede diretamente a tarefa J_v na mesma máquina, então

$$\text{(a)}\quad X_{uv}^{k} + p_{vk} + Y_{uv}^{k} = Y_{tu}^{k+1} + p_{u(k+1)} + X_{uv}^{k+1} \tag{3.12}$$

$$\text{(b) Se}\quad X_{uv}^{k+1} > 0 \quad \text{então}\quad Y_{uv}^{k} = 0 \tag{3.13}$$

$$\text{Se}\quad Y_{uv}^{k} > 0 \quad \text{então}\quad X_{uv}^{k+1} = 0 \tag{3.14}$$

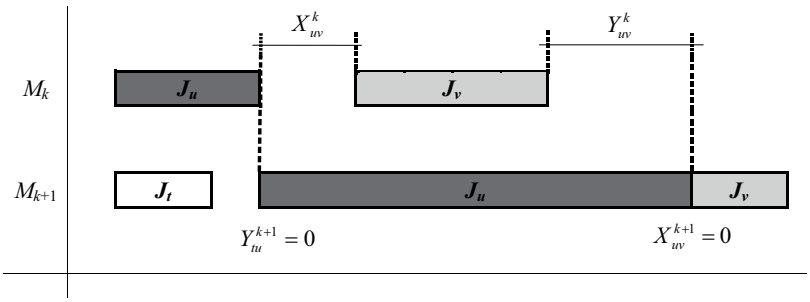
em que X_{uv}^{k}: **tempo ocioso de máquina** (*idle time*) ou intervalo de tempo entre o término da tarefa J_u e o início da tarefa J_v na máquina M_k;

Y_{uv}^{k}: **tempo de espera da tarefa** (*waiting time*) ou intervalo de tempo entre o término da tarefa J_v na máquina M_k e o seu início na máquina M_{k+1}, com a tarefa J_u precedendo diretamente a tarefa J_v.

A demonstração da relação (a) pode ser encontrada em Baker (1974, p. 170) e em Pinedo (2012, p. 157).

As relações (a) e (b) descritas são ilustradas na Figura 22.

FIGURA 22 - Relações de viabilidade para duas tarefas adjacentes no *flow shop* tradicional



Fonte: Nagano; Moccellin (2002). Adaptado.

Na Figura 22, os tempos da relação (a) nas duas máquinas consideradas estão compreendidos entre as linhas tracejadas, sendo $X_{uv}^{k} + p_{vk} + Y_{uv}^{k}$ na

máquina M_k e $Y_{tu}^{k+1} + p_{u(k+1)} + X_{uv}^{k+1}$ na máquina M_{k+1}. No caso da relação (b), esse exemplo ilustra a expressão (3.14), em que $Y_{uv}^{k} > 0$ e $X_{uv}^{k+1} = 0$.

A partir dessas relações de viabilidade entre tarefas adjacentes, Moccellin (1995) apresenta a seguinte proposição:

Limitante superior *UBX*:

Para qualquer sequência σ com as tarefas J_u *e* J_v *programadas respectivamente nas posições* j *e* j+1, *com* j = 1, ..., n-1, *tem-se o limitante superior para* X_{uv}^{k} :

$$UBX_{uv}^{k} = \max[0, UBX_{uv}^{k-1} + (p_{v(k-1)} - p_{uk})]$$

com $UBX_{uv}^{0} = 0$ *e* k = 1, ..., m.

Aplicando-se recursivamente essa relação para k = 1, ..., m, obtém-se o limitante superior UBX_{uv}^{m} para o tempo ocioso da última máquina M_m entre quaisquer pares adjacentes de tarefas J_u e J_v, independentemente das suas posições.

Sabe-se que, para uma determinada sequência de tarefas, o *makespan* pode ser expresso por:

$$C_{\max} = \sum_{j=1}^{n} p_{jm} + \sum_{j=0}^{n-1} X_{[j][j+1]}^{m} \qquad (3.15)$$

em que $X_{[j][j+1]}^{m}$ é o tempo ocioso da última máquina M_m entre o final da tarefa na posição j e o início da tarefa na posição j+1.

Se se considerar $X_{[j][j+1]}^{m}$ como as "distâncias" entre as tarefas nas posições j e j+1 da sequência, o problema de programação em *flow shop* permutacional torna-se análogo ao Problema do Caixeiro Viajante, e a sequência que minimiza o *makespan* é dada pela menor rota entre a tarefa fictícia 0 e a última $J_{[n]}$. Entretanto, evidentemente não se conhecem os valores dos tempos ociosos $X_{[j][j+1]}^{m}$ antes de se definir uma sequência específica.

Desse modo, para qualquer sequência de tarefas, é possível estabelecer o seguinte limitante superior para o *makespan*:

$$UBM = \sum_{j=1}^{n} p_{jm} + \sum_{j=0}^{n-1} UBX^{m}_{[j][j+1]} \ . \tag{3.16}$$

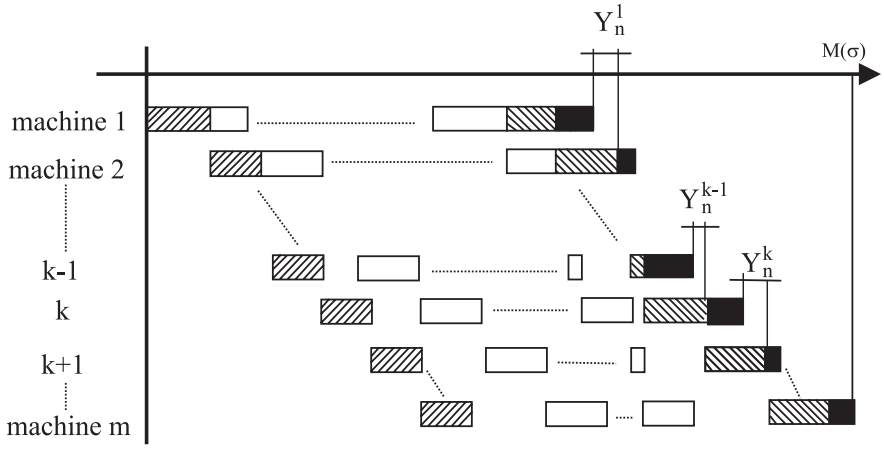
Esse problema pode ser resolvido heuristicamente pela analogia direta com o Problema do Caixeiro Viajante. Existem vários métodos de solução com essa finalidade, como o *farthest insertion travelling salesman procedure* (FITSP) e o *nearest insertion travelling salesman procedure* (NITSP).

De forma diferente, pode-se também calcular o *makespan* de uma determinada sequência por meio da expressão

$$C_{\max} = \sum_{j=1}^{n} p_{[j]1} + \sum_{k=2}^{m} p_{[n]k} + \sum_{k=1}^{m-1} Y^{k}_{[n]} \tag{3.17}$$

em que $Y^{k}_{[n]}$ é o tempo de espera da tarefa na n-ésima posição da sequência, entre o fim da operação na máquina M_k e o início da operação na máquina M_{k+1}.

FIGURA 23 - Tempo total para completar a programação



Fonte: Nagano; Moccellin (2002).

Aqui também, é claro, não se conhecem previamente os valores de $Y^k_{[n]}$ antes de se considerar uma determinada sequência. Contudo, pelas relações de viabilidade entre pares de tarefas adjacentes, pode-se calcular um limitante inferior para o tempo de espera entre essas tarefas sem precisar considerar as suas posições na sequência.

Com base nisso e no limitante superior *UBX*, Nagano e Moccellin (2002) definiram a seguinte proposição:

Limitante inferior *LBY*:

Para uma sequência arbitrária σ com as tarefas J_u *e* J_v *programadas respectivamente nas posições* j *e* j+1, *com* j = 1, ..., n-1, *tem-se o limitante inferior para* Y^k_{uv}:

$$LBY^k_{uv} = \max[0, (p_{u(k+1)} - p_{vk}) - UBX^k_{uv}] .$$

com k = 1, ..., m.

Esse limitante inferior será utilizado na heurística N&M, que será apresentada no quinto capítulo.

Medidas de desempenho

Os problemas de programação da produção sempre objetivam otimizar uma medida de desempenho. No caso do *flow shop*, a mais comum é o *makespan*.

Seja a seguinte notação:

n: número de tarefas a ser programadas;

m: número de máquinas do problema;

p_{jk}: tempo de processamento da tarefa J_j na máquina M_k;

C_{jk}: instante de término da tarefa J_j na máquina M_k;

$J_{[i]}$: tarefa que ocupa a i-ésima posição da sequência.

O *makespan* pode ser assim formulado:

$$C_{[1]1} = p_{[1]1}, \tag{3.18}$$

$$C_{[j]1} = C_{[j-1]1} + p_{[j]1}, \qquad j = 2,..., n \tag{3.19}$$

$$C_{[1]k} = C_{[1]k-1} + p_{[1]k}, \qquad k = 2,..., m \tag{3.20}$$

$$C_{[j]k} = \max\{C_{[j-1]k}, C_{[j]k-1}\} + p_{[j]k}, \qquad j = 2,..., n; k = 2, ..., m \tag{3.21}$$

A expressão (3.18) indica que a data de término da primeira tarefa na primeira máquina ($C_{[1]1}$) é dada simplesmente pelo tempo de processamento da primeira tarefa (tarefa $J_{[1]}$) na máquina M_1. Já a expressão (3.19) representa a data de término da tarefa na j-ésima posição da sequência na primeira máquina, e a expressão (3.20) denota a data de término da primeira tarefa na máquina M_k. A partir disso, pode-se generalizar a expressão (3.21), da data de término da j-ésima tarefa na máquina M_k.

Assim, o *makespan* é definido por:

$$C_{\max} = C_{[n]m}. \tag{3.22}$$

Além disso, outras medidas de desempenho também podem ser consideradas, como tempo total de fluxo (T), tempo médio de fluxo ($\overline{F}$), atraso máximo ($T_{\max}$), atraso total (T) e tempo ocioso de máquina (I).

Essas medidas são descritas a seguir, em que d_j é o prazo ou data de entrega da tarefa J_j:

$$F = \sum_{j=1}^{n} C_{jm} \tag{3.23}$$

$$\overline{F} = \frac{\sum_{j=1}^{n} C_{jm}}{n} \tag{3.24}$$

$$T_{\max} = \max\left\{\max\left\{C_{jm} - d_j, 0\right\} \mid j = 1,..., n\right\} \tag{3.25}$$

$$T = \sum_{j=1}^{n} \max\left\{C_{jm} - d_j, 0\right\} \tag{3.26}$$

$$I = \sum_{k=2}^{m} \left\{\sum_{l=1}^{k-1} C_{[1]l} + \sum_{j=2}^{n} \max\left\{C_{[j]k-1} - C_{[j-1]k}, 0\right\}\right\} \tag{3.27}$$

Complexidade

É bem conhecido o fato de que o problema de minimização do *makespan* em *flow shop*, exceto quando há apenas duas máquinas, é de otimização combinatória da classe de complexidade *NP-hard* (Garey; Johnson; Sethi, 1976), para a qual provavelmente não há algoritmos que a resolvam em tempo polinomial. Além disso, os problemas com vários critérios de otimalidade, como atraso máximo, são também *NP-hard*, exceto alguns casos especiais (Lawler et al., 1993).

Somente alguns casos particulares são resolvidos de forma eficiente (Blazewicz et al., 2001):

- O problema de programação em *flow shop* com duas máquinas é simples. Da mesma forma, o problema com três máquinas pode ser resolvido em tempo polinomial sob várias restrições de tempos de processamento da máquina intermediária.
- O algoritmo de Johnson para *flow shop* com duas máquinas (a ser detalhado no próximo capítulo) pode ser aplicado ao problema com três máquinas se a máquina intermediária não for gargalo.
- O algoritmo de Johnson também resolve o *flow shop* com duas máquinas e interrupção permitida.

4 Métodos de solução exata

A primeira publicação de que se tem notícia sobre o problema de programação em *flow shop* é a de Johnson (1954). Ela apresenta um método de solução exata para minimização do *makespan* para o problema com duas máquinas. Mais tarde, algoritmos como *branch-and-bound* e *beam search*, que também fornecem a solução exata, foram propostos, assim como os modelos de programação linear. Os métodos de solução exata ou ótima sugerem formas de enumerar o espaço de soluções para encontrar a melhor solução do problema. A limitação desses métodos está na baixa eficiência computacional (longos tempos de execução requeridos especialmente em problemas de grande porte), embora muitos problemas possam ser resolvidos por métodos exatos.

Algoritmo de Johnson

O algoritmo de Johnson (1954) fornece a solução ótima para o *flow shop* com duas máquinas e minimização do *makespan*, problema denotado por $F2||C_{\max}$.

Regra de Johnson

Em uma sequência ótima, a tarefa J_i *precede a tarefa* J_j *se* $\min\{p_{i1}, p_{j2}\} \leq \min\{p_{i2}, p_{j1}\}$.

O procedimento é apresentado na Figura 24.

FIGURA 24 - Algoritmo de Johnson

Passo 1 – Determine $\min\{p_{jk}\}$.

Passo 2 – Se $\min\{p_{jk}\}$ pertence à primeira máquina, coloque a tarefa J_j na primeira posição disponível da sequência, ou na última posição disponível.

Passo 3 – Desconsiderando a tarefa alocada, repita os Passos 1 e 2 até que todas as tarefas sejam programadas.

Fonte: Elaborada pelo autor.

De acordo com Yagmahan e Yenisey (2009), essas medidas também são minimizadas pelo algoritmo de Johnson:

- Instantes de término;
- Tempo médio de espera das tarefas;
- Tempo ocioso médio das máquinas.

Exemplo 4.1 (Baker, 1974, p. 143)

Para ilustrar o algoritmo em questão, considere-se o problema com cinco tarefas apresentado na Tabela 4.

TABELA 4 - Dados do Exemplo 4.1

Tarefa J_j	J_1	J_2	J_3	J_4	J_5
p_{j1}	3	5	1	6	7
p_{j2}	6	2	2	6	5

Fonte: Elaborada pelo autor.

A Tabela 5 mostra como a sequência ótima é construída por meio do algoritmo de Johnson em cinco iterações. A cada iteração, a tarefa com o menor tempo de processamento dentre as tarefas não programadas é identificada. Então, tal tarefa ocupa a sua respectiva posição, dependendo da máquina em que seu tempo de processamento é o menor.

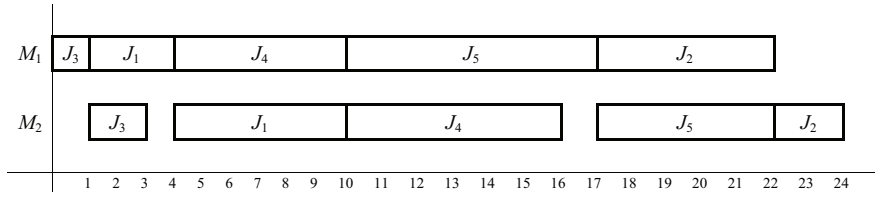
Tabela 5 - Resolução pelo algoritmo de Johnson

Iteração	Tarefas não programadas	Menor p_{jk}	Posição alocada	Sequência parcial
1	$J_1 - J_2 - J_3 - J_4 - J_5$	$p_{31} = 1$	Primeira disponível	J_3 _ _ _ _
2	$J_1 - J_2 - J_4 - J_5$	$p_{22} = 2$	Última disponível	J_3 _ _ _ J_2
3	$J_1 - J_4 - J_5$	$p_{11} = 3$	Primeira disponível	J_3 J_1 _ _ J_2
4	$J_4 - J_5$	$p_{52} = 5$	Última disponível	J_3 J_1 _ J_5 J_2
5	J_4	$p_{41} = p_{42} = 6$	Primeira/Última disponível	J_3 J_1 J_4 J_5 J_2

Fonte: Elaborada pelo autor.

A sequência final é $J_3 - J_1 - J_4 - J_5 - J_2$, cujo *makespan* ótimo é igual a 24, conforme pode ser visto na Figura 25.

Figura 25 - Programação ótima do problema-exemplo 4.1



Fonte: Elaborada pelo autor.

Extensão do algoritmo de Johnson

Não se tem notícia de métodos exatos específicos para o problema de programação em *flow shop* com três máquinas. Entretanto, a solução exata é possível em alguns casos pela extensão do algoritmo de Johnson. Esta pode ser aplicada em problemas em que a segunda máquina possui tempos de processamento uniformemente menores que os da primeira máquina (ou da terceira). Ou seja, a máquina intermediária constitui uma etapa-gargalo do sistema.

Se $\min\{p_{j1}\} \geq \max\{p_{j2}\}$ ou se $\min\{p_{j3}\} \geq \max\{p_{j2}\}$, então o problema pode ser resolvido pelo algoritmo de Johnson como se tivesse apenas duas máquinas, cujos tempos seriam assim definidos: $\widetilde{p}_{j1} = p_{j1} + p_{j2}$ e $\widetilde{p}_{j2} = p_{j2} + p_{j3}$.

Algoritmo *branch-and-bound*

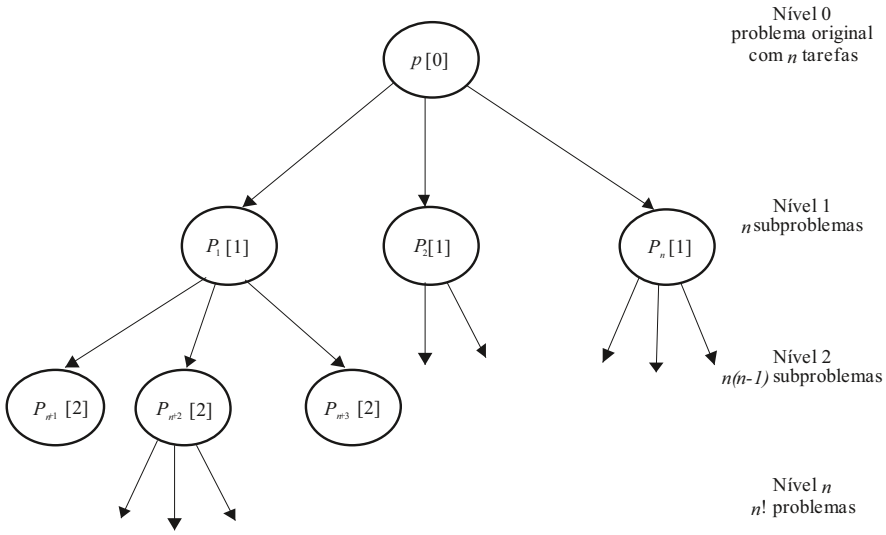
O algoritmo *branch-and-bound* genérico representa o problema a ser resolvido como uma árvore. Cada nó da árvore indica um subproblema e uma solução parcial. O termo *branching* significa "ramificar, dividir", e *bounding* significa "limitar". A ideia é dividir o problema original em subproblemas menores e menos complexos e criar mecanismos de eliminação de alguns nós da árvore e, consequentemente, seus filhos, de forma que não seja necessário percorrer todas as soluções possíveis do problema. Por esse motivo, esse algoritmo consiste num método de enumeração implícita.

O procedimento para programação do *flow shop* com m máquinas foi inicialmente proposto por Ignall e Schrage (1965) e fornece a solução ótima para o problema $Fm||C_{\max}$.

O primeiro nó (ou nó raiz) corresponde ao problema original em que há n tarefas a serem programadas. A partir desse nó, são criadas n ramificações correspondentes às n tarefas que podem ser alocadas na primeira posição da sequência. Cada um desses nós, por sua vez, possui n–1 ramificações correspondentes às n–1 tarefas disponíveis para serem designadas à segunda posição na sequência, e assim por diante.

A Figura 26 ilustra uma árvore do algoritmo *branch-and-bound*, com seus diversos níveis e subproblemas.

FIGURA 26 - Árvore do algoritmo *branch-and-bound*



Fonte: Elaborada pelo autor.

Cada subproblema $P^{[\ell]}$ do nível ℓ representa uma solução viável do problema original, com as ℓ primeiras tarefas programadas e as $(n-\ell)$ tarefas remanescentes consideradas em qualquer ordem.

Para cada nó da árvore ou subproblema é calculado um limitante inferior (*lower bound*) para o *makespan*, correspondente ao término da sequência parcial, considerando-se a carga de processamento ainda não programada em cada máquina.

Para ilustrar os limitantes, considere-se o problema com três máquinas. Seja:

σ: sequência parcial de um subproblema;

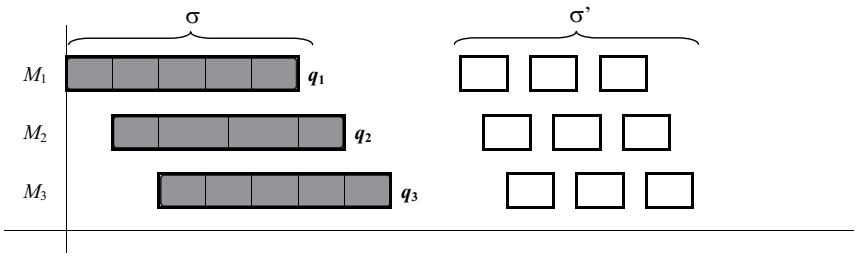
σ': conjunto de tarefas não programadas (não contidas em σ).

Para cada subproblema é definido:

q_k: carga da máquina M_k com a sequência parcial σ (com $k = 1, 2, 3$).

A Figura 27 ilustra graficamente o valor de q_k de cada máquina, as sequências parciais e as tarefas não programadas.

FIGURA 27 - Carga das máquinas



Fonte: Baker (1974, p. 150).

A carga ainda não programada da primeira máquina é $\sum_{J_j \in \sigma'} p_{j1}$. Além disso, haverá uma tarefa específica J_i que será a última na máquina M_1. Após o seu processamento na primeira máquina, a tarefa J_i será completada nas máquinas M_2 e M_3, levando pelo menos $(p_{i2}+p_{i3})$. As situações mais favoráveis que poderão ocorrer são:

- Não haver tempo ocioso na máquina M_1;
- Não haver tempo ocioso entre as operações da última tarefa J_i;
- A tarefa J_i ter a menor soma $(p_{i2}+p_{i3})$ dentre as tarefas de σ'.

Por definição, o limitante inferior é proposto sob a premissa da situação mais otimista para o processamento requerido por cada máquina. Portanto, os limitantes inferiores para as máquinas M_1, M_2 e M_3 são, respectivamente:

$$b_1 = q_1 + \sum_{J_j \in \sigma'} p_{j1} + \min_{J_j \in \sigma'} \{p_{j2} + p_{j3}\} \ , \tag{4.1}$$

$$b_2 = q_2 + \sum_{J_j \in \sigma'} p_{j2} + \min_{J_j \in \sigma'} \{p_{j3}\} \ , \tag{4.2}$$

$$b_3 = q_3 + \sum_{J_j \in \sigma'} p_{j3} \ . \tag{4.3}$$

Finalmente, para um determinado nó, o limitante inferior é definido como:

$$LB = \max\{b_1, b_2, b_3\} \cdot \tag{4.4}$$

Como já foi dito, essa expressão pode ser generalizada para qualquer problema com *m* máquinas,

$$LB = \max_{1\le k\le m}\left\{\max_{h\le k}\left[C_k(\sigma) + \min_{J_j\in\sigma'}\left\{\sum_{u=h}^{k-1} p_{ju}\right\}\right] + \sum_{J_j\in\sigma'} p_{jk} + \min_{J_j\in\sigma'}\left\{\sum_{u=k+1}^{m} p_{ju}\right\}\right\}, \tag{4.5}$$

em que $C_k(\sigma)$ é a data de término da última tarefa da sequência σ na máquina M_k.

Para reduzir o número de problemas, pode-se aplicar a seguinte propriedade de dominância, que consta em Baker (1974, p. 150), a nós do mesmo nível (mesmo número de tarefas) com sequências em ordens diferentes, ou seja, não podem ser filhos do mesmo pai.

Propriedade de dominância:

Suponha-se que as subsequências $\sigma^{(1)}$ e $\sigma^{(2)}$ *contenham as mesmas tarefas (em ordens diferentes). Se* $q_2^{(1)} \le q_2^{(2)}$ *e* $q_3^{(1)} \le q_3^{(2)}$, *então* $\boldsymbol{\sigma^{(1)}}$ ***domina*** $\boldsymbol{\sigma^{(2)}}$, *e* $\boldsymbol{\sigma^{(2)}}$ não precisa ser considerada na busca pela solução ótima.

A partir disso, são denominados **subproblemas ativos** os problemas não ramificados, ou seja, que não têm filhos e que não foram eliminados pela propriedade de dominância.

O esquema básico do algoritmo *branch-and-bound* de Ignall e Schrage (1965) é apresentado na Figura 28.

FIGURA 28 - Algoritmo *branch-and-bound*

Passo 1 – A partir do problema original, gere os n subproblemas do nível 1 e calcule os limitantes inferiores LB.

Passo 2 – Considere o subproblema ativo com o menor LB.

- Se o subproblema for do nível n, a sequência é ótima e $LB = C^*_{\max}$;
- Senão, gere os subproblemas do próximo nível e calcule os LB.

Passo 3 – Aplique a propriedade de dominância entre os subproblemas ativos, desconsiderando os dominados. Volte ao Passo 2.

Fonte: Elaborada pelo autor.

EXEMPLO 4.2 (Baker, 1974, p. 151)

TABELA 6 - Dados do Exemplo 4.2

Tarefa J_j	J_1	J_2	J_3	J_4
p_{j1}	3	11	7	10
p_{j2}	4	1	9	12
p_{j3}	10	5	13	2

Fonte: Elaborada pelo autor.

A partir do problema original, o primeiro nó gerado pelo algoritmo *branch-and-bound* corresponde ao problema P_1^1, em que a tarefa J_1 é alocada na primeira posição da sequência e $\sigma' = \{J_2, J_3, J_4\}$. Para essa sequência parcial:

$q_1 = p_{11} = 3$

$q_2 = p_{11} + p_{12} = 7$

$q_3 = p_{11} + p_{12} + p_{13} = 17.$

Os cálculos para o limitante inferior são:

$b_1 = 3 + 28 + 6 = 37$

$b_2 = 7 + 22 + 2 = 31$

$b_3 = 17 + 30 = 37$

$LB = \max\{37, 31, 37\} = 37.$

Os demais cálculos necessários à resolução do algoritmo encontram-se na Tabela 7.

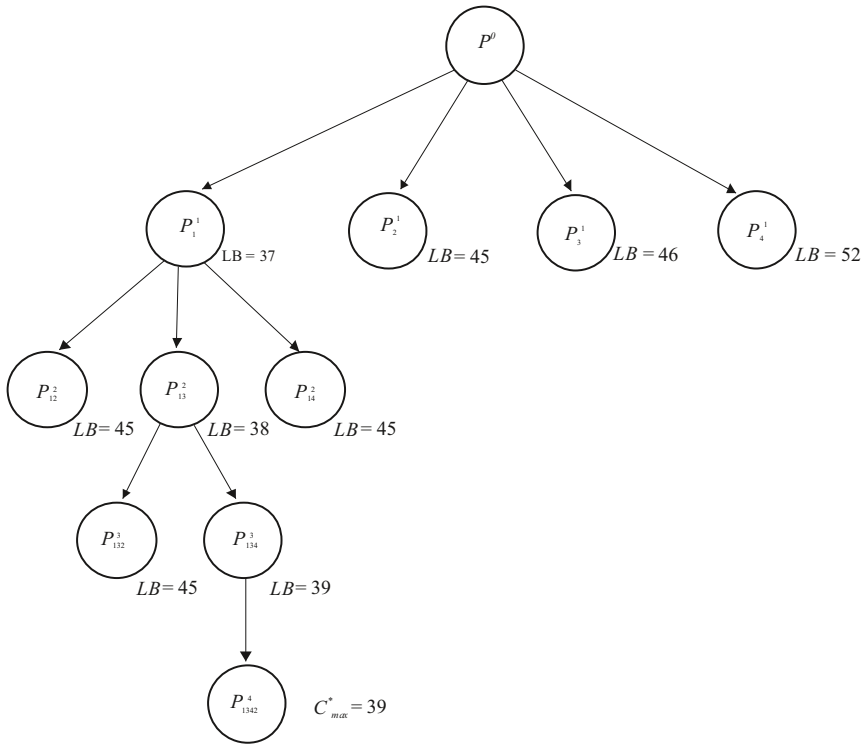
Tabela 7 - Resolução pelo algoritmo *branch-and-bound*

Subsequência s	(q_1, q_2, q_3)	(b_1, b_2, b_3)	***LB***
J_1	(3, 7, 17)	(37, 31, 37)	37
J_2	(11, 12, 17)	(45, 39, 42)	45
J_3	(7, 16, 29)	(37, 35, 46)	46
J_4	(10, 22, 24)	(37, 41, 52)	52
$J_1 - J_2$	(14, 15, 22)	(45, 38, 37)	45
$J_1 - J_3$	(10, 19, 32)	(37, 34, 39)	39
$J_1 - J_4$	(13, 25, 27)	(37, 40, 45)	45
$J_1 - J_3 - J_2$	(21, 22, 37)	(45, 36, 39)	45
$J_1 - J_3 - J_4$	(20, 32, 34)	(37, 38, 39)	39
$J_1 - J_3 - J_4 - J_2$	(31, 33, 39)	(31, 33, 39)	39

Fonte: Elaborada pelo autor.

A árvore de busca da solução desse problema é exibida na Figura 29. Conforme foi definido, a regra de ramificação da árvore consiste em criar um número de nós filhos de acordo com o número de tarefas a serem alocadas. Por exemplo, no nó raiz 0 há quatro tarefas a serem alocadas, portanto criam-se quatro nós filhos; já a partir do nó 1, existem três tarefas a serem alocadas, então criam-se três nós filhos.

FIGURA 29 - Árvore de busca do algoritmo *branch-and-bound* para o problema-exemplo



Fonte: Elaborada pelo autor.

Outros exemplos resolvidos pelo algoritmo *branch-and-bound* podem ser encontrados em Pinedo (2012).

Modelo de programação linear inteira mista

Um modelo matemático de programação linear inteira mista para o problema de programação em um *flow shop* com n tarefas e m máquinas é formulado a seguir, com base em Pinedo (2012, p. 158-159):

Variáveis de decisão:

$$X_{jt} = \begin{cases} 1, \text{ se a tarefa } J_j \text{ é programada na posição } t \text{ da sequência} \\ 0, \text{ caso contrário.} \end{cases}$$

Variáveis auxiliares:

I_{kt}: tempo ocioso na máquina M_k entre as tarefas das posições t e t+1;

W_{kt}: tempo de espera da tarefa na posição t entre as máquinas M_k e M_{k+1}.

Modelo:

$$\text{Min } Z \tag{4.6}$$

$$\text{s. a} \quad \sum_{j=1}^{n} X_{jt} = 1 , \qquad t = 1,..., n \tag{4.7}$$

$$\sum_{t=1}^{n} X_{jt} = 1 , \qquad j = 1,..., n \tag{4.8}$$

$$I_{kt} + \sum_{j=1}^{n} p_{jk} X_{j(t+1)} + W_{k(t+1)} - W_{kt} - \sum_{j=1}^{n} p_{j(k+1)} X_{jt} - I_{(k+1)t} = 0 , \quad t = 1,..., n-1;\ k = 1,..., m-1 \tag{4.9}$$

$$Z = \sum_{j=1}^{n} \sum_{k=1}^{m-1} p_{jk} X_{j1} + \sum_{t=1}^{n-1} I_{mt} , \tag{4.10}$$

$$W_{k1} = 0 , \qquad k = 1,..., m-1 \tag{4.11}$$

$$I_{1t} = 0 , \qquad t = 1,..., n-1 \tag{4.12}$$

$$X_{jt} \in \{0,1\} , \qquad j = 1,..., n;\ t = 1,..., n \tag{4.13}$$

Nesse problema, a minimização do *makespan* equivale à minimização do tempo total ocioso na última máquina M_m, que é igual a $\sum_{k=1}^{m-1} p_{[1]k} + \sum_{j=1}^{n-1} I_{m[j]}$, ou seja, o tempo ocioso que ocorre antes da primeira tarefa da sequência chegar à última máquina e a soma dos tempos ociosos entre as tarefas na última máquina. Usando-se a identidade $p_{[t]k} = \sum_{j=1}^{n} p_{jk} X_{jt}$, é possível formular o modelo. Para a variável p_{jk}, a notação $[j]$ indica a tarefa na j-ésima posição da sequência.

As restrições em (4.7) especificam que exatamente uma tarefa é associada à posição t para qualquer t. As equações em (4.8) definem que a tarefa J_j é alocada a exatamente uma única posição. O conjunto de restrições em

(4.9) relacionam as variáveis de decisão X_{kt} às restrições físicas do problema, garantindo a consistência dos tempos da programação entre todos os pares adjacentes de tarefas nas m máquinas (as relações de viabilidade entre pares adjacentes de tarefas foram apresentadas no terceiro capítulo). A equação em (4.10) determina a função objetivo, conforme descrita anteriormente. As variáveis I_{kt} e W_{kt} são contínuas e não negativas, e as variáveis de decisão X_{jt} são binárias.

As dimensões do modelo em relação ao número de variáveis, com seus respectivos tipos, e as restrições constam na Tabela 8.

Tabela 8 - Tamanho do problema em relação ao modelo

Variáveis	Binárias	n^2
	Inteiras	$2nm$
	Total	$\boldsymbol{n^2+2nm}$
Restrições	(4.7)	n
	(4.8)	n
	(4.9)	$(n-1)(m-1)$
	(4.10)	1
	(4.11)	$(m-1)$
	(4.12)	$(n-1)$
	(4.13)	n^2
	Total	$\boldsymbol{n^2+3n+3nm+m}$

Fonte: Elaborada pelo autor.

A Tabela 9 apresenta alguns exemplos de tamanhos de problema, em relação ao número de variáveis e de restrições, de acordo com o número de tarefas (n) e de máquinas (m).

Tabela 9 - Exemplos de tamanhos numéricos do problema

n	*m*	Variáveis	Restrições
5	3	55	88
5	4	65	104
10	3	160	223
10	4	180	254
10	5	200	285
10	7	240	347
20	3	520	643
20	5	600	765
20	7	680	887
50	5	3.000	3.405
50	10	3.500	4.160
100	5	11.000	11.805
100	10	12.000	13.310
100	20	14.000	16.320

Fonte: Elaborada pelo autor.

5 Métodos heurísticos

Em virtude dos altos tempos computacionais exigidos pelos métodos de solução exata, os métodos de solução heurística têm sido preferidos. Em anos recentes, várias heurísticas foram propostas para a programação do *flow shop*. Estas são processos de solução apoiados em critérios racionais para escolher um caminho entre vários possíveis, sem a exigência de se percorrer todas as possibilidades ou atingir a melhor solução. Buscam uma solução viável, pelo menos próxima da ótima, com tempo de computação aceitável. Nesse caso, a pequena diferença entre a solução heurística e a ótima não justifica o grande esforço computacional requerido.

Regras de prioridade

Dentre as heurísticas existentes para a solução de problemas de programação da produção, as mais simples são as **regras de prioridade**, também conhecidas como *regras de sequenciamento*, *regras de ordenação* ou *regras de despacho*.

Segundo Tubino (2006, p. 155), as regras de sequenciamento são heurísticas usadas para selecionar em uma fila de lotes aquele que terá prioridade

de processamento, com base nas informações dos próprios lotes ou do estado do sistema produtivo.

A necessidade de definir a sequência de tarefas é mais evidente em sistemas de produção empurrados,[2] em que se utilizam critérios predeterminados para emitir ordens de compra, fabricação e montagem dos itens. Nos sistemas puxados,[3] por sua vez, normalmente são implementados *kanbans* para gerenciar a produção.

As regras de prioridade são procedimentos de grande importância prática por serem tecnicamente simples e fáceis de compreender, além de requererem pouco esforço computacional para ser aplicadas. Geralmente a utilização de regras de prioridade é suficiente para a programação em alguns ambientes de produção, como máquina única e *flow shop* permutacional, fornecendo até a solução ótima em alguns casos. Por exemplo, a regra SPT fornece a solução ótima para a minimização do tempo médio de fluxo em problemas de máquina única e máquinas paralelas. A regra EDD produz a sequência ótima na redução do atraso máximo em máquina única. E a solução ótima da minimização do tempo de fluxo ponderado pode ser obtida pela regra WSPT.

Além disso, as regras são fáceis de codificar em linguagens de programação modernas e seus cálculos são bastante rápidos. Servem ainda como base para a concepção de métodos de programação mais complexos.

A exemplo dos estudos de Jayamohan e Rajendran (2000) e de El-Bouri, Balakrishnan e Popplewell (2008), alguns trabalhos da literatura propuseram regras de prioridade para programação de sistemas *flow shop*, além de seus respectivos critérios:

- **SPT** (*shortest processing time*): menor tempo de processamento (p_j)

---

2 Em um **sistema de produção empurrado**, cada centro de trabalho *empurra* o trabalho sem levar em consideração se o centro de trabalho seguinte terá condições de realizá-lo, podendo acarretar tempo ocioso, estoque e filas (Slack et al., 1999, p. 247).

3 Em um **sistema de produção puxado**, o ritmo e as especificações do que é feito são estabelecidos pelo centro de trabalho consumidor, que *puxa* o trabalho do centro antecedente (fornecedor). O consumidor atua como o único "gatilho" do movimento (Slack et al., 1999, p. 247).

- **LPT** (*longest processing time*): maior tempo de processamento (p_j)
- **WSPT** (*weighted shortest processing time*): menor razão entre o tempo de processamento e a prioridade ou penalidade (p_j/w_j)
- **WLPT** (*weighted longest processing time*): maior razão p_j/w_j
- **EDD** (*earliest due date*): menor data de entrega (d_j)
- **MST** (*minimum slack time*): menor folga ou maior urgência ($d_j - p_j$)
- **FIFO** (*first in first out*), **FCFS** (*first come first served*) ou **PEPS** (primeiro que entra, primeiro que sai): "fila", a primeira que chega é a primeira a ser processada, ou seja, deve-se ordená-la pela menor data de liberação (r_j)
- **LIFO** (*last in first out*): "pilha", a última que chega é a primeira a ser processada, ou seja, deve-se ordená-la pela maior data de liberação (r_j)
- **CR** (*critical ratio*): menor razão crítica, ou seja, o tempo disponível dividido pelo tempo de processamento ($[d_j - r_j]/p_j$)
- **SST** (*shortest setup time*): menor tempo de *setup* (s_j).

Heurística de Palmer

Para o problema de programação em *flow shop* com m máquinas e minimização do *makespan*, uma das primeiras heurísticas publicadas é a de Palmer (1965), também conhecida como heurística *slope index* por especificar um índice A_j de prioridade para cada tarefa J_j:

$$A_j = -\sum_{k=1}^{m} \left(m - (2k-1)\right) p_{jk} \ , \tag{5.1}$$

ou seja,

$$A_j = -(m-1)p_{j1} - (m-3)p_{j2} - \ldots + (m-5)p_{j(m-2)} + (m-3)p_{j(m-1)} + (m-1)p_{jm} \cdot \tag{5.2}$$

As tarefas são então sequenciadas pela ordem decrescente desse índice. O raciocínio subjacente a essa heurística é simples. Pelo algoritmo de Johnson, torna-se claro que as tarefas com curto tempo de processamento na primeira máquina e longo tempo de processamento na segunda devem ser posicionadas no início da programação, ao passo que as tarefas com tempo de processamento longo na primeira máquina e curto na segunda devem ser programadas por último.

O índice A_j é grande se os tempos de processamento nas últimas máquinas (a jusante) são longos em relação aos tempos de processamento nas primeiras máquinas (a montante). O índice A_j é pequeno se os tempos de processamento nas últimas máquinas são curtos em comparação aos tempos de processamento nas primeiras máquinas.

Exemplo 5.1 (Pinedo, 2012, p. 153)

Tabela 10 - Dados dos Exemplos 5.1 e 5.2

Tarefa J_j	J_1	J_2	J_3	J_4	J_5
p_{j1}	5	5	3	6	3
p_{j2}	4	4	3	4	4
p_{j3}	4	4	3	4	1
p_{j4}	3	6	3	2	5

Fonte: Elaborada pelo autor.

Os índices A_j são:

$$A_1 = -(3 \times 5) - (1 \times 4) + (1 \times 4) + (3 \times 3) = -6$$
$$A_2 = -(3 \times 5) - (1 \times 4) + (1 \times 4) + (3 \times 6) = +3$$
$$A_3 = -(3 \times 3) - (1 \times 3) + (1 \times 3) + (3 \times 3) = 0$$
$$A_4 = -(3 \times 6) - (1 \times 4) + (1 \times 4) + (3 \times 2) = -12$$
$$A_5 = -(3 \times 3) - (1 \times 4) + (1 \times 1) + (3 \times 5) = +3$$

Considerando-os na ordem decrescente, há duas sequências de tarefas: $J_2 - J_5 - J_3 - J_1 - J_4$ e $J_5 - J_2 - J_3 - J_1 - J_4$. O *makespan* de ambas as sequências é 32. A enumeração completa das soluções mostra que as duas sequências são ótimas.

Heurística de Gupta

Ainda para o problema de *flow shop* com m máquinas e minimização do *makespan*, Gupta (1971) propôs uma regra de prioridade tal como Palmer (1965), que produz boas programações. O índice de prioridade A_j para a tarefa J_j é assim definido:

$$A_j = \frac{e_j}{\min\limits_{1 \le k \le m-1} \left\{ p_{jk} + p_{j(k+1)} \right\}}, \tag{5.3}$$

em que:

$$e_j = \begin{cases} 1 \;\; se\; p_{j1} < p_{jm} \\ -1 \;\; se\; p_{j1} \ge p_{jm} \end{cases} \tag{5.4}$$

Então, a sequência é construída pela ordem decrescente do índice A_j.

Exemplo 5.2

Considere os mesmos dados da Tabela 10. Agora, para a heurística de Gupta, os índices A_j são:

$A_1 = -1 / 7 = -0{,}143$
$A_2 = 1 / 8 = 0{,}125$
$A_3 = -1 / 6 = -1{,}167$
$A_4 = -1 / 6 = -1{,}167$
$A_5 = 1 / 5 = 0{,}200$

Ao sequenciar as tarefas pela ordem decrescente de A_j, têm-se duas sequências possíveis, com diferentes *makespans*: $J_5 - J_2 - J_1 - J_3 - J_4$

(*makespan* 32); $J_5 - J_2 - J_1 - J_4 - J_3$ (*makespan* 33). Portanto, considera-se a que forneceu menor *makespan*; pela enumeração completa, verifica-se que é ótima.

Em geral, a heurística de Gupta fornece soluções melhores que a de Palmer. Uma vez que, nesse exemplo, a heurística de Palmer já apresentou a solução ótima, não foi possível encontrar melhores resultados pela de Gupta. Observe-se, entretanto, o Exemplo 5.3, que será apresentado na próxima seção e compara os resultados das heurísticas de Palmer, Gupta e CDS.

Heurística CDS

Um dos mais notórios métodos heurísticos para o problema de minimização do *makespan* ($Fm||C_{\max}$) é o algoritmo de Campbell, Dudek e Smith (1970), conhecido por heurística CDS. A sua força consiste em duas propriedades: a utilização da regra de Johnson de forma heurística; a criação de várias programações dentre as quais se escolhe a melhor.

A heurística CDS aplica iterativamente, em múltiplas etapas, a regra de Johnson a um novo problema com apenas duas máquinas, derivado do original, em que as tarefas possuem os tempos de processamento modificados p'_{j1} e p'_{j2}.

Assim, na etapa 1, $p'_{j1} = p_{j1}$ e $p'_{j2} = p_{jm}$. Em outras palavras, a regra de Johnson é aplicada à primeira e à *m*-ésima operações, e as operações intermediárias das tarefas são ignoradas.

Na etapa 2, $p'_{j1} = p_{j1} + p_{j2}$ e $p'_{j2} = p_{j(m-1)} + p_{jm}$. Isto é, a regra de Johnson é aplicada à soma dos tempos da primeira e da segunda operações e à soma dos tempos da última e da penúltima operações.

Em termos gerais, na etapa *i*, $p'_{j1} = \sum_{k=1}^{i} p_{jk}$ e $p'_{j2} = \sum_{k=1}^{i} p_{j(m-k+1)}$. Para cada etapa *i*, *i* = 1, 2, ..., *m*–1), a ordem obtida é usada para calcular o *makespan* do problema original. Após ***m*–1 etapas**, o melhor *makespan* e a sequência correspondente, entre as *m*–1 programações, são escolhidos. Evidentemente, pode haver sequências e/ou *makespans* idênticos.

O esquema da heurística CDS é apresentado na Figura 30.

Figura 30 - Algoritmo CDS

Passo 1 – Para cada etapa i, de 1 a m–1:

Considere o problema original como um problema fictício de duas máquinas com os tempos de processamento modificados:

$$p'_{j1} = \sum_{k=1}^{i} p_{jk} \quad \text{e} \quad p'_{j2} = \sum_{k=m-i+1}^{m} p_{jk} \ ;$$

Resolva o problema fictício com base no algoritmo de Johnson;

Com a sequência obtida, calcule o *makespan* do problema original.

Passo 2 – Das m–1 etapas, considere como solução final a sequência que forneceu o melhor *makespan*.

Fonte: Elaborada pelo autor.

Campbell, Dudek e Smith compararam o desempenho de seu algoritmo com a heurística *slope index* de Palmer e verificaram que a heurística CDS é mais eficaz, em problemas tanto de pequeno porte quanto de grande porte.

Exemplo 5.3 (Baker, 1974, p. 165)

Para ilustrar o funcionamento da heurística CDS e ao mesmo tempo comparar seu resultado com as heurísticas de Palmer e de Gupta, apresentadas nas seções anteriores, será resolvido o problema da minimização do *makespan* em *flow shop* com três máquinas e cinco tarefas, cujos dados são apresentados na Tabela 11.

Tabela 11 - Dados do Exemplo 5.3

Tarefa J_j	J_1	J_2	J_3	J_4	J_5
p_{j1}	6	4	3	9	5
p_{j2}	8	1	9	5	6
p_{j3}	2	1	5	8	6

Fonte: Elaborada pelo autor.

Pela heurística de Palmer, os índices A_j são:

$$A_1 = -8 \; A_2 = -6 \; A_3 = 4 \qquad A_4 = -2 \; A_5 = 2$$

Assim, a sequência formada é $J_3 - J_5 - J_4 - J_2 - J_1$, com *makespan* 37.

Ao aplicar-se a heurística de Gupta, os índices A_j são:

$$A_1 = -0{,}100 \qquad A_2 = -0{,}500 \qquad A_3 = 0{,}083$$
$$A_4 = -0{,}077 \qquad A_5 = 0{,}091$$

Portanto, a sequência formada é $J_5 - J_3 - J_4 - J_1 - J_2$, com *makespan* 36.

Agora, será detalhada a resolução pela heurística CDS. Como o problema possui três máquinas, o processo de solução inclui duas etapas (m–1).

A Tabela 12 apresenta os tempos de processamento modificados, a serem usados na etapa 1 da heurística.

Tabela 12 - Resolução da heurística CDS – etapa 1

Tarefa J_j	J_1	J_2	J_3	J_4	J_5
$p'_{j1} = p_{j1}$	6	4	3	9	5
$p'_{j2} = p_{j3}$	2	1	5	8	6

Fonte: Elaborada pelo autor.

Aplicando-se o algoritmo de Johnson, obtém-se a sequência $J_3 - J_5 - J_4 - J_1 - J_2$, com *makespan* 35.

Os tempos de processamento modificados da etapa 2 da heurística estão calculados na Tabela 13.

Tabela 13 - Resolução da heurística CDS – etapa 2

Tarefa J_j	J_1	J_2	J_3	J_4	J_5
$p'_{j1} = p_{j1}+p_{j2}$	14	5	12	14	11
$p'_{j2} = p_{j2}+p_{j3}$	10	2	14	13	12

Fonte: Elaborada pelo autor.

A sequência obtida pelo algoritmo de Johnson é $J_5 - J_3 - J_4 - J_1 - J_2$, com *makespan* 36. Portanto, a melhor programação foi fornecida pela etapa 1 da heurística CDS, com a sequência $J_3 - J_5 - J_4 - J_1 - J_2$ e *makespan* 35.

Observe-se que, para o mesmo problema, a heurística de Palmer obteve um *makespan* de 37, a de Gupta, de 36, e a CDS, de 35.

Heurística NEH

O clássico algoritmo proposto por Nawaz, Enscore Jr. e Ham (1983), conhecido como heurística NEH, para o problema $Fm||C_{\max}$ é ainda hoje utilizado como base para novos métodos de solução de problemas com restrições mais realísticas. Ele pressupõe que as tarefas com maior tempo total de processamento em todas as máquinas devem ter prioridade sobre aquelas com menor tempo total de processamento.

As duas tarefas com maior tempo total de processamento são selecionadas. A melhor sequência parcial para elas é encontrada, considerando-se as duas ordenações possíveis. A posição relativa dessas duas tarefas na sequência é mantida fixa nos passos seguintes do algoritmo. Em seguida, a tarefa com o terceiro maior tempo total de processamento é selecionada e as três sequências parciais são testadas, inserindo-se a nova tarefa no início, no meio e no final da subsequência anterior. A melhor sequência parcial fixará a posição relativa dessas três tarefas para os passos seguintes. Esse processo

é repetido até que todas as tarefas sejam programadas e uma sequência completa seja encontrada.

O número de enumerações nesse algoritmo é $[n(n + 1)/2] - 1$, das quais n são sequências completas e o restante são sequências parciais. O procedimento da heurística NEH é descrito na Figura 31.

FIGURA 31 - Algoritmo NEH

Passo 1 –	Calcule para cada tarefa $P_j = \sum_{k=1}^{m} p_{jk}$.
Passo 2 –	Sequencie as tarefas em ordem decrescente de P_j.
Passo 3 –	Com as duas primeiras tarefas, encontre a melhor sequência parcial, calculando o *makespan* (parcial) para as duas possibilidades. Mantenha a posição relativa dessa melhor sequência parcial nos passos seguintes.
Passo 4 –	Insira a próxima tarefa da lista ordenada em todas as posições possíveis da sequência parcial, mantendo a posição relativa das tarefas já consideradas. Mantenha a melhor sequência parcial em relação ao *makespan*.
Passo 5 –	Se todas as tarefas já foram programadas, pare. Caso contrário, volte ao Passo 4.

Fonte: Elaborada pelo autor.

EXEMPLO 5.4 (Nawaz; Enscore Jr.; Ham, 1983)

Para ilustrar o método em questão, será apresentado a seguir o exemplo numérico de um *flow shop* com quatro tarefas e cinco máquinas, cujos respectivos tempos de processamento são dados na Tabela 14.

Tabela 14 - Dados do Exemplo 5.4

Tarefa J_j	J_1	J_2	J_3	J_4
p_{j1}	5	9	9	4
p_{j2}	9	3	4	8
p_{j3}	8	10	5	8
p_{j4}	10	1	8	7
p_{j5}	1	8	6	2

Fonte: Elaborada pelo autor.

Os tempos P_j são os seguintes:

$P_1 = 33$ $P_2 = 31$ $P_3 = 32$ $P_4 = 29$.

A sequência em ordem decrescente de P_j é: $J_1 - J_3 - J_2 - J_4$.

Considerando-se as duas sequências possíveis para as duas primeiras tarefas, têm-se as datas de término apresentadas nas Tabelas 15 e 16.

Tabela 15 - Datas de término para a sequência parcial $J_1 - J_3$

Tarefas	J_1	J_3	-	-
C_{j1}	5	14		
C_{j2}	14	18		
C_{j3}	22	27		
C_{j4}	32	40		
C_{j5}	33	**46**		

Fonte: Elaborada pelo autor.

Tabela 16 - Datas de término para a sequência parcial $J_3 - J_1$

Tarefas	J_3	J_1	-	-
C_{j1}	9	14		
C_{j2}	13	23		
C_{j3}	18	31		
C_{j4}	26	41		
C_{j5}	32	**42**		

Fonte: Elaborada pelo autor.

Portanto, a melhor sequência parcial é $J_3 - J_1$, com *makespan* 42, como pode ser visto na Tabela 16. A próxima tarefa a ser programada é J_2. Inserindo-a nas três posições possíveis, têm-se as opções de programação apresentadas nas Tabelas 17 a 19.

Tabela 17 - Datas de término para a sequência parcial $J_2 - J_3 - J_1$

Tarefas	J_2	J_3	J_1	-
C_{j1}	9	18	23	
C_{j2}	12	22	32	
C_{j3}	22	27	40	
C_{j4}	23	35	50	
C_{j5}	31	41	**51**	

Fonte: Elaborada pelo autor.

Tabela 18 - Datas de término para a sequência parcial $J_3 - J_2 - J_1$

Tarefas	J_3	J_2	J_1	-
C_{j1}	9	18	23	
C_{j2}	13	21	32	
C_{j3}	18	31	40	
C_{j4}	26	32	50	
C_{j5}	32	40	**51**	

Fonte: Elaborada pelo autor.

Tabela 19 - Datas de término para a sequência parcial $J_3 - J_1 - J_2$

Tarefas	J_3	J_1	J_2	-
C_{j1}	9	14	23	
C_{j2}	13	23	26	
C_{j3}	18	31	41	
C_{j4}	26	41	42	
C_{j5}	32	42	**50**	

Fonte: Elaborada pelo autor.

Assim, a melhor sequência parcial é $J_3 - J_1 - J_2$, com *makespan* 50, como mostra a Tabela 19. Falta apenas a tarefa J_4, a ser inserida nas quatro posições possíveis, conforme as Tabelas 20 a 23.

Tabela 20 - Datas de término para a sequência $J_4 - J_3 - J_1 - J_2$

Tarefas	J_4	J_3	J_1	J_2
C_{j1}	4	13	18	27
C_{j2}	12	17	27	30
C_{j3}	20	25	35	45
C_{j4}	27	35	45	46
C_{j5}	29	41	46	**54**

Fonte: Elaborada pelo autor.

Tabela 21 - Datas de término para a sequência $J_3 - J_4 - J_1 - J_2$

Tarefas	J_3	J_4	J_1	J_2
C_{j1}	9	13	18	27
C_{j2}	13	21	30	33
C_{j3}	18	29	38	48
C_{j4}	26	36	48	49
C_{j5}	32	38	49	**57**

Fonte: Elaborada pelo autor.

Tabela 22 - Datas de término para a sequência $J_3 - J_1 - J_4 - J_2$

Tarefas	J_3	J_1	J_4	J_2
C_{j1}	9	14	18	27
C_{j2}	13	23	31	34
C_{j3}	18	31	39	49
C_{j4}	26	41	48	50
C_{j5}	32	42	50	**58**

Fonte: Elaborada pelo autor.

Tabela 23 - Datas de término para a sequência $J_3 - J_1 - J_2 - J_4$

Tarefas	J_3	J_1	J_2	J_4
C_{j1}	9	14	23	27
C_{j2}	13	23	26	35
C_{j3}	18	31	41	49
C_{j4}	26	41	42	56
C_{j5}	32	42	50	**58**

Fonte: Elaborada pelo autor.

Analisando-se exaustivamente todas as $n!$ = 24 sequências possíveis, verifica-se que esse algoritmo forneceu a solução ótima para o problema com apenas nove enumerações: a sequência $J_4 - J_3 - J_1 - J_2$, com *makespan* igual a 54, conforme a Tabela 20.

Exemplo 5.5

A Tabela 24 reproduz os dados da Tabela 11, juntamente com a soma dos tempos de processamento de cada tarefa P_j, para que seja aplicado o algoritmo NEH.

Tabela 24 - Dados do exemplo comparativo

Tarefa J_j	J_1	J_2	J_3	J_4	J_5
p_{j1}	6	4	3	9	5
p_{j2}	8	1	9	5	6
p_{j3}	2	1	5	8	6
$P_j = \sum_{k=1}^{m} p_{jk}$	**16**	**6**	**17**	**22**	**17**

Fonte: Elaborada pelo autor.

A sequência dada pela ordem decrescente de P_j é $J_4 - J_3 - J_5 - J_1 - J_2$. Após as respectivas inserções em cada iteração, o algoritmo NEH fornece a solução $J_3 - J_4 - J_5 - J_1 - J_2$, com *makespan* 34.

A Tabela 25 apresenta o resultado comparativo dos algoritmos descritos anteriormente. Como pode ser observado, o algoritmo NEH forneceu o melhor resultado dentre as heurísticas.

Tabela 25 - Resultados do exemplo comparativo

Algoritmo	Sequência	C_{max}
Palmer	$J_3 - J_5 - J_4 - J_2 - J_1$	37
Gupta	$J_5 - J_3 - J_4 - J_1 - J_2$	36
CDS	$J_3 - J_5 - J_4 - J_1 - J_2$	35
NEH	$J_3 - J_4 - J_5 - J_1 - J_2$	34

Fonte: Elaborada pelo autor.

Heurística N&M

Nagano e Moccellin (2002) propuseram a heurística construtiva N&M para o problema $Fm||C_{max}$, de modo semelhante à NEH. A única diferença entre elas reside nos passos 1 e 2, em relação à ordenação inicial das tarefas a serem programadas. A heurística N&M, apresentada na Figura 32, utiliza o limitante inferior *LBY* descrito no terceiro capítulo.

FIGURA 32 - Algoritmo N&M

Passo 1 – Calcule para cada tarefa

$$I_v = \sum_{k=1}^{m} p_{vk} - \max_{\substack{u=1,\ldots,n \\ u \neq v}} \left\{ \sum_{k=1}^{m-1} LBY_{uv}^{k} \right\}.$$

Passo 2 – Sequencie as tarefas em ordem decrescente de I_v.

Passo 3 – Com as duas primeiras tarefas, encontre a melhor sequência parcial, calculando o *makespan* para as duas possibilidades. Mantenha a posição relativa dessa melhor sequência parcial nos passos seguintes.

Passo 4 – Insira a próxima tarefa da lista ordenada em todas as posições possíveis da sequência parcial, mantendo a posição relativa das tarefas já consideradas.

Mantenha a melhor sequência parcial em relação ao *makespan*.

Passo 5 – Se todas as tarefas já foram programadas, pare.
Caso contrário, volte ao Passo 4.

Fonte: Elaborada pelo autor.

EXEMPLO 5.5 (Nagano; Moccellin, 2002)

Para ilustrar o funcionamento da heurística N&M, considere-se um *flow shop* com dez tarefas e quatro máquinas, cujos tempos de processamento são dados na Tabela 26.

TABELA 26 - Dados do Exemplo 5.5

Tarefa J_j	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9	J_{10}
p_{j1}	1	2	8	3	6	2	3	4	4	1
p_{j2}	10	3	2	6	4	7	4	3	7	5
p_{j3}	2	2	1	1	1	10	8	2	10	7
p_{j4}	7	6	4	1	7	3	4	10	8	5

Fonte: Elaborada pelo autor.

Para esse caso, tem-se um total de $n(n-1) = 90$ pares de tarefas adjacentes, para as quais deverão ser calculados os limitantes inferiores *LBY*.

Por exemplo, como há quatro máquinas para o par de tarefas $J_1 - J_2$, deve-se calcular os limitantes para $k = 1, 2, 3$ (o somatório dos *LBY* vai de $k = 1$ até $m-1$).

Para $k = 1$:

$$UBX_{12}^{1} = \max[0, UBX_{12}^{0} + (p_{20} - p_{11})] = 0\ ,$$

$$LBY_{12}^{1} = \max[0,\ (p_{12} - p_{21}) - UBX_{12}^{1}] = \max[0, (10-2) - 0] = 8\,.$$

Para $k = 2$:

$$UBX_{12}^{2} = \max[0, UBX_{12}^{1} + (p_{21} - p_{12})] = \max[0, 0 + (2-10)] = 0\,,$$

$$LBY_{12}^{2} = \max[0,\ (p_{13} - p_{22}) - UBX_{12}^{2}] = \max[0, (2-3) - 0] = 0\,.$$

Para $k = 3$:

$$UBX_{12}^{3} = \max[0, UBX_{12}^{0} + (p_{22} - p_{13})] = \max[0, 0 + (3-2)] = 1,$$

$$LBY_{12}^{3} = \max[0,\ (p_{14} - p_{23}) - UBX_{12}^{3}] = \max[0, (7-2) - 1] = 4.$$

Assim, $\sum_{k=1}^{m-1} LBY_{12}^{k} = LBY_{12}^{1} + LBY_{12}^{2} + LBY_{12}^{3} = 12\,.$

O mesmo procedimento deve ser aplicado a cada par de tarefas, obtendo-se a matriz apresentada a seguir:

$$[\sum LBY_{uv}]_{10x10} = \begin{bmatrix} - & 12 & 8 & 9 & 8 & 8 & 7 & 10 & 6 & 9 \\ 2 & - & 0 & 1 & 0 & 1 & 0 & 2 & 0 & 2 \\ 1 & 0 & - & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 5 & 4 & 0 & - & 0 & 4 & 3 & 2 & 2 & 5 \\ 3 & 5 & 1 & 2 & - & 2 & 1 & 3 & 0 & 3 \\ 7 & 13 & 9 & 10 & 9 & - & 10 & 11 & 6 & 11 \\ 3 & 9 & 5 & 6 & 5 & 3 & - & 7 & 1 & 6 \\ 2 & 8 & 4 & 5 & 4 & 1 & 0 & - & 0 & 2 \\ 12 & 18 & 14 & 15 & 14 & 8 & 10 & 16 & - & 12 \\ 4 & 10 & 6 & 7 & 6 & 3 & 5 & 8 & 1 & - \end{bmatrix}$$

Da Tabela 27 têm-se os valores de $\sum p_{vk}$:

Tabela 27 - Soma dos tempos de processamento das tarefas

Tarefa J_j	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9	J_{10}
$\sum p_{vk}$	30	13	15	11	18	22	19	19	29	18

Fonte: Elaborada pelo autor.

Finalmente, pode-se calcular os índices I_v, para v = 1, ..., 10, tal como segue:

$I_1 = 20 - 12 = 8$
$I_2 = 13 - 18 = -5$
$I_3 = 15 - 14 = 1$
$I_4 = 11 - 15 = -4$
$I_5 = 18 - 14 = 4$
$I_6 = 22 - 8 = 14$
$I_7 = 19 - 10 = 9$
$I_8 = 19 - 16 = 3$
$I_9 = 29 - 6 = 23$
$I_{10} = 18 - 12 = 6.$

Assim, chega-se à ordenação inicial de tarefas que a heurística propõe: $J_9 - J_6 - J_7 - J_1 - J_{10} - J_5 - J_8 - J_3 - J_4 - J_2$. Após aplicar os passos 3 a 5 da heurística, o resultado final é $J_2 - J_8 - J_{10} - J_7 - J_5 - J_9 - J_3 - J_1 - J_6 - J_4$, com *makespan* igual a 63.

Para efeito de comparação, ao se resolver o mesmo problema com o algoritmo NEH, a sequência obtida é $J_2 - J_{10} - J_1 - J_3 - J_5 - J_7 - J_8 - J_9 - J_4 - J_6$, com *makespan* 66. E o limitante inferior proposto por Taillard (1993), apresentado na seção "Limitantes inferiores", é igual a 62.

Avaliação de desempenho das heurísticas

Diferença entre eficácia, eficiência e efetividade

- **Eficácia:** refere-se à qualidade da solução (se é ótima, próxima da ótima etc.), ao objetivo, ao fim.
- **Eficiência:** alude ao desempenho computacional do método (se este fornece a solução em tempo de computação aceitável ou não), ao meio.
- **Efetividade:** concerne a utilidade do método e sua aplicação prática, bem como a finalidade.

As principais medidas para avaliar o desempenho das heurísticas referem-se à qualidade da solução e ao tempo computacional:

- **Porcentagem de sucesso:** número de vezes que o método forneceu a melhor solução dividido pelo número de problemas analisados.
- **Desvio relativo percentual (*relative percent deviation* – RPD):** variação percentual correspondente a um valor de referência Z_{ref} (que pode ser a melhor solução obtida entre os métodos, a solução ótima ou um limitante inferior). Se o desvio de um método é zero, isso significa que ele forneceu a melhor solução dentre o conjunto de métodos.

$$RPD = \left(\frac{Z_{heurística} - Z_{ref}}{Z_{ref}} \right).100 \quad (5.5),$$

em que $Z_{heurística}$ é a função objetivo fornecida pela heurística e Z_{ref} é a melhor solução obtida pelos métodos, a solução ótima fornecida por um método exato ou um limitante inferior da solução.

- **Índice de qualidade (percentual):** mede o oposto do desvio relativo percentual.

$$IQ(\%) = \left(1 - \frac{Z_{heurística} - Z_{ref}}{Z_{ref}} \right).100 \quad (5.6)$$

- **Tempo médio de computação:** soma dos tempos de execução do método para todos os problemas dividido pelo número de problemas resolvidos.

6 Meta-heurísticas

Um tipo de heurística bastante estudado nas últimas décadas, o que tem resultado em publicações de aplicações bem-sucedidas, é o das meta-heurísticas. São procedimentos de busca das soluções possíveis que utilizam alguma estratégia de exploração racional, geralmente incluindo algum processo aleatório. Exploram características das soluções encontradas e experimentam diferentes espaços de soluções fora da vizinhança, para escapar de possíveis ótimos locais. A maioria simula algum fenômeno encontrado na natureza, e por isso também são conhecidos como algoritmos bioinspirados. As obras recentes de Gaspar-Cunha, Takahashi e Antunes (2012) e de Lopes, Rodrigues e Steiner (2013) apresentam ferramental teórico e aplicações de meta-heurísticas em problemas de diversas áreas.

Algoritmo genético

O algoritmo genético foi originalmente desenvolvido por Holland (1976), e seu nome é uma analogia à estrutura genética de um cromossomo e à reprodução de plantas e animais. Com base em mecanismos de seleção

natural e genética, o método cria uma *população* de indivíduos $P(t)$ para uma geração t. Em um problema de otimização, cada *indivíduo* representa uma solução que deve ser codificada apropriadamente como *cromossomos*. Os indivíduos da população são avaliados e associados a um *valor de aptidão* (*fitness*). Então, a população sofre operações genéticas, denominadas *cruzamentos*, para originar novos indivíduos. Durante um número de iterações, essa população evolui até atingir algum critério de parada.

Nos problemas de sequenciamento, em geral os cromossomos são codificados como uma permutação de números inteiros que se referem aos índices das tarefas. Um exemplo de cromossomo, então, seria 4 8 3 2 5 1 7 9 6, referente à sequência de tarefas $J_4 - J_8 - J_3 - J_2 - J_5 - J_1 - J_7 - J_9 - J_6$.

A Figura 33 apresenta o esquema geral de um algoritmo genético. O *operador de seleção* escolhe os melhores indivíduos da população (pais) de acordo com a sua aptidão para se reproduzir e originar novos indivíduos (filhos). O operador de *crossover* os cria ao combinar as boas propriedades dos diferentes indivíduos. Sem a necessidade de cruzamento, o operador de mutação produz novos indivíduos modificando um único espécime, visando a aumentar a variabilidade da população.

FIGURA 33 - Algoritmo genético

Passo 1 –	Crie a população inicial *P*.
Passo 2 –	Faça a **seleção** dos pares de indivíduos. Faça o cruzamento por ***crossover*** dos indivíduos selecionados para produzir novos indivíduos. Crie novos indivíduos por **mutação**. Avalie a aptidão dos indivíduos da nova população *P'*. Atualize *P* selecionando os indivíduos que permanecerão na população.
Passo 3 –	Se o critério de parada foi atingido, pare; retorne ao indivíduo mais apto. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

A aplicação do algoritmo genético ao problema de programação em *flow shop* com minimização do *makespan* foi feita inicialmente por Reeves (1995) e Chen, Vempati e Aljaber (1995). Diversas outras pesquisas foram publicadas utilizando essa técnica de solução em problemas de *flow shop*, como Cotta e Troya (1998), Wang e Zheng (2003), Ruiz, Maroto e Alcaraz (2006), Zhang, Wang e Zheng (2006), Nagano, Ruiz e Lorena (2008), Akhshabi, Haddadnia e Akhshabi (2012), e Lin, Lee e Ho (2013).

Simulated annealing

O algoritmo *simulated annealing* genérico faz uma analogia ao fenômeno físico da recristalização de metais, particularmente do aço. O método *annealing* consiste em aquecer o sistema a ser estudado a uma temperatura efetivamente alta e, a partir desse ponto, resfriá-lo em estágios progressivos até que a energia interna seja tão baixa que não permita nenhuma alteração nas características do metal.

Proposto inicialmente por Kirkpatrick, Gelatt e Vecchi (1983), o algoritmo *simulated annealing*, também conhecido como recozimento simulado, baseia-se no procedimento de Metrópolis, aplicado ao estudo da mecânica dos sólidos por um modelo computacional.

Nessa técnica, um átomo do sistema sofre um pequeno deslocamento, escolhido aleatoriamente e representado por uma transição de energia. Se a variação energética for menor ou igual a zero, a nova configuração cristalina é aceita. Caso contrário, o movimento só será aceito mediante uma condição probabilística, ou seja, se a energia do sistema for maior que um valor aleatório; se for menor, o movimento é rejeitado e o método segue para a próxima iteração.

Segundo a analogia de Kirkpatrick, Gelatt e Vecchi (1983), substituindo-se a condição energética por uma função de custo, o procedimento de Metrópolis é perfeitamente capaz de gerar um conjunto de soluções para um problema de otimização em que a condição inicial de temperatura seria um parâmetro de controle.

O algoritmo *simulated annealing* genérico parte de uma solução inicial *S*, escolhe uma solução vizinha *S'* aplicando um operador apropriado

e compara os custos das duas soluções. Se a nova solução *S'* tiver um custo menor, então o algoritmo a aceita, substituindo *S* por *S'*. Caso contrário, aceita *S'* com a probabilidade $e^{-\Delta/T}$, em que Δ é a *diferença entre os custos* de *S* e *S'* e *T* é um parâmetro referido como *temperatura*. O papel da temperatura *T* é bastante relevante na execução do algoritmo. Inicialmente, *T* assume um valor predefinido e decresce a cada iteração de acordo com a função denominada *resfriamento*. O algoritmo termina com um critério de parada específico, que pode ser o número de iterações ou quando a temperatura atinge zero ou valores próximos de zero.

O esquema geral do *simulated annealing* é apresentado na Figura 34.

FIGURA 34 - *Simulated annealing*

Passo 1 –	Gere uma solução inicial *S*. Defina uma temperatura inicial T_i e um fator de resfriamento.
Passo 2 –	Gere uma solução candidata *S'* a partir da solução atual *S*. Se $\Delta < 0$, então faça $S = S'$. Se $\Delta \geq 0$, então faça $S = S'$ com a probabilidade $e^{-\Delta/T}$.
Passo 3 –	Atualize a temperatura de acordo com o fator de resfriamento. Se o critério de parada foi atingido, pare; retorne à solução *S*. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

O desempenho do *simulated annealing* também é influenciado por fatores como critério de parada, escolha do espaço de soluções factíveis, função objetivo e estrutura da escolha na vizinhança (esquema de perturbação).

Diante de resultados bem-sucedidos em uma ampla variedade de problemas, diversos pesquisadores têm recentemente utilizado o algoritmo *simulated annealing* como método de solução para programação da produção, desde o trabalho pioneiro de Osman e Potts (1989).

Vários pesquisadores aplicaram o algoritmo ao problema de *flow shop*, como Ogbu e Smith (1991), Ishibuchi, Misaki e Tanaka (1995), Low, Yeh e Huang (2004), Nearchou (2004a, 2004b), Manjeshwar, Damodaran e Srihari (2009), Naderi, Tavakkoli-Moghaddam e Khalili (2010), Bank et al. (2012) e Jarosław, Czesław e Dominik (2013).

Busca tabu

O algoritmo busca tabu foi proposto por Glover (1989, 1990) e depende dos seguintes parâmetros: solução inicial, movimentos, vizinhança (de soluções), estratégia de busca, lista tabu, critérios de aspiração e de parada. A ideia básica desse método consiste em partir de uma solução inicial e então fazer movimentos sucessivos na vizinhança de soluções, ou seja, operações de busca local. A cada iteração é feito um movimento para a melhor solução vizinha, que pode mesmo não ser melhor que a solução atual.

A estrutura do algoritmo busca tabu é mostrada na Figura 35.

Figura 35 - Busca tabu

Passo 1 –	Gere uma solução inicial S. Faça $S^* = S$.
Passo 2 –	Gere um conjunto de soluções candidatas V, encontrando soluções a partir da solução atual S e que não estejam na lista tabu nem satisfaçam o critério de aspiração.
Passo 3 –	Encontre a melhor solução candidata S de V. Se a solução encontrada for melhor do que S^*, então faça $S^* = S$. Atualize a lista tabu e o critério de aspiração.
Passo 4 –	Se o critério de parada foi atingido, pare; retorne à solução S^*. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

As listas tabus são usadas para evitar ciclos quando o movimento de um ótimo local é feito para uma solução que não melhora. A cada movimento do busca tabu, uma sequência da vizinhança é adicionada à lista tabu, que não poderá ser escolhida por um número sucessivo de iterações. Esse número é o tamanho da lista tabu, que pode ser fixo ou variável. Uma solução candidata *S'* é aceita somente se não estiver na lista tabu ou se um critério de aspiração for satisfeito. Esse critério pode permitir um "movimento tabu" quando a vizinhança tem um valor da função objetivo melhor que o encontrado até então. A busca termina quando algum critério de parada é satisfeito.

Muitos métodos de busca tabu foram propostos para o problema de programação em *flow shop*, a exemplo dos de Taillard (1990), Nowicki e Smutnicki (1996), Ben-Daya e Al-Fawzan (1998), Grabowski e Pempera (2001), Grabowski e Wodecki (2004), Ekşioğlu, Ekşioğlu e Jain (2008), Wang, Li e Wang (2010), Liao e Huang (2011) e Arabameri e Salmasi (2013).

Colônia de formigas

A ideia principal do algoritmo colônia de formigas (*ant colony*), conhecido também por *ant colony optimization* (ACO) ou otimização por colônia de formigas, baseia-se no comportamento de formigas reais que usam trilhas de feromônio para comunicação e cooperação. O caminho feito pela formiga do ninho até a fonte de alimento é uma solução do problema.

Inicialmente a formiga escolhe de forma aleatória qualquer um dos caminhos disponíveis. Quando chega ao objetivo, uma função previamente definida analisa o caminho percorrido e verifica a qualidade da solução gerada. É depositada então uma quantidade de feromônio, que busca privilegiar as melhores soluções. Existe também um decaimento da quantidade de feromônio com o passar do tempo, de forma que os melhores caminhos sejam escolhidos com mais frequência que os demais.

O primeiro exemplo de algoritmo colônia de formigas foi proposto por Dorigo, Maniezzo e Colorni (1991a, 1991b) para o Problema do Caixeiro

Viajante. Estudos posteriores tentaram melhorar o seu desempenho e, consequentemente, vários algoritmos foram propostos. Essas extensões incluem *ant colony system*, *Ant-Q*, *max-min ant system* (MMAS) e *rank-based ant system*.

A estrutura do algoritmo colônia de formigas é apresentada na Figura 36.

FIGURA 36 - Colônia de formigas

Passo 1 –	Gere os caminhos possíveis. Defina a posição inicial das formigas.
Passo 2 –	Construa uma solução completa para cada formiga. Aplique o processo de busca local. Atualize a trilha de feromônio da melhor solução.
Passo 3 –	Se o critério de parada foi atingido, pare; retorne à melhor solução. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

Primeiramente são criados os caminhos possíveis ou trilhas e definidas as posições iniciais de cada formiga, que podem estar todas em um mesmo nó ou dispostas de forma aleatória. Em seguida, cada formiga se move de acordo com uma regra de transição, que contém um componente aleatório, até que se tenha uma solução completa. Uma solução é a sequência de movimentos de uma formiga. Após cada movimentação, os feromônios são atualizados. Depois de todas as formigas terem construído uma solução, é executado um procedimento de busca local que tenta melhorar a solução criada pelas formigas e é feita a atualização da trilha de feromônio da melhor solução. Assim, todas as formigas focarão a melhor solução ao longo do tempo. Esse procedimento é repetido até que o critério de parada seja atingido, que pode ser um número de iterações ou o tempo computacional.

Recentemente, alguns trabalhos foram feitos usando colônia de formigas para programação de *flow shop*, como Stützle (1998), Rajendran e

Ziegler (2004), e Ying e Liao (2004). Mais problemas e pesquisas podem ser encontrados em Dorigo e Stützle (2004) e em Dorigo (2012).

Otimização por nuvem de partículas

O algoritmo de otimização por nuvem de partículas (*particle swarm optimization* – PSO) é uma técnica evolucionária desenvolvida por James Kennedy, um psicólogo social, e Russel Eberhart, um engenheiro eletricista, inspirada na simulação de um sistema social simplificado. A intenção original era simular graficamente o comportamento de um bando de pássaros em voo com seu movimento localmente aleatório, mas globalmente determinado (Kennedy; Eberhart, 1995).

O procedimento é uma abstração desse processo natural, em que a procura pela posição mais apta é a busca pela solução ótima de um problema. O conjunto de posições possíveis das partículas constitui o espaço de busca do problema, e cada posição ocupada por uma partícula é uma solução viável para o problema. O comportamento de cada partícula é baseado na sua experiência anterior e na das outras partículas com as quais se relaciona. De forma semelhante aos algoritmos genéticos, o conjunto das partículas tende a preservar as posições com maior aptidão e descartar as piores.

O algoritmo é iniciado com uma população de partículas, ou seja, soluções aleatórias, cada uma com sua respectiva velocidade, também aleatória. Essa população pode ser também denominada "nuvem" ou "enxame". Cada partícula mantém o rastro de suas coordenadas no espaço de busca, associadas com a melhor solução que tenha alcançado. O valor da solução também é armazenado e chamado de *pbest*. O valor da melhor partícula vizinha é armazenado como *lbest*. Também é rastreado o melhor valor já obtido por qualquer partícula da população, denominado *gbest*. O princípio desse método consiste, a cada iteração e mudança de velocidade, em que as partículas voem em direção às posições de *lbest* e *gbest*.

Diferentemente do algoritmo genético, que utiliza operadores genéticos, nesse algoritmo cada partícula ajusta individualmente o seu voo de

acordo com a sua própria experiência e a das outras partículas. A partir disso, utiliza procedimentos determinísticos de busca local.

Cada movimento é baseado em três parâmetros: fator de sociabilidade, fator de individualidade e velocidade máxima. O algoritmo combina-os com um número gerado aleatoriamente para determinar o próximo local da partícula. O fator de sociabilidade determina a atração das partículas para a melhor posição já encontrada por qualquer elemento da população; o fator de individualidade define a atração da partícula com sua melhor posição já ocupada; a velocidade máxima delimita o movimento, uma vez que é direcional e determinado.

Além desses três fatores, há também o número de partículas da população e o critério de parada. Cada partícula é tratada como um ponto em um espaço multidimensional. A posição da *i*-ésima partícula é representada por $X_i = (x_{i1}, x_{i2}, ..., x_{in})$. A melhor posição anterior da *i*-ésima partícula é denotada por $P_i = (p_{i1}, p_{i2}, ..., p_{in})$. O índice da melhor partícula de todas é indicado pela letra *g*. A velocidade da *i*-ésima partícula é representada por $V_i = (v_{i1}, v_{i2}, ..., v_{in})$.

A nova posição e velocidade da partícula *i* na iteração *t* são calculadas por meio das seguintes equações:

$$V_i^t = V_i^{t-1} + c_1 rand()(P_i^{t-1} - X_i^{t-1}) + c_2 rand()(P_g^{t-1} - X_i^{t-1}) \ , \qquad (6.1)$$

$$X_i^t = X_i^{t-1} + V_i^t \ , \qquad (6.2)$$

em que c_1 e c_2 são duas constantes positivas que correspondem respectivamente ao fator cognitivo e ao fator social, e *rand*( ) é uma função aleatória que gera números uniformemente distribuídos no intervalo [0,1].

O esquema do algoritmo de otimização por nuvem de partículas é apresentado na Figura 37.

FIGURA 37 - Otimização por nuvem de partículas

Passo 1 –	Inicialize aleatoriamente as posições e as velocidades das partículas. Para cada partícula, avalie a função objetivo da sua posição atual.
Passo 2 –	Para cada partícula: Se o valor da solução for melhor do que *lbest*, atualize *lbest*. Se o valor da solução for melhor do que *gbest*, atualize *gbest*.
Passo 3 –	Para cada partícula, atualize a velocidade V_i e a posição X_i.
Passo 4 –	Se o critério de parada foi atingido, pare; retorne à melhor solução. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

Algumas pesquisas com *flow shop* utilizam otimização por nuvem de partículas, tais como Lian, Gu e Jiao (2006, 2008), Liao, Tseng e Luarn (2007), Tasgetiren et al. (2007), Jarboui et al. (2008), Zhang et al. (2008), Zhang, Ning e Ouyang (2010), Bank et al. (2012) e Arabameri e Salmasi (2013).

Scatter search

O algoritmo *scatter search* foi proposto inicialmente por Glover (1977), porém a maioria de suas aplicações é bem mais recente, dentre as quais apenas algumas são encontradas na área de programação da produção. Embora apresente similaridades com o algoritmo genético, o *scatter search* difere deste no uso de estratégias determinísticas, em vez de probabilísticas, para promover a diversificação e intensificação, e no tamanho da população, que aqui é relativamente pequeno.

O procedimento gera um conjunto de soluções iniciais. Um método de melhoria é aplicado por meio da combinação dessas soluções para criar outras novas, na tentativa de que sejam melhores. O conjunto é então atualizado. Esse conjunto de referência contém as melhores soluções já encontradas em relação à função objetivo considerada. Sucessivamente é criado

um subconjunto, por meio da combinação das soluções, que é melhorado e usado para atualizar o conjunto de referências. A busca termina quando o critério de parada é atingido.

O esquema do algoritmo *scatter search* é apresentado na Figura 38.

Figura 38 - *Scatter search*

Passo 1 –	Gere as soluções-teste iniciais. Aplique um método de melhoria para produzir soluções mais eficazes. Atualize o conjunto de referência.
Passo 2 –	Gere novos subconjuntos do conjunto de referência. Combine esses subconjuntos para obter novas soluções. Aplique um método de melhoria às novas soluções. Atualize o conjunto de referência.
Passo 3 –	Se o critério de parada foi atingido, pare; retorne à melhor solução. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

Aplicações do algoritmo *scatter search* ao problema de *flow shop* foram abordadas por Nowicki e Smutnicki (1996), Haq et al. (2007), Saravanan et al. (2008) e Naderi e Ruiz (2014).

Evolução diferencial

A evolução diferencial é um algoritmo de otimização evolucionária, apresentado inicialmente por Storn e Price (1995). No método, as soluções candidatas são representadas por cromossomos baseados em números de ponto flutuante.

Primeiramente, todos os indivíduos são inicializados aleatoriamente e avaliados. A cada geração, operadores de mutação diferencial e *crossover* são aplicados a indivíduos para gerar uma nova população. No processo de mutação, a diferença entre dois indivíduos selecionados é somada a

um terceiro indivíduo, criando assim uma "solução mutante". Essa nova solução é resultado de uma perturbação (aleatória) em algum indivíduo da população. Esse procedimento cria uma população mutante. Então, o operador de *crossover* é aplicado para combinar a solução mutante com a "solução-alvo" para criar a "solução-teste". Um operador de seleção é usado para comparar o valor da função de aptidão dessas duas soluções e determinar aquele que sobreviverá à próxima geração. O processo é executado até atingir o critério de parada.

A Figura 39 apresenta o esquema do algoritmo de evolução diferencial.

FIGURA 39 - Evolução diferencial

Passo 1 –	Crie a população inicial.
Passo 2 –	Avalie o valor da função objetivo para todos os indivíduos e encontre o melhor.
Passo 3 –	Aplique o operador de mutação diferencial. Aplique o operador de *crossover*. Faça a seleção do indivíduo sobrevivente. Atualize o melhor indivíduo da população.
Passo 4 –	Se o critério de parada foi atingido, pare; retorne à melhor solução. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

Algumas pesquisas que aplicam o algoritmo de evolução diferencial ao problema de *flow shop* são as de Tasgetiren et al. (2004), Onwubolu e Davendra (2006), Pan, Tasgetiren e Liang (2008), Qian et al. (2008), Qian et al. (2009), e Li e Yin (2013).

Algoritmo imunológico

O sistema imunológico natural é um complexo e adaptativo sistema de reconhecimento de padrão que defende o corpo de patógenos externos (bactérias ou vírus). É capaz de categorizar todas as células (ou moléculas)

dentro do corpo como pertencentes ao seu próprio tipo ou de origem externa. Possui um mecanismo complexo que faz a recombinação do gene para lidar com os antígenos invasores, produzindo anticorpos e excluindo os antígenos. O processo de infecção envolve a invasão de um patógeno e sua proliferação dentro do organismo. Os patógenos são associados a proteínas específicas (antígenos). O sistema imunológico contém células capazes de reconhecer os antígenos e matar os patógenos. Tais células, mais comumente referidas como células imunológicas ou anticorpos, são aleatoriamente distribuídas pelo sistema imunológico. Nos animais evolutivamente avançados, o sistema imunológico é capaz de aumentar a resposta à reinfecção por um patógeno encontrado previamente, ou seja, esses animais possuem uma imunidade adaptativa (Zandieh; Fatemi Ghomi; Moattar Husseini, 2006).

Esse complexo mecanismo natural de defesa serviu de inspiração para resolver problemas de otimização, dando origem ao algoritmo imunológico, também conhecido por sistema imunológico artificial (*artificial immune system* – AIS). Um esquema genérico do funcionamento do método é apresentado na Figura 40.

FIGURA 40 - Algoritmo imunológico

Passo 1 –	Crie a população inicial de anticorpos.
Passo 2 –	Avalie o valor da função objetivo para cada anticorpo. Selecione o melhor anticorpo. Avalie a afinidade entre cada anticorpo e o melhor. Reduza a probabilidade de seleção dos anticorpos com alta afinidade.
Passo 3 –	Selecione pares de anticorpos e aplique o operador de *crossover*. Aplique o operador de mutação.
Passo 4 –	Se o critério de parada foi atingido, pare; retorne à melhor solução. Se não, volte ao Passo 2.

Fonte: Elaborada pelo autor.

O método em questão simula o processo de defesa do sistema imunológico do organismo: a função objetivo é representada por "antígenos" (invasores do organismo), e as soluções viáveis são os "anticorpos". Geralmente, anticorpos iniciais são gerados aleatoriamente em um espaço de soluções viáveis. A exploração de novos anticorpos ocorre pela geração de um conjunto de soluções candidatas de forma iterativa até que um número predeterminado de gerações seja atingido. Uma função de afinidade entre os anticorpos também é considerada no algoritmo para eliminar aqueles muito similares. O anticorpo que melhor "atacar" o antígeno por meio de uma função de avaliação (*fitness*) é escolhido como solução do problema (Zandieh; Fatemi Ghomi; Moattar Husseini, 2006).

A fundamentação teórica e algumas aplicações do algoritmo imunológico podem ser encontradas em Dasgupta (1999). Mais recentemente surgiram pesquisas empregando o algoritmo em problemas de programação em *flow shop*, como as de Engin e Döyen (2007), Gao e Liu (2007), Tavakkoli-Moghaddam, Rahimi-Vahed e Mirzaei (2007), e Hsieh, You e Liou (2009).

GRASP

A meta-heurística *greedy randomized adaptative search procedure* (GRASP) é um procedimento iterativo que consiste basicamente de duas etapas: uma fase de construção e uma fase de busca local. Na primeira, uma solução viável é produzida, e, na segunda, uma vizinhança de soluções é explorada em busca de um mínimo local. A melhor solução produzida durante as iterações será o resultado obtido pelo algoritmo.

Na fase construtiva, uma solução viável é construída iterativamente, inserindo-se na solução parcial um elemento de cada vez. A cada iteração, são avaliados apenas elementos que podem ser adicionados à solução sem violar as restrições de viabilidade. Esses elementos são chamados elementos candidatos. A escolha do próximo elemento a ser adicionado à solução é determinada ordenando-se os elementos candidatos em uma lista, de acordo com uma função gulosa. Essa função mede o benefício associado à seleção de cada elemento. A heurística é adaptativa porque os benefícios

associados a cada elemento são atualizados a cada iteração da fase construtiva, para incorporar as mudanças causadas pela escolha do último elemento. A componente probabilística é caracterizada pela escolha aleatória de um dos melhores candidatos da lista, que não é necessariamente o melhor. A lista de melhores candidatos é denominada de lista restrita de candidatos (LRC). Na maior parte dos casos, é possível melhorar a solução construída, aplicando-se a ela um procedimento de busca local.

A ideia básica do algoritmo GRASP consiste em usar diferentes soluções iniciais como pontos de partida para a busca local. A Figura 41 esquematiza a heurística em questão.

Figura 41 - GRASP

Passo 1 –	Crie a lista de candidatos.
Passo 2 –	Enquanto a solução não está completa: Construa a LRC. Selecione um elemento aleatoriamente. Atualize os valores da função gulosa.
Passo 3 –	Aplique o procedimento de busca local.
Passo 4 –	Se o critério de parada foi atingido, pare; retorne à melhor solução. Se não, volte ao Passo 1.

Fonte: Elaborada pelo autor.

A meta-heurística GRASP foi aplicada em problemas de *flow shop* por Prabhaharan, Khan e Rakesh (2006), Ravetti et al. (2006), Khan, Prabhaharan e Asokan (2007), Hasanzadeh et al. (2009) e Davoudpour e Ashrafi (2009).

Outras meta-heurísticas

Além dos principais métodos descritos até aqui, existem muitas outras meta-heurísticas, e novos procedimentos puros e híbridos continuam surgindo a cada dia. Esses outros métodos são mais difíceis de ser encontrados

na resolução da programação em *flow shop*. Alguns serão descritos brevemente a seguir.

- **Busca local iterativa (*iterated local search* – ILS):** processo iterativo em que uma solução é perturbada, gerando novas soluções como partida para um método de busca local.
- **Busca em vizinhança variável (*variable neighborhood search* – VNS):** busca local com mudança dinâmica de vizinhança. A vizinhança é expandida ou novos métodos de escolha da vizinhança são utilizados à medida que não se obtêm soluções melhores na vizinhança atual.
- **Algoritmo memético (*memetic algorithm*):** versão híbrida do algoritmo genético com busca local, com o objetivo de aumentar o desempenho.
- ***Beam search*:** método de enumeração implícita, uma adaptação do *branch-and-bound*, em que somente os nós mais promissores de cada nível da árvore de busca são armazenados na memória para serem visitados, enquanto os demais nós são descartados permanentemente.
- **Colônia de abelhas (*bee colony*):** inspirado no comportamento de coleta de alimento das abelhas. Existem três tipos de abelhas: as escoteiras, que voam aleatoriamente no espaço de busca sem orientação específica; as empregadas, que exploram a vizinhança de sua localização para selecionar uma solução aleatória a ser perturbada; as espectadoras, que selecionam probabilisticamente uma solução para explorar sua vizinhança.
- **Colônia de vaga-lumes (*firefly colony*):** baseada no comportamento social de insetos lampirídeos, como vaga-lumes ou pirilampos. Os lampejos são produzidos para atrair presas, para comunicação ou acasalamento, e são associados com o valor da função objetivo a ser otimizada.

- **Algoritmo do morcego (*bat algorithm*):** inspirado no processo de ecolocalização desempenhado pelos morcegos durante o voo para detectar presas e evitar obstáculos. O morcego representa uma solução viável.
- **Busca harmônica (*harmonic search*):** fundamenta-se na observação do desempenho de músicos em uma orquestra, que buscam a harmonia perfeita. A harmonia perfeita refere-se a um padrão de qualidade de áudio e é análoga à busca pela solução ótima. Considera a capacidade de improvisação dos músicos para obtenção de novas harmonias, levando-se em conta a frequência, o timbre e a amplitude de cada instrumento.

Referências

AKHSHABI, M.; HADDADNIA, J.; AKHSHABI, M. Solving flow shop scheduling problem using a parallel genetic algorithm. *Procedia Technology*, v. 1, p. 351-355, 2012.

ALLAHVERDI, A.; GUPTA, J. N. D.; ALDOWAISAN, T. A review of scheduling research involving setup considerations. *OMEGA*: The International Journal of Management Science, v. 27, issue 2, p. 219-239, 1999.

ALLAHVERDI, A.; NG, C. T.; CHENG, T. C. E.; KOVALYOV, M. Y. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, v. 187, issue 3, p. 985-1032, 2008.

ALLAHVERDI, A.; SOROUSH, H. M. The significance of reducing setup times/ setup costs. *European Journal of Operational Research*, v. 187, issue 3, p. 978-984, 2008.

ARABAMERI, S.; SALMASI, N. Minimization of weighted earliness and tardiness for no-wait sequence-dependent setup times flowshop scheduling problem. *Computers & Industrial Engineering*, v. 64, issue 4, p. 902-916, 2013.

ARENALES, M.; ARMENTANO, V.; MORABITO, R.; YANASSE, H. H. *Pesquisa Operacional para cursos de Engenharia*. Rio de Janeiro: Campos, 2007.

BAGCHI, T. P.; GUPTA, J. N. D.; SRISKANDARAJAH, C. A review of TSP based approaches for flowshop scheduling. *European Journal of Operational Research,* v. 169, issue 3, p. 816-854, 2006.

BAKER, K. R. *Introduction to sequencing and scheduling*. New York, NY: John Wiley & Sons, 1974.

BAKER, K. R. *Elements of sequencing and scheduling*. Hanover: Amos Tuck School of Business Administration; Dartmouth College, 1992.

BAKER, K. R.; TRIETSCH, D. *Principles of sequencing and scheduling*. New Jersey, NJ: John Wiley & Sons, 2009.

BANK, M.; FATEMI GHOMI, S. M. T.; JOLAI, F.; BEHNAMIAN, J. Application of particle swarm optimization and simulated annealing algorithms in flow shop scheduling problem under linear deterioration. *Advances in Engineering Software*, v. 47, issue 1, p. 1-6, 2012.

BEN-DAYA, M.; AL-FAWZAN, M. A tabu search approach for the flow shop scheduling problem. *European Journal of Operational Research*, v. 109, issue 1, p. 88-95, 1998.

BURBIDGE, J. L. *The introduction of group technology*. London: Heinemann, 1975.

BŁAŻEWICZ, J.; ECKER, K. H.; PESCH, E.; SCHMIDT, G.; WĘGLARZ, J. *Scheduling computer and manufacturing processes*. 2nd ed. Berlin: Springer, 2001.

CAMPBELL, H. G.; DUDEK, R. A.; SMITH, M. L. A heuristic algorithm for the *n* job *m* machine sequencing problem. *Management Science*, v. 16, issue 10, p. B630-B637, 1970.

CHEN, C. L.; VEMPATI, V. S.; ALJABER, N. An application of genetic algorithms for flow shop problems. *European Journal of Operational Research*, v. 80, issue 2, p. 389-396, 1995.

CHENG, T. C. E.; GUPTA, J. N. D.; WANG, G. A review of flowshop scheduling research with setup times. *Production and Operations Management*, v. 9, issue 3, p. 262-282, 2000.

CONWAY, R. W.; MAXWELL, W. L.; MILLER, L. W. *Theory of scheduling*. Reading: Addison-Wesley, 1967.

CORRÊA, H. L.; GIANESI, I. G. N.; CAON, M. *Planejamento, programação e controle da produção*: MRP II/ERP. 5. ed. São Paulo: Atlas, 2011.

COTTA, C.; TROYA, J. M. Genetic forma recombination in permutation flowshop problems. *Evolutionary Computation*, v. 6, issue 1, p. 25-44, 1998.

DASGUPTA, D. (Ed.). *Artificial immune systems and their applications*. Berlin: Springer-Verlag, 1999.

DAVOUDPOUR, H.; ASHRAFI, M. Solving multi-objective SDST flexible flow shop using GRASP algorithm. *The International Journal of Advanced Manufacturing Technology*, v. 44, issue 7-8, p. 737-747, 2009.

DORIGO, M. *Ant colony optimization*. Iridia: Université Libre de Bruxelle, 2012. Disponível em: <http://iridia.ulb.ac.be/dorigo/ACO/publications.html>. Acesso em: 20 abr. 2013.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Positive feedback as a search strategy. *Technical Report*, n. 91-016, Politecnico di Milano, 1991a.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. The ant system: an autocatalytic optimizing process. *Technical Report*, n. 91-016, Politecnico di Milano, 1991b.

DORIGO, M.; STÜTZLE, T. *Ant colony optimization*. Cambridge: MIT Press, 2004.

EKŞIOĞLU, B.; EKŞIOĞLU, S. D.; JAIN, P. A tabu search algorithm for the flowshop scheduling problem with changing neighborhoods. *Computers & Industrial Engineering*, v. 54, issue 1, p. 1-11, 2008.

EL-BOURI, A.; BALAKRISHNAN, S.; POPPLEWELL, N. Cooperative dispatching for minimizing mean flowtime in a dynamic flowshop. *International Journal of Production Economics*, v. 113, issue 2, p. 819-833, 2008.

ENGIN, O.; DÖYEN, A. A new approach to solve flow shop scheduling problems by artificial immune systems. *Doğuş Üniversitesi Dergisi*, v. 8, issue 1, p. 12-27, 2007.

FERNANDES, F. C. F.; GODINHO FILHO, M. *Planejamento e controle da produção*: dos fundamentos ao essencial. São Paulo: Atlas, 2010.

FRAMINAN, J. M.; GUPTA, J. N. D.; LEISTEN, R. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *Journal of the Operational Research Society*, v. 55, p. 1243-1255, 2004.

FRAMINAN, J. M.; LEISTEN, R.; RUIZ-USANO, R. Comparison of heuristics for flowtime minimization in permutation flowshops. *Computers & Operations Research*, v. 32, issue 5, p. 1237-1254, 2005.

FRENCH, S. *Sequencing and scheduling*: an introduction to the mathematics of the job-shop. New York: John Wiley & Sons, 1982.

GAO, H.; LIU, X. Improved artificial immune algorithm and its applications on permutation flow shop sequencing problems. *Journal of Information Technology*, v. 6, issue 6, p. 929-933, 2007.

GAREY, M. R.; JOHNSON, D. S.; SETHI, R. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, v. 1, issue 2, p. 117-129, 1976.

GASPAR-CUNHA, A.; TAKAHASHI, R.; ANTUNES, C. H. (Coord.). *Manual de computação evolutiva e metaheurística*. Coimbra: Imprensa da Universidade de Coimbra, 2012.

GLOVER, F. Heuristics for integer programming using surrogate constraints. *Decision Science*, v. 8, issue 1, p. 156-166, 1977.

GLOVER, F. First comprehensive description of tabu search, "Tabu Search – Part I". *ORSA Journal on Computing*, v. 1, issue 3, p. 190-206, 1989.

GLOVER, F. The second part of this comprehensive description of tabu search introduces additional mechanisms such as the reverse elimination method, "Tabu Search – Part II". *ORSA Journal on Computing*, v. 2, issue 1, p. 4-32, 1990.

GRABOWSKI, J.; PEMPERA, J. New block properties for the permutation flow shop problem with application in tabu search. *Journal of the Operational Research Society*, v. 52, p. 210-220, 2001.

GRABOWSKI, J.; WODECKI, M. A very fast tabu search algorithm for the permutation flowshop problem with makespan criterion. *Computers & Operations Research*, v. 31, issue 11, p. 1891-1909, 2004.

GUPTA, J. N. D. A functional heuristic algorithm for the flowshop scheduling problem. *Operational Research Quarterly*, v. 22, issue 1, p. 39-47, 1971.

GUPTA, J. N. D.; STAFFORD JR., E. F. Flow shop scheduling research after five decades. *European Journal of Operational Research*, v. 169, issue 3, p. 699-711, 2006.

HARDING, H. A. *Administração da produção*. Tradução de José Marques Junior. São Paulo: Atlas, 1981.

HAQ, A. N.; SARAVANAN, M.; VIVEKRAJ, A. R.; PRASAD, T. A scatter search approach for general flowshop scheduling problem. *The International Journal of Advanced Manufacturing Technology*, v. 31, issue 7-8, p. 731-736, 2007.

HASANZADEH, A.; AFSHARI, H.; KIANFAR, K.; FATHI, M.; JADID, A. O. A GRASP algorithm for the two-machine flow-shop problem with weighed late work criterion and common due date. In: THE IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL ENGINEERING AND ENGINEERING MANAGEMENT, Hong Kong. *Proceedings*… Hong Kong, 2009. p. 1930-1934.

HAX, A. C.; CANDEA, D. *Production and inventory management*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

HILLIER, F. S.; LIEBERMAN, G. J. *Introdução à pesquisa operacional*. São Paulo: McGraw-Hill, 2006.

HOLLAND, J. H. *Adaptation in natural and artificial systems*. Ann Arbor, MI: The University of Michigan Press, 1976.

HSIEH, Y.-C.; YOU, P.-S.; LIOU, C.-D. A note of using effective immune based approach for the flow shop scheduling with buffers. *Applied Mathematics and Computation*, v. 215, issue 5, p. 1984-1989, 2009.

IGNALL, E.; SCHRAGE, L. Application of the branch and bound technique to some flow-shop scheduling problems. *Operations Research*, v. 13, issue 3, p. 400-412, 1965.

ISHIBUCHI, H.; MISAKI, S.; TANAKA, H. Modified simulated annealing algorithms for the flow-shop sequencing problem. *European Journal of Operational Research*, v. 81, issue 2, p. 388-398, 1995.

JARBOUI, B.; IBRAHIM, S.; SIARRY, P.; REBAI, A. A combinatorial particle swarm optimization for solving permutation flowshop problems. *Computers & Industrial Engineering*, v. 54, issue 3, p. 526-538, 2008.

JAROSŁAW, P.; CZESŁAW, S.; DOMINIK, Ż. Optimizing bicriteria flow shop scheduling problem by simulated annealing algorithm. *Procedia Computer Science*, v. 18, p. 936-945, 2013.

JAYAMOHAN, M. S.; RAJENDRAN, C. New dispatching rules for shop scheduling: a step forward. *International Journal of Production Research*, v. 38, issue 3, p. 563-586, 2000.

JOHNSON, S. M. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, Hoboken, v. 1, issue 1, p. 61-68, 1954.

KENNEDY, J.; EBERHART, R. C. Particle swarm optimization. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS, 1995, Perth. *Proceedings*... Piscataway, NJ, 1995. v. 4. p. 1942-1948.

KHAN, B. S. H.; PRABHAHARAN, G.; ASOKAN, P. A GRASP algorithm for m-machine flow shop scheduling problem with bicriteria of makespan and maximum tardiness. *International Journal of Computer Mathematics*, v. 84, issue 12, 2007.

KIRKPATRICK, S.; GELATT, D. C.; VECCHI, M. P. Optimization by simulated annealing. *Science*, v. 220, issue 4598, p. 671-680, 1983.

KURZ, M. E. *Scheduling flexible flow line with sequence dependent setup times*. 2001. 266 p. Dissertation (Doctorate degree) – Department of Systems and Industrial Engineering, The University of Arizona, Tucson, 2001.

LAWLER, E. L.; LENSTRA, J. K.; RINNOOY KAN, A. H. G.; SHMOYS, D. B. Sequencing and scheduling: algorithms and complexity. *Handbooks in Operations Research and Management Science*, v. 4, p. 445-552, 1993.

LEI, D. Multi-objective production scheduling: a survey. *The International Journal of Advanced Manufacturing Technology*, v. 43, issue 9-10, p. 926-938, 2009.

LI, X.; YIN, M. An opposition-based differential evolution algorithm for permutation flow shop scheduling based on diversity measure. *Advances in Engineering Software*, v. 55, p. 10-31, 2013.

LIAEE, M. M.; EMMONS, H. Scheduling families of jobs with setup times. *International Journal of Production Economics*, Amsterdam, v. 51, issue 3, p. 165-176, 1997.

LIAN, Z.; GU, X.; JIAO, B. A similar particle swarm optimization algorithm for permutation flowshop scheduling to minimize makespan. *Applied Mathematics and Computation*, v. 175, issue 1, p. 773-785, 2006.

LIAN, Z.; GU, X.; JIAO, B. A novel particle swarm optimization algorithm for permutation flowshop scheduling to minimize makespan. *Chaos, Solitons & Fractals*, v. 35, issue 5, p. 851-861, 2008.

LIAO, C. J.; TSENG, C. T.; LUARN, P. A discrete version of particle swarm optimization for flowshop scheduling problems. *Computers & Operations Research*, v. 34, issue 10, p. 3099-3111, 2007.

LIAO, L.-M.; HUANG, C.-J. Tabu search heuristic for two-machine flowshop with batch processing machines. *Computers & Industrial Engineering*, v. 60, issue 3, p. 426-432, 2011.

LIN, B. M. T.; CHENG, T. C. E. Two-machine flowshop batching and scheduling. *Annals of Operations Research*, v. 133, issue 1-4, p. 149-161, 2005.

LIN, D.; LEE, C. K. M.; HO, W. Multi-level genetic algorithm for the resource--constrained re-entrant scheduling problem in the flow shop. *Engineering Applications of Artificial Intelligence*, v. 26, issue 4, p. 1282-1290, 2013.

LOPES, H. S.; RODRIGUES, L. C. A.; STEINER, M. T. A. (Ed.). *Meta-heurísticas em Pesquisa Operacional*. Curitiba: Omnipax, 2013.

LOW, C.; YEH, J. Y.; HUANG, K. I. A robust simulated annealing heuristic for flowshop scheduling problems. *The International Journal of Advanced Manufacturing Technology*, v. 23, issue 9-10, p. 762-767, 2004.

LUSTOSA, L.; MESQUITA, M. A.; QUELHAS, O.; OLIVEIRA, R. (Org.). *Planejamento e controle da produção*. Rio de Janeiro: Elsevier, 2008.

MacCARTHY, B. L.; LIU, J. Y. Addressing the gap in scheduling research – a review of optimization and heuristic methods in production scheduling. *International Journal of Production Research*, v. 31, issue 1, p. 59-79, 1993.

MANJESHWAR, P. K.; DAMODARAN, P.; SRIHARI, K. Minimizing makespan in a flow shop with two batch-processing machines using simulated annealing. *Robotics and Computer-Integrated Manufacturing*, v. 25, issue 3, p. 667-678, 2009.

MOCCELLIN, J. V. A new heuristic method for the permutation flow-shop scheduling problem. *Journal of the Operational Research Society*, v. 46, issue 7, p. 883-886, 1995.

MOCCELLIN, J. V.; NAGANO, M. S. *Flow shop* com máquinas paralelas genéricas. In: SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL, 35., 2003, Natal. *Anais*... Rio de Janeiro: Sociedade Brasileira de Pesquisa Operacional, 2003. 1 CD-ROM.

MORTON, T. E.; PENTICO, D. W. *Heuristic scheduling systems*. New York, NY: John Wiley & Sons, 1993.

NADERI, B.; RUIZ, R. A scatter search algorithm for the distributed permutation flowshop scheduling problem. *European Journal of Operational Research*, v. 239, issue 2, p. 323-334, 2014.

NADERI, B.; TAVAKKOLI-MOGHADDAM, R.; KHALILI, M. Electromagnetism-like mechanism and simulated annealing algorithms for flowshop scheduling problems minimizing the total weighted tardiness and makespan. *Knowledge-Based Systems*, v. 23, issue 2, p. 77-85, 2010.

NAGANO, M. S.; MOCCELLIN, J. V. A high quality solution constructive heuristic for flow shop sequencing. *Journal of the Operational Research Society*, v. 53, issue 12, p. 1374-1379, 2002.

NAGANO, M. S.; RUIZ, R.; LORENA, L. A. N. A constructive genetic algorithm for permutation flowshop scheduling. *Computers & Industrial Engineering*, v. 55, issue 1, p. 195-207, 2008.

NAWAZ, M.; ENSCORE JR., E. E.; HAM, I. A heuristic algorithm for the *m*-machine *n*-job flow-shop sequencing problem. *OMEGA*: The International Journal of Management Science, v. 11, issue 1, p. 91-95, 1983.

NEARCHOU, A. C. A novel metaheuristic approach for the flowshop scheduling problem. *Engineering Applications of Artificial Intelligence*, v. 17, issue 3, p. 289-300, 2004a.

NEARCHOU, A. C. Flowshop sequencing using hybrid simulated annealing. *Journal of Intelligent Manufacturing*, v. 15, issue 3, p. 317-328, 2004b.

NOWICKI, E.; SMUTNICKI, C. A fast tabu search algorithm for the permutation flowshop problem. *European Journal of Operational Research*, v. 91, issue 1, p. 160-175, 1996.

OGBU, F. A.; SMITH, D. K. Simulated annealing for the permutation flowshop problem. *OMEGA*: The International Journal of Management Science, v. 19, p. 64-67, 1991.

ONWUBOLU, G.; DAVENDRA, D. Scheduling flowshops using differential evolution algorithm. *European Journal of Operations Research*, v. 171, issue 2, p. 674-692, 2006.

OSMAN, I. H.; POTTS, C. N. Simulated annealing for permutation flow-shop scheduling. *OMEGA*: The International Journal of Management Science, v. 17, issue 6, p. 551-557, 1989.

PALMER, D. S. Sequencing jobs through a multi-stage process in the minimum total time – a quick method of obtaining near optimum. *Journal of the Operational Research Society*, v. 16, p. 101-107, 1965.

PAN, Q. E.; RUIZ, R. A comprehensive review and evalutation of permutation flowshop heuristics to minimize flowtime. *Computers & Operations Research*, v. 40, issue 1, p. 117-128, 2013.

PAN, Q. K.; TASGETIREN, M. F.; LIANG, Y. C. A discrete differential evolution algorithm for the permutation flowshop scheduling problem. *Computers & Industrial Engineering*, v. 55, issue 4, p. 795-816, 2008.

PINEDO, M. *Scheduling*: theory, algorithms, and systems. 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2012.

PRABHAHARAN, G.; KHAN, B. S. H.; RAKESH, L. Implementation of grasp in flow shop scheduling. *The International Journal of Advanced Manufacturing Technology*, v. 30, issue 11-12, p. 1126-1131, 2006.

QIAN, B.; WANG, L.; HU, R.; WANG, W. L.; HUANG, D. X.; WANG, X. A hybrid differential evolution method for permutation flow-shop scheduling. *The International Journal of Advanced Manufacturing Technology*, v. 38, issue 7-8, p. 757-777, 2008.

QIAN, B.; WANG, L.; HUANG, D. X.; WANG, W. L.; WANG, X. An effective hybrid DE-based algorithm for multi-objective flowshop scheduling with limited buffers. *Computers & Operations Research*, v. 36, issue 1, p. 209-233, 2009.

RAJENDRAN, C.; ZIEGLER, H. Ant-colony algorithms for flowshop scheduling to minimize makespan/total flowtime of jobs. *European Journal of Operational Research*, v. 155, issue 2, p. 426-438, 2004.

RAVETTI, M. G.; NAKAMURA, F. G.; MENESES, C. N.; RESENDE, M. G. C.; MATEUS, G. R.; PARDALOS, P. M. *Hybrid heuristics for the permutation flow shop problem*. AT&T Labs Research Technical Report TD-6V9MEV, 2006.

REEVES, C. R. A genetic algorithm for flowshop sequencing. *Computers & Operations Research*, v. 22, issue 1, p. 5-13, 1995.

REZA HEJAZI, S.; SAGHAFIAN, S. Flowshop-scheduling problems with makespan criterion: a review. *International Journal of Production Research*, v. 43, issue 14, p. 2895-2929, July 2005.

RINNOOY KAN, A. H. G. *Machine scheduling problems*: classification, complexity and computations. The Hague: Martinus Nijhoff, 1976.

ROBINSON, A. *Modern approaches to manufacturing improvement*: the Shingo system. Portland, OR: Productivity Press, 1990.

RUIZ, R. *Técnicas metaheurísticas para la programación flexible de la producción*. 2003. 521 p. Tese (Doutorado em Ciência da Computação) – Departamento de Estadística e Investigación Operativa Aplicadas y Calidad, Universitat Politècnica de València, València, 2003.

RUIZ, R.; MAROTO, C. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, v. 165, issue 2, p. 479-494, 2005.

RUIZ, R.; MAROTO, C.; ALCARAZ, J. Two new robust genetic algorithms for the flowshop scheduling problem. *OMEGA*: The International Journal of Management Science, v. 34, issue 5, p. 461-476, 2006.

SARAVANAN, M.; HAQ, A. N.; VIVEKRAJ, A. R.; PRASAD, T. Performance evaluation of the scatter search method for permutation flowshop sequencing problems. *The International Journal of Advanced Manufacturing Technology*, v. 37, issue 11-12, p. 1200-1208, 2008.

SHINGO, S. *Sistemas de produção com estoque zero*: o sistema Shingo para melhorias contínuas. Tradução de Lia Weber Mendes. Porto Alegre: Bookman, 1996.

SHINGO, S. *Sistema de troca rápida de ferramenta*: uma revolução nos sistemas produtivos. Tradução de Eduardo Schaan e Cristina Schumacher. Porto Alegre: Bookman, 2000.

SLACK, N.; CHAMBERS, S.; HARLAND, C.; HARRISON, A.; JOHNSTON, R. *Administração da produção*. Tradução de Ailton Bonfim Brandão. São Paulo: Atlas, 1999.

STORN, R.; PRICE, K. *Differential evolution*: a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, ICSI, 1995.

STÜTZLE, T. An ant approach to the flow shop problem. In: EUROPEAN CONGRESS ON INTELLIGENT TECHNIQUES AND SOFT COMPUTING, 6., 1998, Aachen. *Proceedings*… Aachen: Verlag Mainz, 1998. p. 1560-1564.

SUN, Y.; ZHANG, C.; GAO, L.; WANG, X. Multi-objective optimization algorithms for flow shop scheduling problem: a review and prospects. *The International Journal of Advanced Manufacturing*, v. 55, issue 5-8, p. 723-739, 2011.

TAHA, H. A. *Pesquisa Operacional*: uma visão geral. Tradução de Arlete Simille Marques. São Paulo: Pearson: Prentice Hall, 2008.

TAILLARD, E. Some efficient heuristic methods for the flowshop sequencing problem. *European Journal of Operational Research*, v. 47, issue 1, p. 65-74, 1990.

TAILLARD, E. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, v. 64, issue 2, p. 278-285, 1993.

TASGETIREN, M. F.; LIANG, Y. C.; SEVKLI, M.; GENCYILMAZ, G. Differential evolution algorithm for permutation flowshop sequencing problem with makespan criterion. In: INTERNATIONAL SYMPOSIUM ON INTELLIGENT MANUFACTURING SYSTEMS, 4., 2004, Sakarya. *Proceedings*... Sakarya, 2004. p. 442-452.

TASGETIREN, M. F.; LIANG, Y. C.; SEVKLI, M.; GENCYILMAZ, G. A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. *European Journal of Operational Research*, v. 177, issue 3, p. 1930-1947, 2007.

TAVAKKOLI-MOGHADDAM, R.; RAHIMI-VAHED, A.; MIRZAEI, A. H. A hybrid multi-objective immune algorithm for a flow shop scheduling problem with bi-objectives: weighted mean completion time and weighted mean tardiness. *Information Sciences*, v. 177, issue 22, p. 5072-5090, 2007.

TUBINO, D. F. *Manual de planejamento e controle da produção*. 2. ed. São Paulo: Atlas, 2006.

VALLADA, E.; RUIZ, R.; MINELLA, G. Minimising total tardiness in the *m*-machine flowshop problem: a review and evaluation of heuristics and metaheuristics. *Computers & Operations Research*, v. 35, issue 4, p. 1350-1373, 2008.

VIGNIER, A.; BILLAUT, J. C.; PROUST, C. Les problèmes d'ordonnancement de type flow-shop hybride: état de l'art. *RAIRO*: Recherche Opérationnelle, Paris, v. 33, issue 2, p. 117-183, 1999.

WANG, L.; ZHENG, D. Z. An effective hybrid heuristic for flow shop scheduling. *The International Journal of Advanced Manufactoring Technology*, v. 21, issue 1, p. 38-44, 2003.

WANG, C.; LI, X.; WANG, Q. Accelerated tabu search for no-wait flowshop scheduling problem with maximum lateness criterion. *European Journal of Operational Research*, v. 206, issue 1, p. 64-72, 2010.

YAGMAHAN, B.; YENISEY, M. M. Scheduling practice and recent development in flow shop and job shop scheduling. *Computational Intelligence in Flow Shop and Job Shop Scheduling*, v. 230, p. 261-300, 2009.

YENISEY, M. M.; YAGMAHAN, B. Multi-objective permutation flow shop scheduling problem: literature review, classification and current trends. *OMEGA*: The International Journal of Management Science, v. 45, p. 119-135, 2014.

YING, K. C.; LIAO, C. J. An ant colony system for permutation flowshop sequencing. *Computers & Operations Research*, v. 31, issue 5, p. 791-801, 2004.

ZANDIEH, M.; FATEMI GHOMI, S. M. T.; MOATTAR HUSSEINI, S. M. An immune algorithm approach to hybrid flow shops scheduling with sequence--dependent setup times. *Applied Mathematics and Computation*, v. 180, issue 1, p. 111-127, 2006.

ZHANG, C.; NING, J.; OUYANG, D. A hybrid alternate two phases particle swarm optimization algorithm for flow shop scheduling problem. *Computers & Industrial Engineering*, v. 58, issue 1, p. 1-11, 2010.

ZHANG, C.; SUN, J.; ZHU, X.; YANG, Q. An improved particle swarm optimization algorithm for flowshop scheduling problem. *Information Processing Letters*, v. 108, issue 4, p. 204-209, 2008.

ZHANG, L.; WANG, L.; ZHENG, D. Z. An adaptive genetic algorithm with multiple operators for flowshop scheduling. *The International Journal of Advanced Manufactoring Technology*, v. 27, issue 5-6, p. 580-587, 2006.

ZHU, X.; WILHELM, W. E. Scheduling and lot sizing with sequence-dependent setup: a literature review. *IIE Transactions*, v. 38, issue 11, p. 987-1007, 2006.

Revisão
Gisele Dionísio da Silva

Projeto gráfico da coleção
Alanna Oliva

Editoração eletrônica
Marcus Lisita Rotoli

Dados internacionais de Catalogação-na-Publicação (CIP)
GPT/BC/UFG

F949s

Fuchigami, Hélio.
Sequenciamento da produção em sistemas *flow shop* / Hélio Fuchigami. – Goiânia : Gráfica UFG, 2015.

136 p. (Coleção Lavor)

ISBN: 978-85-68359-37-2

1. Engenharia de produção 2. Planejamento de produção 3. Sistema *flow shop* I. Título.

CDU: 658.5:004.4

Impressão e acabamento Cegraf – UFG
Câmpus Samambaia, Caixa Postal 131
74001-970 – Goiânia – Goiás – Brasil
Fone: (62) 3521 1107 – Fax: (62) 3521 1814
editora@ufg.br – www.cegraf.ufg.br