UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Maria Luísa Gabriel Domingues

Algoritmos para as principais variantes do problema de escalonamento de tarefas

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Maria Luísa Gabriel Domingues

Algoritmos para as principais variantes do problema de escalonamento de tarefas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Paulo Henrique Ribeiro Gabriel

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4144 - http://www.portal.facom.ufu.br/ facom@ufu.br

ATA DE DEFESA - GRADUAÇÃO

Curso de Graduação em:	Bacharelado em Ciência da Computação				
Defesa de:	GBC082 - Projeto de Graduação				
Data:	26/09/2025	Hora de início:	17:00	Hora de encerramento:	18:30
Matrícula do Discente:	12121BCC010				
Nome do Discente:	Maria Luísa Gabriel Domingues				
Título do Trabalho:	Algoritmos para as principais variantes do problema de escalonamento de tarefas				
A carga horária curricular foi cumprida integralmente?	(X) Sim ( ) Não				

Reuniu-se remotamente por meio da plataforma MS Teams, da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Curso de Graduação em Ciência da Computação, assim composta: Professores: Dr. Bruno Augusto Nassif Travencolo - FACOM/UFU; Dra. Márcia Aparecida Fernandes - FACOM/UFU; e Dr. Paulo Henrique Ribeiro Gabriel - FACOM/UFU, orientador da candidata.

Iniciando os trabalhos, o presidente da mesa, Dr. Paulo Henrique Ribeiro Gabriel, apresentou a Comissão Examinadora e a candidata, agradeceu a presença do público, e concedeu à discente a palavra, para a exposição do seu trabalho. A duração da apresentação do discente e o tempo de arguição e resposta foram conforme as normas do curso.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir a candidata. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando a candidata:

(X) Aprovada Nota [95] (Somente números inteiros)

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Paulo Henrique Ribeiro Gabriel**, **Professor(a) do Magistério Superior**, em 26/09/2025, às 18:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Bruno Augusto Nassif Travençolo**, **Professor(a) do Magistério Superior**, em 26/09/2025, às 18:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Márcia Aparecida Fernandes**, **Professor(a) do Magistério Superior**, em 26/09/2025, às 18:29, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Agradecimentos

Agradeço a Deus, acima de tudo, por ser meu amparo, força e fonte de amor incondicional em todos os momentos de minha vida.

À minha família, que me forneceu todo o apoio e carinho em todos os caminhos que tomei em minha vida, e aos meus amigos, que estiveram sempre ao meu lado nos momentos bons e ruins.

Aos meus professores, que me orientaram durante a minha vida acadêmica e me mostraram a alegria que é ser uma pessoa sedenta por conhecimento.

Resumo

O problema de escalonamento de tarefas é um dos vários problemas de otimização da computação que pertencem à Classe NP-Completo, evidenciando a alta complexidade na obtenção de soluções exatas, e abrindo espaço para a produção de algoritmos aproximados. Com o estudo de muitos pesquisadores do ramo no decorrer das décadas, vários algoritmos aproximados foram projetados para obter soluções próximas da solução ótima em tempo hábil para várias instâncias do problema, respeitando uma gama de restrições atreladas a cada uma delas, buscando torná-las úteis para aplicação em ambientes reais. Neste trabalho, abordamos as cinco variantes mais conhecidas do problema de escalonamento: escalonamento de tarefas em uma única máquina, multiprocessamento para máquinas idênticas com memória compartilhada ou distribuída, e multiprocessamento para máquinas não-idênticas com memória compartilhada ou distribuída. Para cada uma dessas variantes, então, é descrito o sistema em que elas se inserem e as restrições e custos que elas consideram, além de ser apresentado alguns algoritmos aproximados, analisando a estratégia, complexidade e razão de aproximação de cada um deles. Por fim, é esperado que essa exposição extensa de várias estratégias de aproximação para vários contextos de escalonamento contribua para a sistematização do conhecimento de algoritmos aproximados para esse problema e sirva de base para outras pesquisas.

Palavras-chave: Escalonamento de tarefas; Algoritmos de aproximação; Heurísticas; Tarefas independentes; DAG.

Abstract

The task scheduling problem is one of several computing optimization problems that belong to the NP-Complete class, highlighting the high complexity of obtaining exact solutions and paving the way for the development of approximate algorithms. Through the study of many researchers in the field over the decades, several approximate algorithms have been designed to obtain solutions close to the optimal solution promptly for various instances of the problem, respecting a range of constraints associated with each one, aiming to make them useful for application in real environments. In this work, we address the five best-known variants of the scheduling problem: task scheduling on a single machine, multiprocessing for identical machines with shared or distributed memory, and multiprocessing for non-identical machines with shared or distributed memory. For each of these variants, we describe the system in which they are inserted and the constraints and costs they consider. We also present some approximate algorithms, analyzing the strategy, complexity, and approximation ratio of each of them. Finally, this extensive exposition of several approximation strategies for various scheduling contexts will contribute to the systematization of knowledge of approximate algorithms for this problem and serve as a basis for further research.

Keywords: Task scheduling; Approximation algorithms; Heuristics; Independent tasks; DAG.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

EDD	*Earliest Due Date rule*
DAG	Grafo Acíclico Direcionado
CP	Caminho Crítico de um DAG de precedência de tarefas
LS	*List Scheduling*
LPT	*Longest Processing Time First Scheduling*
SPT	*Shortest Processing Time First Scheduling*
FFD	*First Fit Decreasing*
MCP	*Modified Critical Path*
ETF	*Earliest Task First Scheduling*

Lista de símbolos

T	Conjunto de tarefas
K	Subconjunto do conjunto de tarefas. $K \subset T$
E	Conjunto de tarefas já escalonadas
P	Conjunto de máquinas
I	Conjunto de máquinas livres para escalonamento. $I \subset P$
m	Número de máquinas
n	Número de tarefas
T_i	i-ésima tarefa de T, onde $1 \leq i \leq n$
$T_{entrada}$	Tarefa de entrada do DAG (Grafo Acíclico Direcionado) de precedência. É uma tarefa que não possui nenhuma tarefa que a precede
T_{saida}	Tarefa de saída do DAG (Grafo Acíclico Direcionado) de precedência. É uma tarefa que não precede nenhuma outra tarefa
P_j	j-ésima máquina de P, onde $1 \leq j \leq m$
$p(T_i)$	Máquina onde a tarefa T_i se encontra escalonada
L	Lista de escalonamento
L_j	Lista de escalonamento com as tarefas escalonadas para a máquina P_j. $L_j \subset L$
r_i	Tempo inicial da janela de execução de uma tarefa T_i
$r_{\min}$	Menor r_i dentre todas as tarefas do sistema
d_i	Tempo final da janela de execução de uma tarefa T_i
$d_{\max}$	Maior d_i dentre todas as tarefas do sistema
e_i	Tempo em que T_i foi escalonada
C_i	Tempo em que T_i finaliza sua execução
μ	Tempo de execução geral, para sistemas onde o custo de execução de toda tarefa é o mesmo

$\mu(T_i)$	Tempo de execução da tarefa T_i, para problemas que envolvem processamento em uma única máquina ou em paralelismo com máquinas idênticas
$\mu_j(T_i)$	Tempo de execução da tarefa T_i na máquina P_j, para problemas que envolvem processamento paralelo com máquinas não-idênticas
$\bar{\mu}(T_i)$	Tempo médio de execução de uma tarefa T_i. Usado em ambientes com máquinas não-idênticas
ϕ_j	Tempo ocioso total da máquina P_j
w_j	Makespan da máquina P_j
w	Makespan do sistema
α_i	Atraso de individual de uma tarefa T_i
$\alpha_{\max}$	Atraso máximo do sistema
w^*	Makespan da solução ótima
$\alpha^*_{\max}$	Atraso máximo da solução ótima
L^*	Lista de escalonamento da solução ótima
$\prec$	Relação de precedência entre tarefas. Se $T_i \prec T_k,\ 1 \leq i, k \leq n$, então T_i precisa finalizar sua execução para que T_k esteja livre para processamento
$P(V, A)$	DAG (Grafo Acíclico Direcionado) que representa as relações de precedência entre todas as tarefas do sistema. Para uma aresta $(v, u) \in A$ é definida uma relação de precedência entre a tarefa do nó v e a do nó u, de tal forma que $v \prec u$. Para problemas que envolvem memória compartilhada, o DAG (Grafo Acíclico Direcionado) é ponderado nas arestas para representar o custo de comunicação entre tarefas
$pred(T_i)$	Conjunto de tarefas que precedem T_i. $T_k \in pred(T_i) \iff T_k \prec T_i$
$succ(T_i)$	Conjunto de tarefas que sucedem T_i. $T_k \in succ(T_i) \iff T_i \prec T_k$
C	Tamanho limite de uma caixa para o problema *bin-packing*
$OPT(T, C)$	Solução ótima do problema *bin-packing* ao empacotar o conjunto de tarefas T em caixas de tamanho máximo C
$FFD(T, C)$	Solução do FFD (*First Fit Decreasing*) ao empacotar o conjunto de tarefas T em caixas de tamanho máximo C

C_l	Limite inferior para o valor de C
$C_l(T, m)$	Para todo $C < C_l[T, m]$, $FFD[T, C] > m$
C_u	Limite superior para o valor de C. É também o resultado do algoritmo MULTIFIT
$C_u(T, m)$	Para todo $C \geq C_u[T, m]$, $FFD[T, C] \leq m$. Portanto, é a solução do algoritmo MULTIFIT
τ_m	Fator de expansão mínimo da capacidade C das caixas para que o FFD (*First Fit Decreasing*) possa escalonar o conjunto T de tarefas em não mais que m caixas
$MF(k)$	Aplicação do MULTIFIT com k iterações para o problema de escalonamento de tarefas
$C_u(k)$	Solução do MULTIFIT com k iterações
$G(V, E)$	Grafo de incomparabilidade do grafo de precedência $P(V, A)$, de forma que se $(v, u) \in E \iff (v, u), (u, v) \notin A$
$ASAP(T_i)$	Tempo mais cedo possível em que a tarefa T_i pode ser executada. É calculado usando o *ASAP Binding*
$ALAP(T_i)$	Tempo mais tardio possível em que a tarefa T_i pode ser executada. É calculado usando o *ALAP Binding*
$M(T_i)$	Mobilidade de uma tarefa T_i. É um valor resultante da diferença entre $ASAP(T_i)$ e $ALAP(T_i)$
$\Gamma(T_i)$	Lista que contém o $ALAP$ de T_i e de todos os seus descendentes em $P(V, A)$ em ordem crescente
CM	Marco do tempo atual da execução do sistema
NM	Próximo marco de tempo atual da execução do sistema
$r(T_i, P_j)$	Tempo em que a última mensagem endereçada a T_i chega na máquina P_j, na qual T_i foi escalonada
$n(T_i, T_k)$	Custo de enviar mensagens de uma tarefa T_j para outra tarefa T_k
$t(P_j, P_k)$	Custo de abrir um canal de comunicação entre as máquinas P_j e P_k
Φ	Número de tarefas de um clã independente

K_i	Subgrupo agregado em um clã independente no algoritmo CLANS, $1 \leq k \leq n$
$\|K_i\|$	Número de tarefas envolvidas em um subgrupo K_i
$\mu(K_i)$	Soma do tempo de execução de todas as tarefas do subgrupo K_i
$y_{k,i}$	é o custo de uma informação entrar na tarefa $T_{k,i}$ do subgrupo K_i
$y'_{k,i}$	é o custo de uma informação sair da tarefa $T_{k,i}$ do subgrupo K_i, $1 \leq k \leq n$
y_i	é o custo de entrada de informação em uma tarefa T_i qualquer do clã independente (sem distinção de subgrupos)
y'_i	é o custo de saída de informação de uma tarefa T_i qualquer do clã independente (sem distinção de subgrupos)
y	Custo constante da entrada de uma informação em uma tarefa qualquer do sistema
y'	Custo constante da saída de uma informação de uma tarefa qualquer do sistema
$B_{j,k}$	Taxa de transmissão de informações entre máquinas P_j e P_k. $1 \leq j, k \leq m$
X_j	Custo de abrir a máquina P_j para comunicação
$\bar{B}$	Taxa de transmissão média do sistema
$\bar{X}$	Custo de abertura média para comunicação no sistema
$rank_u(Ti)$	*Upward Rank* de T_i
$rank_d(Ti)$	*Downward Rank* de T_i
$EST(T_i, P_j)$	Tempo mais cedo possível que uma tarefa T_i pode ser escalonada numa máquina P_j
$EFT(T_i, P_j)$	Tempo mais cedo possível que uma tarefa T_i pode ter sua execução finalizada numa máquina P_j
$AST(T_i, P_j)$	Tempo real em que uma tarefa T_i pode ser escalonada numa máquina P_j
$AFT(T_i, P_j)$	Tempo real em que uma tarefa T_i pode ter sua execução finalizada numa máquina P_j

Sumário

1 Introdução

O problema de escalonamento de tarefas é um dos mais estudados na Ciência da Computação devido à sua ampla aplicabilidade em ambientes industriais, computacionais e logísticos (ROBERT, 2011). Esse problema consiste em atribuir uma lista de tarefas a um conjunto de unidades de processamento, respeitando restrições específicas do contexto, com o objetivo de otimizar métricas como o tempo de execução total ou o atraso máximo. No entanto, por ser um problema NP-Completo (GAREY; JOHNSON, 1979; PAPADIMITRIOU; YANNAKAKIS, 1979), a complexidade das soluções ótimas torna-se intratável à medida que as variantes do problema incorporam mais restrições realistas. Essa inviabilidade prática impulsiona o desenvolvimento e estudo de algoritmos aproximados, os quais são capazes de fornecer soluções de qualidade subótima, porém aceitáveis, em tempo computacional viável.

Devido à vastidão de variantes do problema de escalonamento, cada uma com suas particularidades e algoritmos especializados, torna-se necessário um material que organize e compare essas abordagens. Essa análise pode auxiliar tanto na escolha da melhor estratégia para um cenário aplicado quanto no direcionamento de pesquisas futuras. Diante desse contexto, o objetivo deste trabalho de conclusão de curso é analisar as diferentes variantes clássicas do problema de escalonamento de tarefas, com foco nos algoritmos de aproximação desenvolvidos para cada uma delas.

Neste trabalho abordaremos cinco variantes desse problema, organizadas em três casos principais. Inicialmente, é abordado o escalonamento de tarefas em uma única máquina (WILLIAMSON; SHMOYS, 2011). Em seguida, abordamos o caso do multiprocessamento em máquinas idênticas (GRAHAM, 1966; GRAHAM, 1969; EPSTEIN, 2008; PAPADIMITRIOU; YANNAKAKIS, 1979; KHAN; MCCREARY; JONES, 1994; WU; GAJSKI, 1990; HWANG et al., 1989; LEE et al., 1988; MCCREARY; GILL, 1989) e, finalmente, o multiprocessamento em máquinas não-idênticas (IBARRA; KIM, 1977; HWANG et al., 1989; WU; GAJSKI, 1990; TOPCUOGLU; HARIRI; WU, 2002). Para esses dois últimos casos, são estudados algoritmos projetados para as variações de memória compartilhada e memória distribuída. Esse trabalho busca apresentar essa variedade de algoritmos com o objetivo de contribuir para a sistematização do conhecimento sobre aproximações para o problema de escalonamento e suas principais variantes, e também servir como base para futuras pesquisas desse ramo.

O restante desta monografia é organizado da seguinte forma: no Capítulo 2 apresentamos os conceitos de escalonamento de tarefas, de algoritmos de aproximação e de complexidade de algoritmos, no Capítulo 3, apresentamos as cinco variantes do problema

de escalonamento de tarefas, bem como os principais algoritmos empregados em cada caso. Finalmente, o Capítulo 4 apresenta as conclusões deste trabalho.

2 Fundamentação Teórica

2.1 Problemas de Otimização e o Problema de Escalonamento

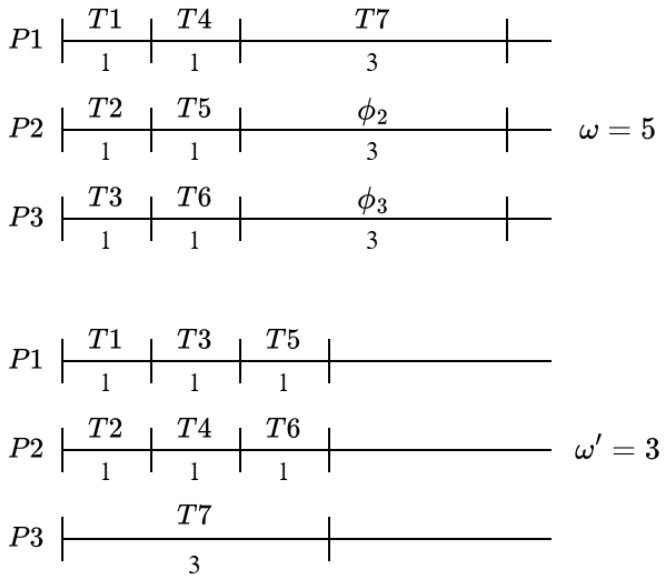
Problemas de otimização são problemas, tanto para a Matemática quanto para a Ciência da Computação, nos quais o objetivo é encontrar a **melhor solução** dentre todas as soluções possíveis, respeitando todas as restrições que o problema traz (GOLDBARG; GOLDBARG; LUNA, 2015). O conjunto de soluções para um problema de otimização pode ser representado como uma função e sua melhor solução pode ser interpretada como sendo um ponto mínimo (ou máximo) global dessa função (GOLDBARG; GOLDBARG; LUNA, 2015). A Ciência da Computação listou, com o passar dos anos de estudo, vários problemas que se enquadram como de otimização, e dentre eles está presente o problema que trataremos neste trabalho, chamado de **problema de escalonamento de tarefas** (ROBERT, 2011).

O problema de escalonamento de tarefas se trata de um desafio clássico de **otimização combinatória**, atribuindo uma lista de **tarefas** a uma quantidade fixa de **unidades de processamento**, de forma que uma unidade de processamento só pode executar uma tarefa por vez (GRAHAM, 1966), e uma solução possível desse problema é justamente o tempo em que é finalizada toda a execução (**makespan**), de acordo com a ordem estipulada para escalonamento das tarefas. Esse problema é de otimização justamente pelo fato de que, mesmo que uma certa solução seja viável, o interesse está em obter a melhor solução dentre as viáveis, sendo essa a **solução ótima** para o problema. Isso é evidenciado quando analisamos a Figura 1.

Observe, na Figura 1, como a ordem de escalonamento das tarefas impacta no makespan w do sistema. Na primeira solução, a organização das tarefas gera tempos ociosos ϕ_2 na máquina P_2, e ϕ_3 na máquina P_3, resultando num makespan $w = 5$, enquanto na segunda solução as tarefas são organizadas de forma diferente, sem causar tempo ocioso nas máquinas e resultando em um outro makespan $w' = 3$, menor que o da primeira solução. Uma simples troca de método de escalonar as tarefas pode causar melhoras significativas nas soluções dadas para o problema.

A definição da **solução ótima** para esse problema se dá, portanto, como a ordem de escalonamento das tarefas que resulta na **minimização** ou **maximização** de alguma variável alvo, como o **makespan**, normalmente com o objetivo de minimizá-lo. O makespan é o mais usado dentre os objetivos que existem para otimização no problema de escalonamento, mas para contextos como o escalonamento em uma única máquina, pode não fazer sentido buscar a minimização desse valor, recorrendo assim a outros objetivos

Figura 1 – Duas soluções possíveis para escalonar sete tarefas (T_1, T_2, T_3, T_4, T_5, T_6 e T_7) em três máquinas (P_1, P_2 e P_3), com os custos de execução totais ω e ω' para cada solução, respectivamente.



Fonte: Autoria própria.

como o atraso de execução das tarefas, como veremos posteriormente nesse trabalho. A Figura 1 representa uma das versões mais simples do problema de escalonamento, porém algumas **variantes** dele podem apresentar outras restrições adicionais, dentre elas estão a definição de **precedência** entre as tarefas, **homogeneidade** ou não das máquinas, o **número de máquinas** envolvidas no escalonamento, o custo adicional de **comunicação** entre as tarefas, determinação de **janelas de execução** para cada tarefa do sistema, dentre vários outros. Para cada variante, especificamente, são necessários **algoritmos** adequados para gerar soluções que respeitem as restrições definidas.

2.2 Algoritmos e Heurísticas de Aproximação

O mais básico método de obtenção de soluções para problemas são os **algoritmos**, que são um conjunto de instruções não ambíguas que se baseiam numa lógica bem estruturada de execução, resultando numa **solução exata** para o problema. Como buscamos pela solução ótima no problema de escalonamento, que é uma solução exata, o ideal seria implementar algoritmos que a obtenham, mas até hoje não foram projetados algoritmos capazes de encontrar a solução ótima em tempo menor que **exponencial**, que é um tempo muito custoso para tamanhos grandes de tarefas e/ou máquinas.

Com isso, foram introduzidas **estratégias** para tentar obter soluções com um custo de tempo menor que o exponencial, em sacrifício da **qualidade da solução**. Essas estratégias, agora, buscam por **soluções próximas** da ótima, e dentre elas se destacam

os **algoritmos de aproximação** e as **heurísticas de aproximação**. Diferentemente dos algoritmos exatos, as **heurísticas de aproximação** procuram resolver problemas de forma **empírica**, ou seja, baseando-se em regras mais práticas ou na intuição, o que pode acelerar o tempo para a obtenção da solução, mas **não há garantia**, além da experiência, sobre a qualidade da solução. Os **algoritmos aproximados**, em contraponto, são construídos baseando-se em métodos de aproximação nos quais há como **provar** a qualidade da solução (WILLIAMSON; SHMOYS, 2011).

Como no momento não foram encontrados algoritmos que encontrem a solução exata em tempo hábil, a Ciência da Computação têm investido em encontrar métodos de aproximação, como as heurísticas de aproximação e os algoritmos aproximados, que possam se aproximar o máximo possível da solução ótima. Posteriormente nesse trabalho veremos alguns desses algoritmos projetados por pesquisadores do ramo (WILLIAMSON; SHMOYS, 2011; GOLDBARG; GOLDBARG; LUNA, 2015).

2.3 Classes de Complexidade

A **complexidade de tempo** de um algoritmo é dita como o **tempo** gasto para obter a solução no qual foi projetado para gerar, o mesmo vale para qualquer outro modelo de resolução de problemas, sendo definida analisando a sequência de passos que definem esses modelos. O conjunto de complexidades desses modelos podem ser divididas em classes, sendo as principais **P** e **NP**.

Segundo Sipser (2013), a **Classe P** de complexidade contém todos os problemas que podem ser **resolvidos em tempo polinomial**, enquanto a **Classe NP** de complexidade contém todos os problemas que **não** possuem garantia de existência de algoritmos que possam resolvê-los em **tempo polinomial**, mas podem ter suas soluções **verificadas** em tempo polinomial. Dessa forma, a Classe P é garantidamente **contida** em NP, já que os problemas da Classe P são justamente algoritmos da classe NP nos quais já foram encontrados algoritmos que os resolvam em tempo polinomial, mas há suspeitas de que NP = P, que permanecerá enquanto não for provado que todos os problemas de NP não possuam algoritmos que os resolvam em tempo polinomial.

O problema de escalonamento pertence à Classe NP, mais especificamente, ele é **NP-Completo**, ou seja, se for encontrado um algoritmo que o resolva em tempo polinomial, **todos** os problemas da classe NP também poderão ser resolvidos em tempo polinomial.

Para saber se um problema é NP-Completo, basta provar que ele **pertence** à classe NP, e que todo problema em NP pode ser **reduzido**, em tempo polinomial, a esse problema. Ou então, basta provar que um problema que já é conhecido por ser NP-Completo pode ser reduzido em tempo polinomial ao problema em questão. No caso do

problema de escalonamento, é provado que ele pode ser reduzido do problema **Number Partition** (MERTENS, 2006), que é NP-Completo (GAREY; JOHNSON, 1979), assim provando que o problema de escalonamento também é NP-Completo.

3 Principais Variantes do Problema de Escalonamento e suas Aproximações

O problema de escalonamento de tarefas é amplo e, por isso, pode ser aplicado em vários **contextos**. Para cada um desses contextos, são envolvidas diferentes formas de funcionamento do sistema em que o escalonamento será aplicado, exigindo a consideração de certas **restrições**, como uma ordem de precedência entre tarefas, o número de máquinas envolvidas e seus comportamentos, bem como a forma de obtenção de informações pelas tarefas. Portanto, neste capítulo vamos abordar as principais variantes desse problema e suas soluções aproximadas.

3.1 Processamento em uma única máquina

O problema de escalonamento em uma **única máquina** é a mais simples dentre as variantes. Ele se caracteriza por estabelecer uma ordem de processamento para n tarefas em uma única máquina, de forma que sempre seja processada no máximo uma tarefa por vez.

Para esse problema, cada tarefa T_i dentre n tarefas deve ser processada por uma quantidade $\mu(T_i)$ de tempo, além de precisar ser executada dentro da sua **janela de execução** (entre o tempo de liberação r_i e o tempo limite de finalização d_i), onde C_i representa o **tempo de finalização** de execução da tarefa T_i. O objetivo desse problema é **minimizar o atraso máximo** $\alpha_{\max} = \max_{i=1,\dots,n} \alpha_i$, onde $\alpha_i = C_i - d_i$ representa o **atraso individual** de cada tarefa.

Esse problema, como descrito aqui, não permite a obtenção de aproximações diretas, já que para d_i positivos, resultando em $\alpha_i \leq 0$, aproximações que envolvem zero e valores negativos acarretam em muitas complicações. Para contornar isso, podemos assumir que os valores d_i serão sempre negativos, gerando valores de α_i positivos (WILLIAMSON; SHMOYS, 2011). Considerando esse caso, podemos obter uma **2-aproximação** para o problema: o **algoritmo EDD** (*Earliest Due Date rule*), um algoritmo guloso que prioriza a tarefa com o menor α_i.

Denotando $\alpha^*_{\max}$ como a solução ótima e $\alpha_{\max}$ como a solução do EDD (*Earliest Due Date rule*), obtemos a seguinte inequação:

$$\alpha^*_{\max} \geq r_{\min} + \sum_{i=1}^{n} \mu(T_i) - d_{\max}, \tag{3.1}$$

onde:

- $r_{\min} = \min_{i=1,\dots,n} r_i$;
- $d_{\max} = \max_{i=1,\dots,n} d_i$.

A partir dessa inequação, Williamson e Shmoys (2011) provam que, sendo k o índice da tarefa T_k que tem o maior atraso do sistema:

$$2\alpha^*_{\max} \geq C_k - d_k = \alpha_{\max} \tag{3.2}$$

Portanto, o algoritmo EDD (*Earliest Due Date rule*) é uma 2-aproximação:

$$\frac{\alpha_{\max}}{\alpha^*_{\max}} \leq 2 \tag{3.3}$$

O custo computacional do EDD (*Earliest Due Date rule*) consiste no custo para ordenação das tarefas, que pode ser $O(n \log n)$ se usarmos um algoritmo eficiente (CORMEN et al., 2024), e no custo para escalonamento $O(n)$, totalizando $O(n \log n)$.

Para **exemplificar** a aplicação desse algoritmo, vamos usá-lo para escalonar quatro tarefas $T = \{T_1, T_2, T_3, T_4\}$ em uma única máquina $P = \{P_1\}$. Para cada tarefa há seu valor de tempo de abertura r_i e fechamento d_i de sua janela de execução, além do seu custo $\mu(T_i)$. Na Tabela 1 a seguir estão representados todos esses dados.

Tabela 1 – Tabela com o valor de tempo de abertura r_i e fechamento d_i da janela de execução, além do custo $\mu(T_i)$ para cada tarefa do exemplo.

Tarefa	r_i	d_i	$\mu(T_i)$
T_1	0	-1	2
T_2	0	-5	4
T_3	2	-2	6
T_4	3	-3	8

Fonte: Autoria própria

O EDD, então, procede em fazer uma **análise de prioridade** de escalonamento das tarefas do conjunto T, seguindo os seguintes passos:

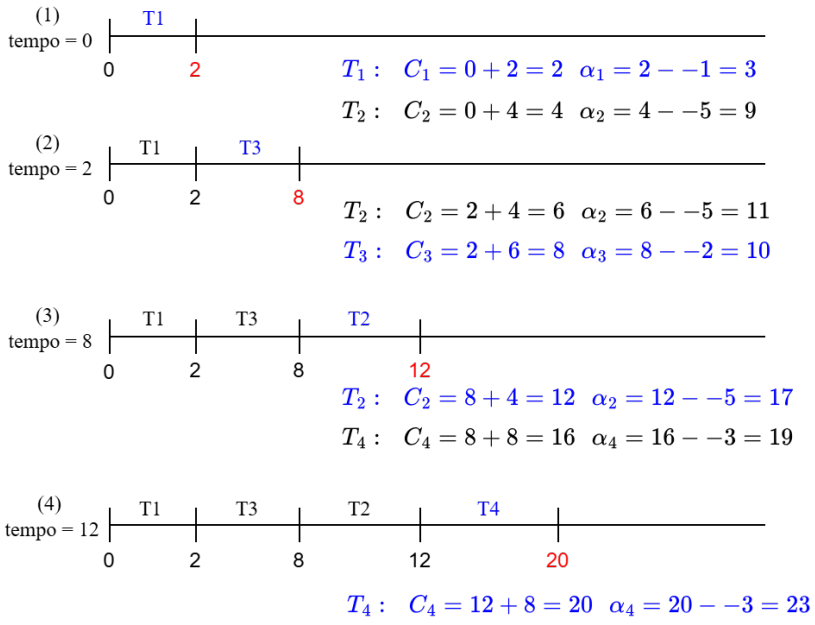
Passo 1. Separa as **tarefas disponíveis** para execução, ou seja, tarefas cujas janelas de execução estão abertas no tempo da iteração atual.

Passo 2. Calcula o possível C_i e o α_i para cada tarefa separada na etapa anterior, **simulando** uma execução no tempo da iteração atual.

Passo 3. O Algoritmo EDD, então, **escolhe** para escalonamento a tarefa com o menor valor de α_i dentre os calculados na etapa passada. A partir do marco de tempo em que a execução da tarefa escolhida finaliza, todos esses passos serão **repetidos**, até que todas as tarefas tenham sido executadas.

A Figura 2 mostra como se deu o escalonamento com o conjunto T de tarefas antes citado, descrevendo os passos efetuados a cada iteração.

Figura 2 – Execução do Algoritmo EDD, escalonando as quatro tarefas antes descritas em uma única máquina em quatro iterações. Por fim, é possível concluir que o atraso máximo α_{max} da solução gerada com o exemplo é 23.



Fonte: Autoria própria.

3.2 Processamento em Máquinas Paralelas Idênticas

O problema de escalonamento de tarefas a máquinas idênticas é a variante mais comum desse problema. Como o próprio nome diz, trata-se de escalonar tarefas num ambiente que envolve **mais de uma máquina**, sendo que todas elas possuem o **mesmo tempo de processamento** para uma mesma tarefa.

Em termos matemáticos, o sistema no qual esse problema é aplicado envolve um conjunto T de m máquinas e um conjunto de tarefas T com n tarefas que possuem entre si uma relação de **precedência** $\prec$, onde para uma tarefa T_i e outra T_k, se $T_i \prec T_k$, então, para T_k ser **disponível** para execução, T_i precisa ter sua execução concluída primeiro. O conjunto de relações de precedência de T é estruturada em forma de **grafo acíclico**

direcionado (DAG) $P(V, A)$, onde cada nó de V é uma tarefa, e cada aresta $(v, u) \in A$ é uma relação de precedência na forma $v \prec u$ (GRAHAM, 1966).

Essas tarefas são escalonadas para execução seguindo uma **lista de escalonamento** L. Como as máquinas são idênticas, todas elas compartilham a mesma função μ de tempo de processamento das tarefas, e o tempo para processar uma tarefa T_i é dado por $\mu(T_i)$. Pode-se demonstrar que esse problema é **NP-Completo** para $m \geq 2$ (GAREY; JOHNSON, 1979).

A seguir, são mostradas duas variações desse problema: o caso em que a memória dos processadores (máquinas) é **compartilhada** e o caso em que a memória **não é compartilhada**.

3.2.1 Memória Compartilhada

O paralelismo com memória compartilhada é o contexto mais simples de escalonamento em máquinas idênticas. Essencialmente, esse caso não possui nenhuma restrição a mais pois, como a **memória** é **compartilhada** entre as tarefas, idealmente não há custo de comunicação, exigindo que somente nos preocupemos com a restrição de precedência e em encontrar o escalonamento que resulte no **menor makespan** w possível. Os principais algoritmos para essa variante do problema são mostrados a seguir.

3.2.1.1 *List Scheduling* - LS

Um método de designar as tarefas às máquinas de forma aproximada foi proposto por Graham (1966) e chamado *List Scheduling* (LS). O LS é um **algoritmo guloso** que escolhe qual tarefa será escalonada para qual máquina de acordo com certa **prioridade**. Essa prioridade pode ser definida de várias formas, como em ordem crescente de tempo de processamento, caminho crítico, entre outros. A execução desse algoritmo se dá da seguinte forma:

Passo 1. **Ordenar** as tarefas em uma lista L de prioridade de escalonamento.

Passo 2. Cada máquina P_j atribui a si uma tarefa de L de forma a **minimizar** seu **makespan individual** w_j, fazendo antes a **verificação** de **disponibilidade** daquela tarefa. Uma tarefa é disponível quando todas as tarefas que a precedem já foram executadas. Quando o tempo de execução da tarefa atribuída é finalizado, P_j procura por uma nova tarefa, e isso se repete até que L esteja vazio.

Passo 3. Caso não haja tarefa disponível, P_j fica **em aguardo** (tempo ocioso ϕ_j) até que outra máquina finalize a execução de sua tarefa, e então ambas as máquinas procuram por novas tarefas livres.

Passo 4. O **tempo total** de execução desse escalonamento é w, que assume o valor do tempo da máquina com maior makespan individual.

Assim como retratado em diferentes trabalhos (GRAHAM, 1966; GRAHAM, 1969; EPSTEIN, 2008), o **limite máximo** do valor da sua solução w em relação à solução ótima w^* é representado pela inequação:

$$\frac{w}{w^*} \leq 2 - \frac{1}{m} \tag{3.4}$$

Visto que essa é uma inequação provada por Graham (1966) como sendo o limite máximo da razão de aproximação para algoritmos que variam somente na lista de escalonamento (preservando outras características do sistema, como o comportamento das máquinas, memória e relação de precedência entre tarefas), não importando muito qual é a estratégia de prioridade. O **custo assintótico** de tempo do algoritmo é $O(n \log n)$, onde n é o número de tarefas e $\log n$ é o custo para ordenar as tarefas em L. O *List Scheduling* é um algoritmo com desempenho ótimo de resultado para $m = 2$ ou 3, mas quando $m \geq 4$ seu desempenho perde para outros algoritmos que veremos em sequência (GRAHAM, 1966).

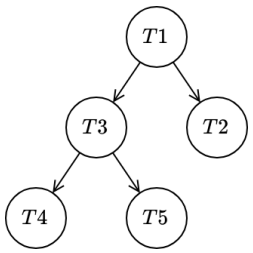
Aplicaremos agora um exemplo de escalonamento do *List Scheduling*, seguindo os passos antes descritos: supondo que a prioridade definida é por menor custo de execução $\mu(T_i)$ e que o conjunto de tarefas T para escalonar em duas máquinas $P = \{P_1, P_2\}$ seja $\{T_1, T_2, T_3, T_4, T_5\}$ (com valores de custo e relação de precedência na Tabela 2 e Figura 3 abaixo, respectivamente), o *List Scheduling* construirá uma solução aproximada conforme descrito na Figura 4.

Tabela 2 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo.

Tarefa	$\mu(T_i)$
T_1	10
T_2	8
T_3	6
T_4	5
T_5	7

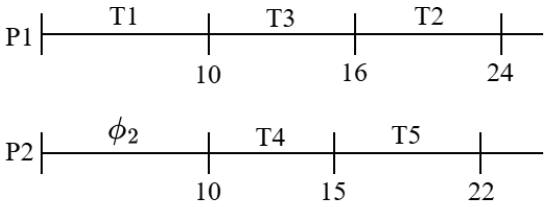
Fonte: Autoria própria

Figura 3 – DAG de precedência das tarefas do exemplo.



Fonte: Autoria própria.

Figura 4 – Execução do *List Scheduling*. Seguindo a prioridade antes descrita, a lista L assume o valor de $[T_4, T_3, T_5, T_2, T_1]$, que define a ordem para escalonar as tarefas de T. As tarefas, então, são escalonadas com base em L e também respeitando a ordem de precedência descrita no DAG (disponibilidade das tarefas para escalonamento). Essa solução aproximada possui makespan w de 24, e tempo ocioso na segunda máquina ϕ_2 de 10 unidades de tempo.



Fonte: Autoria própria.

3.2.1.2 *Longest Processing Time First Scheduling* - LPT

O **LPT** é um **algoritmo guloso** que usa da mesma estratégia do *List Scheduling*, aplicando uma prioridade específica: as primeiras tarefas a serem escalonadas devem ser as com **maior tempo de execução**. No entanto, é usado num contexto onde as tarefas são **independentes**, ou seja, ela possui um DAG completamente desconexo (COFFMAN JR; GAREY; JOHNSON, 1978).

Devido a essas alterações, o limite máximo da sua razão de aproximação, apresentado por Coffman Jr, Garey e Johnson (1978) e provado por Graham (1966), se dá por:

$$\frac{w}{w^*} \leq \frac{4}{3} - \frac{1}{3m} \tag{3.5}$$

O **custo assintótico** de tempo desse algoritmo é $O(n \log n)$, sendo que o custo de ordenação das tarefas em ordem decrescente é $O(n \log n)$ (CORMEN et al., 2024), e o custo de escalonar as tarefas é $O(n)$.

Podemos usar as mesmas tarefas usadas para execução do *List Scheduling* como

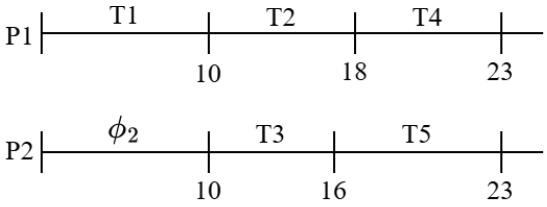
exemplo para execução do LPT *Scheduling*, recordando, também, a partir da Tabela 3 os custos das tarefas envolvidas. A partir disso, então, é construída uma solução aproximada com base no algoritmo do LPT *Scheduling*, como representado na Figura 5.

Tabela 3 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo.

Tarefa	$\mu(T_i)$
T_1	10
T_2	8
T_3	6
T_4	5
T_5	7

Fonte: Autoria própria

Figura 5 – Execução do LPT. Seguindo a prioridade descrita para os algoritmos LPT, a lista L de escalonamento assume o valor de $[T_1, T_2, T_5, T_3, T_4]$, que é a ordem definida para escalonar as tarefas de T, que então são escalonadas baseando-se nessa lista de escalonamento. Para o LPT *Scheduling*, não há a necessidade de considerar a disponibilidade das tarefas, visto que elas são independentes. Essa solução aproximada possui makespan w de 23, e tempo ocioso na segunda máquina ϕ_2 de 10 unidades de tempo.



Fonte: Autoria própria.

3.2.1.3 MULTIFIT

O algoritmo aproximado **MULTIFIT** é um algoritmo projetado como uma **adaptação** do uso de um algoritmo aproximado de outro problema, chamado ***bin-packing*** (CORMEN et al., 2024). O *bin-packing* consiste em atribuir **itens** a **caixas** de forma dinâmica, ou seja, uma **nova caixa** é criada à medida que as caixas já criadas se **enchem**. Cada caixa tem uma **capacidade fixa** C, e o objetivo dos algoritmos aproximados para esse problema é minimizar o número de caixas criados para encaixar todos os itens (COFFMAN JR; GAREY; JOHNSON, 1978).

O MULTIFIT aproveita do fato de que tanto o *bin-packing* quanto o problema de escalonamento são reduzíveis a um problema coincidente chamado PARTITION (COFF-

MAN JR; GAREY; JOHNSON, 1978), evidenciando a proximidade dos dois problemas em estrutura para otimização, para gerar soluções para o problema de escalonamento usando soluções do *bin-packing*. Dessa forma, as soluções geradas pelos algoritmos do *bin-packing* se encaixam como soluções do problema de escalonamento se, ao empacotar as tarefas de T em caixas com uma certa capacidade C (pensando que o peso de cada tarefa é seu custo de execução), o número de caixas geradas na execução **não for maior** que o número m de máquinas estipulado no sistema do problema de escalonamento, tornando C o makespan w da solução gerada. Dessa forma, a solução ótima do problema de escalonamento w^* se equivale à solução ótima do problema *bin-packing* $OPT[T, C]$ da seguinte forma:

$$w^* = min\{C : OPT[T, C] \leq m\} \tag{3.6}$$

Para obtermos soluções do problema *bin-packing*, usaremos o algoritmo aproximado $FFD[T, C]$ (*First Fit Decreasing*). Como os itens a atribuir às caixas são as tarefas, a execução do $FFD[T, C]$ será descrita levando isso em consideração:

Passo 1. Ordenar as tarefas de T por ordem **decrescente** de custo de execução $\mu(T_i)$.

Passo 2. Atribuir as tarefas uma a uma nas caixas, de acordo com a ordem estipulada no passo anterior.

Passo 2.1. Varrer as caixas existentes uma a uma, encaixar a tarefa na primeira caixa que ela couber. Caso a tarefa não caiba em nenhuma das caixas criadas ou não existam caixas ainda, criar uma caixa de capacidade C e atribuí-la à caixa. Repetir esse passo até que todas as tarefas estejam em caixas.

Passo 3. Por fim, o **número de caixas** criadas para atribuir todas as tarefas de T é o valor resultante da execução de $FFD[T, C]$.

É possível demonstrar que a razão de aproximação da solução do FFD (*First Fit Decreasing*), $FFD[T, C]$, perante a solução ótima $OPT[T, C]$, é dada por (COFFMAN JR; GAREY; JOHNSON, 1978; CORMEN et al., 2024):

$$FFD[T, C] \leq \frac{11}{9} OPT[T, C] + 4 \tag{3.7}$$

Esse algoritmo possui **custo assintótico** de $O(n \log n + n \cdot m)$, onde $O(n \log n)$ é o custo para ordenar os itens, e $O(n \cdot m)$ é o custo para alocar os itens às caixas ($O(n^2)$ no pior caso). Para o problema de escalonamento, a capacidade C das caixas corresponde ao **makespan** do sistema, e a **limitação** no número de caixas por m nas soluções geradas do FFD (*First Fit Decreasing*) as tornam elegíveis como o **escalonamento** da melhor

solução aproximada para esse problema. Assim, o objetivo se torna **minimizar** o valor de C de forma que ainda é possível encontrar soluções com o FFD em que a alocação das tarefas é feita em até m caixas.

O algoritmo **MULTIFIT**, então, usa da **busca binária** para encontrar valores de C, definindo um limite mínimo de capacidade C_l e um limite máximo de capacidade C_u para busca. Em seguida, usa do FFD (*First Fit Decreasing*) para validá-los como possíveis soluções para o problema de escalonamento, retornando em $C_u(k)$ a **solução aproximada**, para um número k qualquer de iterações. A execução do MULTIFIT consiste nos seguintes passos (COFFMAN JR; GAREY; JOHNSON, 1978):

1. Sendo m o número de máquinas para escalonamento e n o número de tarefas a serem escalonadas, definir um **limite superior e inferior** para C: C_l e C_u, de tal forma que:
   - $C_l[T, m] = \max\{\frac{\sum_{i=1}^{n} \mu(T_i)}{m}, \max_{T_i \in T}\{\mu(T_i)\}\}$
     (portanto, para todo $C < C_l[T, m]$, $FFD[T, C] > m$).
   - $C_u[T, m] = \max\{\frac{2\sum_{i=1}^{n} \mu(T_i)}{m}, \max_{T_i \in T}\{\mu(T_i)\}\}$
     (portanto, para todo $C \geq C_u[T, m]$, $FFD[T, C] \leq m$).
2. Fazer a **busca binária** em k iterações, sendo k um número qualquer maior que zero:

   $C_l(0) \leftarrow C_l[T, m]$;
   $C_u(0) \leftarrow C_u[T, m]$;
   $i \leftarrow 1$;
   while $i < k$ **do**
   $C \leftarrow [C_l(i-1) + C_u(i-1)]/2$.
   if $FFD[T, C] \leq m$ **then**
   $C_u(i) \leftarrow C$;
   $C_l(i) \leftarrow C_l(i-1)$;
   else
   $C_l(i) \leftarrow C$;
   $C_u(i) \leftarrow C_u(i-1)$;
   end if
   $i \leftarrow i + 1$;
   end while

Finalmente, a **razão de aproximação** desse algoritmo é dada por:

$$\frac{C_u(k)}{w^*} \geq \left(\tau_m + \left(\frac{1}{2}\right)^k\right) \tag{3.8}$$

onde τ_m é o **menor fator de expansão** do valor da solução ótima para o FFD (*First Fit Decreasing*) não usar mais que m caixas na execução, ou seja,

$$\tau_m = \min(\tau : FFD[T, \tau\, w^*] \leq m) \tag{3.9}$$

O **custo assintótico** do algoritmo é $O(n \log n + knm)$ se implementar o FFD (*First Fit Decreasing*) sem ordenar as tarefas, ou $O(n \log n + kn \log m)$ se ordená-las com algum algoritmo eficiente (CORMEN et al., 2024).

Podemos, agora, mostrar como o MULTIFIT é executado na prática. Considerando um exemplo onde aplicaremos quatro iterações do MULTIFIT ($k = 4$), com $T = \{T_1, T_2, T_3, T_4, T_5\}$ e $P = \{P_1, P_2\}$. A Tabela 4 a seguir mostra o custo de execução $\mu(T_i)$ de cada tarefa envolvida no escalonamento. No caso do MULTIFIT, ele somente se aplica para sistemas com tarefas independentes, portanto seu DAG (Grafo Acíclico Direcionado) é completamente desconexo, assim não precisando ser considerado no processo de escalonamento.

Executaremos esse algoritmo passo a passo (como mostra na Figura 6) para melhor visualização da busca binária, que se baseia no resultado de uma execução completa do algoritmo FFD (*First Fit Decreasing*) para decidir o novo espaço de busca.

Tabela 4 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo.

Tarefa	$\mu(T_i)$
T_1	2
T_2	8
T_3	3
T_4	4
T_5	7

Fonte: Autoria própria

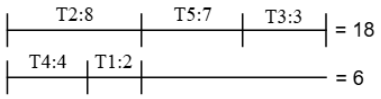
Figura 6 – Execução do MULTIFIT. Usando dos valores dados a k, T e P e com os custos descritos na tabela acima, é feita a execução do algoritmo MULTIFIT. Antes da busca binária, é dado um valor inicial a C_u e C_l, de acordo com as fórmulas para $C_u[T, m]$ e $C_l[T, m]$, e posteriormente a isso, são feitas quatro iterações de busca binária, resultando num makespan w para essa solução aproximada de 12 (é o valor que está em $C_u(4)$). As tarefas presentes na caixa1 e na caixa2 da execução do FFD da última iteração, preservando a ordem em que foram inseridas, devem ser as que serão escalonadas nas máquinas P_1 e P_2 do sistema dessa execução do MULTIFIT, respectivamente, para produzir o makespan $w = 12$.

$$C_l(0) = C_l[T, m] = C_l[\{T_1, T_2, T_3, T_4, T_5\}, 2] = \max\{\frac{24}{2}, 8\} = 12$$
$$C_u(0) = C_u[T, m] = C_u[\{T_1, T_2, T_3, T_4, T_5\}, 2] = \max\{2\frac{24}{2}, 8\} = 24$$

$i = 1$ $\quad C = \frac{12+24}{2} = 18$ $\quad FFD[T, 18] = 2 \leq m$:

$C_u(1) = C = 18,\ C_l(1) = C_l(0) = 12$

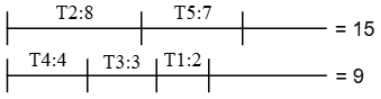
Execução do FFD[T,18]:



$i = 2$ $\quad C = \frac{12+18}{2} = 15$ $\quad FFD[T, 15] = 2 \leq m$:

$C_u(2) = C = 15,\ C_l(2) = C_l(1) = 12$

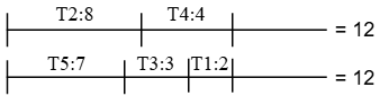
Execução do FFD[T,15]:



$i = 3$ $\quad C = \frac{12+15}{2} = 13$ $\quad FFD[T, 13] = 2 \leq m$:

$C_u(3) = C = 13,\ C_l(3) = C_l(2) = 12$

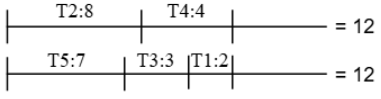
Execução do FFD[T,13]:



$i = 4$ $\quad C = \frac{12+13}{2} = 12$ $\quad FFD[T, 12] = 2 \leq m$:

$C_u(4) = C = 12,\ C_l(4) = C_l(3) = 12$

Execução do FFD[T,12]:



Fonte: Autoria própria.

3.2.1.4 *List Scheduling* para ordens intervalares de precedência

Os DAGs (Grafos Acíclicos Direcionados) que representam a precedência entre tarefas no problema de escalonamento apresentam formatos diversos, como de árvores, florestas ou árvores reversas (PAPADIMITRIOU; YANNAKAKIS, 1979). Além desses casos, existem DAGs com outras propriedades que valem a pena serem notadas, como os DAGs de precedência cujos **grafos de incomparabilidade** são de **corda**.

Para prosseguirmos com as definições desse algoritmo, é importante tomarmos ciência de dois conceitos:

Grafo de Incomparabilidade: O grafo de incomparabilidade de um certo grafo $P(V, A)$ é o grafo não direcionado que contém ligações entre as tarefas de V que não se encontram em A. Ou seja, $P(V, A)$ é um grafo, e $G(V, E)$ é seu grafo de incomparabilidade, de forma que, para todo v, $u \in V$, $(v, u) \in E \iff (v, u), (u, v) \notin A$.

Grafo de Corda: Um grafo G é de corda se, para cada **ciclo** envolvendo mais de 3 vértices, existir uma corda. Ou seja, G é de corda se, para cada ciclo de vértices de G $[v_1, v_2, \ldots, v_k], k \geq 4$, existir uma aresta $(v_i, v_j), j \neq i \pm 1 (mod\, k)$.

Os grafos de corda, por serem detentores de importantes propriedades algorítmicas, são aplicados para vários outros problemas em teoria de grafos, como mínima coloração, clique máxima, entre outros (PAPADIMITRIOU; YANNAKAKIS, 1979). Por isso, podem ter suas propriedades aproveitadas, também, para resolver instâncias do problema de escalonamento, como: dado n tarefas de **tempo unitário**, encontrar uma forma de escaloná-las em m máquinas de forma que todas sejam executadas respeitando a ordem de precedência de $P(V, A)$ e resultem num makespan mínimo.

O fato do grafo de incomparabilidade do DAG (Grafo Acíclico Direcionado) de uma ordem de precedência ser, também, um grafo de corda evidencia que a mesma é uma ordem intervalar (PAPADIMITRIOU; YANNAKAKIS, 1979), ou seja, o grafo de incomparabilidade desse DAG de precedência é um grafo intervalar, onde cada vértice (tarefa) pode ser representado por um intervalo nas reta dos valores reais, e cada aresta do grafo carrega o significado de que seus dois vértices são intervalos que se sobrepõem nessa reta. No contexto do problema de escalonamento, isso significa que esse grafo de incomparabilidade tem a propriedade de apontar quais tarefas podem ser executadas em paralelo nas máquinas do sistema. A partir disso, podemos desenvolver um algoritmo que gere soluções aproximadas em função dos intervalos que representam cada uma das tarefas:

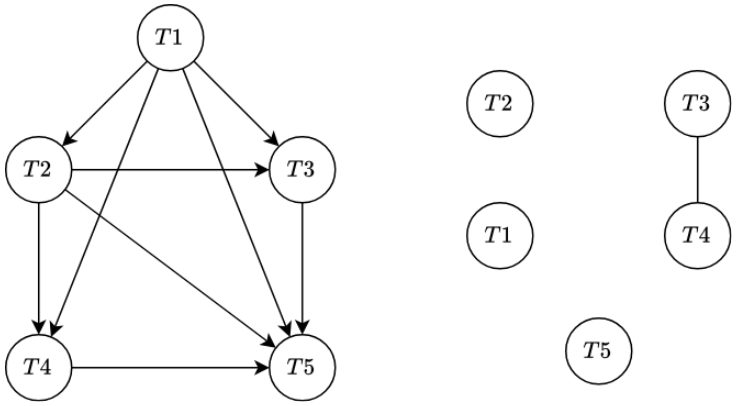
Passo 1. Ordenar as tarefas em ordem decrescente pelo **grau de saída** (número de arestas que sai do nó) dos nós que as representam no DAG (Grafo Acíclico Direcionado) de precedência.

Passo 2. Escalonar as tarefas de acordo com a ordem de prioridade antes estabelecida, respeitando também a disponibilidade das tarefas pela restrição de precedência.

O **custo** computacional desse algoritmo é $O(|V| + |A|)$, sendo V e A, respectivamente, o número de vértices e arestas do Grafo Acíclico Direcionado (DAG) de precedência $P(V, A)$. Esse algoritmo é comprovado como sendo um algoritmo correto, ou seja, sempre retorna um escalonamento válido, respeitando as restrições definidas (PAPADIMITRIOU; YANNAKAKIS, 1979).

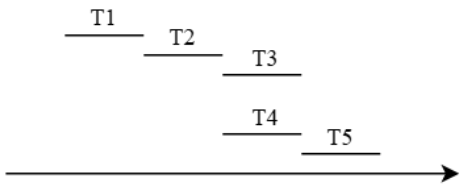
A Figura 9 traz um exemplo do escalonamento de acordo com o algoritmo descrito, cuja prioridade se baseia na Tabela 5, envolvendo cinco tarefas $T = \{T_1, T_2, T_3, T_4, T_5\}$ e duas máquinas $P = \{P_1, P_2\}$, com seu DAG (Grafo Acíclico Direcionado) apresentado na Figura 7, juntamente com seu grafo de incomparabilidade intervalar. A propriedade do grafo de incomparabilidade intervalar antes descrita se apresenta na Figura 8. Todas as tarefas do exemplo são de tempo de execução unitário (todas possuem custo de execução μ com valor 1).

Figura 7 – Grafo de Precedência $P(V, A)$ e seu Grafo de Incomparabilidade $G(V, E)$, contendo cinco tarefas (T_1, T_2, T_3, T_4 e T_5).



Fonte: Autoria própria.

Figura 8 – Ordem intervalar na reta dos números reais $\mathbb{R}$ representada pelo grafo de incomparabilidade intervalar de $P(V, A)$, $G(V, E)$. Observe como a estrutura do grafo intervalar mostrado na figura anterior nos mostra exatamente o comportamento do escalonamento das tarefas: T_3 e T_4 são vértices conectados em $G(V, E)$, mostrando que se sobrepõem na reta dos reais (e posteriormente estarão em paralelo no escalonamento), enquanto as outras tarefas não se conectam, portanto não se sobrepõem na reta dos reais (e posteriormente estarão executadas linearmente no escalonamento).



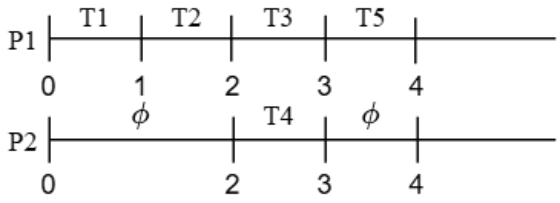
Fonte: Autoria própria.

Tabela 5 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, juntamente com o grau de saída de cada uma delas.

Tarefa	$\mu(T_i)$	grau de saída
T_1	1	4
T_2	1	3
T_3	1	1
T_4	1	1
T_5	1	0

Fonte: Autoria própria

Figura 9 – Execução do *List Scheduling* para ordens intervalares. A ordem de execução das tarefas L assume o valor de $[T_1, T_2, T_3, T_4, T_5]$. O escalonamento, conforme antes descrito, é feito envolvendo duas máquinas ($P = \{P_1, P_2\}$). A solução aproximada gerada possui makespan $w = 4$ e tempo ocioso total na segunda máquina $\phi_2 = 3$.



Fonte: Autoria própria.

3.2.2 Memória Distribuída

O paralelismo com memória distribuída é uma variação do problema de escalonamento que considera a **comunicação** entre as tarefas, tendo em vista que não compartilham a memória, portanto necessitando da transmissão de dados entre tarefas dependentes. Quando uma tarefa é finalizada, informações são passadas dessa tarefa às dependentes, tendo um custo adicional para o sistema.

Agora, o DAG (Grafo Acíclico Direcionado) de precedência é **ponderado** não somente nos **vértices** (que é o comum para representar o tempo de processamento das tarefas), mas também nas **arestas**, para representar o custo da comunicação de uma tarefa a outra.

Conforme descrito por Khan, Mccreary e Jones (1994), as principais heurísticas de aproximação que buscam soluções subótimas para ambientes de multiprocessamento podem ser divididas em três categorias:

1. **Heurísticas de Caminho Crítico**: Denominando que, para grafos acíclicos direcionados de precedência com pesos nos nós (custo de execução) e nas arestas (custo de comunicação), o caminho crítico é o caminho feito no grafo que representa a **sequência de execução mais longa** do sistema, e portanto é o limite mínimo do makespan desse sistema. O objetivo das heurísticas dessa categoria é justamente tentar reduzir o makespan total ao valor do makespan do caminho crítico. Aqui apresentaremos o *Modified Critical Path* (MCP) (WU; GAJSKI, 1990).

2. **Heurísticas de *List Scheduling***: São heurísticas que buscam encontrar o escalonamento ótimo associando **prioridades** às tarefas e escalonando-as de acordo com as mesmas. Nessa categoria são aplicadas extensões no *List Scheduling* para tentar obter melhores resultados, como por exemplo aplicando métodos gulosos, duplicação de tarefas, entre outros. Aqui abordaremos o *Earliest Task First Scheduling* (ETF) (HWANG et al., 1989).

3. **Método da Decomposição de Grafos**: É usado da **Teoria da Decomposição de Grafos** para obter o escalonamento ótimo, recorrendo à redução do DAG (Grafo Acíclico Direcionado) de precedência a uma hierarquia de **subgrafos** (grãos) que representa a relação de independência/dependência entre as tarefas. Aplicando os custos de comunicação e execução, pode-se determinar o tamanho dos grãos ideais para o escalonamento ótimo. Aqui apresentaremos o CLANS (MCCREARY; GILL, 1989), principal representante dessa categoria.

Justamente por não fazer distinção da relação entre as máquinas (se são idênticas ou não), é evidente que os algoritmos aqui apresentados se encaixam nos dois contextos, e serão novamente tratados (no caso do *Modified Critical Path* (MCP) e *Earliest Task First Scheduling* (ETF)) para o problema de escalonamento com máquinas heterogêneas (Seção 3.3.2).

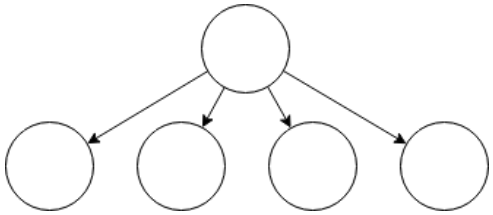
3.2.2.1 *Modified Critical Path* - MCP

O **MCP** de Wu e Gajski (1990) é uma modificação do **algoritmo de caminho crítico** apresentado por Sethi (1976), acrescentando uma estratégia que considera a **flexibilidade** que cada tarefa tem de ser executada em um certo ponto no tempo de execução, de forma que não atrapalhe a execução das outras tarefas.

Para compreender esse algoritmo, é necessário considerar os seguintes termos:

ASAP : É o **tempo mais cedo possível** em que cada tarefa pode ser executada. Para calculá-lo para cada tarefa, se executa o **ASAP *Binding***, atribuindo um $ASAP(T_i)$ para cada tarefa T_i. Simularemos o ASAP *Binding* para a sequência de tarefas $[T_1, T_2, T_3, T_4, T_5]$ com grafo de precedência representado na Figura 10.

Figura 10 – Exemplo de grafo acíclico direcionado de precedência



Fonte: Autoria própria

Passo 1. Atribui a T_1, **tarefa de entrada** do grafo, o menor tempo possível para ser executado, que no caso é zero por ser a primeira tarefa a ser executada.

Passo 2. Quando T_1 finaliza sua execução, envia as mensagens para suas tarefas dependentes, e para todas essas tarefas o seu **menor tempo possível** de execução é o marco de tempo da execução do sistema no qual as **informações** de T_1 **chegam** às tarefas. Esse passo, então, se repetiria para grafos que possuem mais tarefas e mais níveis que o do exemplo, até que todas as tarefas fossem visitadas.

ALAP : É o tempo **mais tardio possível** no qual uma tarefa pode ser executada. O processo de **ALAP *Binding*** é executado após o ASAP *Binding* de forma retrógrada (passando das tarefas de saída até as de entrada), e considerando o tempo mais tardio para escalonamento como o valor que define o $ALAP$ de cada tarefa, denominado $ALAP(T_i)$.

Intervalo de Movimento : O intervalo de movimento de uma tarefa é justamente o **intervalo de tempo** entre $ASAP(T_i)$ e $ALAP(T_i)$, ou seja, é o tempo que ela tem para ser executada sem atrasar a execução de outras tarefas.

Mobilidade: É o **comprimento** do intervalo de movimento, denominado por $M(T_i) = ALAP(T_i) - ASAP(T_i)$.

A execução do MCP (*Modified Critical Path*) consiste nos seguintes passos:

Passo 1. Executar o **ALAP *Binding*** e atribuir $ALAP(T_i)$ para cada nó do DAG (Grafo Acíclico Direcionado) de precedência.

Passo 2. Para cada nó T_i, criar uma **lista** $\Gamma(T_i)$ que contém o $ALAP$ de T_i e de todos os seus descendentes em ordem crescente. **Ordenar** todas as $\Gamma(T_i)$ em ordem lexicográfica, e a partir dela criar uma lista única de escalonamento L com todas as tarefas.

Passo 3. Escalona cada tarefa de L à primeira máquina livre. Assim que uma máquina se liberar, procura por outra tarefa livre em L, respeitando a ordem estipulada e a relação de precedência entre as tarefas.

O custo do primeiro passo é $O(n^2)$. Já o custo do segundo passo é $O(n^2 \log n)$, para gerar n listas, e $O(n^2 \log n)$ para ordená-las. Por fim, o terceiro passo tem custo linear, ou seja $O(n)$, **totalizando** $O(n^2 \log n)$.

A **razão de aproximação** desse algoritmo aproximado segue, também, o do algoritmo no qual foi inspirado, provada por Wu e Gajski (1990), e também por Kohler (1975):

$$\frac{w}{w^*} \leq 2 - \frac{1}{m} \tag{3.10}$$

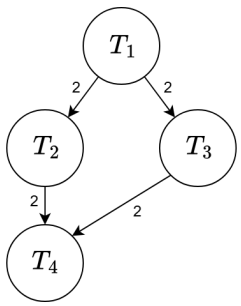
Podemos, então, apresentar um exemplo de aplicação do algoritmo MCP. Considerando que devemos escalonar quatro tarefas ($T = \{T_1, T_2, T_3, T_4\}$) em duas máquinas ($P = \{P_1, P_2\}$), a Tabela 6 representa os custos de execução de cada tarefa envolvida no escalonamento, e suas relações de precedência são representadas pelo DAG (grafo acíclico direcionado) da Figura 11. A Figura 12 mostra a aplicação do algoritmo MCP para o exemplo que foi construído.

Tabela 6 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo.

Tarefa	$\mu(T_i)$
T_1	2
T_2	3
T_3	5
T_4	1

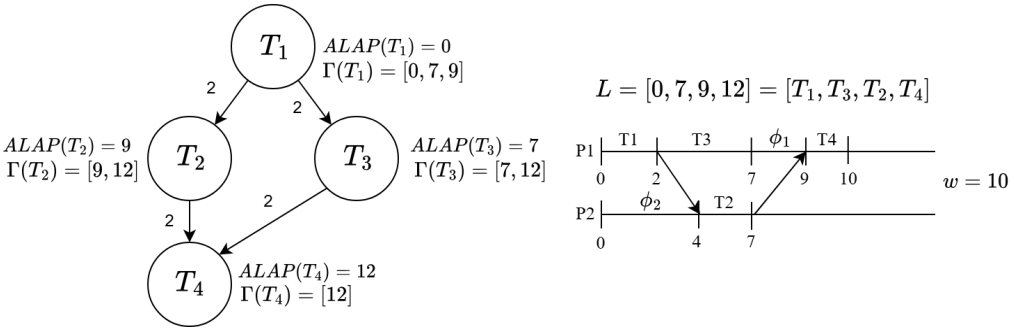
Fonte: Autoria própria

Figura 11 – Grafo acíclico direcionado de precedência das quatro tarefas do exemplo, apresentando seus custos de comunicação.



Fonte: Autoria própria

Figura 12 – Aplicação do algoritmo MCP para escalonar quatro tarefas ($T = \{T_1, T_2, T_3, T_4\}$) em duas máquinas ($P = \{P_1, P_2\}$) em um ambiente de memória distribuída. O makespan da solução aproximada w é 10, com tempo ocioso na primeira máquina ϕ_1 de 2 e segunda máquina ϕ_2 de 4.



Fonte: Autoria própria

3.2.2.2 *Earliest Task First Scheduling* - ETF

O algoritmo de aproximação **ETF** deriva do *List Scheduling* proposto por Graham (1966), alterando o método de ponderação da prioridade das tarefas, considerando as tarefas que podem ser escalonadas **o mais cedo possível** dentre as disponíveis (HWANG et al., 1989).

Para entender o funcionamento do ETF (*Earliest Task First Scheduling*), é necessário saber que a variável CM (*Current Moment*) representa o **tempo atual** do evento de escalonamento, enquanto NM (*Next Moment*) representa um **potencial** próximo marco de tempo que substituirá o valor atual de CM, marcando o menor tempo de finalização entre todas as tarefas escalonadas no CM atual como o próximo CM. Com isso, observaremos a execução do ETF:

Passo 1. Define um conjunto I para todas as **máquinas disponíveis** no tempo CM, define CM como 0, NM como ∞ e um conjunto A para todas as **tarefas de**

entrada, dado que uma tarefa de entrada é qualquer tarefa que não tiver tarefas que a precedem. Inicialmente, I contém todas as máquinas, e é definido o valor de $r(T_{entrada}, P_j)$ para cada tarefa de entrada $T_{entrada}$ como zero, onde $r(T_i, P_j)$ é o **tempo** para a última mensagem endereçada a T_i em P_j chegar. Esse cálculo inclui o tempo de finalização do processamento da tarefa precedente e o tempo de envio da mensagem. Para situações em que a memória é compartilhada entre as tarefas, o custo de envio da mensagem é desconsiderado.

Passo 2. Escolher o **par** tarefa livre T_i e máquina disponível P_j que gera o menor $r(T_i, P_j)$, e atribuir ao tempo de escalonamento de T_i - e_i - o maior valor entre CM e $r(T_i, P_j)$.

Passo 3. Decisão de escalonamento: se T_i antes escolhida tiver um tempo de escalonamento e_i menor ou igual a NM, significa que ela será escalonada nessa iteração. T_i é removida de A, assim como P_j de I. Assim, é calculado o seu tempo de finalização $C_i = e_i + \mu(T_i)$. NM assume o valor do menor tempo de conclusão C_i dentre as tarefas que estão escalonadas no momento. A decisão de escalonamento se repete para quantas tarefas tiver que satisfaçam as exigências feitas anteriormente.

Passo 4. Assim que **não há mais tarefas** com tempo de escalonamento menor que NM, CM assume o valor de NM, reseta I, atualiza A com as novas tarefas livres e $r(T_i, P_j)$ recebe $\max_{T_k \in pred(T_i)}(C_i + n(T_i, T_k) \cdot t(p(T_i), P_j))$ para cada nova tarefa T_i livre. $n(T_i, T_k)$ é o tempo de passar uma informação da tarefa "pai" T_i para a tarefa "filha" T_k, e $t(p(T_i), P_j)$ é o tempo de comunicação entre a máquina em que T_i está e P_j.

Passo 5. O algoritmo volta à **decisão de escalonamento**, até que não sobre nenhuma tarefa para ser executada. O **makespan final** está em CM.

O **custo de execução** desse algoritmo é $O(n^2m)$. A razão de aproximação desse algoritmo pode ser construída pelo limite que Graham (1966) prova para o *List Scheduling (LS)*, mas já que esse limite é para sistemas de memória compartilhada (Seção 3.2.1), é necessário um **adicional** nesse limite que represente a comunicação entre as tarefas do escalonamento do ETF (*Earliest Task First Scheduling*). Portanto, a razão de aproximação do ETF se dá pelo limite do LS de Graham (1966), em soma com o resultado da execução do **algoritmo C**.

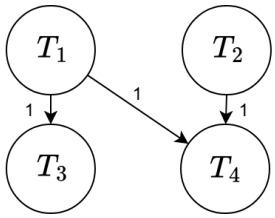
De forma descrita, o algoritmo C busca por momentos em que as máquinas ficaram **ociosas** no escalonamento do ETF, e busca por tarefas que se fossem transferidas para essas máquinas ociosas **não causaria atraso** na execução, tendo em vista que mudar de máquina altera o custo de comunicação entre essa tarefa e as tarefas que a precedem. Para cada tarefa que se encaixa nesse requisito, acumula-se numa variável C o **tempo**

de comunicação atualizado entre a tarefa transferida e as tarefas precedentes a ela. Portanto:

$$w \leq \left(2 - \frac{1}{m}\right) w^* + C \tag{3.11}$$

A fim de ilustrar o funcionamento desse algoritmo, considere a Figura 13 e Tabela 7 que apresentam, respectivamente, a estrutura de precedência entre quatro tarefas e seus custos de execução. Assim, a Figura 14 e a Figura 15 trazem o resultado da aplicação do algoritmo ETF (*Earliest Task First Scheduling*) para escalonar quatro tarefas em duas máquinas.

Figura 13 – Exemplo de grafo acíclico direcionado de precedência das tarefas do exemplo da aplicação do algoritmo *Earliest Task First* - ETF - com os custos de comunicação entre as tarefas.



Fonte: Autoria Própria

Tabela 7 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo.

Tarefa	$\mu(T_i)$
T_1	2
T_2	3
T_3	2
T_4	4

Fonte: Autoria própria

Figura 14 – Aplicação do algoritmo *Earliest Task First* - ETF em cinco iterações para decisão de escalonamento. Cada iteração define um momento no tempo de execução das tarefas nas máquinas. Nesse caso não há tempos de ociosidade das máquinas para justificar aplicação do algoritmo C. O makespan dessa solução é o valor final de $CM = 8$.

Inicialização: $CM = 0$, $NM = \infty$, $A = \{T_1, T_2\}$, $I = \{P_1, P_2\}$

CM = 0:

$e_1 = \max\{CM, 0\} = 0 \quad C_1 = e_1 + \mu(T_1) = 2$
$e_2 = \max\{CM, 0\} = 0 \quad C_2 = e_2 + \mu(T_2) = 3$ $\quad NM = 2$

$e_1 \leq NM$ T1 será escalonado nessa iteração, em P1 $\quad A = \{T_2\}$, $I = \{P_2\}$ $\quad CM = NM = 2$
$e_2 \leq NM$ T2 será escalonado nessa iteração, em P2 $\quad A = \emptyset$, $I = \emptyset$

CM = 2: $\quad A = \{T_3\}$, $I = \{P_1\}$ No tempo atual, somente T1 finalizou sua execução e liberou P1

$e_3 = \max\{CM, 3\} = 3 \quad C_3 = e_3 + \mu(T_3) = 5$ $\quad NM = 3$
$\qquad\qquad\qquad\qquad\quad C_2 = e_2 + \mu(T_2) = 3$

$e_3 \leq NM$ T3 será escalonado nessa iteração, em P1 $\quad A = \emptyset$, $I = \emptyset$ $\quad CM = NM = 3$

CM = 3: $\quad A = \{T_4\}$, $I = \{P_2\}$ No tempo atual, somente T2 finalizou sua execução e liberou P2

$e_4 = \max\{CM, 4\} = 4 \quad C_4 = e_4 + \mu(T_4) = 8$ $\quad NM = 5$
$\qquad\qquad\qquad\qquad\quad C_3 = e_3 + \mu(T_3) = 5$

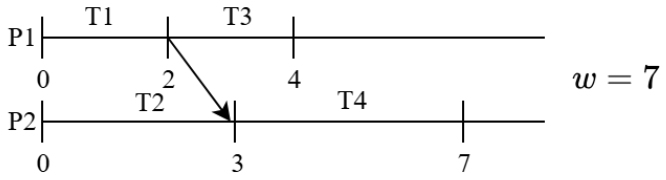
$e_4 \leq NM$ T4 será escalonado nessa iteração, em P2 $\quad A = \emptyset$, $I = \emptyset$ $\quad CM = NM = 5$

CM = 5: T3 finaliza sua execução e libera P1

CM = 8: T4 finaliza sua execução e libera P2

Fonte: Autoria Própria

Figura 15 – Ilustração da aplicação do algoritmo *Earliest Task First* - ETF, conforme o exemplo descrito.



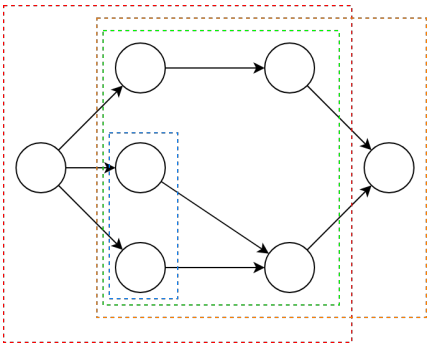
Fonte: Autoria Própria

3.2.2.3 CLANS

O algoritmo **CLANS** apresentado por McCreary e Gill (1989) é um algoritmo que usa de algumas teorias aplicadas a grafos para encontrar uma aproximação da solução ótima. A estratégia dessa heurística é conseguir separar as tarefas representadas no grafo em **grãos**, onde cada grão é um **subgrafo** do grafo acíclico direcionado (DAG) de precedência das tarefas do sistema, de forma que cada subgrafo desses contém tarefas que possam ser executadas **sequencialmente numa mesma máquina**, e assim associar grãos a máquinas.

A partir dessa técnica, só nos resta um método para identificar os melhores grãos e atribuí-los corretamente às máquinas do sistema. A estratégia de McCreary e Gill (1989) é formar **clãs**, que são também subgrafos do grafo acíclico direcionado (DAG) de precedência que se comportam da mesma forma: precedem e sucedem os mesmos nós, ou seja, a comunicação com esse clã se dá sempre da mesma forma. Esses clãs, posteriormente, serão analisados e diferenciados para grãos ou conjunto de grãos.

Figura 16 – Clãs formados do grafo acíclico direcionado dado. Se pegarmos como exemplo o clã $\{2, 3\}$, os nós 2 e 3 pertencem a esse clã pois ambos são filhos de 1 e ambos só possuem 7 como filho, sendo possível abstraí-los como um único nó no grafo.

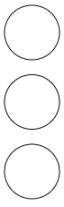


Fonte: Autoria Própria

Dentre os vários **clãs** gerados do grafo acíclico direcionado (DAG) de precedência, os mesmos podem ser divididos em três categorias:

Independente: Clãs independentes são grafos que **não possuem arestas**, como na Figura 17. Ou seja, as tarefas contidas nesse grafo **são independentes**.

Figura 17 – Clã Independente

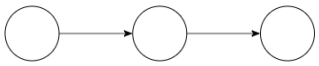


Fonte: Autoria Própria

Linear: Clãs lineares são grafos onde, para cada par de tarefas T_i, T_j, ou $T_i \prec T_j$, ou $T_j \prec T_i$, como na Figura 18

Primitivo: Clãs Primitivos são grafos que não respeitam as regras de nenhuma das classificações antes citadas, e portanto, se fossem subdivididos em mais clãs, resul-

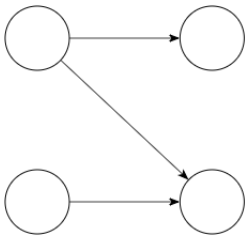
Figura 18 – Clã Linear



Fonte: Autoria Própria

tariam somente em clãs unitários (contendo um único nó). Um exemplo de um clã primitivo é representado na Figura 19.

Figura 19 – Clã Primitivo



Fonte: Autoria Própria

Os clãs previamente separados, agora, serão organizados em uma **árvore de hierarquia** onde os nós-folha são **tarefas**, os nós centrais são **clãs** e as arestas são a **hierarquia** entre esses nós. A hierarquia é dada de forma que um clã ou tarefa é subordinada a outro clã se estiver **contido** no mesmo. Depois, é feito um cálculo do **custo** dos nós dessa árvore de baixo para cima, seguindo os três tipos de clãs possíveis:

Linear: É um tipo de nó onde a paralelização não pode ocorrer, logo seus nós filhos devem ser executados em **sequência**, e o custo desse nó resulta na soma do custo de execução de cada nó filho.

Primitivo: É um nó que agrega todos os seus filhos num mesmo **grão**. O custo desse nó também é a soma do custo dos filhos.

Independente: É o tipo de nó que abre espaço para **paralelização**, e para que um número arbitrário de seus filhos possam compor um mesmo grão. Portanto, é um nó da árvore que pode ser composto por mais de um grão. Como a execução agora é paralela, é preciso considerar novas variáveis no cálculo do custo do nó:

- Caso o custo de executar os nós desse clã seja menor se for feito em sequência quando comparado com paralelizar, tendo em vista que os custos de comunicação das tarefas em serialização é desconsiderado, o clã inteiro será um único

grão, portanto será executado em sequência. Custo é somente a soma do custo de execução das tarefas.

- Caso contrário, será feito o processo de agregação: O nó será dividido em p **subgrupos** $K_1, K_2, \ldots, K_p$, $1 \leq p \leq m$, onde cada um será atribuído a uma máquina P_j diferente. O **custo de comunicação** só é considerado se o par de tarefas dependentes **não** estiver na mesma máquina. Cada subgrupo K_i é composto por tarefas $T_{1,i}$, $T_{2,i}$ $\ldots, T_{k,i}$.
- O **makespan** de K_i é a soma dos tempos de execução dos nós pertencentes a ele, em conjunto com os custos de comunicação de saída ou entrada que forem necessários. O **custo de entrada** de informações para esse subgrupo é $\max\{y_{k,i}\}$, onde $y_{k,i}$ é o custo da informação entrar na tarefa $T_{k,i}$ do subgrupo K_i, e o **custo de saída** de comunicação para esse subgrupo é $\sum y'_{k,i}$, onde $y'_{k,i}$ é o custo de saída de informações da tarefa $T_{k,i}$ do subgrupo K_i.

A estratégia desse algoritmo para encontrar uma solução próxima da ótima é criar os subgrupos dos nós independentes de forma que **minimize** os seus **três custos** envolvidos: o custo de execução de cada subgrupo, o custo de entrada e de saída de informações no subgrupo. Para um caso de sistema onde todas as tarefas possuem o **mesmo tempo de execução** μ, **mesmo custo de entrada** de informações y e de **saída** de informações y', é mais fácil de obter uma agregação em grãos ótima das tarefas dos nós independentes. Para esse caso, o custo total do clã independente será $\left(\lceil \frac{\Phi}{|K_i|} \rceil\right) y + (\Phi - |K_i|)y' + \mu|K_i|$ considerando que Φ é o número de nós desse clã, e o $|K_i|$ ótimo é $\sqrt{\frac{y\Phi}{\mu - y'}}$ quando $\mu > y'$, e 1 quando $\mu \leq y'$. Para casos gerais, o custo total do clã é limitado por no máximo $\sum_{1 \leq i \leq \Phi}(y_i + y'_i) + \max_{1 \leq i \leq \Phi} \mu(T_i)$, que é o custo de separar uma máquina para cada tarefa do clã (subgrupos são de tamanho 1).

O algoritmo CLANS, então, para determinar os grãos, faz:

Passo 1. Transformar o grafo acíclico direcionado (DAG) de precedência em um **árvore hierárquica** de fluxo de dados.

Passo 2. Atribuir **custos de execução** aos nós-folha do grafo acíclico direcionado (DAG) de precedência.

Passo 3. Calcular os **custos** dos nós internos.

Passo 4. Determinar os **grãos**:

- Se o custo do nó for **igual** à soma dos custos de cada nó filho, todos os nós filhos pertencem ao mesmo grão, devendo ser atribuídos à mesma máquina, com senso sobre qual máquina está livre/com menos custo acumulado.

- Já se o custo do nó for **menor** que a soma dos custos de cada nó filho (ou seja, é o caso do **nó clã independente** cujos filhos não são todos nós-folha), separa os nós filhos em grãos. Cada grão é um dos subgrupos mencionados, e cada um deles deve ser atribuído a uma máquina diferente, com senso sobre qual máquina está livre/com menos custo acumulado.

Já o algoritmo de escalonamento em si consiste em varrer a árvore por altura reversa (da maior altura à menor), e escalonar seus nós filhos de acordo com o tipo de clã no qual aquele grão se originou.

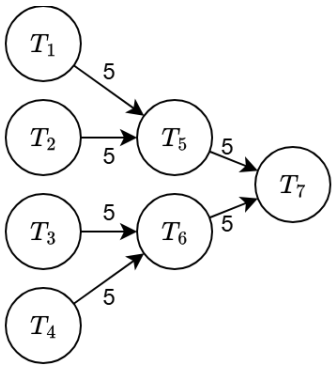
As Figuras 20, 21 e 22 são um exemplo da aplicação do CLANS para escalonar sete tarefas em duas máquinas. Os custos das tarefas estão na Tabela 8.

Tabela 8 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo.

Tarefa	$\mu(T_i)$
T_1	3
T_2	3
T_3	3
T_4	3
T_5	1
T_6	1
T_7	2

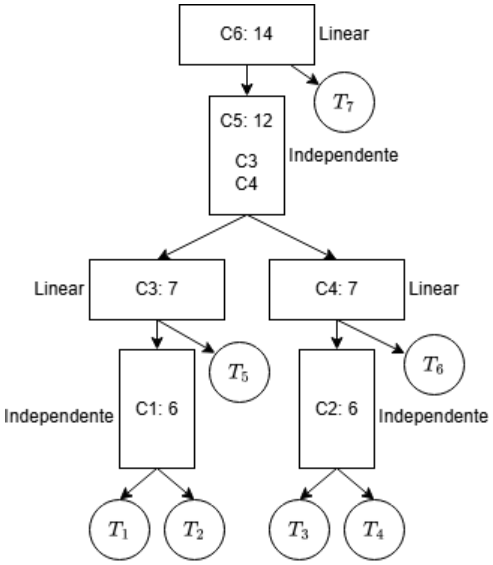
Fonte: Autoria própria

Figura 20 – Grafo acíclico direcionado (DAG) de precedência das sete tarefas envolvidas no exemplo, com seus custos de comunicação.



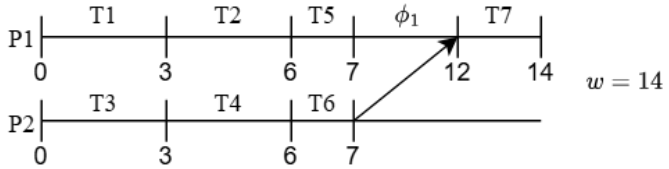
Fonte: Autoria Própria

Figura 21 – Árvore de hierarquia formada pelo grafo acíclico direcionado (DAG) de precedência antes apresentado, onde C1, C2..., C6 são clãs. A forma na qual os clãs organizam as tarefas determinam como as mesmas serão escalonadas posteriormente.



Fonte: Autoria Própria

Figura 22 – Escalonamento após formar os grãos, baseando-se na árvore de hierarquia. Veja que os clãs lineares dispõem seus grãos filhos sequencialmente numa mesma máquina, e os grãos independentes possuem a liberdade de paralelizá-los (como no caso do clã C5) ou de sequenciá-los (como no caso dos clãs C1 e C2), conforme vantagens de custo de execução. O makespan da solulção gerada w é 14, com tempo ocioso na primeira máquina ϕ_1 de 5.



Fonte: Autoria Própria

O **custo** de execução do algoritmo CLANS se comporta de forma **próxima** a $O(n^3)$ (MCCREARY; GILL, 1989), dado que esse é o custo para montar a árvore hierárquica. Não foi encontrado nesse estudo uma razão de aproximação para esse algoritmo.

3.3 Processamento em Máquinas Paralelas não Idênticas

Esse leque de variantes, agora, considera uma nova restrição quanto às máquinas. O problema de escalonamento de máquinas não idênticas prevê que cada máquina envolvida no problema possui um tempo de execução **diferente** para uma mesma tarefa.

Segundo Ibarra e Kim (1977), para esse problema, cada máquina P_j possui uma **função própria** de tempo de execução $\mu_j, 1 \leq j \leq m$ de forma que, para executar uma tarefa $T_i, 1 \leq i \leq n$, o custo é $\mu_j(T_i)$. Esse custo não possui o compromisso de ter o mesmo valor em cada máquina, e além disso, a distribuição desses custos pode ser **uniforme** (custos entre máquinas para uma mesma tarefa é proporcional) ou **não relacionada** (não há padrão fácil de se encontrar entre as funções de custo). Todos os algoritmos que serão apresentados a seguir valem para ambas as distribuições.

3.3.1 Memória Compartilhada

Essa variante tem exatamente as mesmas condições que a sua versão para **máquinas idênticas** (Seção 3.2.1), com o adicional da restrição das máquinas não serem idênticas.

O problema, agora, consiste em encontrar a melhor forma de organizar as tarefas entre as máquinas, de forma que resulte no menor makespan possível, considerando que uma mesma tarefa tem tempos diferentes de execução, dependendo de qual máquina ela é escalonada.

Serão apresentados, aqui, uma coleção de algoritmos aproximados apresentados por Ibarra e Kim (1977), que são nada mais que **adaptações** de algoritmos conhecidos para aproximações no contexto de máquinas idênticas (Seção 3.2.1), como o *List Scheduling* e o LPT *Scheduling* (*Longest Processing Time Scheduling*). Esses algoritmos, também, consideram que as tarefas a serem escalonadas são independentes entre si.

3.3.1.1 *A-Scheduling*

O ***A-Scheduling*** opta por uma **escolha gulosa** ao escalonar uma certa tarefa a uma certa máquina, e é de forma **arbitrária**, tendo em vista que as tarefas envolvidas não serão ordenadas numa lista de escalonamento. Esse algoritmo se dá da seguinte forma:

Passo 1. Cada máquina P_j atribui a si uma tarefa T_i de forma a minimizar o seu próprio makespan, agora considerando que cada máquina tem seu próprio μ_j.

Passo 2. Retorna o resultado $w = \max\{\sum_{T \in L_j} \mu_j(T)\}$, onde L_j é a lista de tarefas que foram escolhidas para serem executadas em P_j.

A **complexidade** do *A-Scheduling* é linear $O(n)$, visto que faz decisões gulosas, mas não ordena as tarefas em lista, ou seja, não tem custo adicional para ordenação das tarefas. A **razão de aproximação** da solução de ambos os algoritmos se comporta da mesma forma:

$$\frac{w}{w^*} \leq m \tag{3.12}$$

Como o *A-Scheduling* é uma adaptação do **LS (*List Scheduling*) arbitrário**, se as máquinas voltassem a ser idênticas, o limite máximo para a razão de aproximação voltaria a ser a proposta em Graham (1966) para *List Scheduling* e apresentada na Seção 3.2.1.

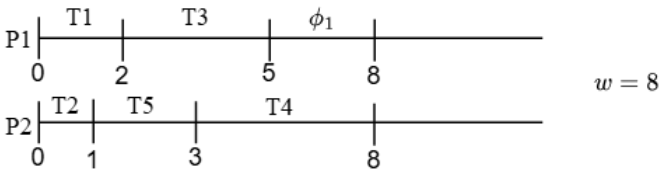
Podemos aplicar esse algoritmo para escalonar cinco tarefas em duas máquinas não-idênticas. na Tabela 9 está os custos das tarefas do exemplo para cada máquina, e a Figura 23 apresenta o escalonamento feito conforme o algoritmo descrito.

Tabela 9 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$
T_1	2	3
T_2	2	1
T_3	3	1
T_4	4	5
T_5	5	2

Fonte: Autoria própria

Figura 23 – Escalonamento do exemplo usando *A-Scheduling*. O makespan da solução w é 8, com tempo ocioso na primeira máquina ϕ_1 de 3.



Fonte: Autoria Própria

3.3.1.2 *B-Scheduling*

O algoritmo ***B-Scheduling*** usa da base de processamento do *A-Scheduling* antes visto. Sua estratégia de execução é a seguinte:

Ordena as tarefas T_i em uma lista de escalonamento L em ordem decrescente pelo menor custo de execução da tarefa dentre as máquinas.

Escalona a lista L como no *A-Scheduling*.

A **complexidade** do *B-Scheduling* é $O(n \log n)$, pois o custo de escalonamento é linear $O(n)$, como visto no *A-Scheduling*, e a ordenação das tarefas é feita em tempo $O(n \log n)$. A **razão de aproximação** de f com f^*, dada como solução ótima, é provada por Ibarra e Kim (1977) como sendo:

$$\frac{w}{w^*} \leq m \tag{3.13}$$

O *B-Scheduling*, como descrito por Ibarra e Kim (1977), é um algoritmo que pode ser reduzido ao algoritmo **LPT** (*Longest Processing Time Scheduling*) no contexto de máquinas idênticas (Seção 3.2.1). Portanto, quando o B-Schedule é aplicado no contexto de **máquinas idênticas** (Seção 3.2), sua razão de aproximação fica subordinada ao limite máximo do LPT.

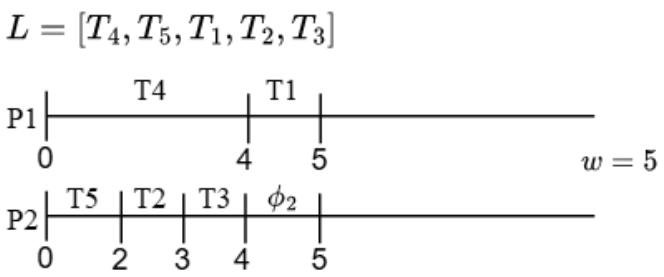
Podemos aplicar esse algoritmo para escalonar cinco tarefas em duas máquinas não-idênticas. na Tabela 10 está os custos das tarefas do exemplo para cada máquina, e a Figura 24 apresenta o escalonamento feito conforme o algoritmo descrito.

Tabela 10 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$
T_1	1	3
T_2	3	1
T_3	2	1
T_4	4	5
T_5	5	2

Fonte: Autoria própria

Figura 24 – Escalonamento do exemplo usando *B-Scheduling*. O makespan da solução w é 5, com tempo ocioso na segunda máquina ϕ_2 de 1.



Fonte: Autoria Própria

3.3.1.3 *C-Scheduling*

O algoritmo ***C-Scheduling*** é um método alternativo do *B-Scheduling*. Sua estratégia de execução se dá por:

Passo 1. Ordena as tarefas T_i em uma lista de escalonamento L em ordem decrescente pelo maior custo de execução da tarefa dentre as máquinas.

Passo 2. Escalona a lista L como no *A-Scheduling*.

A **complexidade** do *C-Scheduling* é a mesma do *B-Scheduling*, $O(n \log n)$, já que o custo de escalonamento é linear $O(n)$ e a ordenação das tarefas é feita em tempo $O(n \log n)$. A **razão de aproximação** de w com w^*, dada como solução ótima, é provada por Ibarra e Kim (1977) como sendo:

$$\frac{w}{w^*} \leq m \tag{3.14}$$

O *C-Scheduling*, como descrito em Ibarra e Kim (1977), também é um algoritmo que pode ser reduzido ao algoritmo **LPT** (*Longest Processing Time Scheduling*) no contexto de máquinas idênticas (Seção 3.2.1), e quando é aplicado nesse contexto, sua razão de aproximação também fica subordinada ao limite máximo do LPT.

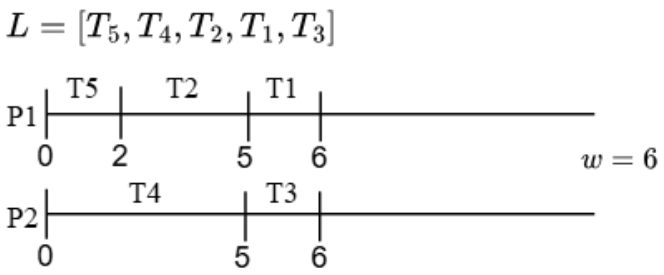
Podemos aplicar esse algoritmo para escalonar cinco tarefas em duas máquinas não-idênticas. na Tabela 11 está os custos das tarefas do exemplo para cada máquina, e a Figura 25 apresenta o escalonamento feito conforme o algoritmo descrito.

Tabela 11 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$
T_1	1	3
T_2	3	1
T_3	2	1
T_4	4	5
T_5	5	2

Fonte: Autoria própria

Figura 25 – Escalonamento do exemplo usando *C-Scheduling*. O makespan da solução w é 6.



Fonte: Autoria Própria

3.3.1.4 *D-Scheduling*

O ***D-Scheduling*** se diverge dos outros algoritmos apresentados em estratégia de escalonamento: usa do **escalonamento dinâmico** (escolher em tempo de execução a tarefa a escalonar) para minimizar o tempo de finalização global do sistema. Com isso, a execução se dá por:

Passo 1. escolher uma tarefa T_i para uma máquina P_j, de forma que esse escalonamento resulte no **menor makespan** do sistema para aquela iteração (observa w iterativamente).

Passo 2. retornar $w = \max\{w_j\}, 1 \leq j \leq m$.

O **custo assintótico** do *D-Scheduling* é $O(n^2)$, tendo em vista que, para cada i-ésima tarefa escalonada, ele procura pela próxima tarefa adequada para escalonamento dentre as $n - i$ tarefas que sobraram. O limite máximo da **razão de aproximação**, no entanto, não muda quando comparado com os outros algoritmos apresentados, como provado por Ibarra e Kim (1977):

$$\frac{w}{w^*} \leq m \tag{3.15}$$

No caso do *D-Scheduling*, esse limite é o **melhor possível** para $m = 2$, mas não é o mais adequado para $m \geq 3$ como nos outros algoritmos (IBARRA; KIM, 1977). O *D-Scheduling* pode ser reduzido a um outro algoritmo se convertermos o contexto para máquinas idênticas (Seção 3.2.1): o **SPT *Scheduling*** (*Shortest Processing Time Scheduling*), que possui a mesma estratégia do LPT *Scheduling* (*Longest Processing Time Scheduling*), mas ordena as tarefas em ordem crescente de tempo de processamento μ. Quando é aplicado no contexto de **máquinas idênticas** (Seção 3.2), sua razão de aproximação também fica subordinada ao limite máximo do SPT, que é idêntico ao do *List Scheduling*, como apresentado por Ibarra e Kim (1977) e Graham (1966):

$$\frac{w}{w^*} \leq 2 - \frac{1}{m} \tag{3.16}$$

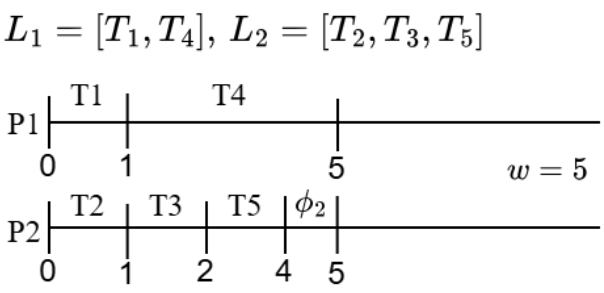
Podemos aplicar esse algoritmo para escalonar cinco tarefas em duas máquinas não-idênticas. na Tabela 12 está os custos das tarefas do exemplo para cada máquina, e a Figura 26 apresenta o escalonamento feito conforme o algoritmo descrito.

Tabela 12 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$
T_1	1	3
T_2	3	1
T_3	2	1
T_4	4	5
T_5	5	2

Fonte: Autoria própria

Figura 26 – Escalonamento do exemplo usando *D-Scheduling*. O makespan da solução w é 5, com tempo ocioso na segunda máquina ϕ_2 de 1.



Fonte: Autoria Própria

3.3.1.5 *E-Scheduling*

O ***E-Scheduling*** é muito semelhante ao *D-Scheduling*, ainda mais que os dois são algoritmos de **escalonamento dinâmico** para minimizar o makespan global do sistema, mas o *E-Scheduling* usa da seguinte estratégia: ele escalona as tarefas nas máquinas observando diretamente o tempo de finalização da tarefa escolhida para escalonamento, resultando no menor tempo de finalização global. Com isso, a execução se dá por:

Escalonar uma tarefa T_i à uma máquina P_j, de forma que esse escalonamento resulte no **menor tempo de finalização** para T_i perante todas as máquinas.

Retornar $w = \max\{w_j\}, 1 \leq j \leq m$.

O **custo assintótico** do *E-Scheduling* é $O(n^2)$, já que usa a mesma estratégia de escalonamento que o *D-Scheduling*. O limite máximo da **razão de aproximação**, como provado por Ibarra e Kim (1977) é:

$$\frac{w}{w^*} \leq m \tag{3.17}$$

O *E-Scheduling* pode ser reduzido ao **LPT** (*Longest Processing Time Scheduling*), assim como o *B-Scheduling* e o *C-Scheduling*. Quando aplicado no contexto de **máquinas idênticas** (Seção 3.2), sua razão de aproximação fica subordinada ao limite máximo do LPT.

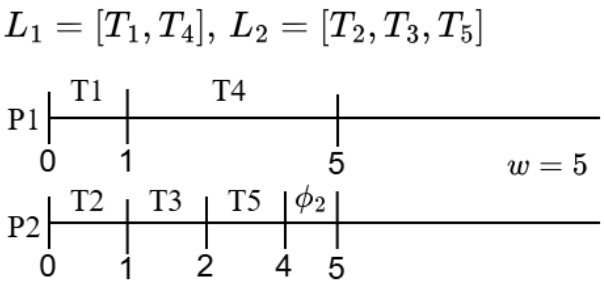
Podemos aplicar esse algoritmo para escalonar cinco tarefas em duas máquinas não-idênticas. na Tabela 13 está os custos das tarefas do exemplo para cada máquina, e a Figura 27 apresenta o escalonamento feito conforme o algoritmo descrito.

Tabela 13 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$
T_1	1	3
T_2	3	1
T_3	2	1
T_4	4	5
T_5	5	2

Fonte: Autoria própria

Figura 27 – Escalonamento do exemplo usando *E-Scheduling*. O makespan da solução w é 5, com tempo ocioso na segunda máquina ϕ_2 de 1.



Fonte: Autoria Própria

3.3.1.6 *F-Scheduling*

O ***F-Scheduling*** foi projetado com o objetivo de possuir um **melhor limite máximo** para $m = 2$ que o dos algoritmos já apresentados, e como o foco é para $m = 2$, o algoritmo procura soluções para escalonamento que envolva **somente duas máquinas**. Esse algoritmo usa da lógica de **corrigir** o escalonamento inicialmente projetado para tentar reduzir o tempo de finalização da máquina com o maior tempo de finalização:

Passo 1. Projetar um **escalonamento completo**, onde as tarefas T_i são associadas à máquina P_1 ou P_2, cujo makespan é o menor possível.

Passo 2. Se o tempo de finalização das duas máquinas é o mesmo, o escalonamento é ótimo, mas caso uma das máquinas possua tempo ocioso, o escalonamento pode não ser ótimo. Para corrigir isso, o algoritmo procura **reescalonar** algumas tarefas da máquina com maior tempo (suponha P_1) para a com menor tempo (suponha P_2), de forma que **reduza ao máximo** o tempo em P_1 e **minimize o aumento** de tempo em P_2:

- Reescalonar T_i de P_1 para P_2, sendo $\frac{\mu_1(T_i)}{\mu_2(T_i)}$ o menor possível.
- Repetir isso até que o makespan de P_1 fique menor que o de P_2.

Passo 3. retornar $w = \max\{w_j\}, 1 \leq j \leq 2$.

A **complexidade** da primeira etapa é $O(n)$, e a segunda é $O(m \log m)$ pelo custo de ordenação e reescalonamento das tarefas. Como $O(n) + O(m \log m) \leq O(n \log n)$, a complexidade total é $O(n \log n)$ (IBARRA; KIM, 1977). Enquanto isso, Ibarra e Kim (1977) prova que a **razão de aproximação** para esse algoritmo é:

$$\frac{w}{w^*} = \frac{\sqrt{5}+1}{2} \tag{3.18}$$

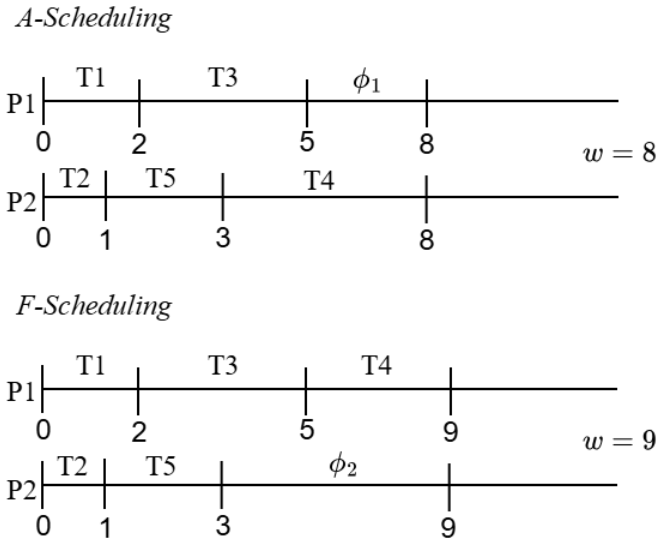
Para exemplificar a aplicação do *F-Scheduling*, podemos aproveitar o escalonamento feito pelo *A-Scheduling* e aplicar a correção conforme descrito. A tabela 14 traz de volta os custos descritos para o exemplo do *A-Scheduling* e a Figura 28 mostra o escalonamento das cinco tarefas em duas máquinas não-idênticas, conforme o descrito para esse algoritmo.

Tabela 14 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$
T_1	2	3
T_2	2	1
T_3	3	1
T_4	4	5
T_5	5	2

Fonte: Autoria própria

Figura 28 – Escalonamento do exemplo usando *F-Scheduling*. As correções feitas para minimizar o tempo ocioso que existia na segunda máquina anteriormente tinha menção de reduzir o tempo ocioso das máquinas, mas, como podemos ver, nesse caso houve aumento no makespan final do sistema e do tempo ocioso das máquinas. O makespan da solução w é 9, com tempo ocioso na segunda máquina ϕ_1 de 6.



Fonte: Autoria Própria

3.3.2 Memória Distribuída

O paralelismo com memória distribuída para o problema de escalonamento em máquinas heterogêneas deve lidar tanto com as **condições** antes impostas para o paralelismo

com memória distribuída em máquinas idênticas (Seção 3.3.1) quanto com as **restrições** para sistemas com máquinas heterogêneas (Seção 3.3).

Primeiramente, vale citar que os algoritmos **MCP** (*Modified Critical Path*) e o **ETF** (*Earliest Task First*), antes tratados para máquinas idênticas, também se encaixam para encontrar soluções subótimas em problemas com máquinas heterogêneas, mas além desses, também serão comentados os algoritmos **HEFT** e **CPOP** de Topcuoglu, Hariri e Wu (2002), que não possuem razões de aproximação formalmente provadas, justamente por serem heurísticas, mas são algoritmos mais novos que representam o curso que está tomando atualmente os novos projetos de algoritmos para o problema de escalonamento, ao lado dos algoritmos de busca guiada.

3.3.2.1 *Modified Critical Path* - MCP

O **MCP** de Wu e Gajski (1990) é uma modificação do algoritmo de caminho crítico apresentado por Sethi (1976), acrescentando uma heurística que considera a flexibilidade que cada tarefa tem de ser executada em um certo ponto no tempo de execução, de forma que não atrapalhe a execução das outras tarefas.

Assim como antes citado (Seção 3.2.2), o algoritmo possui o mesmo método de execução, sendo pontualmente modificado na escolha da máquina para a execução da tarefa no **passo 3**:

Passo 1. Executar o **ALAP *Binding*** e atribuir $ALAP(T_i)$ para cada nó do grafo acíclico direcionado (DAG) de precedência.

Passo 2. Para cada nó T_i, criar uma **lista** $\Gamma(T_i)$ que contém o $ALAP$ de T_i e de todos os seus descendentes em ordem crescente. **Ordenar** todas as $\Gamma(T_i)$ em ordem lexicográfica, e a partir dela criar uma lista única de escalonamento L com todas as tarefas.

Passo 3. Escalona, em ordem, cada tarefa de L à máquina que a executa mais rapidamente, respeitando a função de tempo de execução individual a cada máquina. Assim que uma máquina se liberar, procura por outra tarefa livre em L, respeitando a ordem estipulada.

A **complexidade** do algoritmo não muda, sendo ela $O(n^2 \log n)$, assim como a sua **razão de aproximação** não altera de comportamento, como demonstrado por Wu e Gajski (1990):

$$\frac{w}{w^*} \leq 2 - \frac{1}{m} \tag{3.19}$$

3.3.2.2 *Earliest Task First Scheduling* - ETF

O algoritmo **ETF** (*Earliest Task First Scheduling*) de Hwang et al. (1989), assim como o MCP (*Modified Critical Path*), foi anteriormente citado no contexto de máquinas idênticas (Seção 3.2.2), mas também é um algoritmo efetivo para o contexto de **máquinas heterogêneas**. O código do ETF não precisa ser alterado em nada na mudança de contexto, só vale lembrar agora que, no **Passo 3**, a escolha do escalonamento levará em consideração que as máquinas possuem funções de tempo diferentes, já que a escolha é feita baseando-se em C_i, e $C_i = e_i + \mu(T_i)$.

A **complexidade** e **razão de aproximação** desse algoritmo se repetem, sendo $O(n^2m)$ e $w \leq \left(2 - \frac{1}{m}\right) w^* + C$, sendo C o resultado da execução do algoritmo C, antes descrito (Seção 3.2.2).

3.3.2.3 *Heterogeneous Earliest Finishing Time* - HEFT

O algoritmo **HEFT** atribui prioridade de escalonamento a uma tarefa de acordo com a **contribuição** que ela dará ao makespan (e como essa contribuição se propaga - caminho crítico) do sistema ao ser escalonada. Essa prioridade é calculada com base no grafo acíclico direcionado (DAG) de precedência, resultando em dois **rankings** para a tarefa perante as outras do grafo: *Upward Rank* e/ou *Downward Rank*, definida por Topcuoglu, Hariri e Wu (2002) como:

- **Custo médio entre duas tarefas:** Não é um custo considerado médio pelo fato de ser feita a média entre os custos puros de transmissões $data_{i,j}$ entre duas tarefas T_i e T_j, já que só existe um custo de comunicação entre elas. Para ambos algoritmos HEFT e CPOP, são considerados, também, taxas de transmissão $B_{j,k}$ entre pares de máquinas P_j e P_k $(1 \leq k \leq m)$ e custo de abertura de comunicação X_j para cada máquina P_j. Então, o custo médio de comunicação entre tarefas é calculado usando a média das taxas de transmissão $\bar{B}$ e dos custos de abertura das máquinas $\bar{X}$:

$$\bar{c_{i,j}} = \bar{X} + \frac{data_{i,j}}{\bar{B}} \tag{3.20}$$

- ***Upward Rank* de T_i:** É o **custo médio** de execução do **caminho crítico** formado de T_i até a tarefa de saída do grafo acíclico direcionado (DAG) de precedência.

$$rank_u(T_i) = \bar{\mu}(T_i) + \max_{T_j \in succ(T_i)} \{\bar{c}_{i,j} + rank_u(T_j)\} \tag{3.21}$$

  Onde $\bar{\mu}(T_i)$ é a média de todos tempos de execução para T_i perante todas as máquinas. Para T_{saida}, $rank_u(T_{saida}) = \bar{\mu}(T_{saida})$

- ***Downward Rank* de T_i:** É o **custo médio** de execução **caminho crítico** formado da tarefa de entrada do DAG de precedência e comunicação (tarefa que não depende

de nenhuma outra tarefa) até T_i.

$$rank_d(T_i) = \max_{T_j \in pred(T_i)} \{rank_d(T_j) + \bar{\mu}(T_j) + \bar{c}_{i,j}\} \tag{3.22}$$

Para $T_{entrada}$, $rank_d(T_{entrada}) = 0$.

Sendo que o caminho crítico (CP) é um caminho formado a partir de um nó de partida do DAG (Grafo Acíclico Direcionado) de precedência, e que é construído com os nós filhos que possuem o maior ranking. Além disso, é a **sequência de execução mais demorada** que começa na tarefa de início do caminho, e por ser irredutível (é sequencial), se torna o limite mínimo do makespan de qualquer sequência de execução que comece com essa tarefa.

Além disso, é importante conhecer as variáveis EST e EFT:

- ***Earliest Execution Starting Time*:** É o tempo **mais cedo possível** que uma tarefa pode ser **escalonada** para execução em uma certa maquina. Para uma tarefa de entrada, para qualquer máquina, seu EST é zero.

$$EST(T_i, P_j) = \max\{\, (tempo\, em\, que\, P_j\, se\, libera\,), \max_{T_k \in pred(T_i)} \{AFT(T_k) + c_{i,k}\}\} \tag{3.23}$$

  Onde $AFT(T_i)$ é o tempo em que T_i **de fato** acabou de ser executada no sistema, e $c_{i,j}$ é o custo de comunicação de uma tarefa T_i para uma tarefa T_j.

- ***Earliest Execution Finishing Time*:** É o tempo **mais cedo possível** para uma tarefa ser **finalizada** em uma certa máquina.

$$EFT(T_i, P_j) = \mu(T_i) + EST(T_i, P_j) \tag{3.24}$$

O HEFT é composto por **duas fases** principais:

Definição das prioridades: Ordenar as tarefas em ordem **descrescente por** $rank_u$, com empates resolvidos de forma aleatória, significando que é feita uma **ordenação topológica** que respeita as relações de **dependência** de comunicação entre as tarefas e a **prioridade** estipulada.

Seleção de processador: Passo 1. Seleciona a primeira tarefa da ordem topológica gerada e calcula o EFT da tarefa com todas as máquinas, analisando, também, a possibilidade de "encaixar"essa tarefa em janelas de ociosidade dessas máquinas. Essa técnica se chama **política de inserção** para escalonamento.

Passo 2. Escalona a tarefa à máquina que lhe oferece o menor EFT.

A **complexidade** do HEFT depende do número de relações de precedência presentes entre as tarefas do sistema, ou seja, o tamanho de A no grafo acíclico direcionado (DAG) $P(V, A)$ de precedência, e no número de máquinas, concluindo $O(|A|\, m)$.

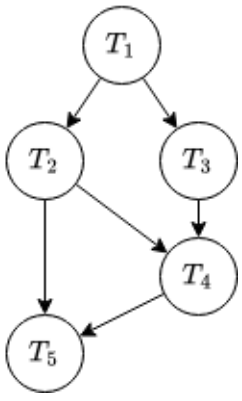
Para mostrar o funcionamento do algoritmo HEFT, podemos aplicá-lo a um exemplo, escalonando cinco tarefas em duas máquinas. A Tabela 15 mostra os custos das tarefas para cada máquina, assim como o custo médio de comunicação $\bar{c}_{i,j}$ de cada tarefa. A Figura 29 mostra a relação de precedência e comunicação entre as tarefas através de um DAG de precedência, e a Figura 30 mostra o escalonamento resultante do algoritmo aplicado no exemplo.

Tabela 15 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina, e o custo médio de comunicação $\bar{c}_{i,j}$ de cada tarefa.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$	$\bar{c}_{i,j}$
T_1	2	1	2
T_2	1	2	2
T_3	2	3	2
T_4	3	3	2
T_5	4	5	2

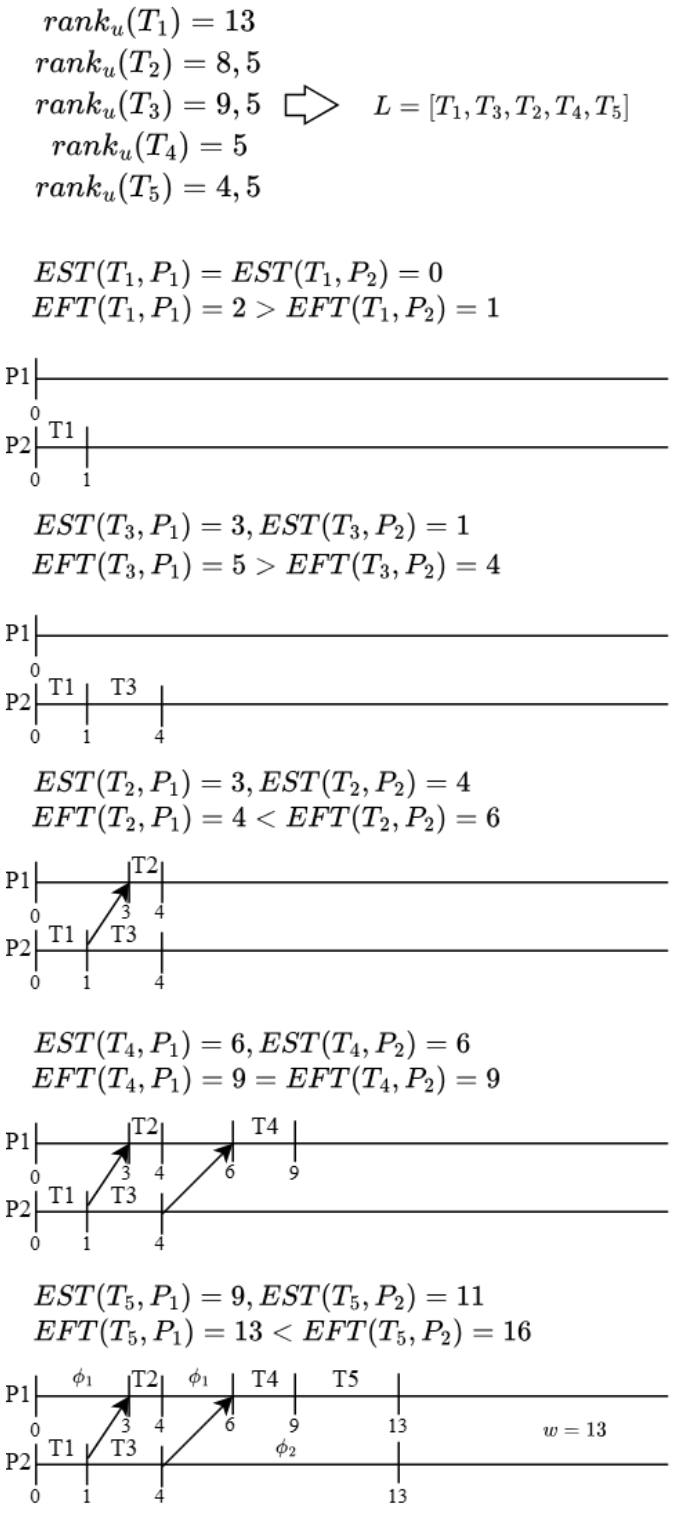
Fonte: Autoria própria

Figura 29 – Grafo acíclico direcionado (DAG) de precedência das cinco tarefas envolvidas no exemplo.



Fonte: Autoria Própria

Figura 30 – Aplicação do algoritmo HEFT no exemplo antes descrito. Nesse caso, não foi possível aplicar política de inserção para reduzir os tempos ociosos nas máquinas envolvidas. o makespan da solução w é 13, com tempo ocioso na primeira máquina ϕ_1 de 5, e na segunda máquina ϕ_2 de 9.



Fonte: Autoria Própria

3.3.2.4 *Critical Path on a Processor* - CPOP

O CPOP de Topcuoglu, Hariri e Wu (2002) usa dos **rankings** das tarefas como **prioridade** de escalonamento assim como o HEFT. Além disso, usam do caminho crítico do grafo acíclico direcionado (DAG) para **minimizar o makespan** do sistema, visto que o tempo de executar as tarefas do caminho crítico (CP) do grafo acíclico direcionado (DAG) é o menor tempo possível para executar o sistema como um todo, e se houver atrasos nas tarefas inclusas nesse caminho, esse atraso impacta diretamente no makespan total.

Para definir quais tarefas serão inclusas no caminho crítico, o CPOP define uma prioridade que é composta pela soma do *Upward Rank* e do *Downward Rank*:

$$prioridade(T_i) = rank_u(T_i) + rank_d(T_i) \tag{3.25}$$

Simbolizando que as tarefas mais importantes para esse algoritmo são as que apresentam muitas conexões, além de um custo alto de comunicação e de processamento. Para montar o **caminho crítico (CP)** do grafo acíclico direcionado (DAG), primeiro se associa a própria **tarefa de entrada** $T_{entrada}$ do DAG como tarefa do caminho crítico, e a partir daí se elege uma tarefa que **sucede** a recentemente escolhida pela **prioridade** antes estabelecida, até chegar em uma **tarefa de saída** T_{saida}.

A **máquina** do sistema que executa o caminho crítico com o menor custo é **reservada** somente para o caminho crítico, enquanto as outras tarefas são distribuídas entre as máquinas restantes. Esse escalonamento é feito usando uma lista de prioridade cujas tarefas com maior valor de $prioridade(T_i)$ vêm primeiro e, assim como no HEFT, considera a **política de inserção** para escalonamento.

A **complexidade** desse algoritmo é $O(|A|\, m)$, onde A é o número de arestas no grafo acíclico direcionado (DAG) de precedência $P(V, A)$ e n é o número de tarefas.

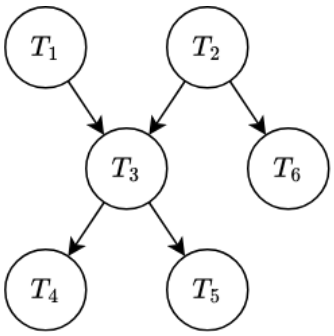
Agora exemplificaremos uma aplicação do algoritmo CPOP, escalonando seis tarefas em duas máquinas. A Tabela 16 mostra os custos das tarefas para cada máquina, assim como o custo médio de comunicação $\bar{c}_{i,j}$ de cada tarefa. A Figura 31 mostra a relação de precedência e comunicação entre as tarefas através de um DAG de precedência, e a Figura 32 mostra o escalonamento resultante do algoritmo aplicado no exemplo.

Tabela 16 – Tabela com o custo $\mu(T_i)$ para cada tarefa do exemplo, para cada máquina, e o custo médio de comunicação $\bar{c}_{i,j}$ de cada tarefa.

Tarefa	$\mu_1(T_i)$	$\mu_2(T_i)$	$\bar{c}_{i,j}$
T_1	10	6	5
T_2	8	10	5
T_3	6	6	5
T_4	5	9	5
T_5	6	2	5
T_5	1	1	5

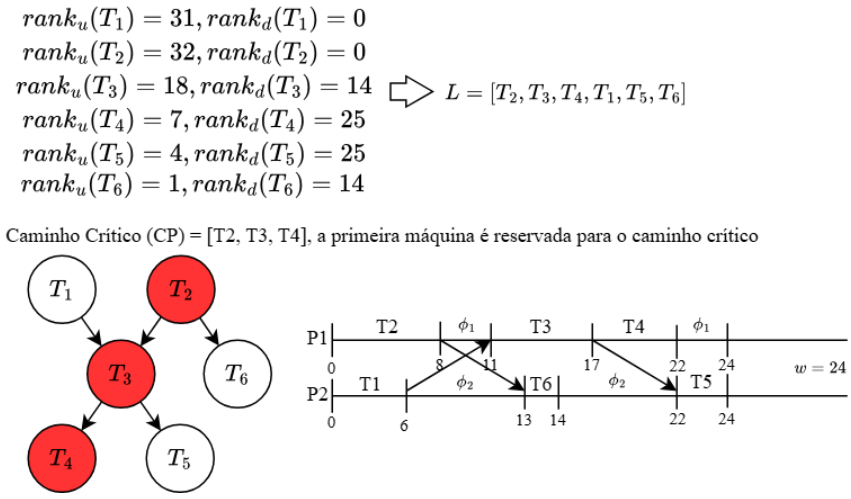
Fonte: Autoria própria

Figura 31 – Grafo acíclico direcionado (DAG) de precedência das seis tarefas envolvidas no exemplo.



Fonte: Autoria Própria

Figura 32 – Aplicação do algoritmo CPOP no exemplo antes descrito. Foi aplicada a política de inserção na sexta tarefa, pois, sem ferir as restrições de precedência das tarefas, era possível encaixá-la em um momento ocioso da segunda máquina. o makespan da solução w é 24, com tempo ocioso na primeira máquina ϕ_1 de 5, e na segunda máquina ϕ_2 de 15.



Fonte: Autoria Própria

3.4 Considerações Finais

Neste capítulo, foram apresentados os principais algoritmos para diferentes variações do problema de escalonamento de tarefas. A seguir, são apresentadas as tabelas comparativas entre esses algoritmos, agrupados pelas variantes que tratam, de modo a resumir as características de cada um deles.

Na Tabela 17 é apresentado o algoritmo EDD (*Earliest Due Date rule*) que é o principal algoritmo de aproximação para o contexto de escalonamento em uma única máquina. Na Tabela 18 são apresentados os algoritmos para o caso de máquinas paralelas idênticas e, na Tabela 19, os algoritmos para máquinas não-idênticas.

Tabela 17 – Algoritmos aproximados para escalonamento em uma única máquina.

Algoritmo	Seção	Complexidade	Aproximação ($\alpha_{\max}/\alpha^*_{max} \leq$)
EDD	3.1	$O(n \log n)$	2

Fonte: Autoria própria

Tabela 18 – Algoritmos aproximados para escalonamento paralelo em máquinas idênticas. A razão de aproximação dos algoritmos LS (*List Scheduling*) para ordens intervalares de precedência e CLANS não é especificada por seus autores.

Algoritmo	Seção	Complexidade	Aproximação ($w/w^* \leq$)	Memória
List Scheduling	3.2.1	$O(n \log n)$	$2 - \frac{1}{m}$	Compartilhada
LPT	3.2.1	$O(n \log n)$	$\frac{4}{3} - \frac{1}{3m}$	Compartilhada
MULTIFIT	3.2.1	$O(n \log n + kn \log m)$	$\tau_m + \frac{1}{2}^k$	Compartilhada
LS ordens intervalares	3.2.1	$O(\|V\| + \|A\|)$	-	Compartilhada
MCP	3.2.2	$O(n^2 \log n)$	$2 - \frac{1}{m}$	Distribuída
ETF	3.2.2	$O(n^2 m)$	$(2 - \frac{1}{m}) + C$	Distribuída
CLANS	3.2.2	$O(n^3)$	-	Distribuída

Fonte: Autoria própria

Tabela 19 – Algoritmos aproximados para escalonamento paralelo em máquinas não idênticas. A razão de aproximação dos algoritmos HEFT e CPOP não é especificada por seus autores.

Algoritmo	Seção	Complexidade	Aproximação ($w/w^* \leq$)	Memória
A-Scheduling	3.3.1	$O(n)$	m	Compartilhada
B-Scheduling	3.3.1	$O(n \log n)$	m	Compartilhada
C-Scheduling	3.3.1	$O(n \log n)$	m	Compartilhada
D-Scheduling	3.3.1	$O(n^2)$	m para $m = 2$	Compartilhada
E-Scheduling	3.3.1	$O(n^2)$	m	Compartilhada
F-Scheduling	3.3.1	$O(n \log n)$	$= \frac{\sqrt{5}+1}{2}$ para $m = 2$	Compartilhada
MCP	3.3.2	$O(n^2 \log n)$	$2 - \frac{1}{m}$	Distribuída
ETF	3.3.2	$O(n^2 m)$	$(2 - \frac{1}{m}) + C$	Distribuída
HEFT	3.3.2	$O(\|A\| m)$	-	Distribuída
CPOP	3.3.2	$O(\|A\| m)$	-	Distribuída

Fonte: Autoria própria

4 Conclusão

Este trabalho teve como objetivo analisar individualmente e expor algoritmos e heurísticas de aproximação aplicadas às principais variantes do problema de escalonamento, fornecendo vários estilos de aproximações da solução ótima em tempo hábil, quando comparado com algoritmos exatos.

A partir da revisão bibliográfica e da análise de diferentes abordagens, foi possível observar a relação entre variantes do problema de escalonamento e algoritmos de aproximação, e a importância de abordagens flexíveis e adaptativas no campo da otimização. Espera-se que o que aqui foi exposto sirva de base para futuras pesquisas e aplicações em problemas reais.

Por fim, vale destacar que há espaço para aprofundamentos, como a inclusão de variantes mais específicas, testar a versatilidade de mais algoritmos de aproximação além das que as já analisadas nesse quesito, quando se trata de adaptá-las para outras variantes, e a abordagem de técnicas mais novas na busca de soluções próximas da ótima, como as metaheurísticas, algoritmos de busca guiada, entre outros.

Referências

COFFMAN JR, E. G.; GAREY, M. R.; JOHNSON, D. S. An application of bin-packing to multiprocessor scheduling. **SIAM Journal on Computing**, v. 7, n. 1, p. 1–17, 1978. <https://doi.org/10.1137/0207001>. Citado 4 vezes nas páginas 27, 28, 29 e 30.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos**: Teoria e prática. 4. ed. Barueri, SP: Editora LTC, 2024. 912 p. Citado 5 vezes nas páginas 23, 27, 28, 29 e 31.

EPSTEIN, L. Load balancing. In: KAO, M.-Y. (Ed.). **Encyclopedia of Algorithms**. Boston, MA: Springer US, 2008. p. 457–459. <https://doi.org/10.1007/978-0-387-30162-4_206>. Citado 2 vezes nas páginas 16 e 26.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability**: A guide to the theory of NP-Completeness. San Francisco, CA: W. H. Freeman and Company, 1979. 338 p. (A Series of Books in the Mathematical Sciences). Citado 3 vezes nas páginas 16, 21 e 25.

GOLDBARG, E. F. G.; GOLDBARG, M. C.; LUNA, H. P. L. **Otimização Combinatória e Meta-heurísticas**: Algoritmos e aplicações. Barueri, SP: Editora LTC, 2015. 416 p. Citado 2 vezes nas páginas 18 e 20.

GRAHAM, R. L. Bounds for certain multiprocessing anomalies. **The Bell System Technical Journal**, v. 45, n. 9, p. 1563–1581, nov. 1966. <https://doi.org/10.1002/j.1538-7305.1966.tb01709.x>. Citado 9 vezes nas páginas 16, 18, 25, 26, 27, 39, 40, 49 e 53.

_____. Bounds on multiprocessing timing anomalies. **SIAM Journal on Applied Mathematics**, v. 17, n. 2, p. 416–429, mar. 1969. <https://doi.org/10.1137/0117039>. Citado 2 vezes nas páginas 16 e 26.

HWANG, J.-J.; CHOW, Y.-C.; ANGER, F. D.; LEE, C.-Y. Scheduling precedence graphs in systems with interprocessor communication times. **SIAM Journal on Computing**, v. 18, n. 2, p. 244–257, 1989. <https://doi.org/10.1137/0218016>. Citado 4 vezes nas páginas 16, 36, 39 e 58.

IBARRA, O. H.; KIM, C. E. Heuristic algorithms for scheduling independent tasks on nonidentical processors. **Journal of the ACM**, v. 24, n. 2, p. 280–289, abr. 1977. <https://doi.org/10.1145/322003.322011>. Citado 8 vezes nas páginas 16, 48, 50, 51, 52, 53, 54 e 55.

KHAN, A. A.; MCCREARY, C. L.; JONES, M. S. A comparison of multiprocessor scheduling heuristics. **International Conference on Parallel Processing**, v. 2, p. 243–250, 1994. <https://doi.org/10.1109/ICPP.1994.19>. Citado 2 vezes nas páginas 16 e 35.

KOHLER, W. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. **IEEE Transactions on Computers**, C-24, n. 12, p.

1235–1238, dez. 1975. <https://doi.org/10.1109/T-C.1975.224171>. Citado na página 38.

LEE, C.-Y.; HWANG, J.-J.; CHOW, Y.-C.; ANGER, F. D. Multiprocessor scheduling with interprocessor communication delays. **Operations Research Letters**, v. 7, n. 3, p. 141–147, 1988. <https://doi.org/10.1016/0167-6377(88)90080-6>. Citado na página 16.

MCCREARY, C.; GILL, H. Automatic determination of grain size for efficient parallel processing. **Communications of the ACM**, v. 32, n. 9, p. 1073–1078, set. 1989. <https://doi.org/10.1145/66451.66454>. Citado 5 vezes nas páginas 16, 36, 42, 43 e 48.

MERTENS, S. The easiest hard problem: Number partitioning. In: PERCUS, A.; ISTRATE, G.; MOORE, C. (Ed.). **Computational complexity and statistical physics**. New York, NY: Oxford University Press, 2006. p. 125–140. Citado na página 21.

PAPADIMITRIOU, C. H.; YANNAKAKIS, M. Scheduling interval-ordered tasks. **SIAM Journal on Computing**, v. 8, n. 3, p. 405–409, 1979. <https://doi.org/10.1137/0208031>. Citado 3 vezes nas páginas 16, 32 e 33.

ROBERT, Y. Task graph scheduling. In: PADUA, D. (Ed.). **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p. 2013–2025. <https://doi.org/10.1007/978-0-387-09766-4_42>. Citado 2 vezes nas páginas 16 e 18.

SETHI, R. Algorithms for minimal length schedules. In: COFFMAN JR, E. G. (Ed.). **Computer and Job-Shop Scheduling Theory**. New York, NY: John Wiley & Sons, 1976. p. 51–100. Citado 2 vezes nas páginas 36 e 57.

SIPSER, M. **Introduction to the Theory of Computation**. 3. ed. Boston, MA: Cengage Learning, 2013. 458 p. Citado na página 20.

TOPCUOGLU, H.; HARIRI, S.; WU, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. **IEEE Transactions on Parallel and Distributed Systems**, v. 13, n. 3, p. 260–274, mar. 2002. <https://doi.org/10.1109/71.993206>. Citado 4 vezes nas páginas 16, 57, 58 e 62.

WILLIAMSON, D. P.; SHMOYS, D. B. **The Design of Approximation Algorithms**. Cambridge, MA: Cambridge University Press, 2011. 518 p. Citado 4 vezes nas páginas 16, 20, 22 e 23.

WU, M.-Y.; GAJSKI, D. Hypertool: a programming aid for message-passing systems. ieee trans parallel distrib syst. **IEEE Transactions on Parallel and Distributed Systems**, v. 1, n. 3, p. 330–343, jul. 1990. <https://doi.org/10.1109/71.80160>. Citado 4 vezes nas páginas 16, 36, 38 e 57.