Busca Local Eficiente para Problemas de Escalonamento da Produção em Máquinas Paralelas com Penalidades por Antecipação e Atraso

Arthur Kramer, Anand Subramanian
Departamento de Engenharia de Produção - Universidade Federal da Paraíba
Centro de Tecnologia, Campus I - Bloco G, Cidade Universitária, 58051-970, João Pessoa, PB
arthurhfrk@gmail.com, anand@ct.ufpb.br

RESUMO

Este trabalho trata dos problemas de escalonamento da produção em máquinas paralelas com penalidades por antecipação e atraso. A metodologia proposta consiste na generalização da abordagem desenvolvida por Ergun e Orlin (2006), na qual reduz o tempo de avaliação das vizinhanças *swap*, *insertion* e *twist* de $\mathcal{O}(n^3)$ para $\mathcal{O}(n^2)$ para o problema $1||\sum_{j=1}^{n} w_j T_j$. Essa generalização se dá tanto no âmbito das vizinhanças quanto no âmbito do problema. O método proposto foi incorporado em uma meta-heurística baseada no *Iterated Local Search* e testado em instâncias da literatura para diferentes problemas. Os resultados obtidos pela meta-heurística, em termos de tempo de execução, demostram que a generalização do método proposto por Ergun e Orlin (2006) foi realizada com sucesso.

PALAVRAS CHAVE. Busca local eficiente, Escalonamento da produção, Antecipação e atraso

Área principal: Metaheurísticas, PO na Administração e Gestão da Produção

ABSTRACT

This paper deals with the parallel machines total weighted earliness and tardiness scheduling problems. The aim of this work is to generalize the proposed methodology by Ergun e Orlin (2006), which reduces the complexity of searching the swap, insertion and twist neighborhoods from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$ for the $1||\sum_{j=1}^{n} w_j T_j$ problem. That generalization occurs both in neighborhoods and problem scopes. The proposed method was embedded on a Iterated Local Search based meta-heuristic and tested in benchmark instances available in the literature. The results obtained by the meta-heuristic, in terms of execution time, shows that the generalization of Ergun e Orlin (2006) methodology was successful.

KEYWORDS. Efficient Local Search, Scheduling, Earliness and Tardiness

Main Area: Metaheuristics, OR in Administration & Production Management

1. Introdução

Problemas de *scheduling* vêm sendo bastante estudados nos últimos 50 anos, recebendo atenção considerável por muitos pesquisadores em todo o mundo. São problemas que estão relacionados à alocação de recursos para a realização de um determinado conjunto de tarefas ao longo de um horizonte temporal.

Motivados pela complexidade desses problemas que estão presentes na vida real e são enfrentados por diferentes tipos de empresas, uma enorme quantidade de variantes e de métodos de resolução foram propostos desde a década de 1950 (Brucker e Knust, 2006). Logo, pode-se afirmar que os problemas de *scheduling* compõem uma das mais importantes áreas no campo da Pesquisa Operacional (PO), conforme pode ser visto no *survey* desenvolvido por Potts e Strusevich (2009). Dentre esses, está o problema de escalonamento da produção (*production scheduling*), um problema clássico de Otimização Combinatória (OC).

Uma área particular de problemas de escalonamento surge em ambientes produtivos que seguem a filosofia *Just-In-Time* (JIT). Estes são comumente chamados de *earliness-tardiness scheduling problems* (problema de escalonamento da produção com penalidades por antecipação e atraso), onde penalidades são aplicadas caso o processamento de uma tarefa seja finalizado antes ou depois de sua data de entrega estipulada. Tais problemas são, normalmente, $\mathcal{NP}$-difíceis, uma vez que eles incluem como caso particular o problema de escalonamento da produção em uma máquina, com penalidades ponderadas por atraso ($1||\sum w_jT_j$ de acordo com a classificação ($\alpha \mid \beta \mid \gamma$) proposta por Graham *et al*. (1979)), que é conhecido por ser $\mathcal{NP}$-difícil (Lenstra *et al*., 1977).

Resolver esses problemas de maneira ótima é uma tarefa extremamente difícil. Logo, uma alternativa natural é a aplicação de métodos (meta-)heurísticos com o objetivo de obter soluções viáveis de alta qualidade em um tempo computacional aceitável. Grande parte das meta-heurísticas utilizadas para a resolução dos problemas de escalonamento da produção são baseados em busca local, etapa que geralmente mais requer tempo de execução. Com esse propósito, alguns autores (Ergun e Orlin, 2006; Ibaraki *et al*., 2008; Kedad-Sidhoum e Sourd, 2010; Liao *et al*., 2012) desenvolveram novos métodos para a avaliação das soluções nas vizinhanças.

Para o $1||\sum w_jT_j$, Ergun e Orlin (2006) propuseram um algoritmo capaz de explorar as vizinhanças *swap*, *insertion* e *twist* em $O(n^2)$. Esse algoritmo utiliza estruturas de dados auxiliares que permitem a avaliação de uma solução na vizinhança em $O(1)$ amortizado, onde o pré-processamento das estruturas auxiliares é realizado em $O(n^2)$, definindo a complexidade da vizinhança. Liao *et al*. (2012) desenvolveram um procedimento para o $1|s_{ij}|\sum w_jT_j$ utilizando conceitos similares aos de Ergun e Orlin (2006), exigindo $O(n^2 \log n)$ operações para pré-processar as estruturas auxiliares necessárias para a avaliação das vizinhanças *swap*, *insertion* e *twist* em tempo constante. Essas complexidades são alcançadas apenas quando o tipo de problema não permite a inserção de tempo ocioso entre o processamento das tarefas. Nesse último caso, a metodologia proposta por Ibaraki *et al*. (2008) pode ser aplicada, resolvendo simultaneamente os problemas de escalonamento das tarefas e de definição dos tempos ótimos de início das tarefas (*timing problem*).

Este trabalho estende a metodologia proposta por Ergun e Orlin (2006) para tratar problemas que envolvam múltiplas máquinas paralelas não-relacionadas, com o objetivo de minimizar a soma ponderada dos atrasos e antecipações sem permitir a inserção de tempo ocioso entre as tarefas, denotado por $R||\sum w'_jE_j + w_jT_j$, no caso mais geral. Essa extensão também envolve a generalização das vizinhanças *insertion* e *swap*, permitindo a remoção/inserção de blocos de vários tamanhos, além da aplicação em vizinhanças que consideram movimentos de trocas entre duas máquinas.

2. Descrição do Problema

Seja $J = \{1, \ldots, n\}$ o conjunto das tarefas a serem sequenciadas em um conjunto $M = \{1, \ldots, m\}$ de máquinas paralelas não-relacionadas. Para cada $j \in J$, sejam p_j^k, d_j, w'_j e w_j seus

tempo de processamento na máquina $k \in M$, data de entrega, peso de penalização por antecipação e peso de penalização por atraso, respectivamente. O objetivo é minimizar $\sum w'_j E_j + w_j T_j$, onde $E_j = \max\{d_j - C_j, 0\}$ e $T_j = \max\{C_j - d_j, 0\}$ representam o tempo em que a tarefa $j \in J$ foi antecipada e atrasada, respectivamente, que dependem dos seus tempos de término C_j. De acordo com a notação $(\alpha|\beta|\gamma)$ proposta por Graham *et al*. (1979), este problema pode ser escrito como $R||\sum w'_j E_j + w_j T_j$.

Um grande número de problemas surgem como casos especiais do problema descrito anteriormente, incluindo problemas bastante conhecidos para o ambiente de uma máquina, tais como $1||\sum w_j T_j$ e $1||\sum w'_j E_j + w_j T_j$, que podem ser eficientemente resolvidos pelo algoritmo de Tanaka e Fujikuma (2012) para instâncias com até 200 tarefas (até 300 tarefas para o $1||\sum w_j T_j$). Apesar do método proposto ser capaz de resolver estes problemas, seu desempenho é inferior quando comparado ao método exato proposto por Tanaka e Fujikuma (2012). Na realidade, devido à excelente performance desse método exato, é difícil conceber heurísticas com desempenho equivalente à esse método exato, mesmo para instâncias de grande dimensões.

A Tabela 1 lista alguns dos principais problemas que aparecem como casos especiais do $R||\sum w'_j E_j + w_j T_j$, os quais o método proposto neste trabalho é capaz de resolver, incluindo os casos descritos anteriormente. Nessa tabela não são explicitados os problemas com pesos unitários, pois eles são casos particulares das versões com pesos não-unitários.

Tabela 1: Problemas considerados

Uma máquina	Máquinas idênticas	Máquinas uniformes	Máquinas não-relacionadas
$1\|\|\sum w_j T_j$	$P\|\|\sum w_j T_j$	$Q\|\|\sum w_j T_j$	$R\|\|\sum w_j T_j$
$1\|\|\sum w'_j E_j + w_j T_j$	$P\|\|\sum w'_j E_j + w_j T_j$	$Q\|\|\sum w'_j E_j + w_j T_j$	$R\|\|\sum w'_j E_j + w_j T_j$
	$P\|\|\sum w_j C_j$	$Q\|\|\sum w_j C_j$	$R\|\|\sum w_j C_j$

Não foram realizados testes computacionais para todos os problemas listados acima, não apenas pelos motivos já mencionados anteriormente, mas também pela indisponibilidade de instâncias. Além disso, em alguns casos, não há na literatura informações sobre os limites inferiores e superiores (*lower/upper bounds*). Detalhes sobre os testes realizados estão presentes no Capítulo 5.

3. Estruturas de Vizinhanças

A solução do problema de escalonamento da produção consiste em determinar sequência em que as tarefas serão processadas em cada máquina. Cada tarefa pode ser representada por um índice inteiro que não deve se repetir. Desta forma, as sequências podem ser representadas por m vetores π_k, onde m é o número de máquinas e $\pi = (\pi_0, \pi_1, \ldots, \pi_m)$ representa a solução por completo. Cada vetor π_k representa uma sequência de tarefas em uma máquina $k \in M$ e $n_k = (|\pi_k| - 2)$ indica o número de tarefas processadas pela máquina k. Cada vetor π_k inicia e finaliza com o elemento 0, que pode ser interpretado como uma tarefa auxiliar com tempo de processamento, pesos e datas de entrega iguais a zero. Define-se ainda $\pi_k(i)$ como sendo a i-ésima tarefa da sequência π_k, onde $\pi_k(0) = \pi_k(n_k + 1) = 0$. Para melhor ilustrar o funcionamento das vizinhanças, bem como os procedimentos utilizados para o cálculo do custo de uma solução vizinha π', uma representação em blocos é apresentada.

Seja $\pi_k = (\pi_k(0), \pi_k(1), \ldots, \pi_k(n_k + 1))$ uma sequência de tarefas composta por $n_k + 2$ tarefas ($\pi_k(0)$ e $\pi_k(n_k + 1)$ são tarefas auxiliares). Podemos dividi-las em blocos B, onde um bloco é definido como uma subsequência de tarefas e o custo de uma sequência pode ser obtido pela soma dos custos dos blocos.

A vizinhança *l–Block Insertion* Intra Máquina consiste simplesmente no deslocamento de um bloco de tamanho l para frente $(i < j)$ ou para trás $(i > j)$, dentro de uma máquina k, conforme ilustrado nas Figuras 1 e 2. Ela é uma generalização da vizinhança *insertion*.

Na vizinhança *Block Swap* Intra Máquina, diferentemente da vizinhança anterior, os movimentos que a compõem são caracterizados pela troca de posições de dois blocos de tamanhos l e

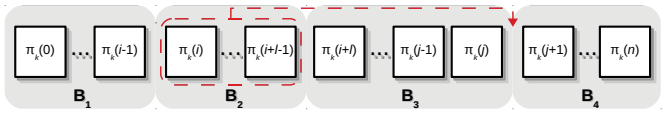


Figura 1: *l–block insertion* intra máquina para frente

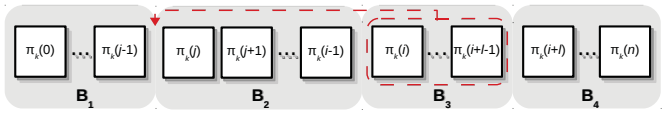


Figura 2: *l–block insertion* intra máquina para trás

l', respectivamente, dentro de uma mesma máquina, como mostrado na Figura 3. Essa vizinhança é uma generalização da vizinhança *swap*.

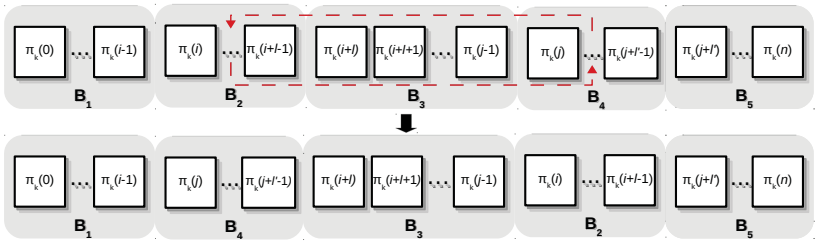
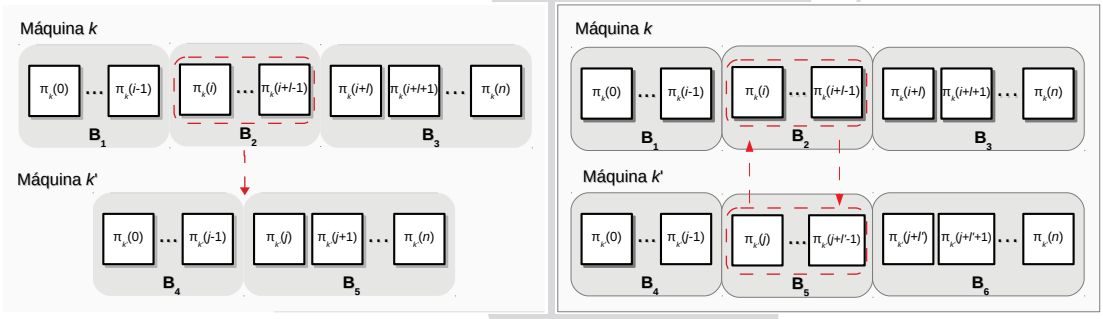


Figura 3: *Block Swap* intra máquina

A vizinhança *l–Block Insertion* Inter Máquinas é composta pelos movimentos de extração de um bloco de tamanho l, iniciando na posição i da máquina k, e inserção do mesmo na posição j, em outra máquina k' distinta, $\forall k' \in M, k \neq k'$. Já a vizinhança *Block Swap* Inter Máquinas se baseia na troca de posições de um bloco de tamanho l, da máquina k por um bloco de tamanho l' da máquina k'. As ilustrações dos movimentos em questão são apresentadas nas Figuras 4(a) e 4(b), respectivamente.



(a) *l–block insertion* inter máquinas (b) *Block Swap* inter máquinas

Figura 4: Vizinhanças inter máquinas

4. Avaliação dos Custos

As estruturas de vizinhanças clássicas, como *swap* e *insertion*, possuem $O(n^2)$ vizinhos, e normalmente são necessárias $O(n)$ operações para computar o custo do vizinho, resultando em

uma complexidade de $O(n^3)$ para a vizinhança por completo quando a avaliação é feita de maneira tradicional.

Em geral, o custo de um bloco de tarefas pode ser computado em $O(|B_t|)$, de modo que computar todos os blocos de uma sequência seguindo essa metodologia resultará em $O(n)$ passos. O Algoritmo 1 demonstra esse processo, onde a e b são as posições iniciais e finais de um bloco B_t, respectivamente, e b' é a posição final do bloco anterior (B_{t-1}) cujo tempo de término é C_b'.

Algoritmo 1 CompCostBlock

Procedimento CompCostBlock(b', a, b, π_k, C_b')
$cost \leftarrow 0$
$Ctemp \leftarrow C\_b' + s^k_{\pi_k(b'),\pi_k(a)} + p_{\pi_k(a)}$ ▷ Variável global contendo um *completion time* temporário
se $Ctemp > d_{\pi_k(a)}$ **então**
    $cost \leftarrow w_{\pi_k(a)} \times \left(Ctemp - d_{\pi_k(a)}\right)$
senão
    $cost \leftarrow w'_{\pi_k(a)} \times \left(d_{\pi_k(a)} - Ctemp\right)$
para $a' = a + 1 \ldots b$ **faça**
    $Ctemp \leftarrow Ctemp + s^k_{\pi_k(a'-1),\pi_k(a')} + p_{\pi_k(a')}$
    se $Ctemp > d_{\pi_k(a')}$ **então**
        $cost \leftarrow cost + w_{\pi_k(a')} \times \left(Ctemp - d_{\pi_k(a')}\right)$
    senão
        $cost \leftarrow cost + w'_{\pi_k(a')} \times \left(d_{\pi_k(a')} - Ctemp\right)$
retorne $cost$

O Algoritmo 1 possibilita o cálculo dos custos das soluções de uma determinada vizinhança da maneira tradicional, ou seja, em $O(n)$ passos para cada solução da vizinhança, quando não há presença de tempo ocioso. Logo, ao se aplicar o Algoritmo 1 em cada bloco, basta que somemos seus custos resultantes a fim de compor o custo total da nova sequência.

Note que, o custo do primeiro bloco de cada sequência não varia, podendo ser pré-calculado antes da avaliação de uma vizinhança. Para o cálculo dos custos desses blocos, define-se W^k_j como a soma dos custos das j primeiras tarefas de uma sequência π_k, dado por:

$$W^k_j = \begin{cases} \sum_{i=1}^{j} w'_{\pi_k(i)}(d_{\pi_k(i)} - C_{\pi^k(i)}), & \text{se} \quad C_{\pi^k(i)} \leq d_{\pi_k(i)} \\ \sum_{i=1}^{j} w_{\pi_k(i)}(C_{\pi^k(i)} - d_{\pi_k(i)}), & \text{se} \quad C_{\pi^k(i)} \geq d_{\pi_k(i)} \end{cases} \quad (1)$$

Por exemplo, o custo de um bloco de tarefas $B = (\pi_k(0), \ldots, \pi_k(5))$ seria obtido em tempo $O(1)$ consultando a estrutura W^k_5.

Para o $1||\sum w_j T_j$, Ergun e Orlin (2006) propuseram um algoritmo capaz de explorar as vizinhanças *swap*, *insertion* e *twist* em $O(n^2)$. Esse método utiliza estruturas de dados auxiliares que permitem a avaliação de uma solução na vizinhança em $O(1)$ amortizado, onde o pré-processamento das estruturas auxiliares é realizado em $O(n^2)$, definindo a complexidade da vizinhança. Essa metodologia foi aqui estendida para as vizinhanças descritas na Seção 3 para a resolução de problemas envolvendo máquinas paralelas não-relacionadas com diversas funções objetivo, desde que o problema não permita tempo ocioso nem possua *setup*.

Uma vez que os valores adotados de l e l', são, em geral, pequenos, decidiu-se pela implementação de um método mais simples com complexidade $\mathcal{O}(l)$ (de acordo com Ergun e Orlin (2006)) ao invés de $\mathcal{O}(1)$, resultando em uma complexidade total de $\mathcal{O}(ln^2)$. Na prática, essa metodologia aparenta ser mais eficiente mesmo para instâncias de tamanho elevados, devido ao fato que o método proposto por Ergun e Orlin (2006) requer uma fase de pré-processamento, com complexidade $\mathcal{O}(n^2)$, no caso dos blocos que tiverem tamanhos maiores que 1. Ao fim da seção, é discutido como avaliar as soluções para as vizinhanças propostas em $\mathcal{O}(1)$ passos.

No caso da vizinhança intra máquina, dois procedimentos similares são necessários para mover um bloco para frente ou para trás, em uma sequência π^k. O Algoritmo 2 mostra como os

movimentos são avaliados quando um bloco de tamanho l é removido e inserido em uma posição posterior na sequência. Os parâmetros de entrada são a sequência atual e seus custo e *completion time* associados, π_k, f_{π_k} e C, respectivamente. Os movimentos são avaliados $\mathcal{O}(l)$ passos (linha 12 do Algoritmo 2) a partir do custo da solução referente ao movimento imediatamente anterior. O mesmo raciocínio é aplicado para o caso em que um bloco de tarefas é movido para uma posição anterior à sua atual. Assim, verifica-se que a complexidade de avaliação de todos os movimentos da vizinhança *l-block insertion* intra máquina é de $\mathcal{O}(ln^2)$.

Algoritmo 2 *l-block* insertion intra forward

1: **Procedimento** *l-block*InsertionIntraF(π_k, f_{π_k}, l, C)
2: $\pi_k^* \leftarrow \pi_k; f_k^* \leftarrow f_{\pi_k}$
3: $p_B \leftarrow 0$ ▷ p_B é o tempo de processamento do bloco considerado para reinserção
4: **para** $i' = 1, \ldots, l$ **faça** ▷ cálculo do tempo de processamento do bloco iniciando na posição 1
5: $\quad p_B \leftarrow p_B + p^k_{\pi_k(i')}$
6: **para** $i = 1, \ldots, n_k - l$ **faça**
7: $\quad f_{\pi'_k} \leftarrow f_{\pi_k}$; ▷ $f_{\pi'_k}$ é o custo da sequência vizinha π'_k sob avaliação
8: $\quad$ **para** $j = i + l, \ldots, n_k - 1$ **faça**
9: $\quad\quad lateness \leftarrow C_{\pi_k(j)} - p_B - d_{\pi_k(j)}$
10: $\quad\quad f_{\pi'_k} \leftarrow f_{\pi'_k} + \max(w_{\pi_k(j)} \times lateness,\ w'_{\pi_k(j)} \times (-lateness))$
11: $\quad\quad f_{\pi'_k} \leftarrow f_{\pi'_k} - f_{\pi_k(j)}$ ▷ $f_{\pi_k(j)}$ é o custo da tarefa $\pi_k(j)$ na sequência π_k
12: $\quad\quad f_{\pi'_k} \leftarrow f_{\pi'_k} + \texttt{CompCostBlock}(j, i, i + l - 1, \pi_k, C_{\pi_k(j)} - p_B)$
13: $\quad\quad$ **se** $f_{\pi'_k} < f_k^*$ **então**
14: $\quad\quad\quad \pi_k^* \leftarrow \pi'_k; f_k^* \leftarrow f_{\pi'_k}$
15: $\quad p_B \leftarrow p_B - p^k_{\pi_k(i)} + p^k_{\pi_k(i+l)}$ ▷ cálculo do tempo de processamento bloco iniciando na posição $i + 1$
16: **retorne** f_k^*

O Algoritmo 3 mostra o pseudocódigo pra a vizinhança *l-block insertion* inter máquinas. Note que quando um bloco de tamanho l iniciando na posição i é removido de uma máquina k e inserido em uma máquina k' distinta, o custo da nova sequência π_k precisa ser calculado apenas uma vez e esse cálculo pode ser feito em $\mathcal{O}(n)$ passos utilizando o procedimento detalhado no Algoritmo 1. Para a máquina k', a qual o bloco será inserido em diversas posições, o custo de cada nova sequência pode ser computado em $\mathcal{O}(l)$ usando a mesma ideia do Algoritmo 2 após computar o custo de inserir o bloco no início da sequência $\pi_{k'}$, em $\mathcal{O}(n)$, também seguindo o Algoritmo 1. Desta maneira, a complexidade de avaliação de todas as soluções na vizinhança *l-block insertion* inter máquinas é $\mathcal{O}(ln^2)$.

Algoritmo 3 *l–block insertion* inter

1: **Procedimento** l-blockInsertionInter(π, f, l)
2: $\pi^* \leftarrow \pi, f^* \leftarrow f_\pi$
3: **para** $k = 1, \ldots m$ **faça**
4: $\quad$ **para** $k' = 1, \ldots m$ **faça**
5: $\quad\quad$ **se** $k \neq k'$ **então**
6: $\quad\quad\quad$ **para** $i = 1, \ldots, n_k - l + 1$ **faça**
7: $\quad\quad\quad\quad f_{\pi'_k} \leftarrow W^k_{i-1} + \texttt{CompCostBlock}(i - 1, i + l, n_k, \pi_k, C_{\pi_k(i-1)})$
8: $\quad\quad\quad\quad \tilde{\pi}_{k'} \leftarrow (\pi_k(i), \ldots, \pi_k(i + l - 1)) \oplus \pi_{k'}$
9: $\quad\quad\quad\quad f_{\pi'_{k'}} \leftarrow \texttt{l-blockInsertionIntraF}(\tilde{\pi}_{k'}, f_{\tilde{\pi}_{k'}}, l)$ ▷ só para $i = 1$ (linha 6 do Alg. 2)
10: $\quad\quad\quad\quad$ **se** $f_\pi - f_{\pi_k} - f_{\pi_{k'}} + f_{\pi'_k} + f_{\pi'_{k'}} < f^*$ **então**
11: $\quad\quad\quad\quad\quad f^* \leftarrow f_\pi - f_{\pi_k} - f_{\pi_{k'}} + f_{\pi'_k} + f_{\pi'_{k'}}$
12: $\quad\quad\quad\quad\quad \pi^* \leftarrow \pi'$
13: **retorne** f^*

Antes de descrever os procedimentos de avaliação para as vizinhanças *block swap* intra e inter máquina, é necessário introduzir algumas funções e estruturas de dados.Seja $\rho^k_j(t)$ uma

função que represente a penalidade associada ao processamento da tarefa j na máquina k dado que ela começa a ser processada no tempo t. Cada função de penalidade $\rho_j^k(t), \forall j \in J, \forall k \in M$ é não-negativa, convexa e linear por partes. As funções $\rho_j^k(t)$ podem ser definidas da seguinte forma:

$$\rho_j^k(t) = \begin{cases} w_j(d_j - p_j^k - t), & t \in [0, d_j - p_j^k] \\ w'_j(d_j - p_j^k + t), & t \in [d_j - p_j^k, +\infty) \end{cases} \quad (2)$$

Note que cada função ρ_j^k será composta por dois segmentos, onde os pontos de transição, denominados de *breakpoints*, serão definidos por d_j e p_j^k.

Definimos $g_j^k(t)$ uma função linear por partes que represente o custo de um bloco de tarefas, compostos pelas tarefas $(\pi_k(j), \pi_k(j+1), \ldots, \pi_k(n_k))$, dado que elas são sequenciadas nessa ordem e que a tarefa $\pi_k(j)$ inicia seu processamento no tempo t. Tais funções podem ser obtidas pela seguinte recursão:

$$g_j^k(t) = \begin{cases} +\infty, & j = n_k + 1, \quad t \in (-\infty, 0) \\ 0, & j = n_k + 1, \quad t \in [0, +\infty) \\ g_{j+1}^k(t + p_j^k) + \rho_j^k(t), & n_k \geq j > 0, \quad -\infty < t < +\infty \end{cases} \quad (3)$$

Perceba que para para obter o custo de um determinado bloco de tarefas, $(\pi_k(j), \ldots, \pi_k(n_k))$ começando no tempo t, por meio das funções $g_j^k(t)$ é necessário primeiro caminhar nos *breakpoints* até encontrar o intervalo onde $t_1 \leq t \leq t_2$, e em seguida calcular o custo. As funções $\rho_j^k(t)$ e $g_j^k(t)$ possuem em comum o fato de serem funções lineares por partes. Para armazená-las na memória, são utilizadas listas encadeadas, onde cada elemento da lista (segmento) é formado por uma estrutura contendo as seguintes informações: b_1 e b_2 indicam o início e fim do intervalo da função, respectivamente; c informa custo da função no tempo b_1; e α representa o coeficiente angular da reta.

De posse dessas informações, o custo de um bloco de tarefas $(\pi_k(j), \pi_k(j+1), \ldots, \pi_k(n_k))$ começando no tempo t (denotado por $g_j^k(t)$), dado que $b_1 \leq t \leq b_2$ será: $c + \alpha \times (t - b_1)$. As informações das funções $g_j^k(t)$ são armazenadas em listas encadeadas `L_j_k`, que contém as informações referentes às funções lineares por partes das penalidades das tarefas $(\pi_k(j), \pi_k(j+1), \ldots, \pi_k(n_k))$, dado que elas são sequenciadas nessa ordem. Note que uma função linear por partes é composta de vários segmentos. A Figura 5 ilustra o armazenamento de uma função $g_j^k(t)$.

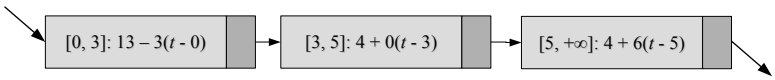


Figura 5: Representação da função $g_1^1(t)$ usando lista encadeada

Nesse trabalho, utilizamos a estrutura de dados `ProcessingList` (Ergun e Orlin, 2006), definida para uma dada sequência π_k e para todos os blocos de tamanho l dessa sequência. Trata-se de uma lista que contém um par de informações para cada posição. A primeira, denominada `pos`, referente à posição do início de um bloco de tarefas de tamanho l na sequência π_k, e a segunda, referenciada por (`p`), indica o tempo de processamento desse bloco em uma determinada máquina k'. A `ProcessingList` deve, então, ser ordenada em ordem decrescente de acordo com os valores de `p`. O procedimento de obtenção da `ProcessingList` é detalhado no Algoritmo 4.

Uma vez definidas as funções e estruturas auxiliares, é possível mostrar como computar os custos de uma sequência π'_k gerada após a realização de um movimento na vizinhança *block swap* intra máquina. O custo dessa nova sequência, obtida pela troca de posições de um bloco de tamanho l, iniciando em i, com um bloco de tamanho l', iniciando em j, é dado por $W_{i-1}^k + c(B'_4) +$

Algoritmo 4 CreateProcessingList

1: **Procedimento** CreateProcessingList(k, k', l)
2: $p_b \leftarrow 0$
3: **para** $i' = 1, \ldots, l$ **faça**
4: $\quad p_b \leftarrow p_b + p^{k'}_{\pi_k(i')}$
5: **para** $i \leftarrow 1, \ldots, n_k - l + 1$ **faça**
6: $\quad$ `ProcessingList.pos[i]` $\leftarrow i$
7: $\quad$ `ProcessingList.p[i]` $\leftarrow p_b$
8: $\quad p_b \leftarrow p_b - p^{k'}_{\pi_k(i)} + p^{k'}_{\pi_k(i+l)}$
9: `sort(ProcessingList,p)`
10: **retorne** `ProcessingList`

$[g^k_{i+l}(t_1) - g^k_j(t_2)] + c(B'_2) + [W^k_{n^k} - W^k_{j+l'}]$, onde W^k_{i-1} representa o custo de B_1; $c(B'_4)$ e $c(B'_2)$ representam os custos dos blocos B_4 e B_2 nas novas posições; $[g^k_{i+l}(t_1) - g^k_j(t_2)]$ informa o custo do bloco B_3, dado que este iniciará no tempo $t_1 = C_{\pi_k(i-1)} + p_{B_4}$, com $t_2 = C_{\pi_k(j+l'-1)} - p_{B_2}$; e $[W^k_{n^k} - W^k_{j+l'}]$ indica o custo do bloco B_5 (ver Fig. 3 para verificar as posições dos blocos).

Os custos dos blocos B_1 e B_5 são obtidos por uma consulta na estrutura W^k em tempo $\mathcal{O}(1)$, onde W^k deve ser calculada/atualizada (em $\mathcal{O}(n_k)$) antes do início da busca na vizinhança. A partir da Figura 3, percebe-se que os blocos B_2 e B_4 possuem tamanhos fixos, l e l'. Logo, os custos destes blocos são obtidos em $\mathcal{O}(l)$ e $\mathcal{O}(l')$, respectivamente, quando feito de maneira tradicional. Para o bloco B_3, o custo será obtido com o auxílio das funções $g^k_j(t)$, como explicado a seguir.

Para uma dada sequência π_k, $g^k_j(t)$ terá, no pior caso, n_k *breakpoints*. Como $g^k_j(t)$ são funções lineares por partes, a complexidade de obter o custo de um bloco $(\pi_k(j), \pi_k(j+1), \ldots, \pi_k(n_k))$, iniciando no tempo t será de $\mathcal{O}(n)$, pois, o tempo t pode estar após o último *breakpoint* da função. Dessa forma, o tempo de avaliação de uma solução na vizinhança seria de $\mathcal{O}(n)$. Entretanto, é possível avaliar um movimento dessa vizinhança em $\mathcal{O}(\max(l, l')n^2)$, caso os custos dependentes das funções $g^k_j(t)$ sejam pré-calculados seguindo uma determinada ordem, possibilitando a obtenção dos custos do bloco B_3 em $\mathcal{O}(1)$ passos, conforme mostrado no Algoritmo 5.

As linhas 4-10 do Alg. 5 mostram como executar a busca na vizinhança block swap intra máquina em $\mathcal{O}(\max(l, l')n^2)$ passos. A etapa de pré-processamento, descrita nas linhas 12-34 computa o valor, para cada par (i, j), da estrutura `costB3` que armazena os custo do bloco B_3 quando um bloco de tamanho l começando na posição i é trocado com um bloco de tamanho l' que se inicia na posição j.

O Algoritmo 5 considera apenas os casos em que os blocos que iniciam na posição i, de tamanho l estão sempre antes dos blocos de tamanho l' que iniciam na posição j. Isso se justifica pelo fato de que se os tamanhos dos blocos forem invertidos, temos o caso complementar da vizinhança. Desta forma, o Algoritmo 5 deve ser executado duas vezes (caso $l \neq l'$) a fim de explorar completamente a vizinhança *block swap* intra máquina.

Assim como nas estruturas de vizinhanças apresentadas anteriormente, o custo de uma solução pertencente à vizinhança *block swap* inter máquinas pode ser obtido pela somas dos custos dos blocos ilustrados na Figura 4(b). O custo da nova sequência da máquina k após a realização de um movimento na vizinhança é dado por $W^k_{i-1} + c(B'_5) + g^k_{i+l}(t_1)$, enquanto que para a máquina k', o custo será $W^{k'}_{j-1} + c(B'_2) + g^{k'}_{j+l'}(t_2)$.

Para a máquina k, o custo de uma solução pode ser obtido em $\mathcal{O}(l')$ passos, enquanto para a máquina k' uma solução pode ser avaliada em tempo $\mathcal{O}(l)$. As parcelas W^k_{i-1} e $W^{k'}_{j-1}$ podem ser obtidas em $\mathcal{O}(1)$ operações através de uma consulta, assim como as parcelas $g^k_{i+l}(t_1)$ e $g^{k'}_{j+l'}(t_2)$, desde que estas sejam pré-processadas. Assim, a complexidade de avaliação de todas as soluções na vizinhança é de $\mathcal{O}(\max(l, l')n^2)$.

Algoritmo 5 *Block swap* intra

1: **Procedimento** blockSwapIntra(π_k, f_{π_k}, l, l')
2: $\pi_k^* \leftarrow \pi_k$; $f_k^* \leftarrow f_{\pi_k}$
3: `costB3` ←`preProcblockSwapIntra`(π_k, l, l') ▷ linhas 12-34
4: **para** $i = 1, \ldots, n_k - l - l' + 1$ **faça**
5: **para** $j = i + l, \ldots, n_k - l' + 1$ **faça**
6: $f_{\pi_k'} \leftarrow W_{i-1}^k +$ `costB3[i,j]` $+ W_{n_k}^k - W_{j+l'}^k$
7: $f_{\pi_k'} \leftarrow f_{\pi_k'} +$ `CompCostBlock`$(i-1, i, i+l'-1, \pi_k^{'}, C_{\pi_k'(i-1)})$
8: $f_{\pi_k'} \leftarrow f_{\pi_k'} +$ `CompCostBlock`$(j-1, j, j+l-1, \pi_k^{'}, C_{\pi_k'(j-1)})$
9: **se** $f_{\pi_k'} < f_k^*$ **então**
10: $\pi_k^* \leftarrow \pi_k^{'}$; $f_k^* \leftarrow f_{\pi_k'}$
11: **retorne** f_k^*
12: **Procedimento** preProcblockSwapIntra(π_k, l, l')
13: Inicialize a matriz `costB3`
14: `ProcessingList` ← `CreateProcessingList`(k, k, l')
15: **para** $i = 1, \ldots, n_k - l' + 1$ **faça**
16: $seg \leftarrow 1^{\text{o}}$ segmento de `L_k_i+l`
17: **para** $j \leftarrow |$`ProcessingList`$|, \ldots, 1$ **faça**
18: **se** `ProcessingList[j].pos` $> i + l - 1$ **então**
19: $t_1 = C_{\pi_k(i-1)} +$ `ProcessingList[j].p`
20: **se** `ProcessingList[j].pos` $> i + l$ **então**
21: **enquanto** $t_1 \geq seg.b_2$ **faça**
22: $seg \leftarrow$ próximo segmento
23: `costB3[i][ProcessingList[j].pos]` $\leftarrow seg.c + seg.\alpha \times (t_1 - seg.b_1)$
24: `ProcessingList` ← `CreateProcessingList`(k, k, l)
25: **para** $j = l + 1, \ldots, n_k - l'$ **faça**
26: $seg \leftarrow 1^{\text{o}}$ segmento de `L_k_j`
27: **para** $i \leftarrow 1, \ldots, |$`ProcessingList`$|$ **faça**
28: **se** `ProcessingList[i].pos` $< j - l + 1$ **então**
29: $t_2 = C_{\pi_k(j+l'-1)} -$ `ProcessingList[i].p`
30: **se** `ProcessingList[i].pos` $< j - l$ **então**
31: **enquanto** $t_2 \geq seg.b_2$ **faça**
32: $seg \leftarrow$ próximo segmento
33: `costB3[ProcessingList[i].pos][j]` $\leftarrow$ `costB3[ProcessingList[i].pos][j]` $-$ $(seg.c + seg.\alpha \times (t_2 - seg.b_1))$
34: **retorne** `costB3`

O Algoritmo 6 ilustra como é feita a avaliação da vizinhança. A linha 6 desse algoritmo se refere ao pré-processamento necessário para o cálculo das parcelas do custo dependentes das funções $g_j^k(t)$. Esse pré-processamento é ilustrado no mesmo algoritmo, nas linhas 15-34.

Note que para todas essas vizinhanças o cálculo do custo dos blocos movidos/trocado é sempre feito da maneira tradicional, o que define a complexidade total de avaliação das soluções das mesmas. Porém, com o auxílio das funções $g_j^k(t)$ e seguindo procedimentos similares aos apresentados nas linhas 12-34 do Alg. 5 e linhas 15-34 do Alg. 6, é possível obter os custos desses blocos em tempo $\mathcal{O}(1)$. Logo, tem-se a complexidade de $\mathcal{O}(n^2)$ para cada uma dessas vizinhanças.

5. Resultados

Esta seção apresenta os resultados obtidos pela aplicação da metodologia proposta incorporada em uma meta-heurística baseada no *Iterated Local Search* (ILS). A estrutura geral e os parâmetros utilizados pela meta-heurística implementada são os mesmos presentes no trabalho de Subramanian *et al*. (2014), porém, na versão a testada não considera a etapa *multi-start*. Na etapa de busca local, o algoritmo utiliza as vizinhanças *l-block insertion* intra máquina com $l \in \{1, 2\}$; *block swap* intra máquina com $(l, l') \in \{(1, 1)\}$; *l-block insertion* inter máquinas com $l \in \{1, 2\}$; e *block swap* inter máquinas com $(l, l') \in \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4)\}$. O

Algoritmo 6 *block swap* inter

Procedimento blockSwapInter(π, f, l, l')
$\pi^* \leftarrow \pi$; $f^* \leftarrow f$
para $k = 1, \ldots, m$ **faça**
  para $k' = 1, \ldots, m$ **faça**
    se $k \neq k'$ **então**
      $\texttt{costB}_3, \texttt{costB}_6 \leftarrow \texttt{preProcblockSwapInter}(\pi_k, \pi_{k'}, l, l')$ ▷ linhas 15-34
      para $i = 1, \ldots, n_k - l + 1$ **faça**
        para $j = 1, \ldots, n_{k'} - l' + 1$ **faça**
          $f_{\pi'_k} \leftarrow W^k_{i-1} + \texttt{CompCostBlock}(i-1, j, j+l'-1, \pi_k, C_{\pi_k(i-1)}) + \texttt{costB}_3\texttt{[i, j]}$
          $f_{\pi'_{k'}} \leftarrow W^{k'}_{j-1} + \texttt{CompCostBlock}(j-1, i, i+l-1, \pi_{k'}, C_{\pi_{k'}(j-1)}) + \texttt{costB}_6\texttt{[i, j]}$
        se $f_\pi - f_{\pi_k} - f_{\pi_{k'}} + f_{\pi'_k} + f_{\pi'_{k'}} < f^*$ **então**
          $f^* \leftarrow f_\pi - f_{\pi_k} - f_{\pi_{k'}} + f_{\pi'_k} + f_{\pi'_{k'}}$
          $\pi^* \leftarrow \pi'$
retorne f^*
Procedimento preProcblockSwapInter($\pi_k, \pi_{k'}, l, l'$)
Inicialize as matrizes $\texttt{costB}_3, \texttt{costB}_5$
$\texttt{ProcessingList} \leftarrow \texttt{CreateProcessingList}(k', k, l')$
para $i = 1, \ldots, n_k - l + 1$ **faça**
  $seg \leftarrow 1^{\text{o}}$ segmento de $\texttt{L\_k\_i+l}$
  para $j = |\texttt{ProcessingList}|, \ldots, 1$ **faça**
    $t_1 \leftarrow C^k_{i-1} + \texttt{ProcessingList[j].p}$
    enquanto $t_1 \geq seg.b_2$ **faça**
      $seg \leftarrow$ próximo segmento
    $\texttt{costB}_3\texttt{[i, ProcessingList[j].pos]} = seg.c + seg.\alpha \times (t_1 - seg.b_1)$
$\texttt{ProcessingList} \leftarrow \texttt{CreateProcessingList}(k, k', l)$

para $j = 1, \ldots, n_{k'} - l' + 1$ **faça**
  $seg \leftarrow 1^{\text{o}}$ segmento de $\texttt{L\_k'\_j+l'}$
  para $i = |\texttt{ProcessingList}|, \ldots, 1$ **faça**
    $t_2 \leftarrow C_{k'}(j-1) + \texttt{ProcessingList[i].p}$
    enquanto $t_2 \geq seg.b_2$ **faça**
      $seg \leftarrow$ próximo segmento
    $\texttt{costB}_6\texttt{[ProcessingList[i].pos, j]} = seg.c + seg.\alpha \times (t_2 - seg.b_1)$
retorne $\texttt{costB}_3, \texttt{costB}_6$

algoritmo foi implementado em linguagem de programação C++ e os experimentos foram executados em computadores com processadores Intel Core i7 de 3.40 GHz, com 16 GB de memória RAM e com sistema operacional Linux Ubuntu 12.04. Os experimentos utilizaram apenas uma *thread*.

O método proposto, ILS_{Fast}, foi comparado com a versão ILS_{Trad}, cuja avaliação de soluções é feita de maneira tradicional. Os experimentos foram realizados em instâncias geradas por Rodrigues *et al*. (2008) e Amorim (2013) (disponíveis em `http://algox.icomp.ufam.edu.br/index.php/benchmark-instances/`) para o $P||\sum w_jT_j$ e para o $P||\sum w'_jE_j + w_jT_j$, respectivamente. O método proposto pode ser empregado em todos os problemas presentes na Tabela 1, porém, por restrições de espaço, apenas os problemas $P||\sum w_jT_j$ e $P||\sum w'_jE_j + w_jT_j$ tiveram resultados reportados.

Os experimentos foram realizados em instâncias contendo $40, 50, 100$ ou 200 tarefas, e com $2, 4$ ou 10 máquinas. Cada grupo de instância, definido pelo número de tarefas a serem escalonadas e o número de máquinas disponíveis para processá-las, é formado por 9 instâncias, exceto para os grupos com 200 tarefas que são formados por 2 instâncias. Os algoritmos foram executados 5 vezes para cada instância. É importante ressaltar que para cada execução os dois métodos utilizaram a mesma semente, ou seja, executaram a mesma sequência de passos, diferindo apenas na forma de avaliação das soluções vizinhas. As colunas t_{med} mostram os tempos médios, em segundos, dessas execuções, enquanto a coluna $prop$ representa a proporção, em porcentagem, entre

os tempos obtidos pelo ILS_{Fast} em relação ao ILS_{Trad}, ou seja, $prop = \frac{\text{ILS}_{\text{Fast}}}{\text{ILS}_{\text{Trad}}}$. As Tabelas 2 e 3 apresentam os resultados desses experimentos.

Tabela 2: Resultados – $P||\sum w_j T_j$

Grupo de Instâncias	ILS_Trad $t_{\text{med}}(s)$	ILS_Fast $t_{\text{med}}(s)$	Prop (%)
wt40-2m	0,48	0,39	**80,66**
wt40-4m	0,48	0,48	**99,14**
wt40-10m	0,17	0,35	206,68
wt50-2m	0,99	0,68	**68,52**
wt50-4m	1,03	0,84	**81,79**
wt50-10m	0,47	0,78	164,97
wt100-2m	19,71	7,78	**39,47**
wt100-4m	18,73	9,20	**49,10**
wt100-10m	9,20	7,62	**82,83**
wt200-2m	453,69	113,83	**25,09**
wt200-4m	450,08	142,68	**31,70**
wt200-10m	227,54	116,49	**51,20**

Tabela 3: Resultados – $P||\sum w'_j E_j + w_j T_j$

Grupo de Instâncias	ILS_Trad t_{med}(s)	ILS_Fast t_{med}(s)	Prop (%)
wet40-2m	0,57	0,48	**84,59**
wet40-4m	0,56	0,57	101,17
wet40-10m	0,18	0,38	205,65
wet50-2m	1,72	1,22	**71,09**
wet50-4m	1,71	1,41	**82,26**
wet50-10m	0,45	0,76	168,77
wet100-2m	35,37	15,55	**43,97**
wet100-4m	36,02	18,24	**50,64**
wet100-10m	13,75	12,27	**89,23**
wet200-2m	1047,07	260,40	**24,87**
wet200-4m	842,24	267,81	**31,80**
wet200-10m	363,45	188,43	**51,85**

Os resultados presentes nas Tabelas 2 e 3 demonstram que os ganhos da metodologia proposta são expressivos, principalmente para as instâncias de dimensões mais elevadas, chegando a ser, em alguns casos, 4 vezes mais rápida que a versão tradicional. De fato, percebe-se que mantendo o número de tarefas constante e aumentando o número de máquinas, os ganhos entre o método proposto e o tradicional tendem a ser menores. Nas instâncias de dimensões menores, com 40 e 50 tarefas e 4 máquinas, os dois métodos tem resultados bem próximos. Para os casos onde o número de máquinas é acrescido, a versão tradicional mostrou-se mais eficiente. Isso se justifica pela necessidade de pré-processamento por parte do método proposto. Em linhas gerais, a metodologia proposta demonstra ser mais eficiente quando as instâncias possuem mais de 100 tarefas a serem processadas.

6. Conclusões

Este trabalho propôs uma extensão da metodologia proposta por Ergun e Orlin (2006) para o problema $1||\sum w_j T_j$. Inicialmente, Ergun e Orlin (2006) reduziram a complexidade de busca nas vizinhanças *insertion*, *swap* e *twist* de $\mathcal{O}(n^3)$ para $\mathcal{O}(n^2)$. Utilizando essa mesma metodologia, a extensão proposta possibilitou a aplicação em problemas envolvendo um ambiente de múltiplas máquinas paralelas não-relacionadas e funções objetivos envolvendo antecipações e atrasos no processamento das tarefas. Esta variante se deu também no âmbito das vizinhanças, que foram generalizadas, considerando blocos não necessariamente unitários e movimentos entre máquinas distintas.

Os resultados apresentados mostram que a abordagem proposta por Ergun e Orlin (2006) foi adaptada com sucesso e que os ganhos, em termos de tempo de execução foram consideráveis, principalmente para os problemas de dimensões mais elevadas.

Referências

Amorim, R. Um estudo sobre formulações matemáticas e estratégias algorítmicas para problemas de escalonamento em máquinas paralelas com penalidades de antecipação e atraso. Dissertação de mestrado, Programa de Pós-Graduação em Informática, Instituto de Computação, Universidade Federal do Amazonas, Manaus, AM, Brazil, 2013.

Brucker, P. e Knust, S. *Complex Scheduling*. GOR-Publications. Springer, 2006.

Ergun, Ö. e Orlin, J. B. (2006), Fast neighborhood search for the single machine total weighted tardiness problem. *Operations Research Letters*, v. 34, n. 1, p. 41 – 45.

Graham, R., Lawler, E., Lenstra, J. e Kan, A. (1979), Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, v. 5, p. 287 – 326.

Ibaraki, T., Imahori, S., Nonobe, K., Sobue, K., Uno, T. e Yagiura, M. (2008), An iterated local search algorithm for the vehicle routing problem with convex time penalty functions. *Discrete Applied Mathematics*, v. 156, n. 11, p. 2050–2069.

Kedad-Sidhoum, S. e Sourd, F. (2010), Fast neighborhood search for the single machine earliness-tardiness scheduling problem. *Computers & Operations Research*, v. 37, n. 8, p. 1464 – 1471. Operations Research and Data Mining in Biological Systems.

Lenstra, J., Kan, A. R. e Brucker, P. Complexity of machine scheduling problems. P.L. Hammer, E.L. Johnson, B. K. e Nemhauser, G. (Eds.), *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, p. 343 – 362. Elsevier, 1977.

Liao, C.-J., Tsou, H.-H. e Huang, K.-L. (2012), Neighborhood search procedures for single machine tardiness scheduling with sequence-dependent setups. *Theoretical Computer Science*, v. 434, p. 45–52.

Potts, C. N. e Strusevich, V. A. (2009), Fifty years of scheduling: a survey of milestones. *JORS*, v. 60, n. S1.

Rodrigues, R., Pessoa, A., Uchoa, E. e de Aragão, M. P. (2008), Heuristic Algorithm for the Parallel Machine Total Weighted Tardiness Scheduling Problem. *Relatório de Pesquisa em Engenharia de Produçao*, v. 8, n. 10.

Subramanian, A., Battarra, M. e Potts, C. N. (2014), An iterated local search heuristic for the single machine total weighted tardiness scheduling problem with sequence-dependent setup times. *International Journal of Production Research*, v. 52, n. 9, p. 2729–2742.

Tanaka, S. e Fujikuma, S. (2012), A dynamic-programming-based exact algorithm for general single-machine scheduling with machine idle time. *Journal of Scheduling*, v. 15, n. 3, p. 347–361.