Meta-heurísticas para o sequenciamento de famílias de tarefas em máquinas paralelas idênticas de processamento em lote

Meta-heuristics for scheduling job families on identical parallel batch processing machines

DOI:10.34117/bjdv8n9-104

Recebimento dos originais: 08/08/2022
Aceitação para publicação: 09/09/2022

Michele Bernardino Fidelis
Mestre em Ciência da Computação pela Universidade Federal de Viçosa (UFV)
Instituição: Universidade Federal de Viçosa (UFV)
Endereço: Campus Universitário, CEP: 36570-900, Viçosa, MG - Brasil
E-mail: michele.bernardino@ufv.br

José Elias Claudio Arroyo
Doutor em Engenharia Elétrica e Computação pela Universidade Estadual de Campinas (UNICAMP)
Instituição: Universidade Federal de Viçosa (UFV)
Endereço: Campus Universitário, CEP: 36570-900, Viçosa, MG - Brasil
E-mail: jarroyo@ufv.br

RESUMO
Este artigo aborda um problema de sequenciamento de tarefas em máquinas paralelas idênticas de processamento em lote. Neste problema uma máquina pode executar um lote de tarefas simultaneamente. Além disso, as tarefas são classificadas em famílias, onde uma família agrupa tarefas que possuam alguma característica em comum. Assim, os lotes devem conter somente tarefas de uma mesma família. O problema também considera tarefas com diferentes tempos de chegada (*release times*). As tarefas possuem ainda uma data de entrega e um peso. O objetivo do problema é determinar os lotes de tarefas para serem sequenciados nas máquinas de tal maneira que o atraso total ponderado das tarefas seja minimizado. O problema envolvendo sequenciamento de lotes é uma extensão do sequenciamento de tarefas clássico (onde uma máquina processa apenas uma tarefa por vez) e possui muitas aplicações reais. Para resolver o problema abordado, três algoritmos baseados em meta-heurísticas foram desenvolvidos: *Adaptive Large Neighborhood Search* (ALNS), *Iterated Greedy* (IG) e *Simulated Annealing* (SA). Todos estes algoritmos utilizam técnicas de busca em vizinhança para melhorar a qualidade de uma solução. Experimentos computacionais, utilizando dados da literatura, foram realizados a fim de avaliar o desempenho dos algoritmos. Para instâncias de grande porte, os algoritmos propostos são comparados com dois algoritmos da literatura (*Memetic Algorithm* e *Variable Neighborhood Search*). Os experimentos e testes realizados demonstram que os algoritmos desenvolvidos neste trabalho geram soluções válidas de excelente qualidade superando as melhores soluções encontradas na literatura.

Palavras-chave: máquinas paralelas de processamento de lotes, meta-heurísticas, busca local, otimização combinatória.

ABSTRACT
This paper addresses a job scheduling problem on identical parallel batch processing machines. In this problem a machine can process a batch of jobs simultaneously. In addition, jobs are classified into families, where a family groups jobs that have some characteristic in common. Thus, batches must contain only jobs of the same family. The problem also considers jobs with different release times. Jobs also have a due date and a priority weight. The objective of the problem is to group the job set into batches and assign the batches to the parallel machines in order to minimize the total weighted tardiness of the jobs. The problem involving batch sequencing is an extension of classical job sequencing (where a machine processes only one job at a time) and has many real applications. To solve the addressed problem, three algorithms based on metaheuristics were developed: Adaptive Large Neighborhood Search (ALNS), Iterated Greedy (IG) and Simulated Annealing (SA). All these algorithms use neighborhood search techniques to improve the quality of a solution. Computational experiments, using instances from the literature, were performed in order to evaluate the performance of the algorithms. For large instances, the proposed algorithms are compared with two algorithms from the literature (Memetic Algorithm and Variable Neighborhood Search). The computational experiments demonstrate that the algorithms developed in this work generate valid solutions of excellent quality, outperforming the best solutions presented in the literature.

Keywords: parallel batch processing machines, meta-heuristics, local search, combinatorial optimization.

1 INTRODUÇÃO

O sequenciamento de lotes de tarefas é um processo muito utilizado em indústrias, uma vez que facilita o processamento das tarefas que possuem alguma característica em comum. Um lote consiste em um conjunto de tarefas que podem ser agrupadas e processadas simultaneamente em uma máquina. Nesse tipo de configuração, todas as tarefas pertencentes ao mesmo lote, iniciam e terminam o processamento simultaneamente. O tempo de processamento do lote consiste no maior tempo de processamento entre as tarefas pertencentes ao lote. Segundo Potts e Kovalyov (2000), operações de processamento em lotes são comuns em indústrias de fabricação de semicondutores nas quais operações para fabricação de placas de circuito são realizadas em fornos que podem agrupar várias tarefas simultaneamente. Também podem ser encontradas em indústrias químicas nas quais algumas operações são realizadas em tanques ou fornos.

Em alguns problemas de sequenciamento de lotes as tarefas são classificadas em famílias. Uma família agrupa tarefas que tenham alguma característica em comum, e os lotes podem conter apenas tarefas de mesma família. Segundo Cheng, Chiang e Fu (2008), problemas de loteamento podem ser classificados em função da compatibilidade

das tarefas. A primeira categoria engloba problemas de loteamento com tarefas de famílias compatíveis, na qual, as tarefas podem ser agrupadas e processadas simultaneamente e o tempo de processamento do lote é determinado pelo maior tempo de processamento entre as tarefas pertencentes ao lote. A outra categoria refere-se a problemas com tarefas de famílias incompatíveis. Encontra-se com frequência em operações de difusão/oxidação em fábricas de semicondutores. Nessas operações, as tarefas são separadas e classificadas em famílias, de acordo com propriedades químicas semelhantes, e somente tarefas da mesma família podem ser processadas simultaneamente.

Neste trabalho aborda-se o Problema de Sequenciamento de Lotes de Tarefas em Máquinas Paralelas Idênticas (PSLTMPI), considerando tarefas incompatíveis. O objetivo é sequenciar os lotes gerados nas máquinas tal que o atraso total ponderado (*total weighted tardiness* (TWT)) seja minimizado. O problema em estudo é considerado NP-difícil, pois o problema mais simples de sequenciamento de tarefas em uma única máquina com a minimização do TWT também é NP-difícil (Pinedo (2012)).

Na literatura alguns trabalhos abordam o problema PSLTMPI. Mönch et al. (2005) propõem dois Algoritmos Genéticos (AG) para o problema PSLTMPI enquanto Chiang et al. (2010) desenvolvem um Algoritmo Memético (AM) para resolver o mesmo problema. Resultados apresentados no trabalho de Chiang et al. (2010) demonstram que o AM supera os AG proposto por Mönch et al. (2005) obtendo maior robustez. Já Bilyk et al. (2014) propõe um algoritmo VNS (*Variable Neighborhood Search*) para o problema PSLTMPI considerando restrições de precedência, no entanto, eles também realizam uma comparação com o trabalho de Chiang et al. (2010). Os resultados obtidos demonstraram que o VNS é capaz de alcançar uma melhoria de 10.28% em relação ao AM.

Na literatura existem alguns trabalhos que consideram tarefas com tamanho diferentes. Jia et al. (2016) consideram o problema de sequenciar *n* tarefas de diferentes tamanhos e de famílias incompatíveis em um conjunto de máquinas paralelas de processamento em lotes com o objetivo de minimizar o *makespan*. Alguns trabalhos da literatura consideram tarefas com tamanho unitário. Lausch e Mönch (2016) abordam um problema com máquinas paralelas idênticas de processamento em lote e com tarefas pertencentes a famílias incompatíveis. A fim de minimizar o atraso total ponderado eles desenvolveram um AG e um algoritmo ACO (*Ant Colony Optimization*). Malve e Uzsoy (2007) consideram tarefas com tempos de chegada dinâmico e pertencentes a diferentes famílias. Eles propõem um AG para minimizar o atraso máximo.

Damodaran e Velez-Gallego (2010) propõem uma heurística para minimizar o *makespan* em máquinas paralelas de processamento em lotes com tarefas que possuem diferentes tempos de chegada. Um algoritmo ACO é proposto pelos autores apresentando bons resultados para o problema. Jia e Leung (2015) consideram tarefas com tamanhos diferentes e máquinas de diferentes capacidades. Para minimizar o *makespan* eles propõem uma heurística construtiva baseada na regra *First-Fit-Decreasing* (FFD), assim como uma meta-heurística baseada em *Max-Min Ant System* (MMAS). Já Herr e Goel (2016) estudam o problema de sequenciamento de tarefas em uma máquina com o objetivo de minimizar o atraso total. Cada tarefa neste problema possui um tempo de processamento, uma data de chegada e pertencem a uma família. Cada tarefa requer uma certa quantidade de recurso antes do início do seu processamento.

Este trabalho considera o problema abordado por Chiang et al. (2010). A complexidade e importância prática do problema são as principais motivações deste trabalho. Para resolver o problema três algoritmos são propostos. Estes algoritmos são, respectivamente, baseados nas meta-heurísticas *Iterated Greedy* (IG), *Adaptive Large Neighborhood Search* (ALNS) e *Simulated Annealing* (SA).

2 DESCRIÇÃO DO PROBLEMA

O PSLTMPI em estudo é definido como em Chiang et al. (2010): Existe um conjunto $J = \{1,...,n\}$ de n tarefas e um conjunto $M = \{1,...,m\}$ de m máquinas idênticas, no qual cada tarefa $j \in J$ deve ser processada por apenas uma máquina $k \in M$. As tarefas possuem tamanhos unitários e as máquinas são de processamento em lote, ou seja, as tarefas podem ser agrupadas e processadas simultaneamente em uma máquina. Cada tarefa j possui uma família f_j, um tempo de processamento p_j, um tempo de chegada r_j (o processamento da tarefa somente pode começar após a chegada da tarefa), um peso w_j e uma data de entrega d_j. Todas as tarefas da mesma família f possuem o mesmo tempo de processamento, portanto somente tarefas da mesma família podem ser processadas juntas. Um lote B_b (com tarefas da mesma família f) estará disponível para ser processado no tempo $R_b = \max\{r_j \mid j \in B_b\}$.

O tempo de processamento de um lote é igual ao maior tempo de processamento entre as tarefas pertencentes ao lote. O tamanho máximo do lote é denotado por B, portanto cada máquina pode processar no máximo B tarefas ao mesmo tempo. Quando

uma máquina começa a processar um lote B_b de tarefas, nenhuma interrupção é permitida, assim, não é possível adicionar ou retirar tarefas do lote até que o processamento finalize.

O problema consiste em agrupar as tarefas em lotes e sequenciar os lotes nas máquinas a fim de minimizar o atraso total ponderado. O atraso da tarefa j é definido como $T_j = \max\{0, C_j - d_j\}$, onde C_j é o tempo de conclusão da tarefa j. O atraso total ponderado é determinado por TWT = $\sum_{i=1}^{n} w_j T_j$.

3 ALGORITMOS PROPOSTOS

Para resolver o PSLTMPI, neste trabalho são propostos três algoritmos baseados nas meta-heurísticas IG, ALNS e SA. Em todos os algoritmos, inicialmente é gerada uma solução inicial utilizando a heurística construtiva *Time Window Decomposition* (TWD), proposta por Bilyk et al. (2014). Essa solução é melhorada por um algoritmo de busca local. Os algoritmos são descritos nas próximas subseções.

3.1 SOLUÇÃO INICIAL

Para gerar uma solução inicial viável é utilizado o algoritmo construtivo TWD proposto por Bilyk et al. (2014). Esta heurística utiliza duas regras de prioridade. A primeira regra, denominada *Apparent Tardiness Cost* (ATC) proposta por Vepsalainen e Morton (1987), determina a prioridade da tarefa j ser inserida em um lote. O índice de prioridade para uma tarefa j é calculado como:

$$I_{ATC,j}(t) = \frac{w_j}{p_j} exp^{\frac{-(d_j - p_j - t + (r_j - t)^+)^+}{k\hat{p}}} \quad (1)$$

onde k é um parâmetro escalar, t é o tempo onde a próxima máquina estará disponível, $\hat{p}$ é a média do tempo de processamento das tarefas que ainda não foram processadas, e $x^+ = \max\{x, 0\}$.

A segunda regra é utilizada para escolher um lote que será sequenciado em uma máquina. A prioridade de um lote é calculada usando a regra BATC-II proposto por Bilyk et al. (2014). De acordo com essa regra, o valor do índice de um lote b com n_b tarefas é calculado como:

$$I_{BATC.II,b}(t) = \frac{n_b}{B} w_j p_j \sum_{j \in J(b)} exp \frac{-(d_j - p_j - t + (r_b - t)^+)^+}{k\hat{p}} \quad (2)$$

onde $J(b)$ é o conjunto de tarefas presentes no lote b e r_b é o maior tempo de chegada das tarefas desse lote. O algoritmo construtivo TWD é explicado nos seguintes passos:

Passo 1: Quando uma máquina se torna disponível no tempo t, considere a janela de tempo $(t, t + \Delta t)$, onde Δ representa um intervalo de tempo pré-definido. Defina o conjunto de tarefas $M(f, t, \Delta t) = \{ j \mid r_j \leq t + \Delta t, f(j) = f\}$ para cada família, isto é, tarefas da família f com tempo de chegada menor do que $t + \Delta t$.

Passo 2: Ordene as tarefas do conjunto $M(f, t, \Delta t)$ em ordem decrescente pelo valor do índice ATC e derive o conjunto de tarefas $M^*(f, t, \Delta t) = \{ j \mid j \in M(f, t, \Delta t), pos(j) \leq Njobs\}$, onde *Njobs* representa o número predefinido de tarefas selecionadas, com as primeiras *Njobs* tarefas de cada família. $pos(j)$ é a posição da tarefa j na sequência de tarefas que são incluídas no conjunto $M(f, t, \Delta t)$.

Passo 3: Considere todas os possíveis lotes das tarefas do conjunto $M^*(f, t, \Delta t)$. Atribua a cada lote um valor de índice usando BATC-II. O lote com o maior valor de índice entre os lotes das diferentes famílias é escolhido e inserido na máquina disponível.

Passo 4: Defina uma nova janela de tempo e volte ao Passo 1, se houver tarefas que não foram processadas.

A heurística TWD, para avaliar todas as formações de lotes, depende dos valores dos parâmetros k, Δt e *Njobs*. Assim como em Bilyk et al. (2014), os valores dos parâmetros utilizados foram $k = 0{,}1*h$ ($h = 1,...,50$), $\Delta t = \hat{p}/2$ e $Njobs = 15$.

3.2 BUSCA LOCAL

Neste trabalho, um algoritmo de busca local (BL) é utilizado para melhorar a solução inicial e as soluções resultantes do processo de perturbação das heurísticas. A BL é baseada em inserção de lotes, ou seja, um lote é removido da sua posição original e inserido em todas as outras possíveis posições do sequenciamento. Um lote pode ser inserido na mesma máquina de origem ou em outras máquinas.

O Algoritmo 1 apresenta o pseudocódigo da BL proposta. O processo de BL recebe como parâmetros de entrada uma solução S e o número máximo de lotes *batchLimit*, para serem removidos aleatoriamente da solução. A melhor solução

conhecida é armazenada na variável S_b (linha 2). Na linha 3, a função *removeRandom* remove aleatoriamente *batchLimit* lotes da solução *S* retornando a solução S_p. Na linha 4, a função *loteInsertion* insere *batchLimit* a solução parcial *Sp*. Neste processo é utilizado a estratégia de *best improvement*, assim, para cada lote removido, todas as possíveis inserções são testadas e a melhor solução da vizinhança S_b é escolhida. O processo de BL irá retornar a melhor solução obtida.

Algoritmo 1: BuscaLocal(S, *batchLimit*)

```
1 início
2     S_b = S;
3     S_p = removeRandom(S, batchLimit);
4     S* = loteInsertion(S_p, batchLimit);
5     se TWT(S*) < TWT(S_b) então
6         S_b = S*;
7     return S_b;
```

Fonte: Autores.

3.3 ITERATED GREEDY (IG)

Iterated Greedy (IG) é uma heurística simples e eficiente originalmente proposta por Ruiz e Stützle (2007), utilizada com muito sucesso para resolver diferentes problemas de sequenciamento de tarefas.

O pseudocódigo da heurística IG é apresentado no Algoritmo 2. Os parâmetros de entrada da heurística são: *Criterio_de_Parada*, *jobLimit* (número de tarefas que serão retiradas da solução *S* para formar a solução parcial S_p) e *batchLimit* (número de lotes retirados durante o processo de BuscaLocal). Na linha 2, uma solução inicial *S* é obtida utilizando a heurística construtiva TWD e em seguida melhorada pelo processo de BuscaLocal (linha 3). Nas linhas 4 e 5, o tamanho da destruição *t* (número de tarefas a serem retiradas da solução na etapa de destruição) e a melhor solução S_b são inicializados, respectivamente. Entre as linhas 6 a 20, as iterações do IG são realizadas até que o critério de parada seja satisfeito. Em cada iteração, a solução atual *S* é submetida ao procedimento de *Destruição_Construção* (linha 7) e a solução obtida é melhorada pelo processo de *BuscaLocal*, obtendo-se a solução S^* (linha 8).

A fase de *Destruição_Construção* é baseada na remoção e inserção de tarefas. Na primeira etapa a solução *S* é parcialmente destruída (*t* tarefas são removidas aleatoriamente obtendo-se uma solução parcial S_p). As tarefas removidas são armazenadas num conjunto *JR*. Na etapa de *Construção*, as tarefas removidas são reinseridas em S_p de forma gulosa. Uma tarefa *i* é aleatoriamente selecionada do conjunto

JR e inserida em todas os lotes da mesma família que não estejam cheios. A tarefa *i* é inserida na posição que resulte no menor valor de atraso ponderado total para a solução S_p. Esse processo é repetido até que todas as tarefas do conjunto *JR* tenham sido reinseridas em S_p, obtendo-se uma nova solução completa *S*.

Algoritmo 2: IG(*jobLimit*, *batchLimit*)

```
1  início
2      S := Solucao_Inicial();
3      S := BuscaLocal(S, batchLimit);
4      t := 1;
5      S_b := S;
6      enquanto Critério_de_Parada faça
7          S* := Destruição_Construção(S, t);
8          S* := BuscaLocal(S*, batchLimit);
9          Δ := TWT(S*)−TWT(S);
10         se TWT(S*) < TWT(S) então
11             t := 1;;
12             S := S*;
13             se TWT(S*) < TWT(S_b) então
14                 S_b := S*;
15         senão
16             t := t + 1;
17             se t > jobLimit então
18                 t := 1;
19             se rand(0, 1) ≤ exp(−Δ)/T) então
20                 S := S*
```

Fonte: Autores.

Entre as linhas 9 a 14 do Algoritmo 2, se a solução S^* for melhor que a solução atual *S*, o tamanho da destruição *t* é reinicializado em 1 e a melhor solução S_b obtida é atualizada. Se a solução atual não é melhorada, o tamanho da destruição *t* é incrementado em um (linha 16). O valor máximo que *t* pode assumir é limitado por *jobLimit*. Se *t* excede *jobLimit*, o valor de *t* é novamente 1 (linhas 17 e 18). Se a solução S^* é admitida pelo critério de aceitação (linha 19 e 20), a solução atual *S* é substituída pela solução S^*, mesmo sendo uma solução pior (regra de Metrópolis). No critério de aceitação, a temperatura *T* é calculado por $T = \sum_{j=1}^{n} \frac{p_j}{100 \times n \times m}$, onde p_j é o tempo de processamento da tarefa *j*, *n* é o número de tarefas e *m* é o número de máquinas.

3.4 ADAPTIVE LARGE NEIGHBORHOOD SEARCH (ALNS)

ALNS (Ropke e Pisinger (2006)) é uma heurística de busca local na qual vários métodos de busca simples competem para modificar a solução atual analisando estatisticamente o histórico de desempenho dos métodos. Em cada iteração um método é

escolhido para destruir a solução atual, e um outro método é escolhido para reparar a solução. A nova solução é aceita se satisfizer alguns critérios definidos pela heurística.

O Algoritmo 3 apresenta o pseudocódigo da heurística ALNS. Uma solução inicial é determinada utilizando a heurística TWT (linha 2), e melhorada pela busca local (linha 3). A melhor solução encontrada S_{best} é inicializada na linha 4. As iterações do algoritmo são executadas entre as linhas 5 a 23, até que o critério de parada seja satisfeito.

Algoritmo 3: ALNS($nMax$, $taskLimit$, $batchLimit$, ρ, $\sigma_1, \sigma_2, \sigma_3$)

```
1  início
2      S := Solucao_Inicial();
3      S := BuscaLocal(S, batchLimit);
4      S_best := S;
5      enquanto Critério_de_Parada faça
6          Iter := 0; t := 0;
7          enquanto Iter < nMax faça
8              Selecione uma vizinhança de destruição N_i ∈ N^-;
9              S* := Destruição(S, N_i);
10             Selecione uma vizinhança de reparação N_j ∈ N^+;
11             S* := Reparação(S*, N_j);
12             Δ = TWT(S*)−TWT(S);
13             se TWT(S*) < TWT(S) então
14                 S := S*;
15                 se TWT(S*) < TWT(S_best) então
16                     S_best := S*;
17             senão
18                 se rand(0,1) ≤ exp(−Δ)/T então
19                     S := S*
20             Atualiza os vetores de resultados π^- e π^+;
21             Iter := Iter + 1;
22         t = t + 1;
23         Atualiza os pesos λ_i^- e λ_j^+ das vizinhanças N_i ∈ N^- e N_j ∈ N^+;
24     Retorna S_best
```

Fonte: Autores.

Nas linhas 7 a 21, a solução atual S é submetida aos métodos de Destruição e Reparação. Estes métodos são baseados em buscas em vizinhanças. As vizinhanças são escolhidas de acordo ao desempenho da solução em iterações passadas. O conjunto de vizinhanças de destruição e reparo são denotados por N^- e N^+, respectivamente. Após a aplicação das vizinhanças, uma nova solução S^* é gerada a partir da solução S (linha 11). Entre as linhas 12 a 19 ocorre a verificação do critério de aceitação. Como no algoritmo IG, a heurística ALNS também utiliza o critério de aceitação baseada na regra de Metrópolis. Na linha 24, os pesos das vizinhanças são atualizados. A linha 23 retorna a melhor solução encontrada.

3.4.1 Métodos de destruição e reparação

A heurística ALNS implementada utiliza diferentes vizinhanças de destruição e reparo. As vizinhanças recebem como entrada uma solução *S* e o número de tarefas *taskLimit* a serem removidas ou inseridas (ou seja, um processo de destruição e reparo é repetido *taskLimit* vezes).

Vizinhanças de destruição

JobRemove(*S, taskLimit*)}: Remove aleatoriamente uma tarefa de um lote.

JobRemoveWorst(*S, taskLimit*): Escolhe aleatoriamente um lote e remove a tarefa que possuir maior valor de atraso ponderado.

JobRemoveReady(*S, taskLimit*): Escolhe aleatoriamente um lote e remove a tarefa com maior valor de tempo de chegada.

JobRemoveDueDate(*S, taskLimit*): Escolhe aleatoriamente um lote e remove a tarefas com menor valor de data de entrega.

BatchRemove(*S, taskLimit*)}: Escolhe um lote e remove todas as tarefas desse lote. A probabilidade do lote ser selecionado é igual ao atraso do lote dividido pelo atraso total dos lotes (Kim et al. (2002)).

Vizinhanças de reparação

JobInsertRandom(*S, taskLimit*): Uma tarefa é inserida aleatoriamente em um dos lotes da mesma família. Caso não exista um lote disponível da mesma família, forma-se um novo lote com a tarefa e ele é inserido de forma aleatória em uma máquina.

JobInsertOrd(*S, taskLimit*): As tarefas são ordenadas em ordem decrescente do índice ATC. Este índice determina a prioridade de uma tarefa *j* ser inserida em um lote.

JobInsertBest(*S, taskLimit*): Esta vizinhança consiste em verificar qual o melhor lote para inserir uma tarefa. Assim, testa-se todas as possíveis inserções para a tarefa e escolhe-se o lote que gere o menor valor de TWT. Se não houver lotes da mesma família disponíveis, um novo lote é formado com a tarefa e é inserido em uma máquina.

JobInsert(*S, taskLimit*): Esta vizinhança verifica se é melhor inserir uma tarefa em um lote ou formar um novo lote com a tarefa. Para isto, testa-se todas as inserções possível para a tarefa e a formação de um novo lote também é verificada. Assim, testa-se, se o TWT gerado é menor ao se inserir a tarefa ou ao formar um novo lote.

Como em Pisinger e Ropke, S. (2007), os pesos λ_j^- e λ_j^+ são designados para cada vizinhança de destruição e reparo, respectivamente. Inicialmente, todas as vizinhanças possuem o mesmo valor de peso (igual a 1). No decorrer das iterações do ALNS, quanto mais uma vizinhança N_i contribui para a melhoria da solução, um peso maior é atribuído e, portanto maior será a probabilidade da vizinhança ser escolhida. Sejam N^- e N^+ os conjuntos de vizinhanças de destruição e reparo, respectivamente ($n_1 = |N^-|$, $n_2 = |N^+|$). Em cada iteração do ALNS, o princípio *roulette wheel* é usado para selecionar uma vizinhança de destruição e reparo. As probabilidades para selecionar uma vizinhança de destruição ($N_i \in N^-$) e reparação ($N_j \in N^+$) são definidas, respectivamente, por $prob_i^- = \frac{\lambda_i^-}{\sum_{k=1}^{n1} \lambda_k^-}$ e $prob_j^+ = \frac{\lambda_j^+}{\sum_{k=1}^{n2} \lambda_k^+}$.

O ALNS utiliza um ajuste automático para atualizar os valores dos pesos λ_j^- e λ_j^+. Estes pesos são atualizados após um número *nMax* de iterações (bloco 7-21 do Algoritmo 3). A ideia básica é acompanhar uma pontuação (*score*) para cada vizinhança, que mede o desempenho recente da vizinhança. Um *score* alto corresponde a uma vizinhança bem sucedida. Os *scores* das vizinhanças de destruição e reparo são armazenados nos vetores $\pi^- = (\pi_i^-)$ e $\pi^+ = (\pi_j^+)$, respectivamente. O *score* de cada vizinhança é definido como zero no início de cada bloco *t* (bloco 7-21). A cada iteração, os *scores* π_i^- e π_i^+ podem ser incrementados utilizando alguns parâmetros pré-estabelecidos. Esses valores representam o quão boa foi a solução encontrada após a aplicação da vizinhança. Portanto, em cada iteração, se a solução obtida na iteração atual for a melhor global, o peso da vizinhança é incrementado por σ_1. Se a solução obtida na iteração não havia sido encontrada ainda e melhora a melhor solução atual, o peso da vizinhança é incrementando em σ_2, caso contrario, o peso é incrementado em σ_3. σ_1, σ_2 e σ_3 são parâmetros do ALNS (Ropke e Pisinger (2006)). No final de cada bloco *t*, (a cada *nMax* iterações) calculam-se novos pesos usando os *scores* registrados. Sejam λ_{it}^- e λ_{jt}^+ os pesos das vizinhanças $N_i \in N^-$ e $N_j \in N^+$ usadas no bloco *t*. Após finalizar o bloco *t*, calculam-se os pesos para todas as vizinhanças a serem usadas no bloco $t + 1$ da seguinte forma:

$$\lambda_{i,t+1}^- = \lambda_{it}^-(1-\rho) + \rho\frac{\pi_i^-}{a_i} \quad ; \quad \lambda_{j,t+1}^+ = \lambda_{jt}^+(1-\rho) + \rho\frac{\pi_j^+}{a_j} \qquad (3)$$

onde π_i^- e π_j^+ são, respectivamente, os *scores* das vizinhanças $N_i \in N^-$ e $N_j \in N^+$ obtidas durante o bloco t, a_i (a_j) é o número de vezes que a vizinhança N_i (N_j) foi utilizada durante o bloco t, e $\rho \in [0,1]$ é o fator de reação que controla o impacto dos pesos.

3.5 SIMULATED ANNEALING (SA)

A heurística SA utilizada neste trabalho é baseada no estudo de Damodaran e Vélez-Gallego (2012), aplicado no contexto de um grupo de máquinas idênticas de processamento de lotes. O SA consiste em inicializar o problema com uma solução com número de lotes reduzidos e pesquisar por soluções melhores sem que o número de lotes seja alterado. Essa pesquisa é realizada até que não ocorra melhora em um número fixo de iterações. Após essas iterações um novo lote é criado e a pesquisa continua. A ideia consiste em controlar o número de lotes criados.

Neste trabalho propõe-se uma heurística SA na qual a vizinhança de soluções é gerada por sete mecanismos de perturbação. A heurística utiliza múltiplos reinícios, isto é, quando a temperatura tem um valor muito próximo de zero, o algoritmo é reiniciado da melhor solução obtida.

Algoritmo 4: SA(T_0, α, $nMax$)

```
1  início
2      S = Solucao_Inicial();
3      S* = BuscaLocal(S, batchLimit);
4      S_best = S*;
5      enquanto Criterio_de_Parada faça
6          T = T0;
7          enquanto T > 0.0001 faça
8              Iter = 0;
9              enquanto Iter < nMax faça
10                 S' = Processo_de_Pertubação(S);
11                 Δ = TWT(S')−TWT(S);
12                 se TWT(S') < TWT(S) então
13                     S = S';
14                     se TWT(S') < TWT(S*) então
15                         S* = S';
16                 senão
17                     se random(0,1) < exp^(−Δ/T) então
18                         S = S';
19                 Iter = Iter + 1;
20             T = α × T ;
21             se TWT(S*) < TWT(S_best) então
22                 S_best = S*;
23                 S= Adiciona_NovoLote(S*);
24             senão
25                 S= Delete_Lote(S*);
26     Retorna S_best;
```

Fonte: Autores.

O pseudocódigo da heurística SA é apresentado no Algoritmo 4. Na linha 2, uma solução inicial *S* é determinada utilizando a heurística TWD. A solução S é melhorada aplicando busca local (linha 3). As iterações do algoritmo são realizadas entre as linhas 5 a 29, até que o critério de parada seja satisfeito. Na linha 6, a temperatura *T* é inicializada. Para um dado valor de temperatura *T*, o algoritmo realiza um número fixo de iterações *nMax* (linha 9) nas quais a solução atual *S* é submetida ao processo de perturbação gerando a solução *S'* (linha 10). Entre as linhas 11 a 18, o critério de aceitação é verificado. No SA, uma solução pior é aceita com uma dada probabilidade dada por $exp^{\frac{-\Delta}{T}}$, onde Δ é a diferença de custo entre a nova solução *S'* e a solução atual *S*. Na linha 22, a temperatura *T* é decrementada por um fator α. O SA proposto neste trabalho também utiliza dois métodos de perturbação denominados *Adiciona_NovoLote* e *Deleta_Lote*. Esses métodos são utilizados para controlar o número de lotes da solução. No primeiro método um lote é criado e adicionado ao processamento. Neste método, uma tarefa é selecionada de forma aleatória de um lote e um novo lote é formado e adicionado ao processamento. O novo lote formado pela tarefa é sequenciado em uma máquina

aleatória. No segundo método, verifica-se se é possível excluir um lote do processamento. A retirada de um lote consiste em verificar se é possível retirar as tarefas pertencentes ao lote e se a retirada do lote levará a um valor de TWT menor que o da solução atual. Se o valor TWT da solução S^* for menor que o valor da melhor solução global S_{best}, o valor de S_{best} é atualizado (linha 22) e o método *Adiciona_NovoLote*(S^*) (linha 23) é aplicado. Caso não ocorra uma melhora, então aplica-se o método *Delete_Lote*(S^*) (linha 25) para verificar se é possível remover um lote. O SA retorna a melhor solução encontrada.

O Processo de Perturbação utilizado na heurística SA consiste em uma adaptação da perturbação utilizada em Kim et al. (2002). Esta etapa possui um conjunto de vizinhanças na qual em cada iteração um vizinho de cada vizinhança é gerado. O vizinho que gerar o menorTWT é escolhido. As seguintes vizinhanças são utilizadas:

swapBatch: Seleciona dois lotes da mesma família e troca suas posições. O primeiro lote é selecionado baseado na taxa de atraso, ou seja, a probabilidade de um lote ser selecionado é igual a soma do atraso das tarefas pertences ao lote dividido pelo atraso total do processamento (Kim et al. (2002)).

batchInsert: Um lote é escolhido de forma aleatória e é inserido em uma nova posição em uma máquina também escolhida de forma aleatória.

batchMerge: Seleciona aleatoriamente dois lotes da mesma família e retiram-se as tarefas de ambos os lotes. Essas tarefas são reinseridas aleatoriamente aos lotes.

batchDiv: Um lote, escolhido de forma aleatória, é dividido em dois novos lotes. Os lotes criados são inseridos de forma aleatória em máquinas escolhidas aleatoriamente.

swapJob: Escolhe dois lotes da mesma família de forma aleatória, então seleciona-se duas tarefas para realizar a troca entre os lotes.

jobInsertReadyTime: Seleciona aleatoriamente um lote e nesse lote seleciona a tarefa com menor valor de tempo de chegada, r_j. Retira-se essa tarefa e aleatoriamente adiciona a outro lote da mesma família que não esteja completo ou um novo lote é formado com essa tarefa.

jobInsertDueDate: Similar ao método *jobInsertReadyTime,* no entanto, esse método seleciona a tarefa com menor valor da data de entrega d_j.

4 EXPERIMENTOS COMPUTACIONAIS

Nesta seção, descrevem-se os experimentos computacionais e análise dos resultados obtidos. As heurísticas desenvolvidas IG, ALNS e SA são comparados com os algoritmos AM proposto por Chiang et al. (2010), e VNS proposto por Bilyk et al. (2014).

Os experimentos computacionais são realizados utilizado um conjunto de 4860 instâncias de grande porte que foram geradas e disponibilizadas por Chiang et al. (2010). As características das instâncias são apresentadas na Figura 1.

Figura 1: Características das instâncias de Chiang et al. (2010).

Parâmetros	**Valores**
Número de Famílias (f)	3, 6, 12
Número de Máquinas (m)	3, 4, 5
Número de Tarefas (n)	180, 240, 300
Tamanho Máximo do Lote (B)	4, 8
Tempo de Processamento da Família (p_j)	2, 4, 10, 16 e 20 com Probabilidade de distribuição dos tempos 0.2, 0.2, 0.3, 0.2 e 0.1, respectivamente
Peso da Tarefa (w_j)	~U(0,1)
Tempo de Chegada (r_j)	$r_j \sim \mathrm{U}(0,\ \alpha \sum p_j/(m*B))$ $\alpha \in \{0.25, 0.5, 0.75\}$
Data de Entrega (d_j)	$d_j - r_j \sim \mathrm{U}(0,\ \beta \sum p_j/(m*B))$ $\beta \in \{0.25, 0.5, 0.75\}$

Fonte: Autores.

Os resultados do algoritmo AM de Chiang et al. (2010) são disponibilizados para todas as 4860 instâncias. Os resultados do algoritmo VNS de Bilyk et al. (2014) não foram disponibilizados. Para que as comparações de resultados sejam realizadas de forma justa, os algoritmos AM e VNS foram reimplementados (seguindo os detalhes dos respectivos artigos) e foram executados no mesmo computador utilizando a mesma condição de parada. Neste trabalho, AM denota o Algoritmo Memético original implementado por Chiang et al. (2010). Denota-se por AM2 o algoritmo AM reimplementado neste trabalho. VNS é o algoritmo proposto por Bilyk et al. (2014) reimplementado neste trabalho.

Todos os algoritmos (IG, ALNS, AS, AM2 e VNS) foram implementados em C++ e executados em um Cluster no qual cada nó possui 2 processadores Intel(R) Xeon(R) CPU X5650 (12M Cache, 2.66 GHz, 6.40 GT/s Intel(R) QPI, 6 cores, 12 threads) e 24 GB de RAM. Os experimentos apresentados neste trabalho foram executados utilizando apenas 1 processador, 4 threads e 4GB de RAM. O critério de parada (*Critério_de_Parada*) utilizado em todos os algoritmos é um limite de tempo de CPU definido como 0,1**n* segundos, onde *n* é o número de tarefas da instância resolvida.

Para avaliar o desempenho das heurísticas IG, ALNS, AS, AM2 e VNS, utiliza-se a métrica RPD (*Relative Percentage Deviantion*). O RPD é definido como:

$$RPD = 100 \times \frac{TWT_h - TWT_{best}}{TWT_{best}}\% \qquad (4)$$

onde TWT_h é valor do TWT obtido por uma heurística e TWT_{best} é o menor valor de TWT para uma instância encontrada entre todas as heurísticas comparadas. Quanto menor o valor do RPD, melhor será a qualidade da solução obtida pela heurística.

Utiliza-se também a métrica RPI (*Relative Percentage Improve*) para avaliar a porcentagem de melhoria provocada por uma heurística em relação ao algoritmo original MA proposto por Chiang et al. (2010). A métrica RPI é definida como:

$$RPI = 100 \times \frac{TWT_{MA} - TWT_h}{TWT_{MA}}\% \quad (5)$$

onde TWT_{MA} corresponde ao valor TWT obtido pelo algoritmo MA proposto por Chiang et al. (2010) para uma determinada instância e TWT_h é o menor valor de TWT encontrada pela heurística. Quanto maior o valor da métrica RPI, maior será a melhoria da heurística em relação ao algoritmo MA.

4.1 CALIBRAÇÃO DOS PARÂMETROS DAS HEURÍSTICAS

Os parâmetros algoritmos IG, ALNS e SA foram calibrados utilizando um conjunto de 486 instâncias geradas de acordo com os parâmetros adotados na Figura 1. Estes algoritmos foram executados 10 vezes para cada instância. Os resultados obtidos são analisados utilizando a métrica RPD. Uma Análise de Variância (ANOVA) multi-fator é realizada para validar os resultados obtidos. Os dados utilizados para os testes da ANOVA são gerados utilizando o método Full Factorial DOE (*Design Of Experiment*), que consiste em gerar todas as combinações de parâmetros (fatores) utilizados na calibração de uma heurística (Eriksson et al. (2000)). Dada a limitação de espaço, os resultados do teste ANOVA não são apresentados.

Para o algoritmo IG, a configuração que apresentou menor média de RPD foi encontrada com os seguintes valores dos parâmetros: *jobLimit* = 0,25**n*, *batchLimit* = 0,25**nL* (onde *nL* é o número de lotes da solução). Os valores dos parâmetros da melhor configuração do ALNS foram: *nMax* = 50, *taskLimit* = 0,25**n*, *batchLimit* = 0,3**nL*, ρ = 0,1, σ_1 = 30, σ_2 = 10 e σ_3 = 15. Para o algoritmo AS, os melhores valores dos parâmetros foram: T_0 = 20000, α = 0,4, *nMax* = *n*/30 e *batchLimit* = 0,3**nL*.

4.2 COMPARAÇÕES ENTRE MÉTODOS HEURÍSTICOS

Nesta seção comparam-se os algoritmos IG, ALNS, AS, MA2 e VNS utilizando as instâncias da literatura (Chiang et al. (2010)). Todos os algoritmos foram executados com o mesmo critério de parada (tempo de CPU igual a 0,1*n segundos). Os algoritmos foram executados 10 vezes para resolver cada instância, e nas comparações considera-se o melhor resultado encontrado. Como em Bilyk et al. (2014), para análise dos resultados, as instâncias são agrupadas em de acordo aos parâmetros do problema.

No primeiro experimento, as heurísticas propostas neste trabalho, ALNS, IG e SA, assim como os algoritmos MA2 e VNS, são comparados com os resultados do algoritmo MA de Chiang et al. (2010). Utilizando a métrica RPI, verifica-se a melhoria de cada heurística em relação ao algoritmo MA.

Tabela 1: Porcentagem de melhoria das heurísticas implementadas em relação ao algoritmo MA.

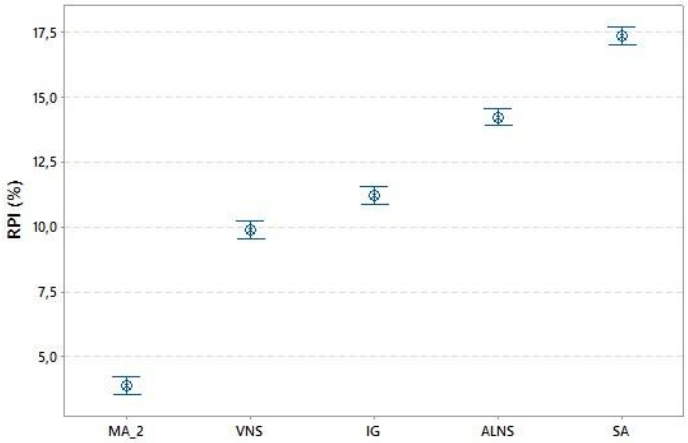
Fator	**Nível**	**AM2**	**VNS**	**IG**	**ALNS**	**SA**
Número de Famílias	3	8,09	13,65	15,01	20,03	**23,66**
	6	4,40	9,21	10,34	13,01	**15,83**
	12	3,79	6,85	8,30	9,70	**12,61**
Número de Máquinas	3	5,87	10,43	13,05	15,47	**18,82**
	4	5,32	9,67	10,79	13,81	**17,16**
	5	5,09	9,61	9,83	13,46	**16,12**
Tamanho Máximo dos Lotes	4	5,97	10,44	13,08	16,46	**20,94**
	8	4,89	9,37	9,36	12,03	**13,80**
Tempo de Chegada	Próximo de zero	4,71	7,66	7,90	9,46	**11,81**
	Moderado	5,88	9,81	11,21	13,94	**17,48**
	Alto	5,69	12,24	14,54	19,32	**22,82**
Data de Entrega	Próximo de zero	2,97	4,91	5,60	6,97	**8,64**
	Moderado	5,43	8,71	11,12	12,63	**15,29**
	Alto	7,89	16,08	15,94	23,13	**28,18**
Média		5,43	9,90	11,22	14,25	**17,37**

Fonte: Autores.

A Tabela 1 mostra os valores dos RPIs médios das heurísticas, para cada categoria de instância. Os maiores percentuais de melhoria estão em negrito. Observa-se que todas as heurísticas implementadas apresentaram uma melhoria significante com relação ao algoritmo AM da literatura. A versão reimplementada AM2 obteve resultados melhores com relação aos resultados originais do AM disponíveis na literatura (em média, AM2 obteve uma melhoria de 5,43% em relação ao AM). Observa-se que a heurística SA se sobressaiu para todas as instâncias (obteve as maiores melhorias). Na Tabela 1, é possível observar também que o algoritmo ALNS apresentou bons resultados seguido pelos

algoritmos IG e VNS. Ou seja, os algoritmos propostos neste trabalho apresentaram um desempenho superior em comparação aos algoritmos da literatura, MA2 e VNS. Em média, as melhorias de SA, ALNS, IG e VNS (em relação ao AM da literatura) foram de 17,37%, 14,25%, 11,22% e 9,90%, respectivamente. De acordo com as características das instâncias do problema, a melhoria das heurísticas, geralmente, aumenta para instâncias com tempos de chegada e datas de entrega altos. Isto mostra que, instâncias com tempos de chegada e datas de entrega apertados (próximo de zero) são mais difíceis de melhorar. Também se observa que, quanto menor o número de famílias das tarefas e menor os tamanhos dos lotes, maior será a melhoria das heurísticas.

Figura 2: Médias e intervalos HSD com 95% nível de confiança para comparação do RPI das heurísticas.



Fonte: Autores.

Os resultados apresentados na Tabela 1 podem ser melhor analisados e visualizados na Figura 2. Esta Figura mostra o gráfico de médias e os intervalos HSD (*Honestly Significant Difference*) de Tukey com nível de confiança de 95% obtidas na execução de um teste ANOVA. Observa-se que o algoritmo SA apresenta os melhores valores de RPI (SA apresentou uma maior melhoria em relação ao algoritmo MA). Pelos resultados obtidos, existem diferenças estatisticamente significantes entre as heurísticas comparadas (seus intervalos não se sobrepõem).

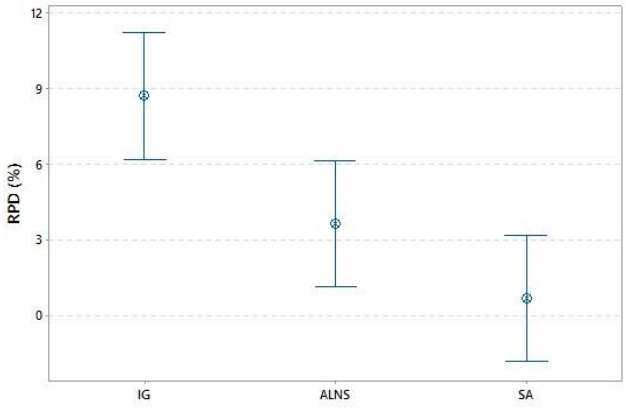
Um segundo experimento consiste em comparar as heurísticas propostas neste trabalho ALNS, IG e SA. Os resultados são comparados utilizando a métrica RPD. Também, os resultados são agrupados em função dos principais parâmetros do problema. O intuito deste experimento consiste em verificar o desempenho das heurísticas propostas para o problema. A Tabela 2 mostra os valores dos RPDs médios dos algoritmos, para cada categoria de instância. As melhores médias estão em negrito.

Tabela 2: Porcentagem de melhoria das heurísticas implementadas em relação ao algoritmo MA.

Fator	Nível	IG	ALNS	SA
Número de Famílias	3	5,18	2,45	**0,49**
	6	15,89	5,96	**1,07**
	12	5,12	2,58	**0,54**
Número de Máquinas	3	5,69	2,92	**0,74**
	4	10,88	4,21	**0,89**
	5	9,61	3,84	**0,47**
Tamanho Máximo dos Lotes	4	11,19	5,63	**0,76**
	8	6,27	1,68	**0,64**
Tempo de Chegada	Próximo de zero	6,88	3,29	**1,32**
	Moderado	8,38	3,17	**0,48**
	Alto	10,93	4,52	**0,30**
Data de Entrega	Próximo de zero	3,03	**0,62**	0,89
	Moderado	8,60	4,05	**0,62**
	Alto	14,57	6,31	**0,58**
Média		8,73	3,66	**0,70**

Fonte: Autores.

Figura 3: Médias e intervalos HSD com 95% nível de confiança para comparação do RPD das heurísticas.



Fonte: Autores.

A fim de validar os resultados obtidos pelos algoritmos ALNS, IG e SA, e verificar se as diferenças observadas são estatisticamente significantes, feita uma análise estatística usando o RPD como variável de resposta. Esta análise pode ser exibida na Figura 3, que mostra o gráfico de médias e os intervalos de Tukey HSD com 95% de nível de confiança.

Na Figura 3 é possível observar que há diferenças significativas entre as heurísticas propostas IG e ALNS. Observa-se que os resultados do ALNS e SA são estatisticamente equivalentes (seus intervalos estão sobrepostos indicando que não existe diferença estatisticamente significante entre as médias). Analisando a média total de RPD, é possível verificar que o algoritmo SA obteve o menor média de RPD (obteve o melhor desempenho entre os algoritmos propostos).

5 CONCLUSÕES

Neste trabalho foram propostas três heurísticas, IG, ALNS e SA, para a minimização do atraso total ponderado no problema de sequenciamento de tarefas em máquinas paralelas de processamento em lote, no qual as tarefas possuem tempos diferentes de chegada e pertencem a famílias incompatíveis (tarefas de diferentes famílias não podem ser processadas juntas).

Os desempenhos das heurísticas propostas foram comparados com o algoritmo memético AM de Chiang et al. (2010) e VNS de Bilyk et al. (2014). Os resultados foram avaliados considerando o percentual de melhoria com relação aos melhores resultados disponíveis na literatura. As abordagens propostas foram capazes de produzir soluções de alta qualidade. O SA foi o algoritmo que apresentou o melhor desempenho seguido ALNS. Os algoritmos propostos AS, ALNS e IG obtiveram melhorias significativas em relação ao algoritmo AM de Chiang et al. (2010). Em média, as melhorias foram de 17.37%, 14.25% e 11,22%, respectivamente.

AGRADECIMENTOS

Os autores agradecem o apoio financeiro da CAPES e FAPEMIG.

REFERÊNCIAS

Bilyk, A., Mönch, L., & Almeder, C. (2014). Scheduling jobs with ready times and precedence constraints on parallel batch machines using metaheuristics. *Computers & Industrial Engineering*, 78, 175-185.

Cheng, H. C., Chiang, T. C., & Fu, L. C. (2008). A memetic algorithm for parallel batch machine scheduling with incompatible job families and dynamic job arrivals. In *2008 IEEE International Conference on Systems, Man and Cybernetics* (pp. 541-546).

Chiang, T. C., Cheng, H. C., & Fu, L. C. (2010). A memetic algorithm for minimizing total weighted tardiness on parallel batch machines with incompatible job families and dynamic job arrival. *Computers & Operations Research*, 37(12), 2257-2269.

Damodaran, P., & Velez-Gallego, M. C. (2010). Heuristics for makespan minimization on parallel batch processing machines with unequal job ready times. *The International Journal of Advanced Manufacturing Technology*, 49(9), 1119-1128.

Damodaran, P., & Vélez-Gallego, M. C. (2012). A simulated annealing algorithm to minimize makespan of parallel batch processing machines with unequal job ready times. Expert systems with Applications, 39(1), 1451-1458.

Eriksson, L., Johansson, E., Kettaneh-Wold, N., Wikström, C., & Wold, S. (2000). Design of experiments. Principles and Applications, Learn ways AB, Stockholm.

Herr, O., & Goel, A. (2016). Minimising total tardiness for a single machine scheduling problem with family setups and resource constraints. *European Journal of Operational Research*, 248(1), 123-135.

Jia, Z. H., Li, K., & Leung, J. Y. T. (2015). Effective heuristic for makespan minimization in parallel batch machines with non-identical capacities. *International Journal of Production Economics*, 169, 1-10.

Jia, Z. H., Wang, C., & Leung, J. Y. T. (2016). An ACO algorithm for makespan minimization in parallel batch machines with non-identical job sizes and incompatible job families. *Applied Soft Computing*, 38, 395-404.

Kim, D. W., Kim, K. H., Jang, W., & Chen, F. F. (2002). Unrelated parallel machine scheduling with setup times using simulated annealing. Robotics and Computer-Integrated Manufacturing, 18(3-4), 223-231.

Lausch, S., & Mönch, L. (2016). Metaheuristic approaches for scheduling jobs on parallel batch processing machines. In *Heuristics, Metaheuristics and Approximate Methods in Planning and Scheduling* (pp. 187-207). Springer, Cham.

Malve, S., & Uzsoy, R. (2007). A genetic algorithm for minimizing maximum lateness on parallel identical batch processing machines with dynamic job arrivals and incompatible job families. *Computers & Operations Research*, 34(10), 3016-3028.

Mönch, L., Balasubramanian, H., Fowler, J. W., & Pfund, M. E. (2005). Heuristic scheduling of jobs on parallel batch machines with incompatible job families and unequal ready times. *Computers & Operations Research*, 32(11), 2731-2750.

Pinedo, M. L. (2012). Scheduling: theory, algorithms, and systems. New York: Springer.

Pisinger, D., & Ropke, S. (2007). A general heuristic for vehicle routing problems. Computers & operations research, 34(8), 2403-2435.

Potts, C. N., & Kovalyov, M. Y. (2000). Scheduling with batching: A review. European journal of operational research, 120(2), 228-249.

Ropke, S., & Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation science, 40(4), 455-472.

Ruiz, R., & Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European journal of operational research, 177(3), 2033-2049.

Vepsalainen, A. P., & Morton, T. E. (1987). Priority rules for job shops with weighted tardiness costs. Management science, 33(8), 1035-1047.