Universidade Federal da Paraíba
Centro de Informática
Graduação em Engenharia da Computação

Uma abordagem heurística para o problema de escalonamento de tarefas em uma máquina com datas de liberação e tempos de setup dependentes da sequência

Rafael Sobral de Morais

João Pessoa - PB
2023

Rafael Sobral de Morais

Uma abordagem heurística para o problema de escalonamento de tarefas em uma máquina com datas de liberação e tempos de setup dependentes da sequência

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal da Paraíba (UFPB), como requisito para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: Prof. Dr. Anand Subramanian

João Pessoa - PB
2023

Catalogação na publicação
Seção de Catalogação e Classificação

M828a Morais, Rafael Sobral de.
Uma abordagem heurística para o problema de escalonamento de tarefas em uma máquina com datas de liberação e tempos de setup dependentes da sequência / Rafael Sobral de Morais. - João Pessoa, 2023.
33 f. : il.

Orientação: Anand Subramanian.
TCC (Graduação) - UFPB/CI.

1. Sequenciamento. 2. Setups dependentes da sequência. 3. Iterated local search. 4. Makespan. I. Subramanian, Anand. II. Título.

UFPB/CI CDU 004

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

Rafael Sobral de Morais

Uma abordagem heurística para o problema de escalonamento de tarefas em uma máquina com datas de liberação e tempos de setup dependentes da sequência

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal da Paraíba (UFPB), como requisito para obtenção do grau de Bacharel em Engenharia da Computação.

Trabalho aprovado. João Pessoa - PB, 17 de novembro de 2023:

gov.br Documento assinado digitalmente
ANAND SUBRAMANIAN
Data: 21/11/2023 13:18:43-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Anand Subramanian
Orientador

gov.br Documento assinado digitalmente
TEOBALDO LEITE BULHOES JUNIOR
Data: 21/11/2023 16:39:39-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Teobaldo Bulhões
Examinador

gov.br Documento assinado digitalmente
BRUNO PETRATO BRUCK
Data: 21/11/2023 17:08:44-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Bruno Bruck
Examinador

João Pessoa - PB
2023

Resumo

Este trabalho aborda o problema de sequenciamento de tarefas em uma máquina considerando datas de liberação e tempos de *setup* não-antecipatórios e dependentes da sequência, com o objetivo de minimizar o *makespan*. Para isso foi proposta uma abordagem heurística híbrida que combina *iterated local search* e *beam search*. Essa abordagem inclui uma estratégia eficiente de avaliação de movimentos na busca local, baseada na concatenação de subsequências, que possibilita avaliação em tempo constante amortizado. Foram conduzidos experimentos computacionais abrangendo um conjunto de 1800 instâncias retiradas da literatura, com tamanhos variando de 25 a 150 tarefas. Os resultados obtidos foram comparados com os das melhores heurísticas conhecidas para esse problema, demonstrando que o método proposto é capaz de gerar soluções altamente competitivas.

Palavras-chave: Sequenciamento. *Setups* dependentes da sequência. *Iterated local search*. *Makespan*.

Abstract

This work addresses the problem of task sequencing on a single machine, considering release dates and non-anticipatory, sequence-dependent setup times, with the aim of minimizing the makespan. For this purpose, a hybrid heuristic approach combining iterated local search and beam search has been proposed. This approach includes an efficient strategy for evaluating moves in local search, based on concatenating subsequences, enabling constant amortized time evaluation. Computational experiments were conducted on a set of 1800 instances taken from the literature, with sizes ranging from 25 to 150 tasks. The results obtained were compared with those of the best-known heuristics for this problem, demonstrating that the proposed method is capable of generating highly competitive solutions.

Keywords: Scheduling. Sequence-dependent setups. Iterated local search. Makespan.

Lista de tabelas

Lista de ilustrações

Sumário

1 Introdução

1.1 Definição do Tema

A eficiência da produção em ambientes industriais está diretamente ligada à gestão do tempo e dos recursos disponíveis. As indústrias necessitam adaptar seus processos para aumentar a produção ou reduzir o tempo de trabalho. Nesse sentido, o sequenciamento de tarefas (*scheduling*) aparece como um processo de tomada de decisão usado regularmente em manufaturas e serviços industriais, lidando com a alocação de recursos em determinado período de tempo, visando otimizar um ou mais objetivos (PINEDO, 2012).

Dentre as funções objetivo mais comuns estão a minimização do *makespan*, que é o tempo de conclusão da última tarefa no sistema, maximização da produção em um horizonte de tempo, minimização da soma dos atrasos (*tardiness* ou *lateness*), minimização do atraso máximo ou da quantidade de atrasos.

Devido à ampla aplicação do escalonamento de tarefas em diversos setores, há uma variedade significativa de características incorporadas a esses problemas. Cada variante têm como objetivo representar de forma precisa as peculiaridades de cada situação, a fim de modelá-las de maneira mais precisa. Os problemas podem considerar que todas as tarefas devem ser processadas em uma única máquina ou processador, ou que tarefas podem ser processadas de forma paralela em máquinas separadas, sendo que estas máquinas podem ser idênticas ou não. Em ambientes com múltiplas máquinas distintas existem os casos em que todas as tarefas devem passar por uma mesma sequência de máquinas para ser concluída (*flow shop*), ou que cada tarefa possui sua própria sequência de máquinas (*job shop*). Existe ainda a situação em que não há uma sequência definida (*open shop*).

Além das diferentes configurações de máquinas, problemas de *scheduling* podem possuir alguns atributos distintos, dentre estes estão: tempos de configuração ou *setup* entre tarefas, isto é, antes ou após o processamento de uma tarefa gasta-se um tempo preparando a máquina, e este tempo pode ser dependente ou independente da sequência de tarefas; datas de liberação, que indicam o momento no qual a tarefa passa a estar disponível para ser executada; datas de entrega, que representam o momento máximo para a conclusão da tarefa.

Neste trabalho é abordado o problema de minimizar o *makespan* (C_{max}) em um escalonamento para uma única máquina, com datas de liberação e tempos de *setup* dependentes da sequência, denotado $1|r_j, s_{ij}|C_{max}$ (GRAHAM et al., 1979). Este problema consiste em sequenciar um conjunto $J = \{1, 2, \ldots, n\}$ contendo n tarefas que devem ser executadas em uma única máquina. Cada tarefa $j \in J$ possui um tempo de processamento $p_j > 0$ e uma data de liberação r_j. Um tempo de *setup* s_{ij} é gasto na transição entre duas

tarefas e antes do processamento da primeira tarefa da sequência. Consideram-se *setups* não-antecipatórios, isto é, o tempo de *setup* s_{ij} só pode ser computado após a liberação da tarefa j.

Uma solução é definida por uma sequência $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ em que cada tarefa $j \in J$ é executada exatamente uma vez. O tempo de conclusão de uma tarefa $j \in J$, denotado por C_j, é dado por $C_j = r_j + s_{0j} + p_j$, se j for a primeira tarefa da sequência, ou seja, $j = \sigma_1$. Caso contrário, tem-se $C_j = \max(C_i, r_j) + s_{ij} + p_j$, sendo i a tarefa que antecede j no escalonamento.

1.2 Justificativa

O planejamento da gestão dos recursos tem um papel crucial no funcionamento de organizações das mais diversas áreas. A competitividade do mercado faz com que empresas estejam sempre em busca de aumentar sua eficiência, e nesse contexto, o sequenciamento de atividades aparece como um importante processo de tomada de decisão com capacidade de otimizar recursos, levando a redução de custos, aumento da produtividade e até da satisfação de clientes e colaboradores. Em suma, a adoção de um escalonamento de tarefas eficiente é um investimento estratégico que pode trazer grandes benefícios e contribuir para o sucesso de uma organização no longo prazo.

O impacto do *scheduling* na produtividade em setores como indústria e serviços faz com que esse tipo de problema seja amplamente estudado, e dentre suas diferentes configurações, a variante que considera máquina única é uma das mais relevantes, uma vez que problemas em ambientes mais complexos frequentemente podem ser decompostos em problemas *single machine*.

Em processos onde diversos serviços devem ser feitos, pode ocorrer o compartilhamento de um recurso entre diferentes tarefas. Por exemplo, uma mesma máquina pode ser usada para execução de um ou mais serviços, sendo que a máquina deve ser utilizada para apenas um serviço por vez. Desta forma, pode ser necessário configurar o ambiente para a execução de cada tarefa, e o tempo gasto na configuração pode variar de acordo com a sequência. Os tempos de *setup* muitas vezes são desprezados para facilitar a resolução do problema, porém sua inclusão pode levar a grande economia em muitos casos. (ALLAHVERDI et al., 2008).

Outra situação presente em processos industriais que demandam diferentes serviços é a indisponibilidade de execução de determinada tarefa em certo momento. Por exemplo, a realização de um serviço pode ser limitada a um profissional específico, assim, o serviço só pode ser executado a partir do momento que este funcionário se faz presente. Neste caso, o problema de sequenciamento deve levar em conta as datas de liberação, que representam o horário mínimo de início de cada tarefa.

O $1|r_j, s_{ij}|C_{max}$ é um problema *single machine* que inclui tempos de *setup* e datas de liberação, e é conhecido ser fortemente $\mathcal{NP}$-difícil, devido aos *setups* dependentes da sequência (PINEDO, 2012). Devido a dificuldade computacional, não são conhecidos algoritmos capazes de resolver sistematicamente o problema de forma ótima em um horizonte de tempo razoável, e a literatura do problema é mais voltada para o desenvolvimento de métodos heurísticos.

Este trabalho propõe um método heurístico híbrido para o $1|r_j, s_{ij}|C_{max}$, combinando elementos de *iterated local search* (ILS) e *beam search* (BS), com o objetivo de obter resultados mais eficientes que os observados na literatura, com um tempo computacional aceitável. Para isso, tal método inclui uma estratégia eficiente de avaliação de movimentos na busca local, baseada nos trabalhos de Kindervater e Savelsbergh (1997) e Vidal et al. (2013), que apresentaram ganhos relevantes para diferentes problemas.

1.3 Objetivos

1.3.1 Objetivo Geral

Desenvolver um método heurístico eficiente para o problema $1|r_j, s_{ij}|C_{max}$, visando encontrar melhores soluções para instâncias encontradas na literatura.

1.3.2 Objetivos específicos

- Incorporar métodos de avaliação eficiente de movimentos ao procedimento de busca local da heurística.
- Testar o algoritmo desenvolvido em instâncias da literatura.
- Comparar os resultados do algoritmo proposto com os obtidos pelos melhores métodos encontrados na literatura.

1.4 Estrutura do Trabalho

O restante do trabalho é organizado da seguinte maneira: o capítulo 2 contém uma revisão da literatura de problemas de *scheduling* com características em comum as abordadas neste trabalho. No capítulo 3 é descrito o algoritmo ILS-BS proposto para resolução do problema estudado. O capítulo 4 apresenta os resultados obtidos pelo algoritmo proposto, além de comparações com os métodos da literatura. Por fim, o capítulo 5 contém as considerações finais sobre o trabalho desenvolvido.

2 Revisão da Literatura

Neste capítulo são descritas algumas contribuições relacionadas a classe de problemas de escalonamento de tarefas que consideram *setups* e datas de liberação.

Até onde se tem conhecimento, alguns poucos trabalhos abordam especificamente o $1|r_j, s_{ij}|C_{max}$. Montoya-Torres, Soto-Ferrari e González-Solano (2010) e Montoya-Torres, González-Solano e Soto-Ferrari (2012) propuseram um algoritmo de inserção aleatória de tarefas para uma versão do problema que não considera o tempo de *setup* inicial. Este atributo foi considerado no trabalho de Velez-Gallego, Maya e Montoya-Torres (2016), que apresentou uma formulação de programação linear inteira mista (MILP) e um algoritmo *beam search*. Enquanto o modelo MILP foi capaz de resolver instâncias de 25 e 50 tarefas em até uma hora, o BS atingiu soluções promissoras para instâncias de até 150 tarefas em um curto tempo de execução.

Fan et al. (2019) apresentaram uma heurística de inserção variável de blocos (VBIH), que emprega um procedimento construtivo baseado no algoritmo NEH. Os autores também adaptaram o algoritmo *iterated greedy* (IG) de Ruiz e Stützle (2007) para o problema a fim de comparação com o VBIH proposto, que foi testado apenas em instâncias com até 75 tarefas, apresentando soluções de qualidade superior ao BS e IG em instâncias de 50 e 75 tarefas, mas sendo superado por ambos em instâncias de 25 tarefas.

Fernandez-Viagas e Costa (2021) propuseram uma heurística baseada em *beam search* e uma meta-heurística baseada em população para o $1|r_j, s_{ij}|C_{max}$ considerando ambos os casos em que os *setups* são antecipatórios e não-antecipatórios. Os algoritmos desenvolvidos foram comparados a reimplementações de diversas heurísticas da literatura propostas para problemas de *scheduling* similares.

Outros problemas em configuração de única máquina tem sido estudados na literatura. Bianco et al. (1988) propuseram uma formulação MILP e um algoritmo branch-and-bound (B&B) para o problema de minimizar o *makespan* com datas de liberação e tempos de processamento dependentes da sequência, o que é equivalente ao $1|r_j, s_{ij}|C_{max}$ com tempos de processamento nulos. Ovacikt e Uzsoy (1994) abordaram o problema de minimizar o *lateness* máximo ($1|r_j, s_{ij}|L_{max}$) por meio de uma heurística de decomposição, denotada *Rolling Horizons Procedure* (RHP) que é testada em um conjunto de instâncias proposto pelos autores. Shin, Kim e Kim (2002) trataram o mesmo problema com um algoritmo baseado em busca tabu, capaz de encontrar soluções de alta qualidade em menor tempo que o RHP.

Chang, Chou e Lee (2004) propuseram uma formulação matemática e uma abordagem heurística para o problema de minimizar o *tardiness* ponderado ($1|r_j, s_{ij}|\sum w_j T_j$), o modelo conseguiu resolver instâncias de até 11 tarefas enquanto a heurística foi capaz de

prover soluções aceitáveis para instâncias de até 500 tarefas em um tempo de execução razoável. Chou, Wang e Chang (2008) minimiza o tempo de conclusão ponderado com um modelo de programação por restrições, uma formulação MILP, um B&B capaz de resolver instâncias de até 40 tarefas, e duas heurísticas foram propostas para lidar com instâncias maiores, indo de 50 a 500 tarefas. O problema de minimizar o *tardiness* total foi abordado por Nogueira et al. (2022) com um algoritmo que combina busca em vizinhança variável (VNS) e relaxação Lagrangiana, em que os multiplicadores Lagrangianos influenciam nos procedimentos de construção e perturbação do VNS.

Problemas de escalonamento com datas de liberação e tempos de *setup* dependentes da sequência em ambientes com máquinas paralelas também tem sido objeto de estudos na literatura. Ying e Cheng (2010) aplicou uma heurística IG para o problema de minimizar o atraso máximo em máquinas paralelas idênticas com datas de liberação e tempos de *setup* dependentes da sequência ($P|r_j, s_{jk}|L_{max}$). Lin et al. (2011) propuseram uma versão aprimorada do IG que incorpora *simulated annealing*. Driessel e Mönch (2011) abordaram o problema de minimizar o *tardiness* ponderado com restrições de precedência $P|r_j, s_{jk}|L_{max}$, apresentando diversas variantes de VNS.

No que se refere a problemas em máquinas uniformes, em que cada máquina possui uma velocidade de processamento distinta, Balakrishnan, Kanet e Sridharan (1999) desenvolveram um modelo de programação inteira mista para o problema de minimizar a soma do *earliness* e *tardiness* penalizados ($Q|r_j, s_{ijk}|e_jE_j + t_jT_j$). Bilge et al. (2004) apresentaram uma busca tabu para o problema de minimizar o *tardiness* total ($Q|r_j, s_{ijk}|T_j$) e Kim, Choi e Lee (2007) propuseram diferentes heurísticas baseadas em busca tabu e *simmulated annealing* para o mesmo problema.

Logendran, McDonell e Smucker (2007) consideraram o problema de minimizar o *tardiness* ponderado em máquinas paralelas não relacionadas ($R|r_j, s_{ijk}|w_jT_j$) com um algoritmo baseado em busca tabu. O mesmo problema foi abordado por Lin e Hsieh (2014), que propuseram um modelo de programação inteira mista e uma meta-heurística baseada em busca local, denotada *iterated hybrid metaheuristic* (IHM). O modelo MILP conseguiu obter soluções ótimas para instâncias com até 3 máquinas e 12 tarefas, enquanto a meta-heurística apresentou soluções de melhor qualidade quando comparada a busca tabu. Diana, de Souza e Filho (2018) abordou o problema com uma meta-heurística ILS-VND que apresentou performance superior ao IHM na maioria das instâncias.

3 Metodologia

Nesse capítulo são apresentados os conceitos e métodos propostos para resolução do problema $1|r_j, s_{ij}|C_{max}$. A seção 3.1 apresenta a representação de uma solução. Na seção 3.2 é detalhado o algoritmo ILS, que serve como base para as heurísticas desenvolvidas. A seção 3.3.1 apresenta um procedimento construtivo baseado em GRASP (*Greedy Randomized Adaptative Search Procedure*), enquanto um segundo método construtivo baseado em *beam search* (BS) é mostrado na seção 3.3.2. O procedimento de busca local e as estruturas de vizinhança são apresentados na seção 3.4, e o procedimento de perturbação é detalhado na seção 3.5.

3.1 Representação da solução

Uma solução para uma problema de escalonamento de tarefas em máquina única pode ser representado por um vetor s de n posições, onde cada elemento s_i, indica que a tarefa j ficou na posição i no escalonamento da produção. Como o problema admite a presença de tempo ocioso, qualquer sequência de tarefas em que todas as tarefas são processadas uma única vez é considerada viável. A Figura 1 mostra a solução ótima obtida para a instância apresentada na Tabela 1, onde pode-se observar a inclusão do tempo ocioso no início da sequência e entre as tarefas 4 e 3.

j	r_j	p_j
1	1	1
2	3	2
3	10	1
4	2	2
5	4	2

s_{ij}	**1**	**2**	**3**	**4**	**5**
0	2	5	1	5	7
1	-	9	8	1	5
2	4	-	6	5	3
3	1	3	-	8	2
4	10	2	3	-	7
5	8	1	5	7	-

Tabela 1 – Exemplo de instância para o $1|r_j, s_{ij}|C_{max}$

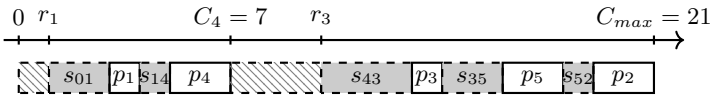

Figura 1 – Solução gerada pela sequência $s = \{1, 4, 3, 5, 2\}$

3.2 *Iterated Local Search*

O ILS opera em uma única solução, combinando um estágio de busca local com um estágio de perturbação. A busca local é uma fase de intensificação em que se busca uma solução ótima local, enquanto a perturbação visa diversificar as soluções evitando que o algoritmo fique preso nesses ótimos locais, aproximando a solução de um valor global.

O pseudocódigo do ILS é apresentado no Algoritmo 1. O procedimento recebe os parâmetros I_R e I_{ILS}. Em I_R iterações uma solução inicial é gerada pelo método construtivo (linha 4), e o contador *IterILS* é iniciado com zero. A solução inicial é então iterativamente melhorada pela busca local, que explora um conjunto de vizinhanças descrito na seção 3.4. Se a busca local e a perturbação falham em melhorar a solução corrente, o contador é incrementado (linha 13), caso contrário o contador é novamente zerado (linha 11). A combinação de busca local e perturbação é interrompida quando *IterILS* atinge o número de iterações limite definido por I_{ILS}, e a solução obtida é comparada a melhor solução obtida em iterações anteriores. Por fim o ILS retorna a melhor solução encontrada dentre todas as iterações.

Algoritmo 1 ILS

```
1:  procedimento ILS(I_R, I_ILS)
2:      f* ← ∞
3:      para i ← 1, ..., I_R faça
4:          s ← Construção()
5:          s' ← s
6:          IterILS ← 0
7:          enquanto IterILS < I_ILS faça
8:              s ← Busca Local(s)
9:              se f(s) < f(s') então
10:                 s' ← s
11:                 IterILS ← 0
12:             s ← Perturbação(s')
13:             IterILS ← IterILS + 1
14:         se f(s') < f(s*) então
15:             s* ← s'
16:             f* ← f(s')
17:     retorne s*
```

3.3 Construção

O procedimento construtivo tem como função gerar uma solução inicial no começo de cada iteração do algoritmo. Neste trabalho foram implementados dois diferentes métodos, um baseado no GRASP e o outro baseado em *beam search*.

3.3.1 Construção baseada em GRASP

O Algoritmo 2 contém o pseudocódigo do método construtivo baseado em GRASP. Uma lista de candidatos CL é inicializada com todas as tarefas disponíveis (linha 3), esta lista é então ordenada de forma crescente de acordo com *release date* de cada tarefa (linha 7), e a partir dessa lista uma lista restrita é criada contendo apenas as $\lfloor(1+\alpha)|CL|\rfloor$ tarefas com menor *release date* (linha 10). Uma tarefa desta lista é selecionada aleatoriamente, inserida na sequência s e removida da lista de candidatos (linhas 11 – 13). Este processo se repete até que a lista de candidatos esteja vazia, sendo que após a inserção da primeira tarefa, a ordenação da lista de candidatos passa a ser feita de acordo com o tempo ocioso gerado pela inserção de cada tarefa ao fim da sequência.

Algoritmo 2 Construção baseada em GRASP

```
1:  procedimento CONSTRUÇÃO
2:      s ← {};
3:      CL ← Lista de todas as tarefas;
4:      α ← Valor aleatório ∈ {0, ..., 1}
5:      enquanto CL ≠ ∅ faça
6:          se s = ∅ então
7:              Ordenar CL de forma crescente de acordo com o release date;
8:          senão
9:              Ordenar CL de forma crescente de acordo com o tempo ocioso gerado;
10:         Criar uma lista RCL com apenas os ⌊(1 + α)|CL|⌋ primeiros candidatos;
11:         Escolher uma tarefa j ∈ RCL aleatoriamente;
12:         s ← s ∪ {j};
13:         CL ← CL − {j};
14:     retorne s
```

3.3.2 Construção baseada em *Beam Search*

Este procedimento construtivo é uma adaptação do algoritmo *beam search* apresentado no trabalho de Velez-Gallego, Maya e Montoya-Torres (2016). O método realiza uma enumeração limitada de soluções parciais em uma árvore, cuja raiz contém uma sequência vazia e novos nós são criados ao adicionar tarefas ao final da sequência. Em cada nível da árvore é adicionada uma tarefa em cada nó, desta forma, no nível n obtém-se soluções completas. A limitação da enumeração se dá de formas: um nó só pode gerar no máximo w filhos; e cada nível da árvore pode conter no máximo N nós. A versão construtiva do *beam search* modifica o procedimento original ao adicionar um fator de aleatoriedade na seleção dos nós sobreviventes em um nível, que é definido pelo parâmetro α. Com isso o método é capaz de gerar uma solução diversificada em cada iteração do ILS.

O pseudocódigo do *beam search* construtivo é apresentado no Algoritmo 3. Na linha 1 o nó raiz é inicializado com uma sequência vazia e adicionado a lista de nós do

nível Π_0 na linha 2. Do nível 0 ao nível $n-1$, para todo nó na lista Π_k são gerados w nós ao se fixar uma tarefa j da lista de tarefas não incluídas $U(\pi)$. Caso o número de tarefas disponível for maior que w, são eliminadas as tarefas que geram o maior tempo ocioso até restar apenas w nós. Cada uma dessas tarefas é adicionada ao final da sequência, o que é representado pela função $\beta(j, i\}$, e os nós resultantes são adicionados a lista de nós Π_{k+1} (linhas 6 – 12). Se essa lista conter mais que N nós, cada nó é avaliado e a lista é ordenada de forma não-decrescente de acordo o *lower bound* (LB) do *makespan*, calculado pela equação 3.1, em que $Q(\pi)$ é a união de $U(\pi)$ com a última tarefa da sequência π (linha 14). Uma lista de candidatos é criada com os $\lfloor(1+\alpha)N\rfloor$ melhores nós de Π_{k+1} (linha 15), e nas linhas 16 – 20 a lista é redefinida para comportar apenas N nós selecionados aleatoriamente da lista de candidatos. Nas linhas 22 – 25 são gerados todos os nós do nível n, que representam soluções completas. O método retorna a solução de menor *makespan*.

A Figura 2 exemplifica a execução do *beam search* construtivo para a instância da Tabela 1. Este exemplo é complementado pela Tabela 2, em que os nós eliminados pelo critério do tempo ocioso são destacados em cinza e os nós eliminados pelo critério do LB do *makespan* estão tachados.

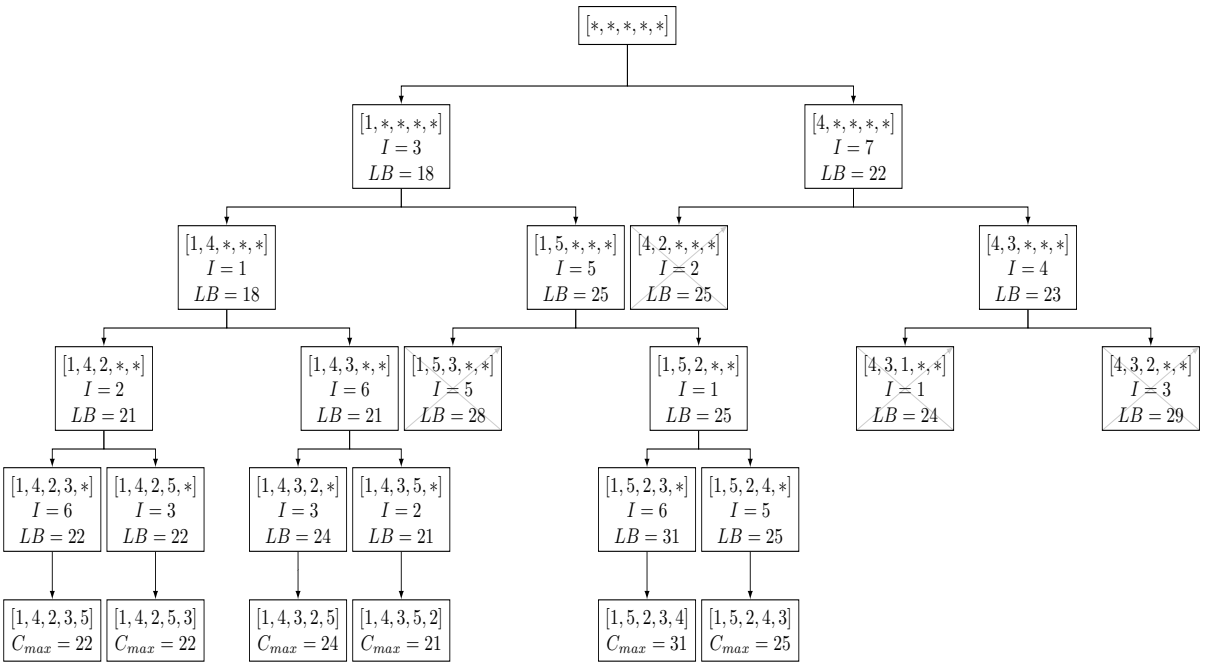

Figura 2 – Árvore do *beam search* para a instância da Tabela 1, com $w = 2$, $N = 3$ e $\alpha = 0.5$

$$LB(\pi) = \max\{C(\pi)\ ,\ \min_{j\in U(\pi)}\{r_j\}\} + \sum_{k\in U(\pi)} \min_{t\in Q(\pi)}\{s_{tk}\} + \sum_{k\in U(\pi)} p_k \qquad (3.1)$$

Tabela 2 – Nós do beam search para cada nível

Nível	**Nó**	**I**	**LB** (3.1)	$\boldsymbol{C(\pi)}$
1	$\pi_1 = [1,*,*,*,*]$	$r_1 + s_{01} = 3$	18	$I + p_1 = 4$
	$\pi_2 = [2,*,*,*,*]$	$r_2 + s_{02} = 8$	23	$I + p_2 = 10$
	$\pi_3 = [3,*,*,*,*]$	$r_3 + s_{03} = 11$	24	$I + p_3 = 12$
	$\pi_4 = [4,*,*,*,*]$	$r_4 + s_{04} = 7$	22	$I + p_4 = 9$
	$\pi_5 = [5,*,*,*,*]$	$r_5 + s_{05} = 11$	25	$I + p_5 = 13$
2	$\pi_6 = [1,2,*,*,*]$	$\max(r_2 - C(\pi_1), 0) + s_{12} = 9$	30	$C(\pi_1) + I + p_2 = 15$
	$\pi_7 = [1,3,*,*,*]$	$\max(r_3 - C(\pi_1), 0) + s_{13} = 14$	33	$C(\pi_1) + I + p_3 = 19$
	$\pi_8 = [1,4,*,*,*]$	$\max(r_4 - C(\pi_1), 0) + s_{14} = 1$	18	$C(\pi_1) + I + p_4 = 7$
	$\pi_9 = [1,5,*,*,*]$	$\max(r_5 - C(\pi_1), 0) + s_{15} = 5$	25	$C(\pi_1) + I + p_5 = 11$
	$\pi_{10} = [4,1,*,*,*]$	$\max(r_1 - C(\pi_4), 0) + s_{41} = 10$	33	$C(\pi_4) + I + p_1 = 20$
	~~$\pi_{11} = [4,2,*,*,*]$~~	~~$\max(r_2 - C(\pi_4), 0) + s_{42} = 2$~~	~~25~~	~~$C(\pi_4) + I + p_2 = 13$~~
	$\pi_{12} = [4,3,*,*,*]$	$\max(r_3 - C(\pi_4), 0) + s_{43} = 4$	23	$C(\pi_4) + I + p_3 = 14$
	$\pi_{13} = [4,5,*,*,*]$	$\max(r_5 - C(\pi_4), 0) + s_{45} = 7$	29	$C(\pi_4) + I + p_5 = 18$
3	~~$\pi_{14} = [4,3,1,*,*]$~~	~~$\max(r_1 - C(\pi_{12}), 0) + s_{31} = 1$~~	~~24~~	~~$C(\pi_{12}) + I + p_1 = 16$~~
	~~$\pi_{15} = [4,3,2,*,*]$~~	~~$\max(r_2 - C(\pi_{12}), 0) + s_{32} = 3$~~	~~29~~	~~$C(\pi_{12}) + I + p_2 = 19$~~
	$\pi_{16} = [4,3,5,*,*]$	$\max(r_5 - C(\pi_{12}), 0) + s_{35} = 2$	26	$C(\pi_{12}) + I + p_5 = 18$
	$\pi_{17} = [1,4,2,*,*]$	$\max(r_2 - C(\pi_8), 0) + s_{42} = 2$	21	$C(\pi_8) + I + p_2 = 11$
	$\pi_{18} = [1,4,3,*,*]$	$\max(r_3 - C(\pi_8), 0) + s_{43} = 6$	21	$C(\pi_8) + I + p_3 = 14$
	$\pi_{19} = [1,4,5,*,*]$	$\max(r_5 - C(\pi_8), 0) + s_{45} = 7$	25	$C(\pi_8) + I + p_5 = 16$
	$\pi_{20} = [1,5,2,*,*]$	$\max(r_2 - C(\pi_9), 0) + s_{52} = 1$	25	$C(\pi_9) + I + p_2 = 14$
	~~$\pi_{21} = [1,5,3,*,*]$~~	~~$\max(r_3 - C(\pi_9), 0) + s_{53} = 5$~~	~~28~~	~~$C(\pi_9) + I + p_3 = 17$~~
	$\pi_{22} = [1,5,4,*,*]$	$\max(r_4 - C(\pi_9), 0) + s_{54} = 7$	28	$C(\pi_9) + I + p_4 = 20$
4	$\pi_{23} = [1,5,2,3,*]$	$\max(r_2 - C(\pi_{20}), 0) + s_{23} = 6$	31	$C(\pi_{20}) + I + p_3 = 21$
	$\pi_{24} = [1,5,2,4,*]$	$\max(r_5 - C(\pi_{20}), 0) + s_{24} = 5$	25	$C(\pi_{20}) + I + p_4 = 21$
	$\pi_{25} = [1,4,3,2,*]$	$\max(r_2 - C(\pi_{18}), 0) + s_{32} = 3$	24	$C(\pi_{18}) + I + p_2 = 19$
	$\pi_{26} = [1,4,3,5,*]$	$\max(r_5 - C(\pi_{18}), 0) + s_{35} = 2$	**21**	$C(\pi_{18}) + I + p_5 = 18$
	$\pi_{27} = [1,4,2,3,*]$	$\max(r_3 - C(\pi_{17}), 0) + s_{23} = 6$	22	$C(\pi_{17}) + I + p_3 = 18$
	$\pi_{28} = [1,4,2,5,*]$	$\max(r_5 - C(\pi_{17}), 0) + s_{25} = 3$	22	$C(\pi_{17}) + I + p_5 = 16$
5	$\pi_{29} = [1,5,2,3,4]$	$\max(r_5 - C(\pi_{23}), 0) + s_{34} = 8$	–	$C(\pi_{23}) + I + p_4 = 31$
	$\pi_{30} = [1,5,2,4,3]$	$\max(r_2 - C(\pi_{24}), 0) + s_{43} = 3$	–	$C(\pi_{24}) + I + p_3 = 25$
	$\pi_{31} = [1,4,3,2,5]$	$\max(r_5 - C(\pi_{25}), 0) + s_{25} = 5$	–	$C(\pi_{25}) + I + p_5 = 24$
	$\boldsymbol{\pi_{32} = [1,4,3,5,2]}$	$\boldsymbol{\max(r_2 - C(\pi_{26}), 0) + s_{52} = 3}$	–	$\boldsymbol{C(\pi_{26}) + I + p_2 = 21}$
	$\pi_{33} = [1,4,2,3,5]$	$\max(r_5 - C(\pi_{27}), 0) + s_{35} = 4$	–	$C(\pi_{27}) + I + p_5 = 22$
	$\pi_{34} = [1,4,2,5,3]$	$\max(r_3 - C(\pi_{28}), 0) + s_{53} = 6$	–	$C(\pi_{28}) + I + p_3 = 22$

3.4 Busca Local

O método de busca local é baseado na descida variável em vizinhança aleatória (RVND) (MLADENOVIC; HANSEN, 1997; SUBRAMANIAN et al., 2010), apresentado no Algoritmo 4. Inicialmente uma lista de estruturas de vizinhança, denominada *NL* é inicializada. Em seguida, uma vizinhança dessa lista é selecionada aleatoriamente, e todas as movimentos possíveis são investigadas e avaliados. Se nenhuma solução melhor do que a atual solução de referência for encontrada, a estrutura de vizinhança é removida da lista *NL*. No entanto, se a exploração da vizinhança resultar em uma solução melhor que a referência, a lista *NL* é restaurada, indicando que todas as estruturas de vizinhança estão novamente disponíveis para exploração.

Algoritmo 3 Beam Search Construtivo

procedimento BEAM SEARCH CONSTRUTIVO(α)
  $\pi_0 \leftarrow \{*, *, ..., *\}$
  $\Pi_0 \leftarrow \{\pi_0\}$
  para $k \leftarrow 0, ..., n-1$ **faça**
    $\Pi_{k+1} \leftarrow \{\emptyset\}$
    para todo $i \in \Pi_k$ **faça**
      $\Theta \leftarrow U(i)$
      enquanto $|\Theta| > w$ **faça**
        $j^* \leftarrow \max\{I(j,i) | j \in \Theta\}$
        $\Theta \leftarrow \Theta \setminus j^*$
      para todo $j \in \Theta$ **faça**
        $\Pi_{k+1} \leftarrow \Pi_{k+1} \cup \beta(j,i)\}$
    se $|\Pi_{k+1}| > N$ **então**
      Ordenar Π_{k+1} de acordo com $LB(\pi)$;
      Criar uma lista de candidatos CL com os $\lfloor (1+\alpha)N \rfloor$ primeiros nós de Π_{k+1};
      $\Pi_{k+1} \leftarrow \{\emptyset\}$
      enquanto $|\Pi_{k+1}| < N$ **faça**
        Escolher um nó $i^* \in CL$ aleatoriamente;
        $\Pi_{k+1} \leftarrow \Pi_{k+1} \cup \{i^*\}$;
        $CL \leftarrow CL \setminus \{i^*\}$;
  $\Pi_k \leftarrow \{\emptyset\}$
  para todo $i \in \Pi_k$ **faça**
    para todo $j \in U(i)$ **faça**
      $\Pi_k \leftarrow \Pi_k \cup \beta(j,i)\}$
  $s^* \leftarrow \min\{C(i) | i \in \Pi_k\}$
  retorne s^*

Algoritmo 4 RVND

procedimento RVND(s)
  Inicializa lista de vizinhanças NL
  Inicializa estrutura de dados de subsequências
  enquanto $NL \neq \emptyset$ **faça**
    Seleciona aleatoriamente uma vizinhança $v \in NL$
    Encontra o melhor vizinho $s' \in v$
    se $f(s') < f(s)$ **então**
      $s \leftarrow s'$
      $f(s) \leftarrow f(s')$
      Atualiza a estrutura de subsequências
      Restaura a lista NL
    senão
      Remove v de NL

3.4.1 Estruturas de Vizinhança

Dois tipos de estrutura de vizinhança foram utilizados no RVND:

- *Swap*: permuta duas tarefas da sequência. A Figura 3 mostra um exemplo usando dados da Tabela 1 em que as tarefas 5 e 4 da sequência $(1, 5, 3, 4, 2)$ são trocadas, resultando em uma nova sequência $(1, 4, 3, 5, 2)$. Pode-se observar que houve melhora na solução mesmo com a introdução de tempo ocioso para satisfazer a data de liberação da tarefa 3.
- *l-block insertion*: Um bloco de l tarefas adjacentes é removido de sua posição atual e reinserido em outra posição da sequência. A Figura 4 representa um exemplo utilizando também os dados fornecidos na Tabela 1 e considerando $l = 2$, na qual as duas últimas tarefas consecutivas da mesma sequência inicial $(1, 5, 3, 4, 2)$ são removidas e reinseridas imediatamente após a tarefa 1, resultando em uma sequência melhorada dada por $(1, 4, 2, 5, 3)$. Nesse caso, não foi necessário incluir tempo ocioso para garantir a viabilidade. Uma vizinhança distinta é associada a cada valor diferente de l.

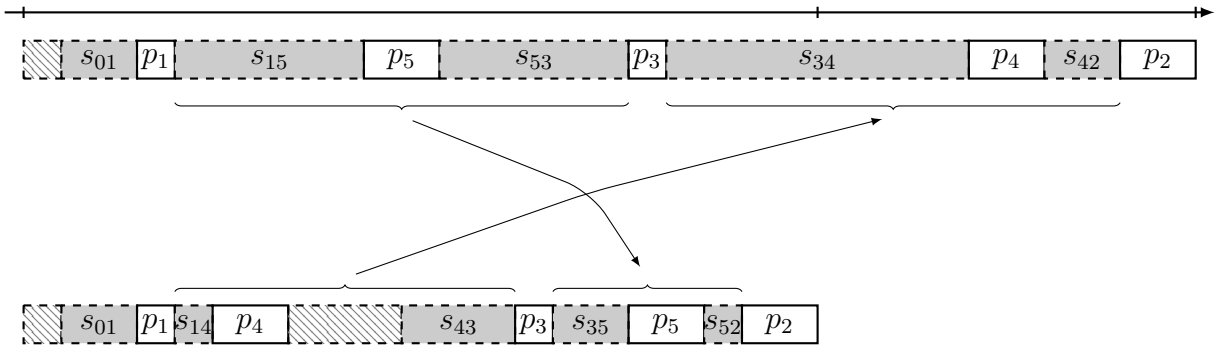

Figura 3 – *Swap* entre as tarefas 5 e 4.

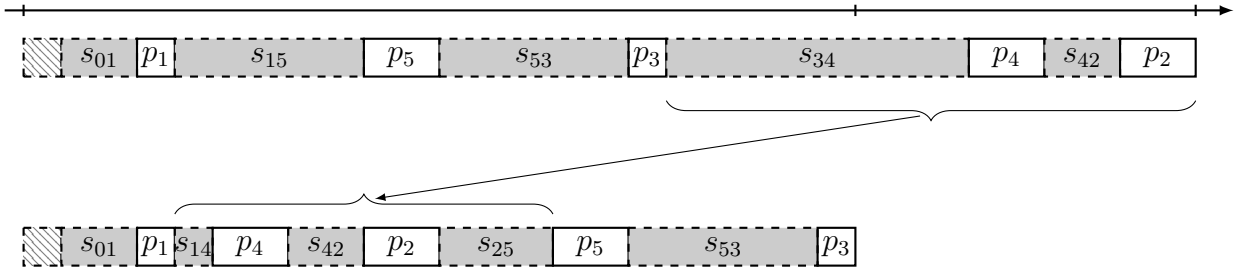

Figura 4 – l-block insertion: $l = 2$, bloco $(4, 2)$ inserido na posição $i = 1$.

Como mencionado anteriormente, é necessário ajustar a solução para satisfazer as restrições de disponibilidade de tarefas, incluindo um tempo ocioso caso a data de liberação

de uma tarefa tenha sido violada. Portanto, avaliar um movimento pode ser demorado se realizado de forma direta, ou seja, verificando se a data de liberação de todas as tarefas afetadas por um movimento é satisfeita e introduzindo tempo ocioso, se necessário. Isso pode comprometer rapidamente o desempenho do método em termos de tempo de CPU à medida que o tamanho da instância aumenta. A próxima seção descreve uma maneira eficiente de realizar a avaliação do movimento para superar esse problema.

3.4.2 Avaliação de Movimentos

O processo de avaliação do movimento consiste em calcular o *makespan* da sequência resultante produzida por um movimento na tentativa de encontrar uma melhoria na solução de referência. Isso poderia ser feito simplesmente iterando pela sequência e calculando o tempo de conclusão de cada tarefa, resultando em uma complexidade de $\mathcal{O}(n)$. Nesse caso, como o tamanho de todas as estruturas de vizinhança consideradas é $\mathcal{O}(n^2)$, a complexidade geral de explorar e avaliar cada vizinhança é $\mathcal{O}(n^3)$. Em vista disso, para reduzir a complexidade computacional, foi implementado um esquema eficiente de avaliação de movimentos baseado nos trabalhos de Kindervater e Savelsbergh (1997) e Vidal et al. (2013) que permite que o movimento seja avaliado em operações de complexidade $\mathcal{O}(1)$ amortizada, implicando em uma redução da complexidade de exploração da vizinhança para $\mathcal{O}(n^2)$.

Nesse esquema, a avaliação de um movimento é feita concatenando subsequências de tarefas. Para cada subsequência $\sigma = (\sigma_1, \sigma_2, ..., \sigma_{|\sigma|})$ em uma solução dada, os seguintes atributos são armazenados em uma estrutura de dados auxiliar:

- $F(\sigma)$ = primeira tarefa;
- $L(\sigma)$ = última tarefa;
- $E(\sigma)$ = menor tempo para iniciar o processamento sem tempo ocioso;
- $D(\sigma)$ = soma dos tempos de processamento e tempos de configuração.

Seja $\sigma = (j)$ uma subsequência unitária composta por uma tarefa $j \in J$. Nesse caso, $F(\sigma) = j$, $L(\sigma) = j$, $E(\sigma) = r_j$ e $D(\sigma) = p_j$. Qualquer subsequência σ, $|\sigma| \geq 1$ pode ser derivada da concatenação de outras duas subsequências σ^1 e σ^2. Esse processo é descrito no Algoritmo 5. O *makespan* de qualquer subsequência pode ser definido por $C(\sigma) = E(\sigma) + D(\sigma)$. No entanto, o *makespan* de uma solução s representada pela subsequência $\sigma = (\sigma_1, \sigma_2, ..., \sigma_{|\sigma|})$ não corresponde necessariamente a $C(\sigma)$, uma vez que o tempo de configuração inicial não é considerado. O *makespan* real de s é igual a $C(\sigma' \oplus \sigma)$, onde σ' é uma subsequência unitária contendo uma tarefa fictícia 0 com $r_0 = 0$ e $p_0 = 0$.

Ao realizar a operação $\sigma^1 \oplus \sigma^2$, existem diferentes casos em relação aos tempo ocioso gerado pela concatenação. No primeiro caso, como mostrado na Figura 5, o tempo

mínimo necessário para concluir a primeira subsequência $C(\sigma^1)$ é maior ou igual ao tempo mais cedo $E(\sigma^2)$ para iniciar σ^2, sem produzir tempo ocioso. Nos demais cenários, a subsequência σ^1 é concluída antes de $E(\sigma^2)$. No cenário mostrado na Figura 6, a máquina fica ociosa entre a conclusão do último trabalho de σ^1 (i) e o início de σ^2. No entanto, o tempo ocioso é reduzido se o *setup* do primeiro trabalho de σ^2 (j) puder ser processado antes do tempo mínimo para iniciar a sequência, como mostrado na Figura 7.

Algoritmo 5 Concatenação das subsequências $\sigma^1 \oplus \sigma^2$

1: **procedimento** CONCATENAÇÃO(σ^1, σ^2)
2: $F(\sigma^1 \oplus \sigma^2) \leftarrow F(\sigma^1)$;
3: $L(\sigma^1 \oplus \sigma^2) \leftarrow L(\sigma^2)$;
4: $I \leftarrow 0$;
5: $i, j \leftarrow L(\sigma^1), F(\sigma^2)$;
6: **se** $C(\sigma^1) < E(\sigma^2)$ **então**
7: **se** $C(\sigma^1) < r_j$ **então**
8: $I \leftarrow r_j - C(\sigma^1)$;
9: **se** $C(\sigma^1) + I + s_{ij} < E(\sigma^2)$ **então**
10: $I \leftarrow E(\sigma^2) - (C(\sigma^1) + s_{ij})$;
11: $E(\sigma^1 \oplus \sigma^2) \leftarrow E(\sigma^1) + I$;
12: $D(\sigma^1 \oplus \sigma^2) \leftarrow D(\sigma^1) + D(\sigma^2) + s_{ij}$;
13: **retorne** $\sigma^1 \oplus \sigma^2$

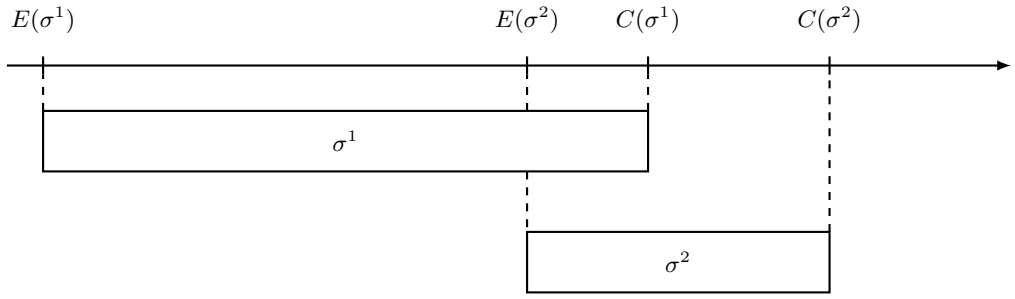

Figura 5 – Concatenação com $C(\sigma^1) \geq E(\sigma^2)$. O tempo ocioso gerado deriva apenas de σ^1.

3.5 Perturbação

Se o procedimento de busca local falha em melhorar a solução de referência, uma perturbação é aplicada a melhor solução da iteração. Esta perturbação consiste em permutar duas subsequências disjuntas selecionadas aleatoriamente. Esse procedimento é similar ao *double bridge* desenvolvido por Martin, Otto e Felten (1991) para o problema do caixeiro viajante.

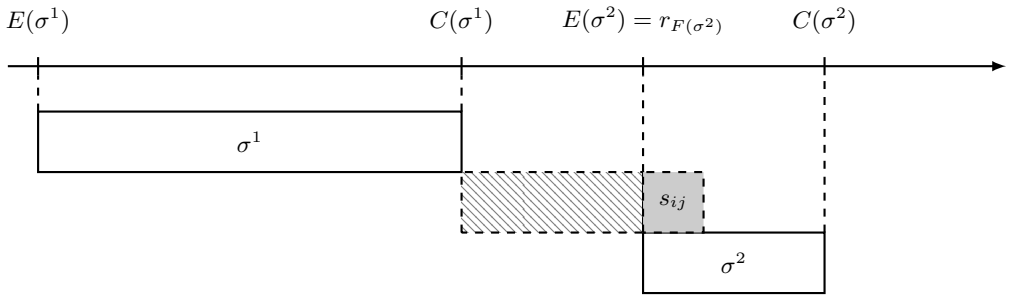

Figura 6 – Um caso de concatenação em que $C(\sigma^1) \leq E(\sigma^2)$: o *setup* entre a última tarefa de σ^1 e a primeira tarefa de σ^2 não pode ser processado antes de $E(\sigma^2)$.

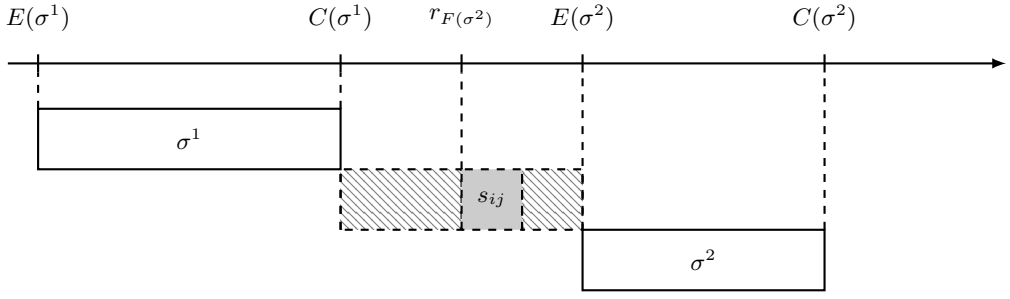

Figura 7 – Um caso diferente de concatenação em que $C(\sigma^1) \leq E(\sigma^2)$: o *setup* entre $L(\sigma^1)$ e $F(\sigma^2)$ pode ser computado antes de $E(\sigma^2)$, reduzindo o tempo ocioso. Note que um dos períodos ociosos (antes ou depois do *setup*) pode ser nulo.

4 Resultados Computacionais

Experimentos foram realizados em duas versões do ILS: uma utilizando a construção baseada em GRASP (GILS), e outra que utiliza *beam search* (ILS-BS). Ambos foram implementados em C++ (g++ 5.4.0) e todos os experimentos foram realizados em uma máquina com processador Intel i7-2600 com 3.40 GHz e 16 GB RAM e sistema operacional Linux Ubuntu 16.04 64 bits.

4.1 Instâncias

As instâncias utilizadas nos experimentos foram propostas por Ovacikt e Uzsoy (1994). Os autores utilizaram uma distribuição uniforme no intervalo $[1, 200]$ para gerar os tempos de processamento e de *setup*. As datas de liberação são definidas por uma distribuição uniforme com um intervalo entre 0 e um limite superior que é o produto entre um fator de dispersão R e o *makespan* esperado das tarefas. Desta forma é possível dividir o conjunto de instâncias pela quantidade de tarefas $n \in \{25, 50, 75, 100, 125, 150\}$ e pelo fator de dispersão de datas de liberação $R \in \{0.2, 0.4, 1.0, 1.4, 1.8\}$.

4.2 Calibração dos Parâmetros

Os parâmetros I_R e I_{ILS} foram calibrados por meio de um procedimento similar ao descrito em Silva, Teixeira e Subramanian (2021), no qual os autores propuseram um algoritmo ILS para o problema de minimizar o *makespan* em um ambiente com máquinas paralelas idênticas, servidor comum e tempos de *setup* dependentes da sequência. No procedimento realizado, um conjunto de 90 instâncias de teste foi definido, abrangendo todas as combinações possíveis de n e R. Inicialmente, I_R foi fixado em 1 e diferentes valores de I_{ILS} (50, 100 e 150) foram testados. Ao analisar os resultados, decidiu-se utilizar $I_{ILS} = 100$ para o GILS e para o ILS-BS, por apresentar um boa relação entre *gap* médio e os tempos de execução. Com este parâmetro definido, foram realizados testes com diferentes valores de I_R (10, 20, 50), onde o melhor valor para o ILS-BS foi $I_R = 10$, e para o GILS utilizou-se $I_R = 20$.

A Figura 8 mostra o tempo médio de execução e os *gaps* para valores de diferentes valores de l, que é o tamanho máximo de blocos da vizinhança *l-block-insertion* descrita na seção 3.4.1. Este experimento foi realizado com 10 execuções do ILS-BS para cada instância do mesmo conjunto de 90 instâncias supracitado. Os *gaps* médios foram computados em relação ao valor objetivo da melhor solução conhecida da literatura ($Melhor_{Lit}$), i.e., $gap = 100 \times ((Média_{ILS\text{-}BS} - Melhor_{Lit})/Melhor_{Lit})$. Para o GILS foi realizado

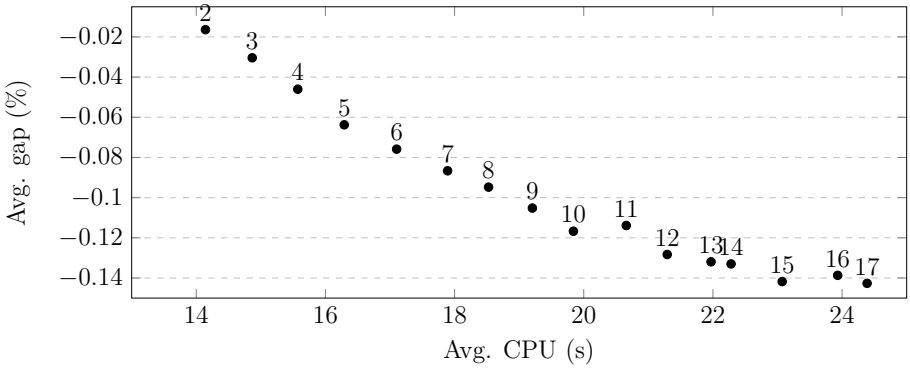

Figura 8 – *Gap* médio e tempo de execução para cada valor de l

experimento semelhante, e em ambos os casos notou-se que adoção de $l = 13$ apresenta melhor equilíbrio entre tempo de execução e qualidade de solução.

A Tabela 3 reporta os *gaps* médios obtidos para diferentes valores do parâmetro α. Estes valores estão no intervalo $[0, 1]$, em que 0 significa que no *beam search* construtivo apenas os melhores nós candidatos serão selecionados para gerar novas ramificações, enquanto 1 indica que a seleção dos nós se dá pela escolha aleatória de nós em uma lista com o dobro do tamanho máximo de nós por nível N. Os valores de N e do número máximo de ramificações por nó w foram definidos com base nos resultados reportados por Velez-Gallego, Maya e Montoya-Torres (2016), onde se verificou que os valores mais promissores foram $w = 5$ e $N = 100$.

α	**0.00**	**0.01**	**0.05**	**0.10**	**0.50**	**1.00**
Gap	−0.08	−0.12	−0.06	0.01	0.22	0.22

Tabela 3 – *Gap* médio para cada valor de α

4.3 Resultados para instâncias com até 75 tarefas

Esta seção apresenta uma comparação entre os resultados obtidos pelo ILS-BS e GILS com o *beam search* (VELEZ-GALLEGO; MAYA; MONTOYA-TORRES, 2016), e os algoritmos VBIH e *iterated greedy* (IG) apresentados em Fan et al. (2019), neste último os autores reportaram resultados apenas para instâncias de até 75 tarefas.

A Tabela 4 mostra os *gaps* médios entre as melhores soluções encontradas pelos algoritmos e a melhor solução conhecida (BKS), incluindo aquelas encontradas pelos métodos propostos neste trabalho. Em média pode-se observar que tanto o ILS-BS quanto o GILS são capazes de encontrar soluções de melhor qualidade, com o GILS conseguindo

superar o ILS-BS nas instâncias em que $n = 25$ e $R \in \{1.0, 1.4\}$, e também em $n = 50$ e $R = 1.4$. Em todas as outras combinações o ILS-BS obteve melhores resultados.

Tabela 4 – *Gaps* para o BKS considerando as melhores soluções encontradas por cada algoritmo: $n \in \{25, 50, 75\}$

	$n = 25$					$n = 50$					$n = 75$				
R	**ILS-BS**	**GILS**	**BS**	**VBIH**	**IG**	**ILS-BS**	**GILS**	**BS**	**VBIH**	**IG**	**ILS-BS**	**GILS**	**BS**	**VBIH**	**IG**
0.2	0.00	0.00	0.67	0.47	0.03	0.03	0.07	0.78	0.71	0.68	0.02	0.20	0.55	0.92	1.03
0.6	0.00	0.00	0.12	0.00	0.11	0.02	0.03	0.55	0.01	0.93	0.12	0.18	0.68	0.22	1.64
1.0	0.01	0.00	0.10	0.17	0.09	0.03	0.07	0.37	0.23	1.03	0.05	0.24	0.49	0.54	1.72
1.4	0.01	0.00	0.08	0.27	0.06	0.08	0.07	0.36	0.56	0.78	0.04	0.23	0.42	0.78	1.39
1.8	0.00	0.00	0.14	0.24	0.04	0.05	0.07	0.78	0.65	0.62	0.03	0.20	0.65	0.86	1.14

Os *gaps* entre o BKS e a solução média encontrada pelos algoritmos ILS-BS e GILS são apresentados na Tabela 5, onde nota-se que ambos os algoritmos conseguem encontrar soluções de alta qualidade de forma consistente, com os *gaps* sendo menores que os encontrados pelas melhores soluções dos métodos existentes. Deve-se ressaltar que algoritmo *beam search* é determinístico e os resultados reportados para o VBIH e IG contém apenas as melhores soluções encontradas por cada um. Desta forma, não foi possível obter a média para estes algoritmos.

Tabela 5 – *Gaps* médios para o BKS considerando a solução média encontrado por cada algoritmo: $n \in \{25, 50, 75\}$

	$n = 25$		$n = 50$		$n = 75$	
R	**ILS-BS**	**GILS**	**ILS-BS**	**GILS**	**ILS-BS**	**GILS**
0.2	0.02	0.02	0.12	0.27	0.14	0.43
0.6	0.01	0.01	0.11	0.15	0.22	0.31
1.0	0.02	0.01	0.10	0.24	0.17	0.44
1.4	0.02	0.01	0.17	0.29	0.16	0.49
1.8	0.02	0.02	0.13	0.24	0.13	0.43

4.4 Resultados para instâncias com ao menos 100 tarefas

A Tabela 6 apresenta os *gaps* médios entre as melhores soluções encontradas pelos procedimentos baseados em ILS e as melhores soluções conhecidas. Já a Tabela 7 exibe os *gaps* médios entre as soluções médias e os BKSs para cada par $n \geq 100$ e R. É importante destacar que os resultados para $n \geq 100$ só estão disponíveis no trabalho de Velez-Gallego, Maya e Montoya-Torres (2016), e, por isso, a comparação considera apenas o algoritmo BS proposto pelos autores.

Os resultados mostram claramente a superioridade do ILS-BS em relação aos métodos GILS e BS. Embora o GILS ainda consiga encontrar *gaps* médios menores que 1% na maioria dos casos, ele não é competitivo com o ILS-BS, especialmente para instâncias maiores. Isso confirma a vantagem do procedimento construtivo baseado em BS em

Tabela 6 – *Gaps* para o BKS considerando as melhores soluções encontradas por cada algoritmo: $n \in \{100, 125, 150\}$

	$n = 100$			$n = 125$			$n = 150$		
R	**ILS-BS**	**GILS**	**BS**	**ILS-BS**	**GILS**	**BS**	**ILS-BS**	**GILS**	**BS**
0.2	0.02	0.33	0.56	0.01	0.54	0.49	0.03	0.70	0.44
0.6	0.09	0.21	0.58	0.06	0.21	0.51	0.03	0.31	0.37
1.0	0.01	0.32	0.41	0.00	0.45	0.36	0.00	0.59	0.32
1.4	0.02	0.44	0.36	0.00	0.54	0.38	0.01	0.80	0.30
1.8	0.01	0.36	0.62	0.01	0.45	0.60	0.00	0.66	0.47

comparação com a abordagem construtiva baseada em GRASP. Adicionalmente, o GILS é também superado pelo BS nas instâncias de 150 tarefas.

Tabela 7 – *Gaps* médios para o BKS considerando a solução média encontrado por cada algoritmo: $n \in \{100, 125, 150\}$

	$n = 100$		$n = 125$		$n = 150$	
R	**ILS-BS**	**GILS**	**ILS-BS**	**GILS**	**ILS-BS**	**GILS**
0.2	0.11	0.57	0.11	0.74	0.13	0.93
0.6	0.20	0.34	0.15	0.35	0.12	0.43
1.0	0.11	0.54	0.10	0.65	0.09	0.79
1.4	0.13	0.70	0.12	0.80	0.13	1.07
1.8	0.13	0.58	0.11	0.66	0.10	0.88

4.5 Sumário dos resultados

A Tabela 8 apresenta o número total de BKSs e soluções ótimas encontradas por cada algoritmo heurístico. É evidente a superioridade do ILS-BS em relação aos demais métodos, visto que esse método foi capaz de encontrar $88,5\%$ das BKSs considerando todas as 1800 instâncias, enquanto o GILS e o BS encontraram apenas $35,3\%$ e $15,5\%$, respectivamente. Quando consideramos apenas as instâncias com $n \leq 75$, o ILS-BS obteve $96,5\%$ dos BKSs, enquanto o GILS, o BS, o VBIH e o IG alcançaram $68,0\%$, $30,4\%$, $47,0\%$ e $37,8\%$ dos BKSs, respectivamente.

Tabela 8 – Número de BKSs encontradas por cada algoritmo.

	ILS-BS	**GILS**	**BS**	**VBIH**[1]	**IG**[1]
#BKS	1596	636	279	375	304

[1]Considerando apenas 900 instâncias: $n \in \{25, 50, 75\}$

4.6 Análise dos tempos de execução

As médias dos tempos de CPU gastos pelos algoritmos ILS-BS, GILS e BS, agregados por n, são apresentadas na tabela 9. Para este experimento, o BS proposto por Velez-Gallego, Maya e Montoya-Torres (2016) foi cuidadosamente re-implementado em C++ e executado na mesma máquina que o ILS-BS e o GILS. Os algoritmos VBIH e IG apresentados em Fan et al. (2019) foram originalmente codificados em C++, e os autores os executaram com um limite de tempo de 25 segundos para cada instância em uma máquina com o mesmo processador que a máquina descrita anteriormente, mas com 8 GB de RAM. Por brevidade, este valor constante na não foi incluído na tabela. Os resultados mostram claramente que o BS autônomo é muito mais rápido do que o ILS-BS e o GILS, mas com perda na qualidade das soluções, como visto anteriormente. Por outro lado, tanto o ILS-BS quanto o GILS requereram, em média, muito menos do que 25 segundos para $n \leq 75$, superando assim o VBIH e o IG em termos de tempo de CPU.

Tabela 9 – Tempos de execução médios

n	**ILS-BS**	**GILS**	**BS**
25	0.63	0.48	0.20
50	3.63	4.00	1.00
75	10.24	13.68	2.63
100	21.28	33.43	5.25
125	37.27	66.47	8.99
150	57.38	97.58	13.97

As comparações entre o ILS-BS e o BS autônomo foram estendidas ao modificar os parâmetros deste último para que o algoritmo tenha mais nós na árvore para encontrar soluções melhores. Mais precisamente, o valor de w foi aumentado de 5 para 20 e N de 500 para 2000. Esta versão alternativa do BS é referida como BS*. Como resultado, o BS* tornou-se mais lento do que o ILS-BS. A Tabela 10 apresenta o tempo médio de CPU e as lacunas médias alcançadas pelo BS* em relação às soluções médias encontradas pelo ILS-BS. Pode-se observar que, embora expandir a árvore do BS possa levar a soluções melhores, as melhorias associadas não são suficientes para alcançar a qualidade de solução obtida pelo ILS-BS para $R = 1,4$ e $R = 1,8$. Para os valores restantes de R, o BS* produziu resultados competitivos, especialmente para $R = 1$, mas, em geral, ainda não melhores ou tão bons quanto o ILS-BS, com exceção de $n = 25$ e $R = 1$, bem como $n = 150$ e $R = 1$.

4.7 Impacto da avaliação eficiente de movimentos

Para quantificar os benefícios de adotar o esquema de avaliação de movimentos eficiente descrito na Seção 3.4.2, foi realizado um experimento em que o algoritmo ILS-BS

Tabela 10 – *Gaps* (%) encontrados pelo BS* em relação a solução média obtida pelo ILS-BS

R	$n = 25$	$n = 50$	$n = 75$	$n = 100$	$n = 125$	$n = 150$
0.2	0.01	0.24	0.26	0.22	0.21	0.14
0.6	0.00	0.12	0.15	0.12	0.12	0.07
1.0	-0.01	0.03	0.08	0.11	0.05	-0.01
1.4	0.07	0.51	0.44	0.36	0.43	0.23
1.8	0.61	0.56	0.34	0.34	0.25	0.20
Avg. Time (s)	1.35	6.74	16.48	31.62	54.70	83.59

é executado sem tal esquema. Essa versão alternativa avalia movimentos em $\mathcal{O}(n)$ tempo. A Figura 9 compara o desempenho de tempo de execução de ambas as versões, em que se pode observar visivelmente os benefícios da implementação de um esquema de avaliação de movimentos eficiente. Além disso, quanto maior a instância, maiores são os ganhos. Para as instâncias de 150 tarefas, o ILS-BS com avaliação de movimento eficiente foi aproximadamente 7,76 vezes mais rápido do que a versão não eficiente.

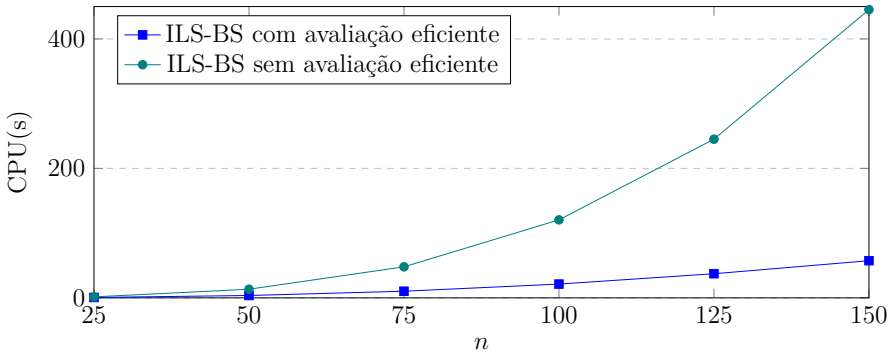

Figura 9 – Tempos de execução do ILS-BS com e sem a avaliação eficiente de movimentos

5 Considerações finais

Este trabalho abordou o problema de sequenciamento em uma máquina com datas de liberação e tempo de *setup* dependentes da sequência ($1|r_j, s_{ij}|C_{max}$). Foi apresentada uma abordagem heurística híbrida baseada em *iterated local search* para o problema, em que dois diferentes métodos construtivos foram testados, um baseado em GRASP e outro baseado em *beam search*. Além disso, foi implementada uma estratégia de avaliação de eficiente de movimentos na busca local, reduzindo a complexidade computacional deste procedimento.

A abordagem proposta foi testada no conjunto de 1800 instâncias descrito em Ovacikt e Uzsoy (1994), e os resultados obtidos foram comparados aos encontrados pelo *beam search* (VELEZ-GALLEGO; MAYA; MONTOYA-TORRES, 2016), VBIH e *iterated greedy* (FAN et al., 2019).

O ILS-BS se mostrou superior aos demais algoritmos, sendo capaz de encontrar 88,5% das melhores soluções conhecidas. Em média, o GILS só apresentou melhores resultados que o ILS-BS em instâncias de 25 tarefas, porém apresentou *gaps* maiores que os do BS e do próprio ILS-BS em instâncias com ao menos 100 tarefas. Assim, a adoção da construção baseada em *beam search* gerou grande melhoria nos resultados em instâncias maiores, apresentando os melhores *gaps* para qualquer conjunto de instâncias com $n > 25$.

Além do ganho na qualidade de solução, o ILS-BS precisou de menos iterações que o GILS para atingir resultados competitivos, chegando a ser 42% mais rápido nas instâncias de 150 tarefas. A adoção da avaliação eficiente de movimentos baseada em concatenação de subsequências foi capaz de reduzir consideravelmente o tempo de execução do algoritmo, com os ganhos aumentando conforme o tamanho da instância aumenta, chegando a ser aproximadamente 7,76 vezes mais rápido que a versão do algoritmo que não utiliza do esquema de avaliação eficiente. Apesar de todos estes ganhos o ILS-BS ainda apresentou tempos de execução maiores que o BS. Por este motivo foi realizado um experimento complementar em que o BS foi executado com parâmetros que permitem um maior número de nós na árvore, levando a um tempo de execução maior que o do ILS-BS, mas mostrando que mesmo com a modificação desses parâmetros o BS ainda não é capaz de atingir a mesma qualidade de solução do ILS-BS.

O ILS-BS proposto se apresenta como uma forma eficiente de abordar o $1|r_j, s_{ij}|C_{max}$, e pode ser aplicado em diferentes problemas de escalonamento de tarefas compatíveis com a representação da solução por uma sequência de tarefas. Futuramente pode-se explorar este método em problemas *single machine* com atributos extras, ou problemas em configurações distintas (*flow-shop*, *open-shop*) que podem ser codificadas em sequências únicas de tarefas.

Referências

ALLAHVERDI, A. et al. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, v. 187, p. 985–1032, 2008.

BALAKRISHNAN, N.; KANET, J. J.; SRIDHARAN, V. Early/tardy scheduling with sequence dependent setups on uniform parallel machines. *Computers & Operations Research*, v. 26, n. 2, p. 127–141, 1999.

BIANCO, L. et al. Scheduling tasks with sequence- dependent processing times. *Naval Research Logistics*, v. 35(2), p. 177–184, 1988.

BILGE Ümit et al. A tabu search algorithm for parallel machine total tardiness problem. *Computers & Operations Research*, v. 31, n. 3, p. 397–414, 2004.

CHANG, T.-Y.; CHOU, F.-D.; LEE, C.-E. A heuristic algorithm to minimize total weighted tardiness on a single machine with release dates and sequence-dependent setup times. *Journal of the Chinese Institute of Industrial Engineers*, v. 21(3), p. 289–300, 2004.

CHOU, F.-D.; WANG, H.-M.; CHANG, T.-Y. Algorithms for the single machine total weighted completion time scheduling problem with release times and sequence-dependent setups. *Int J Adv Manuf Technol*, v. 43, n. 810, 2008.

DIANA, R. O.; de Souza, S. R.; FILHO, M. F. A variable neighborhood descent as ils local search to the minimization of the total weighted tardiness on unrelated parallel machines and sequence dependent setup times. *Electronic Notes in Discrete Mathematics*, v. 66, p. 191–198, 2018. ISSN 1571-0653. 5th International Conference on Variable Neighborhood Search.

DRIESSEL, R.; MöNCH, L. Variable neighborhood search approaches for scheduling jobs on parallel machines with sequence-dependent setup times, precedence constraints, and ready times. *Computers & Industrial Engineering*, v. 61, n. 2, p. 336–345, 2011. Combinatorial Optimizatiion in Industrial Engineering.

FAN, J. et al. A variable block insertion heuristic for single machine with release dates and sequence dependent setup times for makespan minimization. In: *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*. Xiamen, China: IEEE, 2019. p. 1676–1683.

FERNANDEZ-VIAGAS, V.; COSTA, A. Two novel population based algorithms for the single machine scheduling problem with sequence dependent setup times and release times. *Swarm and Evolutionary Computation*, v. 63, 2021.

GRAHAM, R. et al. Optimization and approximation in deterministic sequencing and scheduling: a survey. In: HAMMER, E. J. P.; KORTE, B. (Ed.). *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*. [S.l.]: Elsevier, 1979, (Annals of Discrete Mathematics, v. 5). p. 287 – 326.

KIM, S.-I.; CHOI, H.-S.; LEE, D.-H. Scheduling algorithms for parallel machines with sequence-dependent set-up and distinct ready times: minimizing total tardiness. *PROCEEDINGS OF THE INSTITUTION OF MECHANICAL ENGINEERS PART B-JOURNAL OF ENGINEERING MANUFACTURE*, v. 221, n. 6, p. 1087–1096, 2007.

KINDERVATER, G.; SAVELSBERGH, M. Vehicle routing: handling edge exchanges. *In: Aarts, E., Lenstra, J. (Eds.), Local Search in Combinatorial Optimization. Wiley, New York*, p. 337–360, 1997.

LIN, S.-W. et al. Minimization of maximum lateness on parallel machines with sequence-dependent setup times and job release dates. *Computers & Operations Research*, v. 38, n. 5, p. 809–815, 2011.

LIN, Y.-K.; HSIEH, F.-Y. Unrelated parallel machine scheduling with setup times and ready times. *International Journal of Production Research*, v. 52, n. 4, p. 1200 – 1214, 2014. ISSN 1366588X.

LOGENDRAN, R.; MCDONELL, B.; SMUCKER, B. Scheduling unrelated parallel machines with sequence-dependent setups. *Computers & Operations Research*, v. 34, n. 11, p. 3420–3438, 2007. ISSN 0305-0548.

MARTIN, O.; OTTO, S.; FELTEN, E. Large-step markov chains for the traveling salesman problem. *Complex Systems*, v. 5, n. 3, p. 299–326, 1991.

MLADENOVIC, N.; HANSEN, P. Variable neighborhood search. *Computers & Operations Research*, v. 24(11), p. 1097–1100, 1997.

MONTOYA-TORRES, J.; GONZáLEZ-SOLANO, F.; SOTO-FERRARI, M. Deterministic machine scheduling with release times and sequence-dependent setups using random-insertion heuristics. *International Journal of Advanced Operations Management*, v. 4(1/2), p. 4–26, 2012.

MONTOYA-TORRES, J.; SOTO-FERRARI, M.; GONZáLEZ-SOLANO, F. Production scheduling with sequence-dependent setups and job release times. *Dyna*, v. 77, n. 163, p. 260–269, 2010.

NOGUEIRA, T. H. et al. A hybrid VNS-Lagrangean heuristic framework applied on single machine scheduling problem with sequence-dependent setup times, release dates and due dates. *Optimization Letters*, v. 16, p. 59–78, 2022.

OVACIKT, I.; UZSOY, R. Rolling horizon algorithms for a single-machine dynamic scheduling problem with sequence-dependent setup times. *International Journal of Production Research*, v. 32(6), p. 1243–1263, 1994.

PINEDO, M. L. *Scheduling: Theory, Algorithms, and Systems*. Nova Iorque: Springer-Verlag, 2012.

RUIZ, R.; STüTZLE, T. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, v. 177(3), p. 2033–2049, 2007.

SHIN, H.; KIM, C.-O.; KIM, S. A tabu search algorithm for single machine scheduling with release times, due dates, and sequence-dependent set-up times. *The International Journal of Advanced Manufacturing Technology*, v. 19(11), p. 859–866, 2002.

SILVA, J. M. P.; TEIXEIRA, E.; SUBRAMANIAN, A. Exact and metaheuristic approaches for identical parallel machine scheduling with a common server and sequence-dependent setup times. *Journal of the Operational Research Society*, Taylor & Francis, v. 72, n. 2, p. 444–457, 2021.

SUBRAMANIAN, A. et al. A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery. *Computers & Operations Research*, v. 37, n. 11, p. 1899–1911, 2010. ISSN 0305-0548. Metaheuristics for Logistics and Vehicle Routing.

VELEZ-GALLEGO, M. C.; MAYA, J.; MONTOYA-TORRES, J. A beam search heuristic for scheduling a single machine with release dates and sequence dependent setup times to minimize the makespan. *Computers & Operations Research*, v. 73, p. 132–140, 2016.

VIDAL, T. et al. A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. *Computers & operations research*, Elsevier, v. 40, n. 1, p. 475–489, 2013.

YING, K.-C.; CHENG, H.-M. Dynamic parallel machine scheduling with sequence-dependent setup times using an iterated greedy heuristic. *Expert Systems with Applications*, v. 37, n. 4, p. 2848–2852, 2010.