UNIVERSIDADE FEDERAL DE OURO PRETO

# Novos Algoritmos para o Problema de Sequenciamento em Máquinas Paralelas Não-Relacionadas com Tempos de Preparação Dependentes da Sequência

Luciano Perdigão Cota
Universidade Federal de Ouro Preto

Orientador: Marcone Jamilson Freitas Souza

Coorientador: Alexandre Xavier Martins

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Ouro Preto, como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação.

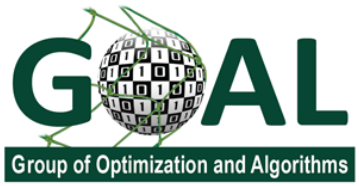
Ouro Preto, Julho de 2014

# Novos Algoritmos para o Problema de Sequenciamento em Máquinas Paralelas Não-Relacionadas com Tempos de Preparação Dependentes da Sequência

Luciano Perdigão Cota
Universidade Federal de Ouro Preto

Orientador: Marcone Jamilson Freitas Souza

Coorientador: Alexandre Xavier Martins


C843n Cota, Luciano Perdigão.
Novos algoritmos para o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência [manuscrito] / Luciano Perdigão Cota – 2014.
134 f.: il. color.; grafs.; tabs.

Orientador: Prof. Dr. Marcone Jamilson Freitas Souza.
Co-orientador: Prof. Dr. Alexandre Xavier Martins.

Dissertação (Mestrado) - Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação. Programa de Pós-graduação em Ciência da Computação.
Área de concentração: Ciência da Computação

1. Otimização combinatória - Teses. 2. Programação heurística - Teses. 3. Programação (Matemática) - Teses. I. Souza, Marcone Jamilson Freitas. II. Martins, Alexandre Xavier. III. Universidade Federal de Ouro Preto. IV. Título.

CDU: 004.421

Catalogação: sisbin@sisbin.ufop.br

**Ata da Defesa Pública de Dissertação de Mestrado**

Aos 14 dias do mês de julho de 2014, às 10 horas na Sala de Seminários do DECOM no Instituto de Ciências Exatas e Biológicas (ICEB), reuniram-se os membros da banca examinadora composta pelos professores: **Prof. Dr. Marcone Jamilson Freitas Souza (presidente e orientador), Prof. Dr. Alexandre Xavier Martins (coorientador), Prof. Dr. Frederico Gadelha Guimarães e Prof. Dr. José Elias Cláudio Arroyo**, aprovada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação, a fim de arguirem o mestrando **Luciano Perdigão Cota**, com o título **"Novos Algoritmos para o Problema de Sequenciamento em Máquinas Paralelas Não-Relacionadas com Tempos de Preparação Dependentes da Sequência"**. Aberta a sessão pelo presidente, coube ao candidato, na forma regimental, expor o tema de sua dissertação, dentro do tempo regulamentar, sendo em seguida questionado pelos membros da banca examinadora, tendo dado as explicações que foram necessárias.

Recomendações da Banca:

( X ) Aprovada sem recomendações

( ) Reprovada

( ) Aprovada com recomendações: _____

Banca Examinadora:

Prof. Dr. Marcone Jamilson Freitas Souza

Prof. Dr. Alexandre Xavier Martins

Prof. Dr. Frederico Gadelha Guimarães

Prof. Dr. José Elias Cláudio Arroyo

Prof. Dr. Luiz Henrique de Campos Merschmann
Coordenador do Programa de Pós-Graduação em Ciência da Computação
DECOM/ICEB/UFOP

**Ouro Preto, 14 de julho de 2014.**

*Dedico este trabalho aos meus pais José Perdigão (in memoriam) e Maria Aparecida, aos meus irmãos Renato Assis e Ana Paula, e a Paula Amora.*

# Resumo

Este trabalho trata o Problema de Sequenciamento em Máquinas Paralelas Não-Relacionadas com Tempos de Preparação Dependentes da Sequência (UPMSPST, do inglês *Unrelated Parallel Machine Scheduling Problem with Setup Times*), objetivando minimizar o *makespan*. Para resolvê-lo foram desenvolvidos três algoritmos heurísticos e um algoritmo híbrido. O primeiro algoritmo heurístico, denominado HIVP, tem uma solução inicial gerada por um procedimento construtivo parcialmente guloso baseado no método *Heuristic-Biased Stochastic Sampling* e na regra *Adaptive Shortest Processing Time* – ASPT. Essa solução é, posteriormente, refinada pelo procedimento *Iterated Local Search* – ILS, tendo o *Random Variable Neighborhood Descent* como método de busca local. Além disso, periodicamente a busca é intensificada e diversificada por meio de um procedimento *Path Relinking* – PR. No segundo algoritmo heurístico, denominado GIAP, a solução inicial é criada por um procedimento inspirado no *Greedy Randomized Adaptive Search Procedures*. Nesse segundo algoritmo, a solução é refinada por um procedimento ILS que utiliza como método de busca local o procedimento *Adaptive Local Search* – ALS. A busca é também intensificada e diversificada por meio de um procedimento PR. O terceiro e último algoritmo heurístico, denominado AIRP, tem sua solução inicial gerada por um procedimento construtivo guloso baseado na regra ASPT. Essa solução é refinada por um procedimento ILS que tem como busca local um procedimento chamado RIV. De forma análoga aos algoritmos anteriores, a busca também passa por uma intensificação e diversificação periodicamente por meio de um procedimento PR. O algoritmo híbrido, denominado AIRMP, tem o funcionamento similar ao do algoritmo heurístico AIRP, diferindo deste por acrescentar um módulo de programação linear inteira mista. Para a aplicação desse módulo são selecionados um par de máquinas e subconjuntos de tarefas nessas máquinas. Esses subconjuntos são combinados e passam por uma busca local que consiste em acionar um resolvedor de programação matemática aplicado à melhor das formulações de programação matemática dentre aquelas estudadas e desenvolvidas.

Pelos experimentos computacionais foi possível concluir que o AIRP obteve os melhores resultados dentre os algoritmos heurísticos propostos, conseguindo superar vários outros algoritmos da literatura. Também foram realizados experimentos para comparar os algoritmos AIRMP e AIRP. Como o AIRMP necessita de um tempo maior para acionar o módulo de programação matemática, esses experimentos utilizaram um tempo maior de execução. Observou-se, no entanto, que a adição do módulo de programação matemática não melhorou o desempenho do AIRMP no tempo testado e na estrutura utilizada de subconjuntos de tarefas. Esses testes também mostraram que aumentando-se o tempo de processamento do AIRP, o algoritmo é capaz de encontrar melhores soluções.

Palavras-chave: Máquinas Paralelas, *Makespan*, *Iterated Local Search*, *Adaptive Local Search*, *Random Variable Neighborhood Descent*, *Path Relinking*, *Greedy Randomized Adaptive Search Procedures*, *Adaptive Shortest Processing Time*.

# Abstract

This paper deals with the Unrelated Parallel Machine Scheduling Problem with Setup Times – UPMSPST, having as goal to minimize the *makespan*. In order to solve it, three heuristic algorithms and a hybrid algorithm were developed. The first heuristic algorithm, called HIVP, has an initial solution generated by a greedy constructive procedure based on the *Heuristic-Biased Stochastic Sampling* and on the *Adaptive Shortest Processing Time* – ASPT rule. This solution is then refined by the *Iterated Local Search* – ILS procedure, having the *Random Variable Neighborhood Descent* as local search method. Moreover, the search is periodically intensified and diversified by a *Path Relinking* – PR procedure. In the second algorithm, called GIAP, the initial solution is created by a procedure inspired on the *Greedy Randomized Adaptive Search Procedures*. This solution is then refined by an ILS procedure that uses the procedure *Adaptive Local Search* – ALS as local search method. The search is also intensified and diversified by a PR procedure. The third and final heuristic algorithm, called AIRP, has its initial solution generated by a greedy constructive procedure based on ASPT rule. This solution is then refined by the ILS, having as local search a procedure called RVI. Analogously to the previous algorithms, the search also applies periodically an intensification and diversification strategy based on the PR procedure. The hybrid algorithm, named AIRMP, is similar to the AIRP heuristic algorithm, differing from it by adding a module of mixed integer linear programming. To apply this module one pair of machines are selected and subsets of jobs of these machines. These subsets are combined and they pass through a local search that consists in running a mathematical programming solver applied to the best formulation among the studied and developed ones. By computational experiments it was concluded that the AIRP algorithm obtained the best results among the proposed heuristic algorithms, outperforming several other algorithms from the literature. Experiments were also conducted to compare the AIRMP and AIRP algorithms. As the AIRMP needs more time to execute the mathematical programming module, these

experiments utilized a higher runtime. It was observed, however, that the addition of the mathematical programming module does not improved the performance of the AIRMP algorithm in the tested time and in the used structure of subsets of jobs. These tests also showed that increasing the processing time of the AIRP, the algorithm is able to find better solutions.

Keywords: Parallel Machine, *Makespan*, *Iterated Local Search*, *Adaptive Local Search*, *Random Variable Neighborhood Descent*, *Path Relinking*, *Greedy Randomized Adaptive Search Procedures*, *Adaptive Shortest Processing Time*.

# Declaração

Esta dissertação é resultado de meu próprio trabalho, exceto onde referência explicitatória é feita ao trabalho de outros, e não foi submetida para outra qualificação nesta nem em outra universidade.

Luciano Perdigão Cota

# Agradecimentos

À minha mãe Maria Aparecida que, mesmo nos momentos mais difíceis, foi minha grande fonte de incentivo e perseverança.

Ao meu pai José Perdigão *(in memoriam)*, por ser uma grande pessoa e será sempre meu grande herói.

Aos meus irmãos Renato Assis e Ana Paula pelo apoio contínuo e união.

Ao meu amor Paula Amora que sempre acreditou no meu pontecial e sempre me incentivou em todos os momentos.

À todos os demais familiares pela confiança.

Aos amigos, que sempre estiveram dispostos a ajudar.

Aos professores do Programa de Pós-Graduação em Ciência da Computação da UFOP, que sempre estiveram presentes para sanar as minhas dúvidas e colaborar na minha formação.

Ao meu orientador Marcone que, nestes dois anos, me proporcionou um imensurável conhecimento que me despertou para a pesquisa científica. Não obstante à sua ocupação de Reitor, esteve sempre disponível a me auxiliar nas reuniões com grande dedicação, paciência e motivação. Seu profissionalismo será sempre fonte de inspiração e admiração.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Algoritmos

# Nomenclatura

| | |
|---|---|
| ACO | *Ant Colony Optimization* |
| *ALS* | *Adaptive Local Search* |
| ANOVA | *Analysis of Variance* - Análise de Variância |
| AGCR | Algoritmo genético desenvolvido por Haddad et al. (2011) |
| AGVL | Algoritmo genético desenvolvido por Haddad et al. (2011) |
| AIRMP | Algoritmo baseado nos procedimentos ASPT, ILS, VND, $PLIM_{UPMSPST}$ e PR |
| AIRP | Algoritmo baseado nos procedimentos ASPT, ILS, VND, e PR |
| $BkPR_{AM}$ | Procedimento PR onde o atributo é a alocação de todas as tarefas de uma máquina |
| $BkPR_{AT}$ | Procedimento PR onde o atributo é a posição de uma tarefa |
| AIRP | Algoritmo baseado no ASPT, ILS, VND e *Path Relinking* |
| $CG_{ASPT}$ | Procedimento construtivo guloso baseado na regra ASPT |
| $CPG_{ASPT}$ | Procedimento construtivo parcialmente guloso baseado na regra ASPT |
| $CPG^{HBSS}_{ASPT}$ | Procedimento construtivo parcialmente guloso baseado no HBSS |
| $CPG_{GRASP}$ | Procedimento construtivo parcialmente guloso baseado em GRASP |
| $BI_{TMM}$ | Método de busca local que utiliza a estrutura de vizinhança $N^{TMM}(.)$ |
| $FI^{1}_{MI}$ | Método 1 de busca local que utiliza a estrutura de vizinhança $N^{MI}(.)$ |
| $FI^{2}_{MI}$ | Método 2 de busca local que utiliza a estrutura de vizinhança $N^{MI}(.)$ |

| | |
|---|---|
| $FI_{TMD}$ | Método de busca local que utiliza a estrutura de vizinhança $N^{TMD}(.)$ |
| GA1 | Algoritmo genético desenvolvido por Vallada e Ruiz (2011) |
| GA2 | Algoritmo genético desenvolvido por Vallada e Ruiz (2011) |
| GIAP | Algoritmo baseado nos procedimentos GRASP, ILS, VND e *Path Relinking* |
| GIVMP | Algoritmo implementado em Haddad (2012) |
| GIVP | Algoritmo implementado em Haddad (2012) |
| GRASP | *Greedy Randomized Adaptive Search Procedures* |
| HBSS | *Heuristic-Biased Stochastic Sampling* |
| HIVP | Algoritmo baseado nos procedimentos HBSS, ILS, VND e *Path Relinking* |
| ILS | *Iterated Local Search* |
| IVP | Algoritmo implementado em Haddad (2012) |
| LB | *Lower Bound* - Limite Inferior |
| Meta-RaPS | *Metaheuristic for Randomized Priority Search* |
| MIP | *Mixed Integer Programming Heuristics* |
| MPLIM_IT | Formulação de programação matemática, indexada no tempo, para o PSUMAA |
| $N^{TMD}(.)$ | Estrutura de vizinhança de movimentos de trocas em máquinas diferentes |
| $N^{MI}(.)$ | Estrutura de vizinhança de movimentos de múltipla inserção |
| $N^{TMM}(.)$ | Estrutura de vizinhança de movimentos de trocas na mesma máquina |
| $Perturb_{TMD}$ | Procedimento de perturbação que utiliza a estrutura de vizinhança $N^{TMD}(.)$ |
| $Perturb_{ILS}$ | Procedimento de perturbação baseado em movimentos de inserção |
| PMSP | *Parallel Machine Scheduling Problem* |
| PLIM | Programação Linear Inteira Mista |
| $PLIM_{UPMSPST}$ | Procedimento de busca local utilizando PLIM |
| PR | *Path Relinking* |

| | |
|---|---|
| PSUMAA | Problema de sequenciamento em uma máquina com penalidades por antecipação e atraso na produção |
| $RPD$ | Desvio percentual relativo |
| $RPD_{avg}$ | Desvio percentual relativo médio |
| $RPD_{best}$ | Desvio percentual relativo à melhor solução |
| *RVI* | Procedimento de busca local baseado no RVND e no ILS |
| *RVND* | *Random Variable Neighborhood Descent* |
| VND | *Variable Neighborhood Descent* |
| VNS | *Variable Neigborhood Search* |
| UPMSPST | *Unrelated Parallel Machine Scheduling Problem with Setup Times* |
| UPMSPST-VR | Formulação Matemática para o UPMSPST desenvolvida por Vallada e Ruiz (2011) |
| UPMSPST-RA | Formulação Matemática para o UPMSPST desenvolvida por Rabadi et al. (2006) |
| UPMSPST-CV | Formulação Matemática para o UPMSPST baseada no problema do Caixeiro Viajante Assimétrico |
| UPMSPST-IT | Formulação Matemática indexada no tempo para o UPMSPST |

# Capítulo 1

# Introdução

A vida moderna proporciona uma concorrência cada vez maior dentre as organizações, onde não há limite de fronteiras para os mercados. Nessas concorrências as organizações buscam formas constantes e inteligentes de aperfeiçoamento da produção e/ou prestação de serviços. Na busca pela melhoria dos processos de produção, uma questão importante são os custos de produção, os quais têm grande influência no preço final do produto e no lucro obtido pelas organizações.

Em muitas empresas, o processo de produção envolve o atendimento a um conjunto de demandas (tarefas) usando-se um conjunto de máquinas. Tal problema, conhecido como Problema de Sequenciamento em Máquinas, consiste em procurar a ordem ótima de alocação dessas tarefas nas máquinas disponíveis.

Este trabalho trata o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência (UPMSPST, do inglês *Unrelated Parallel Machine Scheduling Problem with Setup Times*). Neste problema, tem-se um conjunto de máquinas $M$ e um conjunto de tarefas $N$, as máquinas de $M$ são consideradas paralelas por poderem executar qualquer tarefa do conjunto $N$. As máquinas são ditas não-relacionadas porque o tempo de execução de cada tarefa de $N$ é diferente em cada máquina de $M$. Têm-se, também, um tempo de processamento de cada tarefa de $N$ em uma máquina de $M$, e um tempo de preparação das máquinas que depende da ordem da alocação das tarefas e da máquina para a qual foi alocada. O objetivo é alocar todas as tarefas de $N$ nas máquinas de $M$ buscando minimizar o tempo máximo de conclusão do sequenciamento, o chamado *makespan*.

O UPMSPST tem grande importância téoria e prática. A importância teória se deve

ao fato de o UPMSPST pertencer à classe de problemas $\mathcal{NP}$-difíceis (Ravetti et al., 2007), porque ele é uma generalização do *Parallel Machine Scheduling Problem with Identical Machines and without Setup Times* (Garey e Johnson, 1979; Karp, 1972). Já a importância prática segue do fato de o UPMSPST estar presente em indústrias de diferentes áreas, como na área téxtil (Pereira Lopes e de Carvalho, 2007).

Dada a dificuldade de resolver o UPMSPST na otimalidade em tempos de processamento aceitáveis para a tomada de decisão em casos reais, a utilização de métodos exatos fica limitada a problemas de pequenas dimensões. Em vista disso, as heurísticas têm sido as técnicas mais utilizadas para resolver esse problema (Rabadi et al., 2006).

Entre os principais trabalhos da literatura que abordam o UPMSPST estão Rabadi et al. (2006), Vallada e Ruiz (2011) e Haddad (2012).

Em (Rabadi et al., 2006) é desenvolvido um algoritmo baseado na metaheurística Meta-RaPS (*Metaheuristic for Randomized Priority Search*), a qual combina procedimentos de construção e refinamento, utilizando ideias próximas do *Greedy Randomized Adaptive Search Procedures* – GRASP (Feo e Resende, 1995).

Em (Vallada e Ruiz, 2011) são desenvolvidos Algoritmos Genéticos (Goldberg, 1989; Holland, 1975) para o UPMSPST, cuja principal característica é a realização de cruzamentos com buscas locais "limitadas".

Em (Haddad, 2012) foram propostos algoritmos baseados em *Iterated Local Search* – ILS (Lourenço et al., 2003) e *Variable Neighborhood Descent* – VND (Hansen et al., 2008). O melhor deles, o algoritmo GIVMP, usa um procedimento de construção gulosa aletória para gerar uma solução inicial, o ILS como metaheurística para guiar o processo de busca, o procedimento *Random Variable Neighborhood Descent* – RVND (Souza et al., 2010) para realizar as buscas locais, e o *Path Relinking* – PR (Glover, 1996) como técnica de intensificação e diversificação. Ao final é feita uma pós-otimização usando um módulo de programação linear inteira mista – PLIM para uma máquina, o qual atua como mecanismo de busca local. O autor mostrou que o algoritmo é superior aos de (Vallada e Ruiz, 2011) e (Rabadi et al., 2006).

Dado o bom desempenho do GIVMP na resolução do UPMSPST, o presente trabalho se propõe a aperfeiçoá-lo. Entre os aperfeiçoamentos propostos, destacamos: 1) novas heurísticas para construção da solução incial; 2) novos procedimentos heurísticos para a busca local; 3) novos procedimentos para intensificar e diversificar a busca e, finalmente, 4) desenvolvimento de um novo módulo de programação matemática, desta vez,

específico para problemas de sequenciamento. Ao contrário de (Haddad, 2012), neste trabalho este módulo é usado periodicamente para realizar buscas locais e não como pós-otimização.

## 1.1 Objetivos

Nesta seção são apresentados os principais objetivos deste trabalho.

### 1.1.1 Objetivo Geral

O objetivo deste trabalho é desenvolver algoritmos eficientes, baseados em técnicas heurísticas e/ou de programação matemática, que sejam capazes de produzir soluções de boa qualidade em um tempo restrito, para resolver o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência (UPMSPST).

### 1.1.2 Objetivos Específicos

Para alcançar o objetivo geral se faz necessário a obtenção dos seguintes objetivos específicos:

- Analisar e estudar trabalhos da literatura que tratam o problema abordado e relacionados;
- Analisar e estudar técnicas de solução de problemas de otimização combinatória;
- Propor duas formulações de programação matemática, uma indexada no tempo e outra com restrições do problema do caixeiro viajante assimétrico;
- Desenvolver algoritmos heurísticos, com a utilização de diferentes heurísticas, metaheurísticas e técnicas de intensificação e diversificação;
- Desenvolver um algoritmo híbrido, que combine procedimentos heurísticos com um módulo de programação linear inteira mista;
- Realizar experimentos computacionais utilizando problemas-teste da literatura, buscando comprovar a eficiência dos algoritmos desenvolvidos;

## 1.2 Motivação

Devido à grande concorrência existente entre as organizações, o custo de produção tem grande importância para as empresas. A redução desses custos pode implicar em um menor custo do produto e/ou serviço, proporcionando à organização maior competitividade frente ao mercado global.

O UPMSPST está presente em organizações de diferentes ramos, como: téxteis, químicos, tintas, semicondutores e papéis (Rabadi et al., 2006). Por isso, este problema tem grande importância prática. A resolução de forma eficiente deste problema pode implicar em menores custos de produção para as organizações, melhorando a competitividade dessas; daí a importância do desenvolvimento de algoritmos eficientes para a solução do problema.

Por outro lado, o UPMSPST tem grande importância teórica, por ser um problema da classe $\mathcal{NP}$-difícil. Desta forma, é desafiador o desenvolvimento de algoritmos eficientes para sua solução. Entre esses algoritmos, destacam-se atualmente os algoritmos híbridos, que combinam procedimentos heurísticos e formulações de PLIM. Esses procedimentos, chamados *matheuristics*, ou heurísticas MIP (*Mixed Integer Programming Heuristics*), buscam utilizar a flexibilidade das heurísticas e o poderio das técnicas exatas.

## 1.3 Estrutura do Trabalho

O restante deste trabalho está estruturado como segue.

No capítulo 2 é caracterizado o problema UPMSPST. A revisão bibliográfica é apresentada no capítulo 3. O capítulo 4 apresenta os algoritmos desenvolvidos para resolver o problema. Já no capítulo 5 são apresentados os experimentos computacionais. O capítulo 6 conclui esta dissertação e apresenta perspectivas de trabalhos futuros.

# Capítulo 2

# Caracterização do Problema

O problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência, ou UMPSPST, é caracterizado por se ter um conjunto de tarefas $N = \{1, ..., n\}$ e um conjunto de máquinas $M = \{1, ..., m\}$, com as características abaixo:

(a) Cada tarefa deve ser alocada a uma única máquina;

(b) O tempo de processamento, ou execução, de uma tarefa $j \in N$ em uma máquina $i \in M$ é $p_{ij}$. Este tempo depende da máquina para qual a tarefa será alocada, por isso as máquinas pertencem à categoria de máquinas paralelas não-relacionadas.

(c) Existe um tempo de preparação, $S_{ijk}$, para se processar a tarefa $k \in N$ após a tarefa $j \in N$ na máquina $i \in M$. O tempo de preparação $S_{hjk}$ em uma máquina $h \in M$, com $h \neq i$, pode ser diferente do tempo de preparação $S_{ijk}$.

(d) Para facilitar a modelagem do problema é definida uma tarefa fictícia, denominada tarefa 0, que será a primeira tarefa a ser executada em todas as máquinas. Desta forma, existe um tempo de preparação $S_{i0j}$ para processar a tarefa $j \in N$ na primeira posição da máquina $i \in M$. O tempo de processamento da tarefa fictícia, $p_{i0}$, em uma máquina $i \in M$ é zero.

O objetivo é alocar as tarefas do conjunto $N$ nas máquinas do conjunto $M$, buscando minimizar o *makespan*. O *makespan*, denotado por $C_{\max}$, é o tempo consumido pela máquina que conclui suas tarefas por último.

De acordo com as características do UPMSPST, o problema pode ser representado por $R_M \mid s_{ijk} \mid C_{\max}$ (Graham et al., 1979). Nesta representação, $R_M$ indica as máquinas não-relacionadas, $S_{ijk}$ os tempos de preparação e $C_{\max}$ o *makespan*.

O UPMSPST pertence à classe de problemas $\mathcal{NP}$-difíceis, por se tratar de uma generalização do PMSP com máquinas idênticas e sem tempos de preparação ($P_M||C_{\max}$), o qual é da classe $\mathcal{NP}$-difícil mesmo com duas máquinas (Garey e Johnson, 1979; Karp, 1972).

Para ilustrar o problema, é apresentado a seguir um problema-teste do UPMSPST envolvendo 6 tarefas e 2 máquinas. Na Tabela 2.1 estão descritos os tempos de processamento das 6 tarefas nas 2 máquinas e nas tabelas 2.2 e 2.3 estão os tempos de preparação para essas tarefas nas máquinas 1 e 2, respectivamente.

**Tabela 2.1:** Tempos de processamento nas máquinas $M_1$ e $M_2$

| | $M_1$ | $M_2$ |
|---|---|---|
| 1 | 1 | 4 |
| 2 | 87 | 21 |
| 3 | 28 | 68 |
| 4 | 32 | 17 |
| 5 | 38 | 43 |
| 6 | 9 | 48 |

**Tabela 2.2:** Tempos de preparação na máquina $M_1$

| $M_1$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 3 | 1 | 8 | 1 | 3 | 9 |
| 2 | 4 | 3 | 7 | 3 | 7 | 8 |
| 3 | 7 | 3 | 1 | 2 | 3 | 5 |
| 4 | 3 | 8 | 3 | 2 | 5 | 2 |
| 5 | 8 | 3 | 7 | 9 | 1 | 5 |
| 6 | 8 | 8 | 1 | 2 | 2 | 3 |

Na Figura 2.1 é ilustrado um exemplo de possível alocação para as 6 tarefas nas 2 máquinas. Como pode ser observado, na máquina 1, denominada $M_1$, estão alocadas as

**Tabela 2.3:** Tempos de preparação na máquina $M_2$

| $M_2$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 8 | 5 | 1 | 6 | 1 | 7 |
| 2 | 6 | 1 | 7 | 7 | 6 | 2 |
| 3 | 7 | 6 | 3 | 9 | 6 | 9 |
| 4 | 3 | 7 | 3 | 2 | 1 | 7 |
| 5 | 5 | 8 | 5 | 6 | 1 | 9 |
| 6 | 7 | 4 | 1 | 7 | 9 | 2 |

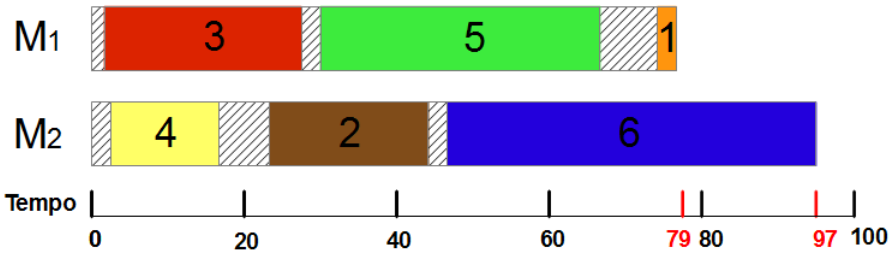



**Figura 2.1:** Exemplo de um possível sequenciamento

tarefas 3, 5 e 1, respectivamente. Já na máquina 2, $M_2$, estão alocadas as tarefas 4, 2 e 6, respectivamente. A parte hachurada do desenho representa os tempos de preparação, e a parte colorida o tempo de processamento de cada tarefa. Para a criação desta figura foram utilizadas os dados das tabelas 2.1, 2.2 e 2.3. Por exemplo, para identificar o tempo de processamento da tarefa 5 na máquina 1, $p_{15}$, basta acessar o elemento da quinta linha e primeira coluna da Tabela 2.1. Já para identificar o tempo de preparação, $S_{151}$, para processar a tarefa 1 após a tarefa 5 na máquina 1, deve-se acessar o elemento que está na quinta linha e primeira coluna da Tabela 2.2.

Desta forma, pode-se calcular o tempo de conclusão da máquina $M_1$ como:

$$\begin{aligned} C_{M_1} &= s_{103} + p_{13} + s_{135} + p_{15} + s_{151} + p_{11} \\ C_{M_1} &= 1 + 28 + 3 + 38 + 8 + 1 = 79 \end{aligned}$$

De modo equivalente, também é calculado o tempo de conclusão da máquina $M_2$:

$$\begin{aligned} C_{M_2} &= s_{204} + p_{24} + s_{242} + p_{22} + s_{226} + p_{26} \\ C_{M_2} &= 2 + 17 + 7 + 21 + 2 + 48 = 97 \end{aligned}$$

Neste exemplo, o *makespan* é 97, visto que a máquina 2 é a que termina suas tarefas por último.

# Capítulo 3

# Revisão Bibliográfica

Primeiramente são apresentados neste capítulo, na seção 3.1, os trabalhos relacionados ao tema. Posteriormente, são apresentadas na seção 3.2, quatro formulações matemáticas para o UPMSPST, sendo duas delas contribuições deste trabalho. Nas seções seguintes, 3.3 e 3.4, são apresentados os conceitos de heurísticas e metaheurísticas, respectivamente. Finalmente, na seção 3.5, é feita uma revisão da técnica *Path Relinking*.

## 3.1 Trabalhos Relacionados

Pelo reduzido número de trabalhos que tratam o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência, na primeira subseção são tratados os problemas similares ao UPMSPST. Na subseção seguinte é feita uma revisão dos trabalhos que tratam especificamente o UPMSPST.

### 3.1.1 Problemas Similares ao UPMSPST

Em (Weng et al., 2001) trata-se o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência somente das tarefas, tendo-se como objetivo minimizar a média ponderada do tempo de conclusão. Foram desenvolvidas sete heurísticas construtivas. Para os experimentos foram utilizados problemas-teste com até 12 máquinas e 120 tarefas. As heurísticas foram definidas utilizando critérios gulosos. O algoritmo 7 se mostrou o mais eficiente. Esse algoritmo aloca, a cada passo, a tarefa $j$ na máquina $k$, onde $j$ e $k$ são tais seja mínima a di-

visão da soma dos tempos de término da máquina $k$, de processamento da tarefa $j$ e de preparação dessa tarefa nessa máquina pela penalidade por atraso da tarefa $j$.

Em (Kim et al., 2002) é tratado um problema de sequenciamento com lotes, tendo como objetivo minimizar o tempo total de atraso. Para a resolução deste problema foi desenvolvido um algoritmo *Simulated Annealing*. Foram geradas seis estruturas de vizinhança, com os seguintes movimentos: 1) troca entre lotes; 2) inserções entre lotes; 3) fusões dos lotes; 4) separações dos lotes; 5) trocas de itens; e 6) inserções de itens. Nos experimentos computacionais o *Simulated Annealing* foi comparado a um método de descida, que utiliza as mesmas estruturas de vizinhança. Segundo os autores, o algoritmo *Simulated Annealing* obteve melhor desempenho.

Já em (Kim et al., 2003), é abordado o problema anterior com a inserção de datas de entregas iguais, tendo como objetivo minimizar o tempo total de atraso. Foram desenvolvidas quatro heurísticas, sendo elas: 1) heurística construtiva baseada na data de entrega mais cedo ponderada; 2) heurística construtiva usando o tempo de processamento ponderado; 3) heurística construtiva baseada no sequenciamento em lote de dois níveis e 4) algoritmo *Simulated Annealing*. Os experimentos foram realizados utilizando dados de uma fábrica de semicondutores. A heurística de sequenciamento em lote de dois níveis e o *Simulated Annealing* foram os algoritmos que obtiveram os melhores resultados.

No trabalho (Rocha et al., 2006) é tratado o problema de sequenciamento em máquinas paralelas com datas de entrega e tempo de preparação dependentes da sequência. É desenvolvido um algoritmo *branch-and-bound* que utililiza como limite superior os resultados obtidos por um algoritmo GRASP. Os resultados desse algoritmo são comparados com os gerados por duas formulações de programação linear inteira mista "puros" executadas no solver CPLEX; o algoritmo *branch-and-bound* obteve os melhores resultados. Também foi gerado um conjunto de novos problemas-teste para o problema tratado.

Em (Logendran et al., 2007) o problema abordado é o de sequenciamento em máquinas paralelas não-relacionadas com tempos dependentes da sequência das tarefas, considerando entrada dinâmica das tarefas e disponibilidade dinâmica das máquinas. O objetivo é minimizar o tempo total de atraso ponderado. Foram desenvolvidas seis variações do algoritmo Busca Tabu, os quais utilizam quatro regras de despacho para gerar as soluções iniciais: 1) data de entrega mais cedo; 2) menor atraso ponderado; 3) menor proporção entre o peso e a data de entrega; e 4) menor proporção entre a data de entrega dividida pela soma do tempo de preparação com o tempo de liberação da tarefa vezes o peso. Já as seis variações da Busca Tabu são baseadas na memória de curto e longo prazo,

reinício da busca e tamanho da lista tabu. Ao final dos experimentos, os autores concluíram que a Busca Tabu com memória de curto prazo e tamanho fixo da lista tabu são recomendadas para problemas de pequeno porte; já a Busca Tabu com frequência mínima e tamanho fixo da lista tabu é o indicado para resolver problemas de médio porte e, finalmente, para a solução de problemas de grande porte a Busca Tabu com frequência mínima e lista tabu de tamanho variável é a mais indicada.

No trabalho (Randall e Kurz, 2007) é tratado o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência das tarefas com datas de entrega das tarefas, objetivando minimizar o atraso total ponderado. Para a resolução deste problema é proposto um algoritmo genético adaptativo. A representação da solução é realizada utilizando chaves aleatórias. A população inicial é gerada aleatoriamente. São usados quatro operadores de cruzamento: um ponto de corte, dois pontos de corte, uniforme e uniforme parametrizados. Os resultados desses algoritmos foram comparados com um algoritmo genético tradicional e uma Busca Tabu. Foram utilizados problemas-teste com até 200 tarefas e 20 máquinas. Segundo os autores, o algoritmo genético adaptativo gerou soluções de melhor qualidade que os demais, mas exigiu um maior tempo computacional.

Para um problema de sequenciamento em máquinas paralelas com tempos de preparação dependentes da sequência das tarefas com datas de disponibilidade das máquinas e datas de liberação para as tarefas, Pereira Lopes e de Carvalho (2007) desenvolveram um algoritmo *Branch-and-Price*. O autor propôs um método de geração de colunas, chamado *primal box*, e um método de seleção de variáveis a serem ramificadas que reduzem de forma significativa o número de nós explorados. Os experimentos mostraram que os algoritmos desenvolvidos são capazes de resolver problemas com até 50 máquinas e 150 tarefas em um tempo computacional razoável.

Em (de Paula et al., 2007) é implementado um *Variable Neigborhood Search* – VNS para o problema de sequenciamento em máquinas paralelas não-relacionadas idênticas com datas de entrega para as tarefas e penalidade para atraso nas datas de entrega. O objetivo é minimizar a soma do *makespan* com atrasos ponderados. Os experimentos foram realizados comparando o VNS com três versões de um *Greedy Randomized Adaptive Search Procedures* – GRASP. O VNS obteve melhores resultados médios nos problemas-teste com 60 tarefas ou mais.

Em (Rosa et al., 2009), o problema abordado é o de sequenciamento em uma máquina com penalidades por antecipação e atraso da produção (PSUMAA). Nesse trabalho fo-

ram propostas três formulações de programação matemática para sua resolução. Destaca-se a formulação MPLIM_IT, indexada no tempo, que obteve desempenho muito superior às demais nos problemas-teste maiores considerados.

No trabalho (Peydro e Ruiz, 2010), o problema analisado é o de sequenciamento em máquinas paralelas não-relacionadas sem considerar os tempos de preparação dependentes da sequência, objetivando minimizar o *makespan*. Foram desenvolvidos sete algoritmos, baseados no algoritmo *Iterated Greedy Search* (Peydro e Ruiz, 2010) e no *Variable Neighborhood Descent* – VND (Hansen et al., 2008). Para os experimentos foram utilizados 1400 problemas-teste. Segundo os autores, os algoritmos se mostraram competitivos quando comparados com outros das literatura.

Em (Peydro e Ruiz, 2011) os autores desenvolveram novos algoritmos e um método de redução da dimensão do problema tratado no trabalho anterior (Peydro e Ruiz, 2010). A redução de tamanho se baseia na ideia de executar somente as alocações das tarefas mais promissoras, utilizando o tempo de processamento. Nos experimentos computacionais foram utilizados os mesmos 1400 problemas-teste do trabalho anterior. Os algoritmos desenvolvidos foram comparados com os da versão anterior e se mostraram superiores na maioria dos casos.

### 3.1.2 UPMSPST

Al-Salem (2004) foram os primeiros a abordar o UPMSPST. É desenvolvido um algoritmo de particionamento, chamado *Partitioning Heuristic* (PH), baseado em três fases, como seguem: *i*) a primeira fase consiste em aplicar uma engenhosa heurística construtiva para alocar as tarefas às máquinas, tendo por base um valor dado pela soma do tempo de processamento de uma tarefa em uma máquina com a média dos tempos de preparação dessa tarefa nessa máquina. *ii*) a segunda fase é uma heurística de refinamento que utiliza movimentos de remoção das tarefas nas máquinas mais sobrecarregadas e as inserem nas máquinas menos sobrecarregadas e *iii*) a terceira e última fase consiste em aplicar uma heurística baseada no problema do caixeiro viajante assimétrico (TSP) para encontrar a melhor sequência das tarefas em cada uma das máquinas. O autor compara os resultados do algoritmo PH com os valores dos limites inferiores projetados para os problemas-teste testados e conclui que os valores das melhores soluções do algoritmo distam, no máximo, 11% dos limites inferiores.

Em (Rabadi et al., 2006) é desenvolvido um algoritmo baseado na metaheurística

Meta-RaPS (*Metaheuristic for Randomized Priority Search*). Essa metaheurística combina procedimentos de construção e refinamento, utilizando ideias próximas do *Greedy Randomized Adaptive Search Procedures* – GRASP (Feo e Resende, 1995). A construção é feita tal como na fase de construção do algoritmo GRASP, tendo-se como função guia para definir o sequenciamento, o tempo de processamento da tarefa candidata somado ao tempo de preparação desta tarefa na máquina que estiver menos sobrecarregada. Já o refinamento realizado utiliza três buscas locais baseadas em movimentos de inserção, trocas na mesma máquina e trocas entre máquinas diferentes. É também desenvolvida uma formulação de programação matemática para o problema e disponibilizados problemas-teste em (SchedulingResearch, 2005) para o UPMSPST.

No trabalho (Helal et al., 2006) é implementado um algoritmo Busca Tabu para a resolução do UPMSPST. Esse algoritmo utiliza duas fases de busca local, sendo estas: 1) intra-máquina, que otimiza a sequência de tarefas sobre as máquinas; e 2) inter-máquina, que equilibra a atribuição das tarefas nas máquinas. Para realizar os experimentos foram utilizados os problemas-teste de (SchedulingResearch, 2005). Os resultados encontrados mostraram o bom desempenho do algoritmo desenvolvido.

Em (Arnaout et al., 2010) é proposto um algoritmo de *Ant Colony Optimization*(Colônia de Formigas) para a resolução do UPMSPST. Nesse trabalho foram considerados problemas-teste onde a proporção do número de tarefas e do número de máquinas é grande. O problema foi resolvido em dois estágios. O primeiro consistia na atribuição das tarefas e o segundo no sequenciamento das tarefas. Foram realizadas buscas locais baseadas em movimentos de múltipla inserção, trocas na mesma máquina e trocas em máquinas diferentes. Nos experimentos foram utilizados os problemas-teste de (SchedulingResearch, 2005). O algoritmo Colônia de Formigas obteve um melhor desempenho que outros da literatura no subconjunto de problemas-teste considerado.

No trabalho (Ying et al., 2010) é implementado um algoritmo *Restricted Simulated Annealing* para resolver o UPMSPST. O algoritmo desenvolvido utiliza uma estratégia de busca restrita, que tem por objetivo eliminar movimentos ineficientes das tarefas. A solução inicial é gerada de forma aleatória. O *Restricted Simulated Annealing* utiliza movimentos de inserções e trocas de tarefas, mas o movimento é realizado somente se houver uma melhora na solução corrente. Os experimentos foram realizados utilizando os problemas-teste de (SchedulingResearch, 2005). O algoritmo desenvolvido conseguiu melhorar as soluções da literatura em problemas-teste de maior porte.

Chang e Chen (2011) implementaram um Algoritmo Genético e um *Simulated Anne-*

*aling*, além de realizarem um estudo sobre as propriedades do UPMSPST. Estas propriedades desenvolvidas utilizam movimentos inter-máquinas e intra-máquinas. Ao aplicar essas propriedades chega-se a uma solução próxima da solução ótima. Os experimentos mostraram a eficiência do Algoritmo Genético, o qual conseguiu melhorar as soluções da literatura em problemas-teste maiores, utilizando como *benchmark* os problemas-teste de (SchedulingResearch, 2005).

Em (Fleszar et al., 2011) os autores propõem um algoritmo híbrido, que combina os procedimentos *Variable Neigborhood Descendent* - VND e *Multi-start* com programação matemática. A solução inicial é gerada de forma parcialmente aleatória. A seguir, a solução é refinada por buscas locais baseadas em movimentos de inserção e trocas. O modelo de programação matemática é responsável por definir qual o outro tipo de movimento será aplicado para melhorar a solução. Nos experimentos também foram utilizados os problemas-teste de (SchedulingResearch, 2005). Esse algoritmo foi capaz de resolver problemas-teste de grande porte com qualidade em pouco tempo.

Em (Vallada e Ruiz, 2011) são desenvolvidos Algoritmos Genéticos (Goldberg, 1989; Holland, 1975) para o UPMSPST, cuja principal característica é realizar cruzamentos com buscas locais "limitadas". O primeiro elemento da população é criado utilizando uma heurística de múltipla inserção e os demais indíviduos são gerados aleatoriamente. A seleção é feita por torneios $n$-ários. As buscas locais "limitadas" são baseadas em movimentos de múltipla inserção. Foram definidos dois algoritmos genéticos, denominados GA1 e GA2, sendo que ambos se diferenciam entre si pelos parâmetros. Foi disponibilizado um novo conjunto de problemas-teste em (SOA, 2011). É importante citar que para todos os problemas-teste de (SOA, 2011) não são considerados os tempos de preparação para as primeiras tarefas alocadas nas máquinas, se tornando uma característica particular do UPMSPST. A partir destes problemas-teste foram executados os experimentos computacionais e o algoritmo GA2 obteve o melhor desempenho. Os autores também desenvolveram uma nova formulação de programação matemática para o UPMSPST baseada na proposta por (Rabadi et al., 2006).

Em (Haddad et al., 2011), os autores desenvolveram dois algoritmos genéticos: AGVL e AGCR. No método AGVL, os cruzamentos e mutações são realizados por meio de operações convencionais envolvendo vetores de listas de inteiros, enquanto no algoritmo AGCR, essas operações envolvem vetores de chaves aleatórias. Os autores testaram esses algoritmos em problemas-teste de pequeno porte (envolvendo até 5 máquinas e 12 tarefas). Nesses problemas-teste, o algoritmo AGCR superou os algoritmos GA1 e GA2 de (Vallada e Ruiz, 2011).

Em (Haddad et al., 2012) foi desenvolvido um algoritmo híbrido denominado GIVMP. O algoritmo usa um procedimento de construção gulosa e aletória para gerar uma solução inicial, o ILS como metaheurística para guiar o processo de busca, o procedimento *Random Variable Neighborhood Descent* - RVND (Souza et al., 2010) para realizar as buscas locais, e o *Path Relinking* - PR (Glover, 1996) como técnica de intensificação e diversificação. Ao final é feita uma pós-otimização usando um módulo de programação matemática, baseado no problema do caixeiro viajante assimétrico, para atuar como busca local.

Em (Haddad, 2012), foram propostos três algoritmos baseados em *Iterated Local Search* - ILS (Lourenço et al., 2003) e *Variable Neighborhood Descent* - VND (Hansen et al., 2008): IVP, GIVP e GIVMP. Desses, o algoritmo GIVMP, publicado em (Haddad et al., 2012), foi o de melhor desempenho.

Este trabalho aperfeiçoa os algoritmos de (Haddad, 2012). O aperfeiçoamento consiste no desenvolvimento de novos procedimentos construtivos, novos procedimentos para buscas locais, novas perturbações, e novos procedimentos para intensificação e diversificação da busca. Também é proposto um novo algoritmo híbrido, que combina técnicas heurísticas com programação matemática.

## 3.2 Formulações de Programação Matemática

São apresentadas a seguir quatro formulações de programação matemática para o problema em estudo. As duas primeiras são formulações encontradas na literatura, enquanto as duas últimas são formulações propostas neste trabalho. Para a clareza de entendimento a notação utilizada é reapresentada em todas as subseções.

### 3.2.1 Formulação UPMSPST-VR

Vallada e Ruiz (2011) propuseram um modelo de programação linear inteira mista (PLIM) para resolver o UPMSPST, aqui nomeado UPMSPST-VR, baseado na formulação de (Guinet, 1993). As seguintes notações são utilizadas por este modelo:

- $M = \{1, ..., m\}$: conjunto de máquinas, sendo $m$ o número de máquinas;
- $N = \{1, ..., n\}$: conjunto de tarefas, com $n$ representando o número de tarefas;

- $N_0 = N \cup \{0\}$: conjunto de tarefas com a adição da tarefa 0 (fictícia);
- $p_{ij}$: tempo de processamento da tarefa $j$ na máquina $i$;
- $S_{ijk}$: tempo de preparação necessário para poder processar a tarefa $k$ se ela for sequenciada após a tarefa $j$ na máquina $i$;
- $B$: constante suficientemente grande

As variáveis de decisão utilizadas pelo modelo são:

- $x_{ijk}$: 1 se a tarefa $j$ antecede imediatamente a tarefa $k$ na máquina $i$ e 0 caso contrário;
- $C_{ij}$: tempo de conclusão da tarefa $j$ na máquina $i$;
- $C_{\max}$: tempo máximo de conclusão do sequenciamento.

O modelo PLIM é apresentado pelas equações (3.1) a (3.10):

$$\min \quad C_{\max} \tag{3.1}$$

$$s.a.$$

$$\sum_{i=1}^{m} \sum_{\substack{j=0 \\ j \neq k}}^{n} x_{ijk} = 1 \qquad \forall k \in N \tag{3.2}$$

$$\sum_{i=1}^{m} \sum_{\substack{k=1 \\ j \neq k}}^{n} x_{ijk} \leq 1 \qquad \forall j \in N \tag{3.3}$$

$$\sum_{k=1}^{n} x_{i0k} \leq 1 \qquad \forall\, i \in M \tag{3.4}$$

$$\sum_{\substack{h=0 \\ h \neq k \\ h \neq j}}^{n} x_{ihj} \geq x_{ijk} \qquad \forall j,k \in N, j \neq k, \forall i \in M \tag{3.5}$$

$$C_{ik} + B\,(1 - x_{ijk}) \geq C_{ij} + S_{ijk} + p_{ik} \quad \forall j \in N_0, \forall k \in N, j \neq k, \forall i \in M \tag{3.6}$$

$$C_{i0} = 0 \qquad \forall i \in M \tag{3.7}$$

$$C_{ij} \geq 0 \qquad \forall j \in N, \forall i \in M \tag{3.8}$$

$$C_{\max} \geq C_{ij} \qquad \forall j \in N, \forall i \in M \tag{3.9}$$

$$x_{ijk} \in \{0,\, 1\} \quad \forall j \in N_0, \forall k \in N, j \neq k, \forall i \in M \tag{3.10}$$

O objetivo (3.1) é minimizar o *makespan*. As restrições (3.2) garantem que cada tarefa é alocada a apenas uma máquina e possui apenas uma predecessora. Com as restrições (3.3), limita-se a 1 o número máximo de tarefas sucessoras de uma dada tarefa. Igualmente, pelas restrições (3.4), limita-se a 1 o número máximo de sucessoras das tarefas fictícias em cada máquina. As restrições (3.5) garantem que se uma tarefa $j$ é imediatamente predecessora de uma tarefa $k$, deve existir uma tarefa imediatamente predecessora de $j$ na mesma máquina. Já as restrições (3.6) servem para controlar os tempos de conclusão das tarefas nas máquinas. Se a tarefa $k$ está alocada imediatamente após a tarefa $j$ na máquina $i$ ($x_{ijk} = 1$), o tempo de conclusão relacionado, $C_{ik}$, deve ser maior ou igual ao tempo de conclusão da tarefa $j$, $C_{ij}$, somado ao tempo de preparação entre $j$ e $k$ e o tempo de processamento de $k$. Caso $x_{ijk} = 0$, a constante $B$ fará com que essas restrições sejam redundantes. As restrições (3.7) e (3.8) definem os tempos de conclusão como 0 para as tarefas fictícias e não negativas para as tarefas regulares, respectivamente. As restrições (3.9) definem o tempo máximo de conclusão (*makespan*). Finalmente, as restrições (3.10) definem as variáveis como binárias.

### 3.2.2 Formulação UPMSPST-RA

A seguir é apresentado o modelo de programação linear inteira mista (PLIM) para resolver o UPMSPST proposto em Rabadi et al. (2006), aqui nomeado UPMSPST-RA. Este modelo é uma adaptação do de (Guinet, 1991) e considera as seguintes notações:

- $M = \{1, ..., m\}$: conjunto de máquinas, sendo $m$ o número de máquinas;
- $N = \{1, ..., n\}$: conjunto de tarefas, com $n$ representando o número de tarefas;
- $N_0 = N \cup \{0\}$: conjunto de tarefas com a adição da tarefa 0 (fictícia);
- $p_{jk}$: tempo de processamento da tarefa $j$ na máquina $k$;
- $S_{ijk}$: tempo de preparação necessário para poder processar a tarefa $j$ se ela for sequenciada após a tarefa $i$ na máquina $k$;
- $S_{0jk}$: tempo de preparação necessário para poder processar a tarefa $j$ se ela for a primeira tarefa a ser processada na máquina $k$;
- $B$: constante suficientemente grande

As seguintes variáveis de decisão são utilizadas no modelo:

- $x_{ijk}$: 1 se a tarefa $j$ está sequenciada após a tarefa $i$ na máquina $k$ e 0 caso contrário;
- $x_{0jk}$: 1 se a tarefa $j$ for a primeira a ser processada na máquina $k$ e 0 caso contrário;
- $x_{i0k}$: 1 se a tarefa $i$ for a última a ser processada na máquina $k$ e 0 caso contrário;
- $C_j$: tempo de conclusão da tarefa $j$;
- $C_{\max} = max_{j \in N}\, C_j$: é o *makespan* da solução

O modelo PLIM é apresentado pelas equações (3.11) a (3.17):

$$\min \qquad C_{\max} \tag{3.11}$$

$$s.a.$$

$$0 \leq C_j \leq C_{\max} \qquad \forall j \in N_0 \tag{3.12}$$

$$\sum_{\substack{i=0 \\ i \neq j}}^{n} \sum_{k=1}^{m} x_{ijk} = 1 \qquad \forall j \in N \tag{3.13}$$

$$\sum_{\substack{i=0 \\ i \neq h}}^{n} x_{ihk} = \sum_{\substack{j=0 \\ j \neq h}}^{n} x_{hjk} \qquad \forall h \in N,\ \forall k \in M \tag{3.14}$$

$$C_i + \sum_{k=1}^{m} (S_{ijk} + p_{jk}) x_{ijk} + B \left( \sum_{k=1}^{m} x_{ijk} - 1 \right) \leq C_j \qquad \forall i \in N_0,\ \forall j \in N \tag{3.15}$$

$$\sum_{j=0}^{n} x_{0jk} = 1 \qquad \forall\ k \in M \tag{3.16}$$

$$x_{ijk} \in \{0,\ 1\} \qquad \forall\ i, j \in N_0, \forall k \in M \tag{3.17}$$

O objetivo (3.11) visa a minimização do *makespan*. As restrições (3.12) garantem que o *makespan* seja o tempo máximo de conclusão das tarefas. As restrições (3.13) asseguram que cada tarefa será alocada apenas uma vez e processada por apenas uma máquina. As restrições (3.14) garantem que para cada tarefa e para cada máquina, o número de predecessores da tarefa é igual ao número de sucessores da mesma. As restrições (3.15) são utilizadas para calcular os tempos de conclusão de cada tarefa e ainda assegurar que uma tarefa não possa preceder e suceder a mesma tarefa, ou seja, impede a criação de ciclos de processamento. As restrições (3.16) garantem que exatamente uma tarefa está sequenciada como a primeira tarefa de cada máquina. Finalmente, as restrições (3.17) definem os domínios das variáveis.

### 3.2.3 Formulação UPMSPST-CV

Esta formulação, proposta neste trabalho e nomeada UPMSPST-CV, é uma extensão daquela de (Vallada e Ruiz, 2011). A extensão consiste em incluir uma variável de decisão e restrições do problema do caixeiro viajante assimétrico, conforme proposta de (Sarin et al., 2005), na formulação de Vallada e Ruiz (2011). São utilizadas as seguintes notações:

- $M = \{1, ..., m\}$: conjunto de máquinas, sendo $m$ o número de máquinas;
- $N = \{1, ..., n\}$: conjunto de tarefas, com $n$ representando o número de tarefas;
- $N_0 = N \cup \{0\}$: conjunto de tarefas com a adição da tarefa 0 (fictícia);
- $p_{ij}$: tempo de processamento da tarefa $j$ na máquina $i$;
- $S_{ijk}$: tempo de preparação necessário para poder processar a tarefa $k$ se ela for sequenciada após a tarefa $j$ na máquina $i$;
- $B$: constante suficientemente grande

As variáveis de decisão utilizadas pelo modelo são:

- $y_{ijk}$: 1 se a tarefa $j$ antecede, imediatamente ou não, a tarefa $k$ na máquina $i$, 0 caso contrário;
- $x_{ijk}$: 1 se a tarefa $j$ antecede imediatamente a tarefa $k$ na máquina $i$ e 0 caso contrário;
- $C_{ij}$: tempo de conclusão da tarefa $j$ na máquina $i$;
- $C_{\max}$: tempo máximo de conclusão do sequenciamento.

O modelo PLIM é apresentado pelas equações (3.18) a (3.30):

$$\min \quad C_{\max} \tag{3.18}$$

$$s.a.$$

$$\sum_{i=1}^{m} \sum_{j=0, j\neq k}^{n} x_{ijk} = 1 \qquad \forall k \in N \tag{3.19}$$

$$\sum_{i=1}^{m} \sum_{k=1, j\neq k}^{n} x_{ijk} \leq 1 \qquad \forall j \in N \tag{3.20}$$

$$\sum_{k=1}^{n} x_{i0k} \leq 1 \qquad \forall\, i \in M \tag{3.21}$$

$$\sum_{h=0, h\neq k, h\neq j}^{n} x_{ihj} \geq x_{ijk} \qquad \forall j,k \in N, j \neq k, \forall i \in M \tag{3.22}$$

$$C_{ik} + B\left(1 - x_{ijk}\right) \geq C_{ij} + S_{ijk} + p_{ik} \quad \forall j \in N_0, \forall k \in N, j \neq k, \forall i \in M \tag{3.23}$$

$$C_{i0} = 0 \qquad \forall i \in M \tag{3.24}$$

$$C_{ij} \geq 0 \qquad \forall j \in N, \forall i \in M \tag{3.25}$$

$$C_{\max} \geq C_{ij} \qquad \forall j \in N, \forall i \in M \tag{3.26}$$

$$y_{ijk} \geq x_{ijk} \qquad \forall i \in M, \forall k, j \in N, j \neq k \tag{3.27}$$

$$y_{ijk} + y_{ikj} = 1 \qquad \forall i \in M, \forall k, j \in N, j \neq k \tag{3.28}$$

$$y_{ijk} + y_{ika} + y_{iaj} \leq 2 \qquad \forall i \in M, \forall k, j, a \in N, j \neq k \neq a \tag{3.29}$$

$$x_{ijk}, y_{ijk} \in \{0,\, 1\} \quad \forall j \in N_0, \forall k \in N, j \neq k, \forall i \in M \tag{3.30}$$

A função objetivo (3.18) visa a minimização do *makespan*. As restrições (3.19) definem que cada tarefa é alocada a apenas uma máquina e possui apenas uma predecessora. Nas restrições (3.20) limita-se a 1 o número máximo de tarefas sucessoras de uma dada tarefa. Com as restrições ((3.21)), limita-se a 1 o número máximo de sucessoras das tarefas fictícias em cada máquina. As restrições (3.22) garantem que se uma tarefa $j$ é imediatamente predecessora de uma tarefa $k$, deve existir uma tarefa imediatamente predecessora de $j$ na mesma máquina. As restrições (3.23) controlam os tempos de conclusão das tarefas nas máquinas. As restrições (3.24) e (3.25) definem os tempos de conclusão como 0 para as tarefas fictícias e não negativas para as tarefas regulares, respectivamente. As restrições (3.26) definem o tempo máximo de conclusão (*makespan*). As restrições (3.27) asseguram que para toda tarefa $j$ que antecede imediatamente uma tarefa $k$ em uma máquina $i$, esta tarefa $j$ também deve anteceder, imediatamente ou não, a tarefa $k$ na mesma máquina $i$. Em (3.28) garante-se que dadas duas tarefas $j$ e

$k$, apenas uma pode anteceder a outra. Já em (3.29), assegura-se que se uma tarefa $j$ antecede uma tarefa $k$ em uma máquina $i$ , e esta tarefa $k$ antecede a uma outra tarefa $a$ na mesma máquina $i$, então a tarefa $a$ não pode anteceder a tarefa $j$ nesta máquina $i$. Finalmente, as restrições (3.30) definem as variáveis como binárias.

### 3.2.4 Formulação UPMSPST-IT

A UPMSPST-IT é outra formulação de programação linear inteira mista (PLIM) indexada no tempo proposta neste trabalho. Ela foi baseada nas formulações de (Rosa et al., 2009) e (Ravetti et al., 2007), que desenvolveram formulações deste tipo para problemas similares ao UPMSPST. As seguintes notações são utilizadas:

- $M = \{1, ..., m\}$: conjunto de máquinas, sendo $m$ o número de máquinas;
- $N = \{1, ..., n\}$: conjunto de tarefas, com $n$ representando o número de tarefas;
- $H = \{1, ..., hl\}$: conjunto de tempos discretizados, sendo $hl$ o tempo máximo;
- $p_{ij}$: tempo de processamento da tarefa $j$ na máquina $i$;
- $S_{ijk}$: tempo de preparação necessário para poder processar a tarefa $k$ se ela for sequenciada após a tarefa $j$ na máquina $i$;

O parâmetro $hl$ é o limite superior do *makespan* para um dado problema-teste. Nos testes realizados com os problemas-teste de (SOA, 2011), o $hl$ foi fixado no valor correspondente à melhor solução disponibilizada para cada problema-teste.

As variáveis de decisão utilizadas pelo modelo são:

- $x_{ijh}$: 1 se a tarefa $j$ está alocada no tempo $h$ na máquina $i$ e 0, caso contrário;
- $C_{ij}$: tempo de conclusão da tarefa $j$ na máquina $i$;
- $C_{\max}$: tempo máximo de conclusão do sequenciamento.

O modelo PLIM indexado no tempo é apresentado pelas equações (3.31) a (3.35):

$$\min \quad C_{\max} \tag{3.31}$$

$$s.a.$$

$$\sum_{i=1}^{m} \sum_{h=0}^{hl-p_{ij}} x_{ijh} = 1 \qquad \forall j \in N \tag{3.32}$$

$$x_{ijh} + \sum_{u=h}^{\min(h+p_{ij}+S_{ijk}-1,hl)} x_{iku} \leq 1 \qquad \forall h \in H, \forall i \in M, \forall j,k \in N, j \neq k \tag{3.33}$$

$$C_j \geq \sum_{i=1}^{m} \sum_{h=0}^{hl-p_{ij}} (h + p_{ij}) * x_{ijh} \qquad \forall j \in N \tag{3.34}$$

$$C_{max} \geq C_j \qquad \forall j \in N \tag{3.35}$$

$$x_{ijh} \in \{0, 1\} \qquad \forall j \in N, \forall i \in M, \forall h \in H \tag{3.36}$$

A função objetivo (3.31) busca a minimização do *makespan*. As restrições de (3.32) definem que todas as tarefas devem ser alocadas a uma máquina e em um determinado tempo. Em (3.33) são garantidas as reservas dos tempos, ou seja, para toda tarefa $k$ alocada a uma máquina $i$ em um tempo $u$, não deverá existir uma tarefa $j$ nessa mesma máquina $i$ em um tempo $h$ que coincida com o tempo que $u$ irá necessitar. Já em (3.34) define-se que o tempo de conclusão de uma tarefa $j$ é o tempo acumulado $h$ em que ela se inicia acrescido do tempo de processamento dessa tarefa. As restrições (3.35) definem o tempo máximo de conclusão. Finalmente, as restrições (3.36) definem que as variáveis são binárias.

### 3.2.5 Comparação entre as formulações

As formulações foram implementadas no otimizador CPLEX, versão 12.6, com seus parâmetros de entrada padrão. Nos testes realizados, observou-se que na formulação UPMSPST-IT, que é indexada no tempo, o número de variáveis criadas cresce muito rapidamente com o tamanho dos problemas-teste, tornando-a não atrativa. Além disso, mesmo nos problemas-teste pequenos, ela não teve um resultado satisfatório quando comparada com todas as outras. A formulação UPMSPST-CV também não surtiu o efeito desejado, apresentando desempenho inferior à das formulações UPMSPST-RA e UPMSPST-VR. O melhor desempenho foi da formulação UPMSPST-VR, sendo ela a escolhida para ser utilizada no algoritmo híbrido proposto nesta dissertação. Contudo, é

importante ressaltar que não foram feitos testes exaustivos para assegurar a supremacia da formulação UPMSPST-VR.

## 3.3 Heurísticas

Heurísticas são técnicas baseadas em procedimentos intuitivos que procuram por uma boa solução em um tempo computacional aceitável, porém não existe garantia de otimalidade e nem de proximidade da solução ótima.

Nas subseções posteriores são apresentadas as heurísticas construtivas e as heurísticas de refinamento.

### 3.3.1 Heurísticas Construtivas

Para inicializar a busca por uma solução adequada em um problema qualquer, precisa-se de uma solução inicial. A técnica responsável por construir uma solução inicial é denominada heurística construtiva. Ela tem por objetivo construir uma solução elemento a elemento. A escolha de cada elemento a ser inserido na solução varia de acordo com uma função de avaliação, a qual depende do problema abordado.

Nas heurísticas clássicas, geralmente os elementos são ordenados segundo uma função gulosa, e a cada passo é inserido o "melhor" elemento segundo um critério previamente definido.

Para exemplificar o funcionamento de heurística construtiva, o pseudocódigo do Algoritmo 3.1 apresenta uma heurística construtiva gulosa que utiliza uma função de avaliação $g(.)$. O elemento $t_{melhor}$ é aquele que possui o melhor valor segundo a função $g(.)$. Caso o problema seja de minimização o $g$ será o menor valor, porém se o problema for de maximização, $g$ será o maior valor.

#### 3.3.1.1 Heuristic-Biased Stochastic Sampling

*Heuristic-Biased Stochastic Sampling* – HBSS é um procedimento heurístico construtivo proposto em (Bresina, 1996). Nessa heurística, os elementos candidatos a pertencerem à solução parcial são escolhidos de acordo com uma probabilidade associada a uma função *bias*. Os candidatos são ordenados de pelo valor da função de avaliação, e a cada

**Algoritmo 3.1**: Heurística Construtiva Gulosa

**Entrada**: $g(.)$
**Saída**: Solução $s$ construída
Inicialize o conjunto $C$ de elementos candidatos;
$s \leftarrow \emptyset$;
**enquanto** $(|C| \neq 0)$ **faça**
    $g(t_{melhor}) = melhor\{g(t) \mid t \in C\}$;
    $s \leftarrow s \cup \{t_{melhor}\}$;
    Atualize o conjunto $C$ de elementos candidatos;
**fim**
**Retorne** $s$;

candidato é associada uma probabilidade, a qual depende da ordem de classificação. A função *bias* influencia diretamente no valor dessas probabilidades.

Na Tabela 3.1 são apresentadas as probabilidades de escolha para várias funções *bias*, de acordo com o número de candidatos. Nesta Tabela são apresentados os resultados relativos às seguintes funções *bias*: Uniforme, Logaritmica, Polinomial de grau 2, Polinomial de grau 3, Polinomial de grau 4 e Exponencial. A primeira coluna representa a ordem de classificação dos candidatos. Cada célula da tabela representa a probabilidade de o candidato $i$ ser escolhido usando a função *bias* $j$. Por exemplo, se a função *bias* escolhida for a Exponencial, o quinto elemento da lista de candidatos terá 1,2% de chance de ser escolhido.

**Tabela 3.1:** Tabela de funções *bias*[1]

| Candidatos | Funções $bias(j)$ | | | | | | |
|---|---|---|---|---|---|---|---|
| $i$ | Unif. | Log. | Linear | Pol.(grau 2) | Pol.(grau 3) | Pol.(grau 4) | Exp. |
| 1 | 0,033 | 0,109 | 0,250 | 0,620 | 0,832 | 0,924 | 0,632 |
| 2 | 0,033 | 0,069 | 0,125 | 0,155 | 0,104 | 0,058 | 0,233 |
| 3 | 0,033 | 0,055 | 0,083 | 0,069 | 0,031 | 0,011 | 0,086 |
| 4 | 0,033 | 0,047 | 0,063 | 0,039 | 0,013 | 0,004 | 0,031 |
| 5 | 0,033 | 0,042 | 0,050 | 0,025 | 0,007 | 0,001 | 0,012 |
| 6 | 0,033 | 0,678 | 0,429 | 0,092 | 0,013 | 0,002 | 0,006 |

[1]Fonte: Adaptado de (Bresina, 1996)

### 3.3.2 Heurísticas de Refinamento

As heurísticas de refinamento são também denominadas buscas locais, as quais têm por objetivo refinar uma solução previamente gerada. O refinamento da solução é realizado por meio de movimentos. Um movimento é uma modificação realizada na solução para gerar uma nova solução diferente, chamada de vizinho dessa solução. O conjunto de movimentos de um mesmo tipo aplicados sobre uma solução define uma vizinhança dessa solução.

Na heurística de refinamento caminha-se, a cada iteração, de vizinho para vizinho de acordo com a vizinhança adotada, até que se chegue a um critério de parada.

As duas principais heurísticas de refinamento são apresentadas a seguir.

#### 3.3.2.1 Método da Descida/Subida

A idéia deste método é partir de uma solução inicial e a cada passo analisar todos os seus vizinhos, e se mover para o melhor vizinho de acordo com a função de avaliação. Ao final o método chega a uma solução ótima local. Pelo fato de analisar todos os vizinhos a cada iteração e escolher o melhor, esta técnica é comumente referenciada na literatura inglesa por *Best Improvement Method*. O método é chamado de descida para problemas de minimização e de subida para problemas de maximização.

O Algoritmo 3.2 apresenta um pseudocódigo do método de descida para um problema de minimização a partir de uma solução inicial $s$ com uma função de avaliação $f$ e considerando a busca em uma dada vizinhança $N(.)$.

**Algoritmo 3.2**: Método da Descida

**Entrada**: $(f(.), N(.), s)$
**Saída**: Solução $s$ refinada

```
1 V ← {s' ∈ N(s) | f(s') < f(s)};
2 enquanto (|V| > 0) faça
3 |   Selecione s' ∈ V, onde s' = arg min{f(s') | s' ∈ V};
4 |   s ← s';
5 |   V ← {s' ∈ N(s) | f(s') < f(s)};
6 fim
7 retorna s;
```

#### 3.3.2.2 Método de Primeira Melhora

O método de subida/descida faz uma pesquisa exaustiva por todos os vizinhos a cada iteração. O método de Primeira Melhora, ao contrário, evita esta grande exploração da vizinhança, sendo também chamado de *First Improviment Method*. Este método interrompe a exploração dos vizinhos assim que um vizinho melhor é encontrado. Somente no pior caso todos os vizinhos são pesquisados. No entanto, assim como no método de subida/descida, o algoritmo fica preso no primeiro ótimo local encontrado.

## 3.4 Metaheurísticas

Diferentemente das heurísticas clássicas, as metaheurísticas são procedimentos de carácter geral que tem mecanismos que evitam a parada prematura em ótimos locais ainda distantes do ótimo global. A forma como isso é feito varia de metaheurística para metaheurística.

A seguir é feita uma breve revisão das metaheurísticas utilizadas neste trabalho.

### 3.4.1 Greedy Randomized Adaptive Search Procedures

A metaheurística *Greedy Randomized Adaptive Search* – GRASP (Feo et al., 1994) consiste em duas fases: *i*) fase de construção, na qual é gerada uma solução, elemento a elemento; *ii*) fase de refinamento, onde é aplicada uma busca local na solução gerada. A melhor solução encontrada ao longo das iterações do GRASP é retornada como resultado.

Um pseudocódigo do GRASP é apresentado no Algoritmo 3.3. Ele recebe como parâmetro uma função de avaliação $f$, uma função adaptativa $g$ responsável por definir o quanto de melhora cada elemento trará para a solução no processo de construtivo, um conjunto de vizinhanças $V$ que será utilizada pela busca local e um valor $\alpha$ que definirá o quão aleatória serão as escolhas no método construtivo. Até que um critério de parada seja satisfeito, a cada iteração uma solução $s$ recebe uma solução proveniente da fase de construção, e a submete a uma busca local. Após a busca local, a solução $s$ é avaliada. Se ela for a melhor solução até o momento, ela é armazenada; do contrário, ela é descartada. Ao final, o GRASP retorna a melhor solução encontrada.

**Algoritmo 3.3**: *Greedy Randomized Adaptive Search*

**Entrada**: $f(.)$, $g(.)$, $V(.)$, $\alpha$
**Saída**: Solução $s$ refinada
1 **enquanto** *(critério de parada não satisfeito)* **faça**
2 | $s \leftarrow$ *Construtivo*$(g(.), \alpha)$;
3 | $s' \leftarrow$ *BuscaLocal*$(f(.), V(.), s)$;
4 | *AtualizaMelhor*$(s', s^\star)$ ; `/* s* é a melhor solução encontrada */`
5 **fim**
6 **Retorne** $s^\star$;

### 3.4.2 Iterated Local Search

A metaheurística *Iterated Local Search* – ILS (Lourenço et al., 2003) é um procedimento de busca local onde são geradas novas soluções de partida obtidas por meio de perturbações na solução ótima local.

O Algoritmo 3.4 apresenta o pseudocódigo do ILS. Ele recebe como parâmetro a função de avaliação. Na linha 1 é gerada uma solução inicial $s_0$. Essa solução passa por uma busca local na linha 2. Um laço de iterações inicia-se até que um critério de parada seja satisfeito. Dentro deste laço, na linha 4 uma solução $s'$ recebe uma solução gerada pela perturbação na solução $s$. A perturbação consiste em uma espécie de mutação ou modificação na solução. Na linha 5 a solução $s'$ passa por uma busca local e a solução gerada é armazenada na solução $s''$. Já na linha 6 a solução $s''$ passa por um critério de aceitação em relação a solução $s$. Esse critério define se esse ótimo local será aceito. O histório é avaliado na perturbação para definir o grau de perturbação aplicado, e também no critério de aceitação para avaliar se compensa aceitar a solução gerada. Ao final do laço é retornada a melhor solução encontrada durante a busca.

**Algoritmo 3.4**: *Iterated Local Search*

**Entrada**: função de avaliação
**Saída**: Solução $s$ refinada
1 $s_0 \leftarrow$ *GeraSolucaoInicial*();
2 $s \leftarrow$ *BuscaLocal*$(s_0)$;
3 **enquanto** *(critério de parada não satisfeito)* **faça**
4 | $s' \leftarrow$ *Perturbacao*$(s, histórico)$;
5 | $s'' \leftarrow$ *BuscaLocal*$(s')$;
6 | $s \leftarrow$ *CriterioAceitacao*$(s, s'', histórico)$;
7 **fim**
8 **Retorne** $s$;

### 3.4.3 Variable Neighborhood Descent

O método *Variable Neighborhood Descent* – VND (Hansen et al., 2008) consiste em realizar um refinamento por meio de trocas sistemáticas de vizinhança no espaço de soluções. Quando uma solução melhor que a corrente é encontrada, ela é armazenada, retornando-se à primeira estrutura de vizinhança. O método termina quando não há melhora da solução corrente em nenhuma das vizinhanças exploradas.

De acordo com os autores de (Hansen et al., 2008), o método VND se baseia em três princípios básicos:

- Um ótimo local com relação à uma estrutura de vizinhança não é necessariamente um ótimo local relativo à outra estrutura de vizinhança;
- Um ótimo global é um ótimo local com relação para todas as estruturas de vizinhanças;
- Para muitos problemas, ótimos locais com relação à uma ou mais estruturas de vizinhanças são relativamente próximos.

**Algoritmo 3.5**: *Variable Neighborhood Descent*

**Entrada**: $f(.), V(.), r, s$
**Saída**: Solução $s$ refinada

```
1  k ← 1 ;                                  /* Estrutura de vizinhança corrente */
2  enquanto (k ≤ r) faça
3  |  Encontre o melhor vizinho s' ∈ V^(k)(s);
4  |  se (f(s') < f(s)) então
5  |  |  s ← s';
6  |  |  k ← 1
7  |  fim
8  |  senão
9  |  |  k ← k + 1
10 |  fim
11 fim
12 Retorne s;
```

## 3.5 Path Relinking

O método *Path Relinking* – PR foi proposto em (Glover, 1996). O PR é uma técnica de intensificação e diversificação da busca. Este método se baseia na exploração de caminhos no espaço de soluções partindo de uma ou mais soluções do conjunto elite e levando a outras soluções elite. O conjunto elite é um conjunto formado por boas soluções já encontradas. Para efetuar estes caminhos são escolhidos movimentos que possam levar atributos da solução guia para a solução corrente. Os atributos são características da solução, que variam de acordo com o problema abordado.

O Algoritmo 3.6 apresenta um pseudocódigo do PR. Ele recebe como parâmetros de entrada uma solução base $s_{base}$ e uma solução guia $s_{guia}$. Inicialmente todos os atributos da solução $s_{guia}$ são armazenados em $\delta$. Existe um laço de iterações até que o conjunto $\delta$ não esteja vazio. Dentro desse laço o melhor atributo da solução $s_{guia}$ é inserido na solução $s_{base}$, e a melhor solução até o momento é armazenada. Ao final do laço, é retornada a melhor solução encontrada no caminho percorrido entre a solução $s_{guia}$ e a solução $s_{base}$.

**Algoritmo 3.6**: *Path Relinking*

**Entrada**: $s_{base}$, $s_{guia}$, $f()$
**Saída**: Melhor solução $s^{\star}$ encontrada

```
1  Inicialize Δ com todos os atributos de s_guia;
2  s' ← arg min{f(s_base), f(s_guia)};
3  x ← s_base;
4  enquanto (|Δ| > 1) faça
5  |  l* ← arg min{f(x ⊕ l) : l ∈ Δ};
6  |  Δ ← Δ \ {l*};
7  |  x ← x ⊕ l*;
8  |  AtualizaMelhor(x, s') ;          /* s' é a melhor solução encontrada */
9  fim
10 s* ← BuscaLocal(s');
11 Retorne s*;
```

Segundo (Rossetti, 2003), o PR pode ser aplicado de duas formas: *i*) como estratégia de intensificação a cada ótimo local; *ii*) como pós-otimização entre todos os pares de solução pertencentes ao conjunto elite. A aplicação como estratégia de intensificação se mostrou mais eficiente de acordo com esta autora.

# Capítulo 4

# Metodologia

## 4.1 Representação da Solução

Uma solução $s$ para o problema UPMSPST é representada por um vetor de inteiros com $m$ posições, onde cada posição representa uma máquina e a esta máquina está associada uma lista contendo as tarefas a ela alocadas.

A Figura 4.1 ilustra a representação da solução para a Figura 2.1. Nesta Figura as tarefas 3, 5 e 1 estão alocadas na máquina 1, enquanto as demais à máquina 2.

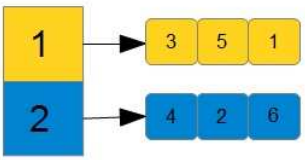



**Figura 4.1:** Exemplo de uma representação da solução

## 4.2 Avaliação de uma Solução

O valor de avaliação de uma solução $s$, denominado *makespan*, é o tempo de conclusão das tarefas, no caso, o tempo de conclusão da máquina que termina de executar suas tarefas por último.

## 4.3 Algoritmos Propostos

Nesta seção são apresentados os quatro algoritmos propostos para a resolução do UPMSPST. Nas subseções 4.3.1, 4.3.2 e 4.3.3 são apresentados os algoritmos heurísticos HIVP, GIAP e AIRP, respectivamente, enquanto na subseção 4.3.4 é apresentado o algoritmo híbrido AIRMP.

### 4.3.1 Algoritmo Heurístico HIVP

O algoritmo heurístico HIVP combina quatro procedimentos para explorar o espaço de soluções: 1) O procedimento parcialmente guloso baseado na heurística HBSS, nomeado $CPG_{\underline{A}SPT}^{HBSS}$; 2) O procedimento de refinamento baseado na metaheurística *Iterated Local Search*, nomeado $\underline{I}LS$; 3) o procedimento de busca local *Random Variable Neighborhood Descent*, nomeado $R\underline{V}ND$; e 4) O procedimento *Path Relinking*, nomeado $Bk\underline{P}R_{AT}$, para intensificar e diversificar a busca.

O procedimento parcialmente guloso $CPG_{ASPT}^{HBSS}$ (ver subseção 4.4.3) é usado para gerar a solução inicial. O ILS é usado para guiar a busca tendo o procedimento *RVND* (ver subseção 4.4.8) para fazer as buscas locais e o procedimento $Perturb_{ILS}$ (ver subseção 4.4.7.2) para perturbar os ótimos locais gerados pelo *RVND*. Já o procedimento $BkPR_{AT}$ (ver subseção 4.4.13) é usado para fazer um balanço entre intensificação e diversificação da busca.

O Algoritmo 4.1 apresenta o pseudocódigo do HIVP.

O HIVP possui cinco parâmetros de entrada: *i*) $vezesNivel$, que representa o número de vezes de execução para cada nível de perturbação; *ii*) $tempoExec$, tempo limite em milisegundos para execução do algoritmo; *iii*) $bias$, utilizado pelo procedimento construtivo para guiar a escolha do elemento candidato a ser inserido na solução parcial *iv*) $\beta$, utilizado no procedimento *RVND* para limitar a quantidade de máquinas que passa pela busca local $BI_{TMM}$ (ver subseção 4.4.6.3) e *v*) $nivelMax$, que representa o número máximo de perturbações simultâneas.

Na linha 1, a contabilização do tempo de execução é iniciada. Depois, na linha 2, são criadas 3 soluções para o problema, como segue: *i*) $s$, a solução corrente; *ii*) $s'$, a solução modificada; *iii*) $melhorSol$, que armazena a melhor solução. Na linha 3 o conjunto elite é iniciado (ver subseção 4.4.11). Na linha 4 a solução $s$ recebe a solução inicial criada

**Algoritmo 4.1**: HIVP

```
   entrada  : vezesNivel, tempoExec, bias, β, nivelMax
   saida    : melhorSol
 1 tempoAtual ← 0;
 2 Solução s, s', melhorSol;
 3 elite ← {};
 4 s ← CPG^{HBSS}_{ASPT}(bias) ;                                  /* ver subseção 4.4.3 */
 5 s ← RVND(s, β) ;                                               /* ver subseção 4.4.8 */
 6 melhorSol ← s;
 7 elite ← elite ∪ {melhorSol};
 8 nivel ← 1;
 9 Atualiza tempoAtual;
10 enquanto tempoAtual ≤ tempoExec faça
11     s' ← s;
12     vezes ← 0;
13     perturbMax ← nivel + 1;
14     enquanto vezes < vezesNivel faça
15         perturb ← 0;
16         s' ← s;
17         enquanto perturb < perturbMax faça
18             perturb ++;
19             s' ← Perturb_ILS(s') ;                            /* ver subseção 4.4.7.2 */
20         fim
21         s' ← RVND(s', β) ;                                    /* ver subseção 4.4.8 */
22         elite ← atualiza(s');
23         pr ← aleatorio(0,1);
24         se pr ≤ 0.05 e |elite| ≥ 5 então
25             el ← aleatorio(1,5);
26             se f(elite[el]) < f(s') então
27                 s' ← BkPR_AT(elite [el], s') ;                /* ver subseção 4.4.13 */
28             fim
29             senão
30                 se f(elite[el]) > f(s') então
31                     s' ← BkPR_AT(s', elite [el]) ;            /* ver subseção 4.4.13 */
32                 fim
33             fim
34         fim
35         se f(s') < f(s) então
36             s ← s';
37             vezes ← 0;
38             atualizaMelhor(s, melhorSol);
39             elite ← atualiza(s);
40         fim
41         vezes ++;
42         Atualiza tempoAtual;
43     fim
44     nivel ++;
45     se nivel ≥ nivelMax então
46         nivel ← 1;
47     fim
48 fim
49 retorne melhorSol ;
```

pelo procedimento parcialmente guloso $CPG^{HBSS}_{ASPT}$. A seguir, a solução $s$ passa por uma busca local utilizando o procedimento *RVND* e a variável *melhorSol* recebe a solução $s$ resultante da busca local. Na linha 7 a solução *melhorSol* é inserida no conjunto elite.

O nível de perturbação é inicializado com valor 1 na linha 8, e o tempo de execução é atualizado na linha 9. O processo iterativo do HIVP inicia-se na linha 10 e termina na linha 48. Este processo iterativo é interrompido quando o limite de tempo é excedido.

Nas linhas 12 e 13 são inicializadas a variável que controla o número de vezes em que cada nível de perturbação (*nivel*) é aplicado, assim como o nível máximo de perturbações ($perturbMax$). O laço entre as linhas 14 e 43 é responsável por controlar o número de vezes em cada nível de perturbação.

Entre as linhas 17 e 20 são feitas as perturbações na solução corrente. Na linha 21, após a perturbação, a solução corrente passa pela busca local *RVND*. Na linha 22 o conjunto elite é atualizado.

Na linha 23 é gerado um número real aleatório entre 0 e 1, que representará a probabilidade de se aplicar o procedimento $BkPR_{AT}$. Esse procedimento é aplicado com uma probabilidade de 5%, desde que o conjunto elite esteja completo, isto é, com 5 elementos. São passados como parâmetros a esse procedimento a solução corrente e uma solução escolhida de maneira aleatória dentro do conjunto elite. A solução de menor *makespan* é a solução base (primeiro parâmetro do procedimento $BkPR_{AT}$) e a de maior *makespan* a solução guia (como segundo parâmetro).

Entre as linhas 35 e 40 é analisado se as alterações feitas na solução corrente $s'$ foram boas o suficiente para continuar a busca a partir dela. Ao fim do tempo de execução, a variável *melhorSol* guarda a melhor solução encontrada. Quando o nível de perturbação é maior ou igual a 4 (Linha 45), ele é reiniciado com 1.

### 4.3.2 Algoritmo Heurístico GIAP

O algoritmo heurístico GIAP tem o funcionamento similar ao do algoritmo heurístico HIVP apresentado na subseção 4.3.1, diferenciando-se dele apenas na utilização de dois procedimentos: o da construção de uma solução inicial e o da busca local.

No algoritmo GIAP é utilizado o procedimento parcialmente guloso $CPG_{\underline{G}RASP}$ (ver subseção 4.4.4 a seguir) para gerar a solução inicial. Já as buscas locais são realizadas pelo procedimento $\underline{A}LS$ (ver subseção 4.4.10 a seguir).

Os demais procedimentos do algoritmo GIAP são os mesmos do algoritmo HIVP, ou seja, as perturbações do ILS são realizadas utilizando o procedimento $Perturb_{\underline{I}LS}$ (ver subseção 4.4.7.2), e as intensificações e diversificações da busca são feitas pelo procedimento $Bk\underline{P}R_{AT}$ (ver subseção 4.4.13).

O Algoritmo 4.2 apresenta o pseudocódigo do GIAP. Ele possui oito parâmetros de entrada: *i*) $vezesNivel$, que representa o número de vezes de execução para cada nível de perturbação; *ii*) $tempoExec$, tempo limite em milisegundos para execução do algoritmo; *iii*) $iterMax$, utilizado pelo procedimento de busca local para controlar o número máximo de iterações; *iv*) $iterAtualizaProb$, utilizado no procedimento de busca local para definir o momento em que as probabilidades dos métodos são atualizadas; *v*) $\alpha$, define o valor de aleatoriedade na escolha de candidados para o procedimento construtivo; *vi*) $tempoLimite$, tempo limite em milisegundos para execução do procedimento construtivo; *vii*) $\beta$, utilizado no procedimento $ALS$ para limitar a quantidade de máquinas que passa pela busca local $BI_{TMM}$ (ver subseção 4.4.6.3); e *viii*) $nivelMax$, que representa o número máximo de perturbações simultâneas.

### 4.3.3 Algoritmo Heurístico AIRP

O algoritmo heurístico AIRP também tem o funcionamento similar ao do algoritmo heurístico HIVP apresentado na subseção 4.3.1, diferenciando-se dele apenas na utilização dos procedimentos de construção, busca local e *Path Relinking*.

No algoritmo AIRP é utilizado o procedimento construtivo guloso $\underline{A}daptive\ Shortest\ Processing\ Time - CG_{\underline{A}SPT}$ (ver subseção 4.4.1 a seguir) para gerar a solução inicial. As buscas locais são realizadas por meio do procedimento $\underline{R}VI$ (ver subseção 4.4.9). O procedimento $Bk\underline{P}R_{AM}$ (ver subseção 4.4.12) é utilizado para intensificar e diversificar a busca.

Assim como no algoritmo HIVP, as perturbações do $\underline{I}LS$ são realizadas utilizando-se o procedimento $Perturb_{\underline{I}LS}$ (ver subseção 4.4.7.2).

O Algoritmo 4.3 apresenta o pseudocódigo do AIRP. Ele possui quatro parâmetros de entrada: *i*) $vezesNivel$, que representa o número de vezes de execução para cada nível de perturbação; *ii*) $tempoExec$, tempo limite em milisegundos para execução do algoritmo; *iii*) $\beta$, utilizado no procedimento $RVI$ para limitar a quantidade de máquinas que passa pela busca local $BI_{TMM}$ (ver subseção 4.4.6.3); e *iv*) $nivelMax$, que representa o número máximo de perturbações simultâneas.

**Algoritmo 4.2**: GIAP

```
   entrada : vezesNivel, tempoExec, iterMax, iterAtualizaProb, α, tempoLimite, β, nivelMax
   saida   : melhorSol
 1 tempoAtual ← 0;
 2 Solução s, s', melhorSol;
 3 elite ← {};
 4 s ← CPG_GRASP(α, tempoLimite) ;                          /* ver subseção 4.4.4 */
 5 s ← ALS(s, iterMax, iterAtualizaProb, β) ;               /* ver subseção 4.4.10 */
 6 melhorSol ← s;
 7 elite ← elite ∪ {melhorSol};
 8 nivel ← 1;
 9 Atualiza tempoAtual;
10 enquanto tempoAtual ≤ tempoExec faça
11     s' ← s;
12     vezes ← 0;
13     perturbMax ← nivel + 1;
14     enquanto vezes < vezesNivel faça
15         perturb ← 0;
16         s' ← s;
17         enquanto perturb < perturbMax faça
18             perturb ++;
19             s' ← Perturb_ILS(s') ;                      /* ver subseção 4.4.7.2 */
20         fim
21         s' ← ALS(s', iterMax, iterAtualizaProb, β) ;    /* ver subseção 4.4.10 */
22         elite ← atualiza(s');
23         pr ← aleatorio(0,1);
24         se pr ≤ 0.05 e |elite| ≥ 5 então
25             el ← aleatorio(1,5);
26             se f(elite[el]) < f(s') então
27                 s' ← BkPR_AT(elite [el], s') ;          /* ver subseção 4.4.13 */
28             fim
29             senão
30                 se f(elite[el]) > f(s') então
31                     s' ← BkPR_AT(s', elite [el]) ;      /* ver subseção 4.4.13 */
32                 fim
33             fim
34         fim
35         se f(s') < f(s) então
36             s ← s';
37             vezes ← 0;
38             atualizaMelhor(s, melhorSol);
39             elite ← atualiza(s);
40         fim
41         vezes ++;
42         Atualiza tempoAtual;
43     fim
44     nivel ++;
45     se nivel ≥ nivelMax então
46         nivel ← 1;
47     fim
48 fim
49 retorne melhorSol ;
```

**Algoritmo 4.3**: AIRP

```
   entrada    : vezesNivel, tempoExec, β, nivelMax
   saida      : melhorSol
 1 tempoAtual ← 0;
 2 Solução s, s', melhorSol;
 3 elite ← {};
 4 s ← CG_ASPT() ;                                          /* ver subseção 4.4.1 */
 5 s ← RVI(s, β) ;                                          /* ver subseção 4.4.9 */
 6 melhorSol ← s;
 7 elite ← elite ∪ {melhorSol};
 8 nivel ← 1;
 9 Atualiza tempoAtual;
10 enquanto tempoAtual ≤ tempoExec faça
11     s' ← s;
12     vezes ← 0;
13     perturbMax ← nivel + 1;
14     enquanto vezes < vezesNivel faça
15         perturb ← 0;
16         s' ← s;
17         enquanto perturb < perturbMax faça
18             perturb ++;
19             s' ← Perturb_ILS(s') ;                      /* ver subseção 4.4.7.2 */
20         fim
21         s' ← RVI(s', β) ;                               /* ver subseção 4.4.9 */
22         elite ← atualiza(s');
23         pr ← aleatorio(0,1);
24         se pr ≤ 0.05 e |elite| ≥ 5 então
25             el ← aleatorio(1,5);
26             se f(elite[el]) < f(s') então
27                 s' ← BkPR_AM(elite [el], s') ;          /* ver subseção 4.4.12 */
28             fim
29             senão
30                 se f(elite[el]) > f(s') então
31                     s' ← BkPR_AM(s', elite [el]) ;      /* ver subseção 4.4.12 */
32                 fim
33             fim
34         fim
35         se f(s') < f(s) então
36             s ← s';
37             vezes ← 0;
38             atualizaMelhor(s, melhorSol);
39             elite ← atualiza(s);
40             vezes ← 0;
41         fim
42         vezes ++;
43         Atualiza tempoAtual;
44     fim
45     nivel ++;
46     se nivel ≥ nivelMax então
47         nivel ← 1;
48     fim
49 fim
50 retorne melhorSol ;
```

### 4.3.4 Algoritmo Híbrido AIRMP

O algoritmo híbrido AIRMP é similar ao algoritmo heurístico AIRP, diferenciando-se deste apenas por adicionar um módulo de programação linear inteira mista. Esse módulo é acionado periodicamente após um determinado número de iterações sem melhora. O Algoritmo 4.4 apresenta o pseudocódigo do AIRMP.

Como pode ser observado pelo Algoritmo 4.4, o AIRMP possui seis parâmetros de entrada: *i*) $vezesNivel$, que representa o número de vezes de execução para cada nível de perturbação; *ii*) $tempoExec$, tempo em milisegundos para execução do algoritmo; *iii*) $tempoParticao$, utilizado no procedimento $PLIM_{UPMSPST}$ para controlar o tempo de execução em cada partição; *iv*) $tamParticao$, define o tamanho das partições utilizado pelo procedimento $PLIM_{UPMSPST}$; *v*) $\beta$, utilizado no procedimento $RVI$ para limitar a quantidade de máquinas que passa pela busca local $BI_{TMM}$ (ver subseção 4.4.6.3); e *vi*) $nivelMax$, que representa o número máximo de perturbações simultâneas.

Na linha 1, a contabilização do tempo de execução é iniciada. Depois, na linha 2, são criadas 3 soluções para o problema, como segue: *i*) $s$, a solução corrente; *ii*) $s'$, a solução modificada; *iii*) $melhorSol$, que armazena a melhor solução. Na linha 3 o conjunto elite é iniciado (ver subseção 4.4.11). Na linha 4 a solução $s$ recebe a solução inicial criada pelo procedimento guloso $CG_{ASPT}$. A seguir, essa passa por uma busca local utilizando o procedimento $RVI$ e a $melhorSol$ recebe a solução $s$ resultante da busca local. Na linha 7 a solução $melhorSol$ é inserida no conjunto elite.

O nível de perturbação é inicializado com valor 1 na linha 8, e o tempo de execução é atualizado na linha 10. O processo iterativo do ILS inicia-se na linha 11 e termina na linha 67. Este processo iterativo é interrompido quando o limite de tempo é excedido.

Nas linhas 13 e 14 são inicializadas a variável que controla o número de vezes em que cada nível de perturbação ($nivel$) é aplicado, assim como o nível máximo de perturbações ($perturbMax$). Entre as linhas 18 e 21 são feitas as perturbações na solução corrente. Após a perturbação (linha 22), a solução corrente passa pela busca local $RVI$. Na linha 23 o conjunto elite é atualizado.

Na linha 24 é gerado um número real aleatório entre 0 e 1. Se esse número for menor ou igual a 0,05 e o conjunto elite estiver completo (com 5 elementos) será realizada uma intensificação e diversificação da busca utilizando o procedimento $BkPR_{AM}$. São passados como parâmetros para o $BkPR_{AM}$ a solução corrente e uma solução escolhida de maneira aleatória dentro do conjunto elite. A solução com o menor valor para o *makespan*

**Algoritmo 4.4**: AIRMP

```
   entrada : vezesNivel, tempoExec, tempoParticao, tamParticao, β, nivelMax
   saida   : melhorSol
 1 tempoAtual ← 0;
 2 Solução s, s', melhorSol;
 3 elite ← {};
 4 s ← CG_ASPT() ;                                              /* ver subseção 4.4.1 */
 5 s ← RIV(s, β) ;                                              /* ver subseção 4.4.9 */
 6 melhorSol ← s;
 7 elite ← elite ∪ {melhorSol};
 8 nivel ← 1;
 9 iterSemMelhora ← 0;
10 Atualiza tempoAtual;
11 enquanto tempoAtual ≤ tempoExec faça
12     s' ← s;
13     vezes ← 0;
14     perturbMax ← nivel + 1;
15     enquanto vezes < vezesNivel faça
16         perturb ← 0;
17         s' ← s;
18         enquanto perturb < perturbMax faça
19             perturb ++;
20             s' ← Perturb_ILS(s') ;                          /* ver subseção 4.4.7.2 */
21         fim
22         s' ← RVI(s', β) ;                                  /* ver subseção 4.4.9 */
23         elite ← atualiza(s');
24         pr ← aleatorio(0,1);
25         se pr ≤ 0,05 e |elite| ≥ 5 então
26             el ← aleatorio(1,5);
27             se f(elite[el]) < f(s') então
28                 s' ← BkPR_AM(elite [el], s') ;              /* ver subseção 4.4.12 */
29             fim
30             senão
31                 se f(elite[el]) > f(s') então
32                     s' ← BkPR_AM(s', elite [el]);           /* ver subseção 4.4.12 */
33                 fim
34             fim
35         fim
36         se iterSemMelhora ≤ maxSemMelhora então
37             m' ← total de máquinas de melhorSol;
38             M' ← {1, .., m'};
39             Seja K o conjunto M' ordenado decrescentemente pelos tempos de conclusão;
40             maqMakespan ← o índice da primeira máquina do conjunto K;
41             para cada posição i ∈ K tal que i ≠ maqMakespan faça
42                 s'' ← PLIM_UPMSPST(melhorSol, maqMakespan, i, tempoParticao, tamParticao) ;   /* ver
                   subseção 4.4.14 */
43                 se f(s'') < f(melhorSol) então
44                     atualizaMelhor(s", melhorSol);
45                     elite ← atualiza(s");
46                     iterSemMelhora ← 0;
47                 fim
48             fim
49         fim
50         se f(s') < f(s) então
51             s ← s';
52             vezes ← 0;
53             atualizaMelhor(s, melhorSol);
54             elite ← atualiza(s);
55             iterSemMelhora ← 0;
56         fim
57         senão
58             iterSemMelhora ++;
59         fim
60         vezes ++;
61         Atualiza tempoAtual;
62     fim
63     nivel ++;
64     se nivel ≥ nivelMax então
65         nivel ← 1;
66     fim
67 fim
68 retorne melhorSol ;
```

é considerada como solução base (primeiro parâmetro do procedimento $BkPR_{AM}$) e a de maior valor para o *makespan* a solução guia (como segundo parâmetro do procedimento).

Na linha 36 é verificado se existiu $maxIterSemMelhora$ iterações sem melhora na solução corrente. O controle dessas iterações é realizado pela variável $iterSemMelhora$. Caso seja verdadeiro, são aplicadas buscas locais utilizado o módulo de programação linear inteira mista $PLIM_{UPMSPST}$ (ver subseção 4.4.14). Cada uma dessas buscas locais envolve um par de máquinas de $melhorSol$, sendo uma delas a que conclui suas tarefas por último, ou seja, a que define o *makespan*. A outra máquina varia começando-se da de segundo maior tempo de conclusão até à de menor tempo de conclusão. Caso este módulo encontre uma melhor solução, o conjunto elite e $melhorSol$ são atualizados e a variável $iterSemMelhora$ é reiniciada com valor 0.

Entre as linhas 50 e 59 é analisado se as alterações feitas na solução corrente $s'$ foram boas o suficiente para continuar a busca a partir dela. Se houver melhoria na solução corrente, a variável $iterSemMelhora$ é reiniciada em 0, caso contrário, a variável $iterSemMelhora$ é incrementada em uma unidade. Ao fim do tempo de execução, a variável $melhorSol$ guarda a melhor solução encontrada. Quando o nível de perturbação é maior ou igual a 4 (Linha 65), ele é reiniciado com 1.

## 4.4 Módulos Implementados

Nesta seção são apresentados os módulos implementados que foram utilizados na seção 4.3.

### 4.4.1 Procedimento $CG_{ASPT}$

O procedimento construtivo $CG_{ASPT}$ utiliza a regra *Adaptive Shortest Processing Time* – ASPT para gerar uma solução inicial. O ASPT é uma extensão da heurística *Shortest Processing Time* – SPT (Baker, 1974).

O Algoritmo 4.5 apresenta o pseudocódigo desse procedimento. Ele recebe como parâmetro uma função $g$ que define a classificação dos candidatos. Inicialmente todas as tarefas são inseridas em uma lista de candidatos $LC$. Enquanto existir elementos em $LC$, é executado um laço de repetição. Dentro deste laço a função $g$ classifica todos os pares $(i, j)$ de máquina $i \in M$ e tarefa $j \in LC$ de acordo com o valor de $g$. No caso,

para cada máquina $i$ e tarefa $j$ considera-se uma função $g_{ij}$ dada pela soma do tempo de processamento dessa tarefa nessa máquina, do tempo de preparação dessa tarefa nessa máquina considerando a alocação anterior e do tempo de conclusão dessa máquina. A seguir, armazena-se em $g_{\min}$ o custo do par (máquina, tarefa) que produzirá o menor tempo de conclusão considerando a inserção da tarefa na última posição da máquina respectiva, isto é, $g_{\min} = \min\{g_{ij}, \forall i \in M, \forall j \in LC\}$, sendo $g_{ij} = p_{ij} + S_{iyj} + T_i$, $j$ a tarefa candidata, $y$ a última tarefa alocada à máquina $i$ e $T_i$ o tempo de conclusão da máquina $i$. Se a máquina $i$ estiver vazia $T_i$ será 0.

O par $(i, j)$ associado a $g_{\min}$ é, então, inserido na solução parcial $s$, isto é, a tarefa $j \in LC$ é inserida na última posição da máquina $i \in M$. Após a inserção, a tarefa $j$ é retirada da lista $LC$. O processo se repete até que $LC$ esteja vazia. Ao final do procedimento é retornada uma solução $s$ factível.

**Algoritmo 4.5**: $CG_{ASPT}$

**Entrada**: $g(.)$
**Saída**: $s$

```
1  Sejam M e N os conjuntos de máquinas e tarefas, respectivamente;
2  s ← ∅;
3  Inicialize a lista de candidatos LC com todas as tarefas de N;
4  enquanto (|LC| > 0) faça
5  |  g_min ← min{g_ij | i ∈ M, j ∈ LC};
6  |  Selecione a tarefa j e a máquina i que geraram o custo g_min;
7  |  Insira na solução s a tarefa j na última posição da máquina i;
8  |  Atualize a lista de candidatos LC, retirando a tarefa j ∈ LC;
9  fim
10 Retorne s;
```

### 4.4.2 Procedimento CPG$_{\text{ASPT}}$

O procedimento $CPG_{ASPT}$ é um método construtivo parcialmente guloso que também utiliza a regra ASPT. O que diferencia este procedimento do apresentado na subseção 4.4.1, é que neste é utilizado uma aleatoriedade na escolha da tarefa candidata a pertencer à solução.

O Algoritmo 4.6 apresenta seu pseudocódigo. Ele recebe como parâmetros uma função de avaliação $g$ que tem a mesma finalidade da função $g$ apresentada na subseção 4.5 e um valor $\alpha$ que define o nível de aleatoriedade na escolha das tarefas candidatas. Inicialmente todas as tarefas são inseridas em uma lista de candidatos $LC$. É executado

um laço de repetição até que existam candidatos em $LC$. Dentro do laço é calculado o valor $g$min da mesma maneira em que foi calculado na subseção 4.5. O valor $g$max é calculado de forma similar ao $g$min, a única diferença é que o $g$max é o "maior" valor de soma do tempo de processamento, tempo de preparação e do tempo de conclusão da máquina.

Uma Lista Restrita de Candidatos $LRC$ armazena todos os pares $(i, j)$, de tarefa $j \in LC$ e máquina $i \in M$, em que o valor do custo $g$ satisfaça a equação (4.1).

$$g_{ij} \leq g_{min} + \alpha(g_{\max} - g_{\min}) \tag{4.1}$$

A seguir é selecionado em $LRC$ um par $(i, j)$, de tarefa e máquina, de maneira aleatória. Na solução $s$ é inserida a tarefa $j$ na última posição da máquina $i$. Após a inserção a tarefa $j$ é retirada da lista de candidatos $LC$. O processo se repete até que $LC$ esteja vazia. Ao final do procedimento é retornada uma solução $s$ factível.

**Algoritmo 4.6**: $CPG_{ASPT}$

**Entrada**: $g(.)$,$\alpha$
**Saída**: $s$

1 Sejam $M$ e $N$ os conjuntos de máquinas e tarefas, respectivamente;
2 $s \leftarrow \emptyset$;
3 Inicialize a lista de candidatos $LC$ com todas as tarefas de $N$;
4 **enquanto** $(|LC| > 0)$ **faça**
5 $\quad g_{\min} \leftarrow \min\{g_{ij} \mid i \in M,\, j \in LC\}$;
6 $\quad g_{\max} \leftarrow \max\{g_{ij} \mid i \in M,\, j \in LC\}$;
7 $\quad LRC \leftarrow \{i \in M,\, j \in LC \mid g(i,j) \leq g_{min} + \alpha(g_{\max} - g_{\min})\}$;
8 $\quad$ Selecione aleatoriamente um par tarefa e máquina $(i, j) \in LRC$;
9 $\quad$ Insira na solução $s$ a tarefa $j$ na última posição da máquina $i$;
10 $\quad$ Atualize a lista de candidatos $LC$, retirando a tarefa $j \in LC$;
11 **fim**
12 **Retorne** $s$;

### 4.4.3 Procedimento $\mathbf{CPG}^{\mathbf{HBSS}}_{\mathbf{ASPT}}$

O procedimento construtivo $CPG^{HBSS}_{ASPT}$ é um método construtivo parcialmente guloso que utiliza a regra ASPT para classificar os candidatos e se baseia na heurística *Heuristic-Biased Stochastic Sampling* – HBSS (Bresina, 1996) para definir a probabilidade de

escolha de cada candidato.

O Algoritmo 4.7 apresenta seu pseudocódigo. Ele recebe como parâmetros a função $g$ para classificar os candidatos e uma função *bias* de acordo com a Tabela 3.1 da subseção 3.3.1.1. Inicialmente todas as tarefas são inseridas na lista de candidatos $LC$. Um laço de repetição é executado até que existam candidatos na lista $LC$. Dentro do laço é criada uma lista ordenada $Rank$ de pares $(i, j)$, sendo $i \in M$ uma máquina e $j \in LC$ uma tarefa. Essa lista $Rank$ é ordenada, em ordem crescente, pelos valores de $g$, tal como descrito na subseção 4.4.2. A seguir, a cada par desta lista é calculada uma probabilidade utilizando a função *bias*. Essas probabilidades são armazenadas em uma lista chamada $Prob$. Dentre todos os pares $(i, j)$ é escolhido um par aleatoriamente pelo método da roleta. Dessa forma, todos os pares têm chances de serem escolhidos, sendo aquele com maior valor de *bias* o de maior chance de ser escolhido. Quando o par $(i, j)$ é escolhido, a tarefa $j$ é inserida na última posição da máquina $i$ na solução $s$, e a lista $LC$ é atualizada removendo-se a tarefa $j$. Esse laço é repetido até que todas as tarefas sejam inseridas na solução. Ao final é retornada uma solução factível $s$.

**Algoritmo 4.7**: $CPG^{HBSS}_{ASPT}$

**Entrada**: $g(.)$,*bias*
**Saída**: $s$

```
1  Sejam M e N os conjuntos de máquinas e tarefas, respectivamente;
2  s ← ∅;
3  Inicialize a lista de candidatos LC com todas as tarefas de N;
4  enquanto (|LC| > 0) faça
5  |  Defina uma lista Rank de pares (i, j), com j ∈ LC e i ∈ M;
6  |  Ordene esta lista Rank de acordo com a função g;
7  |  Defina uma lista Prob para armazenar as probabilidades de cada par (i, j),
   |  j ∈ LC e i ∈ M;
8  |  Classifique a lista Prob pela função bias;
9  |  Aplique o método da roleta para selecionar um par de tarefa e máquina
   |  (i, j) ∈ Prob;
10 |  Insira na solução s a tarefa j na última posição da máquina i;
11 |  Atualize a lista de candidatos LC, removendo a tarefa j ∈ LC;
12 fim
13 Retorne s;
```

### 4.4.4 Procedimento $CPG_{GRASP}$

O procedimento $CPG_{GRASP}$ implementa a fase de construção do algoritmo GRASP (Feo et al., 1994) para gerar uma solução inicial.

O Algoritmo 4.8 apresenta seu pseudocódigo. Ele recebe como parâmetros uma função de avaliação da solução $f$, uma função de avaliação $g$ para classificação dos candidatos, uma variável $\alpha$ para definir a aleatoriedade e um *tempoLimite* para execução. Inicialmente é gerada uma solução $s$ utilizando o procedimento construtivo guloso $CG_{ASPT}$ definido na subseção 4.4.1. Na linha 3 é iniciado um laço de repetição enquanto existir tempo suficiente. Dentro do laço é gerada uma nova solução $s'$, construída através do procedimento construtivo parcialmente guloso $CPG_{ASPT}$ definido na subseção 4.4.2. Na linha 5 é avaliada se a solução $s'$ é melhor que a solução de $s$. Caso isso ocorra, a solução $s$ recebe $s'$; caso contrário ela é desprezada. A seguir o *tempoAtual* é atualizado. Ao final do laço é retornada a melhor solução encontrada.

É possível observar que este procedimento retorna uma solução igual ou melhor que o procedimento guloso $CG_{ASPT}$, já que ela parte de uma solução construída de forma gulosa e tenta gerar novas soluções parcialmente gulosas que sejam melhores.

**Algoritmo 4.8**: $CPG_{GRASP}$

**Entrada**: $f(.)$, $g(.)$, $\alpha$, *tempoLimite*
**Saída**: $s$
1 Sejam $M$ e $N$ os conjuntos de máquinas e tarefas, respectivamente;
2 $s \leftarrow CG_{ASPT}(g(.), M, N)$;
3 **enquanto** *(tempoAtual* $<$ *tempoLimite)* **faça**
4 $\quad s' \leftarrow CPG_{ASPT}(g(.), \alpha, M, N)$;
5 $\quad$ **se** $f(s) > f(s')$ **então**
6 $\quad\quad s \leftarrow s'$;
7 $\quad$ **fim**
8 $\quad$ *Atualiza*(*tempoAtual*);
9 **fim**
10 **Retorne** $s$;

### 4.4.5 Estruturas de Vizinhança

Para explorar o espaço de soluções são aplicados três tipos diferentes de movimentos, cada qual dando origem a um tipo de vizinhança. As subseções posteriores apresentam essas estruturas de vizinhança.

#### 4.4.5.1 Múltipla Inserção

Este movimento, que dá origem à vizinhança $N^{MI}(.)$, consiste em realocar uma tarefa de uma máquina para qualquer outra posição na mesma máquina ou realocar esta tarefa para qualquer posição de outra máquina. Considerando o exemplo de solução $s$ da Figura 2.1, a Figura 4.2 ilustra a aplicação deste movimento, em que a tarefa 6 da máquina $M_2$ é transferida para a máquina $M_1$ antes da tarefa 1.

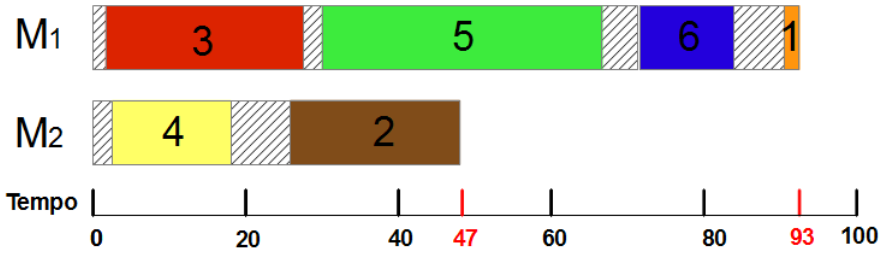



**Figura 4.2:** Exemplo do movimento de Múltipla Inserção

#### 4.4.5.2 Trocas na Mesma Máquina

Este movimento, que dá origem à vizinhança $N^{TMM}(.)$, consiste em trocar de posição duas tarefas de uma mesma máquina. Para exemplificar, a Figura 4.3 ilustra a troca de posição das tarefas 2 e 4 da máquina $M_2$, relativa à Figura 2.1.

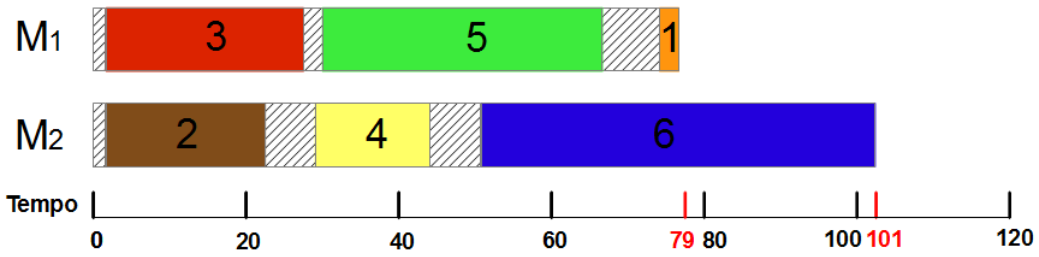



**Figura 4.3:** Exemplo do movimento de Troca na Mesma Máquina

#### 4.4.5.3 Trocas em Máquina Diferentes

Este movimento, que dá origem à vizinhança $N^{TMD}(.)$, consiste em trocar uma tarefa de uma máquina por outra tarefa de outra máquina. A Figura 4.4 ilustra a troca das

tarefas 3 e 6, das máquinas $M_1$ e $M_2$, respectivamente, ambas relativas à Figura 2.1.

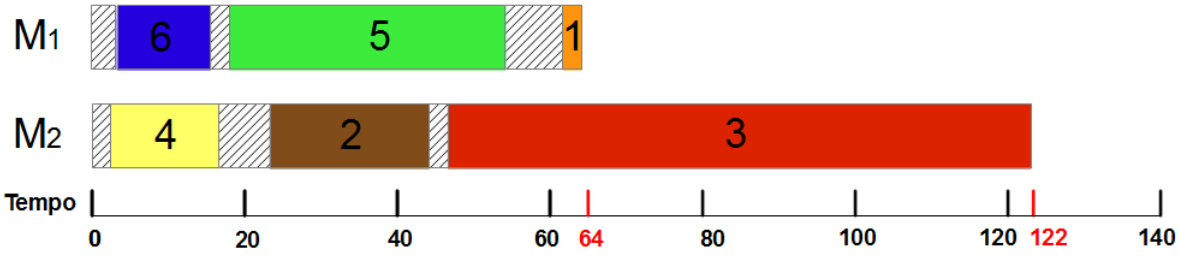



**Figura 4.4:** Exemplo do movimento de Troca em Máquinas Diferentes

### 4.4.6 Buscas Locais

Nesta subseção são apresentadas as buscas locais.

#### 4.4.6.1 Busca Local $FI^1_{MI}$

Esta busca local utiliza a vizinhança $N^{MI}(.)$ através da estratégia *First Improvement*.

Ela funciona como segue: Inicialmente, consideram-se dois conjuntos $K_1$ e $K_2$, sendo que em $K_1$ são armazenadas as máquinas em ordem decrescente, de acordo com a função custo de cada máquina e em $K_2$, as máquinas ordenadas, em ordem crescente. A seguir, são aplicados movimentos envolvendo todas as combinações de $K_1$ e $K_2$, ou seja, das máquinas mais sobrecarregadas com as máquinas menos sobrecarregadas. A inserção da tarefa $j$ pertencente à máquina de $K_1$ é avaliada em todas posições da máquina selecionada em $K_2$.

Um movimento é aceito, ou seja, um vizinho $s' \in N^{MI}(s)$ é aceito se: $i$) houver redução no custo das duas máquinas envolvidas (ou redução do custo na máquina envolvida, caso o movimento envolva uma única máquina); $ii$) houver melhora no custo em uma das máquinas e piora no custo na outra máquina, mas no cômputo geral, a melhoria for maior que a piora e não haja piora do *makespan*. Este critério é aplicado apenas para movimentos envolvendo duas máquinas.

Em caso de aceitação, $s'$ passa a ser a nova solução corrente $s$ e este procedimento é reaplicado a partir da solução corrente. Caso contrário, outro vizinho $s'$ é gerado. A

busca local é interrompida quando não houver mais movimentos de melhora em relação à solução corrente, caracterizando um ótimo local com relação a esta vizinhança.

O Algoritmo 4.9 apresenta o pseudocódigo da busca local $FI^1_{MI}$.

**Algoritmo 4.9**: $FI^1_{MI}$

```
   Entrada: s
   Saída: s
 1 Seja M o conjunto de máquinas;
 2 Seja K1 o conjunto M ordenado decrescentemente pelos tempos de conclusão;
 3 Seja K2 o conjunto M ordenado crescentemente pelos tempos de conclusão;
 4 para cada máquina k1 ∈ K1 faça
 5     para cada tarefa j ∈ k1 faça
 6         para cada máquina k2 ∈ K2, comk2 ≠ k1 faça
 7             para cada posição i de 0 até tamanho de k2 + 1 faça
 8                 Seja s' o resultado da realocação de j na posição de i;
 9                 se critério de aceitação for atendido então
10                     s ← s';
11                     FI^1_MI(s);
12                 fim
13             fim
14         fim
15     fim
16 fim
17 Retorne s;
```

#### 4.4.6.2 Busca Local $\mathbf{FI^2_{MI}}$

Esta busca local utiliza a vizinhança $N^{MI}(.)$ através da estratégia *First Improvement* e tem o funcionamento similar à da busca local apresentada na subseção 4.4.6.1.

Ela funciona como segue: Inicialmente, consideram-se dois conjuntos $K_1$ e $K_2$, sendo que em $K_1$ são armazenadas as máquinas em ordem decrescente, de acordo com a função custo de cada máquina e em $K_2$, as máquinas ordenadas, em ordem crescente. A seguir, são aplicados movimentos envolvendo todas as combinações de $K_1$ e $K_2$, ou seja, das máquinas mais sobrecarregadas com as máquinas menos sobrecarregadas. A inserção da tarefa $j$ pertencente à máquina de $K_1$ é avaliada em todas posições da máquina selecionada em $K_2$.

Um movimento é aceito, ou seja, um vizinho $s' \in N^{MI}(s)$ é aceito se houver melhoria no valor da solução corrente $s$, tendo-se como referência o *makespan*.

Em caso de aceitação, o algoritmo é finalizado e a solução $s$ retornada. Caso contrário, outro vizinho $s'$ é gerado. A busca local é interrompida quando uma solução é aceita ou se esgotem todas as combinações de $K_1$ e $K_2$.

O Algoritmo 4.10 apresenta o pseudocódigo da busca local $FI^2_{MI}$.

**Algoritmo 4.10**: $FI^2_{MI}$

**Entrada**: $s$, $f(.)$
**Saída**: $s$

```
1  Seja M o conjunto de máquinas;
2  Seja K1 o conjunto M ordenado decrescentemente pelos tempos de conclusão;
3  Seja K2 o conjunto M ordenado crescentemente pelos tempos de conclusão;
4  para cada máquina k1 ∈ K1 faça
5      para cada tarefa j ∈ k1 faça
6          para cada máquina k2 ∈ K2, comk2 ≠ k1 faça
7              para cada posição i de 0 até tamanho k2 + 1 faça
8                  Seja s' o resultado da realocação de j na posição de i;
9                  se f(s') < f(s) então
10                     s ← s';
11                     Pare a Busca;
12                 fim
13             fim
14         fim
15     fim
16 fim
17 Retorne s;
```

#### 4.4.6.3 Busca Local $\mathbf{BI_{TMM}}$

Esta busca local utiliza a vizinhança $N^{TMM}(.)$ através da estratégia *Best Improvement*.

Ela funciona como segue: Inicialmente forma-se um conjunto $K_1$ com máquinas ordenadas, em ordem decrescente, de acordo com a função custo de cada máquina.

A seguir, são analisados movimentos envolvendo trocas entre tarefas da máquina mais sobrecarregada. O vizinho $s' \in N^{TMM}(s)$ com o melhor valor da função custo é armazenado em $s''$ se $f(s') < f(s'')$. Ao final de todas as trocas envolvendo a máquina mais sobrecarregada é realizado o melhor movimento armazenado na solução $s''$; desta forma, $s' \leftarrow s''$.

Enquanto houver melhora, o procedimento é repetido a partir da máquina mais

sobrecarregada. Não havendo melhora na máquina atual, são analisados os movimentos envolvendo as tarefas da segunda máquina mais sobrecarregada. Este procedimento é aplicado até um máximo de $\beta\%$ da quantidade de máquinas. A exploração não envolve a totalidade das máquinas em virtude do elevado custo desta busca.

O Algoritmo 4.11 mostra o pseudocódigo da busca local $BI_{TMM}$.

**Algoritmo 4.11**: $BI_{TMM}$

```
Entrada: s, f(.), β
Saída: Solução s refinada
Seja M o conjunto de máquinas;
Seja K_1 o conjunto M ordenado decrescentemente pelos tempos de conclusão;
para cada máquina k_1 até β% de K_1 faça
    melhorou ← TRUE;
    s'' ← s;
    enquanto melhorou faça
        melhorou = FALSE;
        para cada tarefa j ∈ k_1 faça
            para cada tarefa i ∈ k_1 faça
                se i ≠ j então
                    Seja s' a solução resultante da melhor troca entre as tarefas i e
                    j;
                    se f(s') < f(s'') então
                        s'' ← s';
                    fim
                fim
            fim
        fim
        se f(s'') < f(s) então
            s ← s'';
            melhorou ← TRUE;
        fim
    fim
fim
Retorne s;
```

#### 4.4.6.4 Busca Local $FI_{TMD}$

Este procedimento utiliza a vizinhança $N^{TMD}(.)$ por meio da estratégia *First Improvement*.

Ela funciona como segue: Inicialmente são formados dois conjuntos $K_1$ e $K_2$ de

máquinas, sendo que o conjunto $K_1$ recebe as máquinas ordenadas, em ordem decrescente, de acordo com a função custo de cada máquina e $K_2$, as máquinas ordenadas, em ordem crescente. A seguir, são aplicadas as trocas de tarefas envolvendo todas as combinações de máquinas de $K_1$ e $K_2$.

Um movimento é aceito, ou seja, um vizinho $s' \in N^{TMD}(s)$ é aceito se: *i*) houver redução no custo das duas máquinas envolvidas; *ii*) houver melhora no custo em uma das máquinas e piora no custo na outra máquina, mas no cômputo geral, a melhoria for maior que a piora.

Caso o movimento seja aceito, o procedimento é interrompido; caso contrário, são analisadas outra combinação de máquinas, respeitando-se a ordem dos conjuntos $K_1$ e $K_2$. O procedimento é encerrado quando uma troca for aceita ou se esgotarem as trocas envolvendo tarefas de todos os pares de máquinas.

O Algoritmo 4.12 apresenta o pseudocódigo da busca local $FI_{TMD}$.

**Algoritmo 4.12**: $FI_{TMD}$

**Entrada**: $s$
**Saída**: $s$

```
1  Seja M o conjunto de máquinas;
2  Seja K1 o conjunto M ordenado decrescentemente pelos tempos de conclusão;
3  Seja K2 o conjunto M ordenado crescentemente pelos tempos de conclusão;
4  para cada máquina k1 ∈ K1 faça
5      para cada tarefa j ∈ k1 faça
6          para cada máquina k2 ∈ K2 faça
7              para cada tarefa i ∈ k2 faça
8                  se k1 ≠ k2 então
9                      Seja s' o resultado da troca de j com i;
10                     se critério de aceitação for atendido então
11                         s ← s';
12                         Pare a busca;
13                     fim
14                 fim
15             fim
16         fim
17     fim
18 fim
19 Retorne s;
```

### 4.4.7 Perturbações

Nesta subseção são apresentadas as perturbações.

#### 4.4.7.1 Perturbação Perturb$_{\text{TMD}}$

Este procedimento utiliza a vizinhança $N^{TMD}(.)$ e, portanto, consiste em aplicar um movimento de troca envolvendo tarefas de máquinas diferentes.

Ela funciona como segue: Inicialmente, tal como anteriormente, sejam dois conjuntos $K_1$ e $K_2$, com $K_1$ recebendo as máquinas ordenadas, em ordem decrescente, pela função custo de cada máquina e $K_2$ recebendo as máquinas ordenadas, em ordem crescente.

O procedimento consiste em aplicar uma perturbação envolvendo um par de máquinas de $K_1$ e $K_2$. Inicialmente, respeitando-se a ordem dos conjuntos $K_1$ e $K_2$, são aplicadas perturbações envolvendo a máquina mais sobrecarregada e a máquina menos sobrecarregada. Uma perturbação $s' \in N^{TMD}(s)$ é aceita apenas se houver redução no custo de uma das duas máquinas envolvidas. Observe que este procedimento pode gerar uma solução de qualidade inferior, ou seja, com piora no valor do *makespan*.

Caso a perturbação seja aceita, o procedimento é interrompido; caso contrário, são analisadas outras combinações de máquinas, respeitando-se a ordem estabelecida. O procedimento é interrompido quando uma perturbação for aceita ou se esgotarem as perturbações envolvendo tarefas de todos os pares de máquinas.

O Algoritmo 4.13 apresenta o pseudocódigo da perturbação $Perturb_{TMD}$.

#### 4.4.7.2 Perturbação Perturb$_{\text{ILS}}$

Este procedimento consiste em perturbar a solução ótima local por meio de movimentos de inserção, de forma a explorar outras regiões do espaço de soluções. Ele funciona como segue: Inicialmente selecionam-se duas máquinas de maneira aleatória, $i_1$ e $i_2$. Depois, retira-se uma tarefa $j$ de forma aleatória da máquina $i_1$. A tarefa $j$ é, então, inserida na melhor posição da segunda máquina $i_2$. A melhor posição é aquela que acarretar o menor custo à máquina $i_2$.

O Algoritmo 4.14 apresenta o pseudocódigo da perturbação $Perturb_{ILS}$.

Para controlar a quantidade de pertubações que são realizadas no algoritmo proposto

**Algoritmo 4.13**: *Perturb*$_{TMD}$

**Entrada**: $s$
**Saída**: $s$

```
1  Seja M o conjunto de máquinas;
2  Seja K_1 o conjunto M ordenado decrescentemente pelos tempos de conclusão;
3  Seja K_2 o conjunto M ordenado crescentemente pelos tempos de conclusão;
4  para cada máquina k_1 ∈ K_1 faça
5      para cada tarefa j ∈ k_1 faça
6          para cada máquina k_2 ∈ K_2 faça
7              para cada tarefa i ∈ k_2 faça
8                  se k_1 ≠ k_2 então
9                      Seja s' o resultado da troca de j com i;
10                     se critério de aceitação for atendido então
11                         s ← s';
12                         Pare a busca;
13                     fim
14                 fim
15             fim
16         fim
17     fim
18 fim
19 Retorne s;
```

**Algoritmo 4.14**: *Perturb*$_{ILS}$

**Entrada**: $s$
**Saída**: $s$

```
1  Aleatoriamente selecione uma máquina i_1 e uma máquina i_2 de s;
2  Aleatoriamente selecione uma tarefa j de i_1;
3  Remova j de i_1;
4  Insira j na melhor posição na máquina i_2;
5  Retorne s;
```

é utilizado um nível de perturbação. Este nível $l$ de perturbação é definido por $l + 1$ movimentos de inserção. O número máximo de níveis de perturbação permitidos é 3; assim ocorrerão 4 movimentos de inserção simultâneos, no máximo. O objetivo desta técnica é diversificar a busca e, ao mesmo tempo, não ficar preso nos ótimos locais. O nível $l$ de perturbação aumenta após *timeslevel* soluções perturbadas sem que haja melhoria na solução corrente. Por outro lado, quando é encontrada uma solução melhor, o nível de perturbação volta para seu nível mais baixo ($l = 1$).

### 4.4.8 Procedimento RVND

O procedimento *Random Variable Neighborhood Descent – RVND* foi proposto por (Souza et al., 2010). Nesse trabalho foi mostrada a eficiência do *RVND* diante do clássico *Variable Neighborhood Descent* – VND (Hansen et al., 2008).

Seu pseudocódigo é apresentado pelo Algoritmo 4.15.

O *RVND* utiliza as três buscas locais $FI^1_{MI}$, $BI_{TMM}$ e $FI_{TMD}$. Não é definida uma ordem fixa para exploração das buscas locais utilizadas; desta forma, uma ordem de processamento destas buscas é definida aleatoriamente. A solução $s'$ resultante de uma busca local é aceita se ela possuir um menor *makespan* que a solução $s$. Caso isso seja verdade, a solução $s$ recebe a solução $s'$, e a busca é reiniciada com o a primeira busca local; caso contrário, a busca continua na próxima busca local e encerra quando todas as buscas locais forem analisadas.

No Algoritmo 4.15 suponha, por exemplo, que em uma chamada, ao final do embaralhamento do vetor $v$, seus elementos sejam $v = \{2, 3, 1\}$. Isso significa, então, que a ordem de exploração será: 1) vizinhança $BI_{TMM}$, 2) vizinhança $FI_{TMD}$ ; e 3) vizinhança $FI^1_{MI}$.

### 4.4.9 Procedimento RVI

O procedimento *RVI* é inspirado nos algoritmos *Random Variable Neighborhood Descent – RVND* (Souza et al., 2010) e *Iterated Local Search* – ILS (Lourenço et al., 2003).

Seu pseudocódigo é apresentado pelo Algoritmo 4.16. Este algoritmo tem o funcionamento próximo do procedimento apresentado na subseção 4.4.8, diferenciando-se apenas nos métodos utilizados para realizar as buscas locais.

**Algoritmo 4.15**: *RVND*

**entrada**: $s$, $f(.)$, $\beta$
**saida** : $s$

$v \leftarrow \{1, 2, 3\}$;
`embaralhar` $(v)$;
$k \leftarrow 1$;
**enquanto** $(k \leq 3)$ **faça**
  **se** $k = v[1]$ **então**
    $s' \leftarrow FI^{1}_{MI}(s)$ ; `/* ver subseção 4.4.6.1 */`
  **fim**
  **se** $k = v[2]$ **então**
    $s' \leftarrow BI_{TMM}(s, \beta)$;
    ; `/* ver subseção 4.4.6.3 */`
  **fim**
  **se** $k = v[3]$ **então**
    $s' \leftarrow FI_{TMD}(s)$ ; `/* ver subseção 4.4.6.4 */`
  **fim**
  **se** $f(s') < f(s)$ **então**
    $s \leftarrow s'$;
    `atualizaMelhor`$(s)$;
    $k \leftarrow 1$;
  **fim**
  **senão**
    $k++$;
  **fim**
**fim**
**Returne** $s$;

**Algoritmo 4.16**: *RVI*

**entrada**: $s$, $f(.)$, $\beta$
**saida** : $s$

```
1  v ← {1, 2, 3};
2  embaralhar (v);
3  k ← 1;
4  enquanto (k ≤ 3) faça
5      se k = v[1] então
6          s' ← FI¹_MI(s) ;                    /* ver subseção 4.4.6.1 */
7      fim
8      se k = v[2] então
9          s' ← BI_TMM(s, β) ;                 /* ver subseção 4.4.6.3 */
10     fim
11     se k = v[3] então
12         s' ← Perturb_TMD(s) ;               /* ver subseção 4.4.7.1 */
13     fim
14     se f(s') < f(s) então
15         s ← s';
16         atualizaMelhor(s);
17         k ← 1;
18     fim
19     senão
20         k + +;
21     fim
22 fim
23 Returne s;
```

O *RVI* utiliza as duas buscas locais $FI^{1}_{MI}$ e $BI_{TMM}$, e a perturbação $Perturb_{TMD}$. Não é definida uma ordem fixa para exploração dos métodos utilizados; desta forma, a cada iteração uma ordem de aplicação destes métodos é definida aleatoriamente. Se a solução $s'$ resultante de um dos métodos possuir um menor *makespan* que a solução $s$, ela é aceita e a solução $s$ receberá $s'$, a busca é então reiniciada com o primeiro método na ordem aleatoriamente predefinida; caso contrário, a busca continua no próximo método e encerra quando todos os métodos forem analisados.

### 4.4.10 Procedimento ALS

O procedimento *Adaptive Local Search – ALS* é um procedimento que utiliza os métodos $FI^{1}_{MI}$ e $BI_{TMM}$ como buscas locais, e um método de perturbação $Perturb_{TMD}$. A ideia deste procedimento é aumentar progressivamente, de forma adaptativa, a probabilidade dos métodos que tiverem o melhor desempenho em buscas pregressas.

O Algoritmo 4.17 apresenta o pseudocódigo do *ALS*.

Inicialmente todos os métodos ($FI^{1}_{MI}$, $BI_{TMM}$ e $Perturb_{TMD}$) têm a mesma probabilidade de ser escolhidos (linha 5). A seguir, é executado um laço de repetição (entre as linhas 6 e 33) por $iterSemMelhora$ iterações sem melhoria na solução corrente. Em cada iteração um método é selecionado por um mecanismo de roleta utilizando a sua probabilidade, na linha 14.

Quando o método selecionado proporciona uma melhoria na solução corrente, o número de iterações é reinicializado (linha 28); caso contrário, o número de iterações é incrementado (linha 31).

A cada $iterMax$ iterações do laço de $iterSemMelhora$ iterações, as probabilidades de cada método são atualizadas. Para atualizar as probabilidades é calculada a média $M_i$ do valor das soluções encontradas até o momento para cada método $i \in \{1, 2, 3\}$, na linha 10. A seguir, na linha 11, é verificada a "distância", dada por $q_i$, de cada média para a melhor solução $M^{\star}$ encontrada pelo algoritmo, isto é, $q_i = M_i/M^{\star}$. De acordo com esta "distância" é calculada a nova probabilidade $p_i$ de escolher o método $i$, calculada pela expressão $p_i = q_i / \sum_{j=1}^{3} q_j$, na linha 12. Quanto mais distante a média de um método estiver da melhor solução menor será sua probabilidade. Por outro lado, quanto mais próximo a "distância" da média de um método estiver da melhor solução, maior será a sua probabilidade. Desta forma, o método que obtiver um melhor desempenho terá maior probabilidade de ser escolhido. Ao final é retornada a melhor solução encontrada.

**Algoritmo 4.17**: *ALS*

**entrada**: $s$, $f(.)$, $iterMax$, $iterAtualizaProb$, $\beta$
**saida** : $s$

$v \leftarrow \{1, 2, 3\}$;
$iterSemMelhora \leftarrow 0$;
$iter \leftarrow 0$;
$M^{\star} \leftarrow$ melhor solução até o momento;
$p_i \leftarrow 1/3\%$ para cada método $i \in \{1, 2, 3\}$;
**enquanto** $iterSemMelhora \leq iterMax$ **faça**
    $iter \leftarrow 0$;
    $iter \leftarrow iter + 1$;
    **se** $(mod(iter, iterAtualizaProb)) = 0$ **então**
        $M_i \leftarrow$ para cada método $i \in \{1, 2, 3\}$;
        $q_i = M_i/M^{\star}$ para cada método $i \in \{1, 2, 3\}$;
        $p_i = q_i / \sum_{j=1}^{3} q_j$ para cada método $i \in \{1, 2, 3\}$;
    **fim**
    $itemEscolhido \leftarrow Roleta(Prob)$;
    **se** $itemEscolhido = v[1]$ **então**
        $s' \leftarrow FI^{1}_{MI}(s)$ ; `/* ver subseção 4.4.6.1 */`
    **fim**
    **se** $itemEscolhido = v[2]$ **então**
        $s' \leftarrow BI_{TMM}(s, \beta)$ ; `/* ver subseção 4.4.6.3 */`
    **fim**
    **se** $itemEscolhido = v[3]$ **então**
        $s' \leftarrow Perturb_{TMD}(s)$ ; `/* ver subseção 4.4.7.1 */`
    **fim**
    **se** $f(s') < f(s)$ **então**
        $s \leftarrow s'$;
        `atualizaMelhor(`$s$`)`;
        $atualiza(A*)$;
        $iterSemMelhora \leftarrow 0$;
    **fim**
    **senão**
        $iterSemMelhora \leftarrow iterSemMelhora + 1$;
    **fim**
**fim**
**Returne** $s$;

### 4.4.11 Definição do Conjunto Elite

A técnica Reconexão por Caminhos (PR, do inglês *Path Relinking*) faz um balanço entre intensificação e diversificação da busca. Seu objetivo é explorar caminhos que conectam soluções de alta qualidade. Para que isso seja feito as soluções de alta qualidade são armazenadas em um conjunto de soluções, chamado elite.

O conjunto elite utilizado neste trabalho tem no máximo 5 soluções. Para entrar neste conjunto, uma solução $s'$ deve satisfazer a pelo menos uma dentre as seguintes condições:

1. Ser melhor que a melhor solução do conjunto elite, ou seja, possuir um valor de *makespan* menor que a solução do conjunto elite que possui o menor *makespan*.
2. Ser melhor que a pior solução do conjunto elite, ou seja, possuir um menor valor de *makespan* que a solução do conjunto elite de valor mais alto de *makespan*; e se diferenciar de todas as demais soluções do conjunto elite em pelo menos 10%. Adota-se como critério de diversidade, o percentual de tarefas alocadas em posições diferentes na mesma sequência de máquinas.

Estando o conjunto elite completo, ao entrar uma solução sai a de pior *makespan*.

### 4.4.12 Procedimento $\mathsf{BkPR_{AM}}$

O procedimento $BkPR_{AM}$ utiliza a estratégia *Backward Path Relinking*, que consiste em caminhar de uma solução base para uma solução guia, sendo a solução base a melhor solução e a solução guia, a pior solução.

Esta estratégia é aplicada sobre as duas soluções assim constituídas: *i*) uma solução aleatoriamente escolhida dentro do conjunto elite; *ii*) o ótimo local resultante do procedimento de busca local adotado (podendo ser, conforme o caso, o *RVND*, *RVI* ou *ALS*.

O atributo escolhido para caracterizar o caminho é a sequência ordenada de tarefas de uma máquina; daí a nomenclatura *AM*, de Atributo Máquina.

Inicialmente, as sequências da solução guia $s_{\text{guia}}$ são inseridas em uma lista $\Delta$. A cada iteração é analisada a inserção, na solução base $s_{\text{base}}$, de um atributo (sequência) da solução $s_{\text{guia}}$.

A seguir, são eliminadas as tarefas repetidas na solução $s_{\text{base}}$. Além disso, se a máquina da solução $s_{\text{base}}$ que receber esta sequência possuir alguma tarefa diferente da sequência da máquina da solução $s_{\text{guia}}$, então esta tarefa é realocada para a melhor posição de outra máquina que ainda não teve fixado seu atributo da solução $s_{\text{guia}}$. A melhor posição é aquela que produz o menor custo em relação à máquina.

Feitas todas as análises dos atributos da solução $s_{\text{guia}}$, é adicionado à solução $s_{\text{base}}$ o atributo cujo custo na solução $s_{\text{base}}$ seja o menor. Este custo é dado pelo somatório dos custos de cada máquina da solução $s_{\text{base}}$. Essa solução $s_{\text{base}}$ modificada é submetida, então, à busca local $FI^2_{MI}$, definida na subseção 4.4.6.2. Observa-se que, uma vez inserido um atributo na solução $s_{\text{base}}$, esse atributo não pode ser alterado. Em seguida, este atributo (sequência) da solução $s_{\text{guia}}$ é retirado da lista $\Delta$. Esse procedimento é repetido até que a lista $\Delta$ esteja vazia. Ao final do algoritmo a solução $s_{\text{base}}$ será uma solução idêntica à solução $s_{\text{guia}}$. É retornada a melhor solução encontrada no caminho entre a solução $s_{\text{guia}}$ e a solução $s_{\text{base}}$.

O Algoritmo 4.18 apresenta um pseudocódigo do procedimento $BkPR_{AM}$.

**Algoritmo 4.18**: $BkPR_{AM}$

**Entrada**: $s_{\text{base}}$, $s_{\text{guia}}$, $f(.)$
**Saída**: $s$

```
1  Seja Δ o conjunto de todos os atributos (sequência de tarefas de uma máquina) de s_guia;
2  s ← s_guia;
3  enquanto (|Δ| > 0) faça
4  |   l* ← arg min{somaMaq(s_base ⊕ l) : l ∈ Δ};
5  |   Δ ← Δ \ {l*};
6  |   s_base ← s_base ⊕ l*;
7  |   s'' ← FI²_MI(s_base) ;                    /* ver subseção 4.4.6.2 */
8  |   se f(s'') < f(s) então
9  |   |   s ← s'';
10 |   |   atualizaMelhor(s);
11 |   fim
12 fim
13 Retorne s;
```

A Figura 4.5 ilustra um exemplo de inserção de um atributo da solução guia na máquina $M_1$ da solução base. Na nova solução base percebe-se que a tarefa 6, que antes ocupava a máquina $M_1$, foi realocada para a máquina $M_2$. Por outro lado, foi eliminada a repetição da ocorrência da tarefa 1.

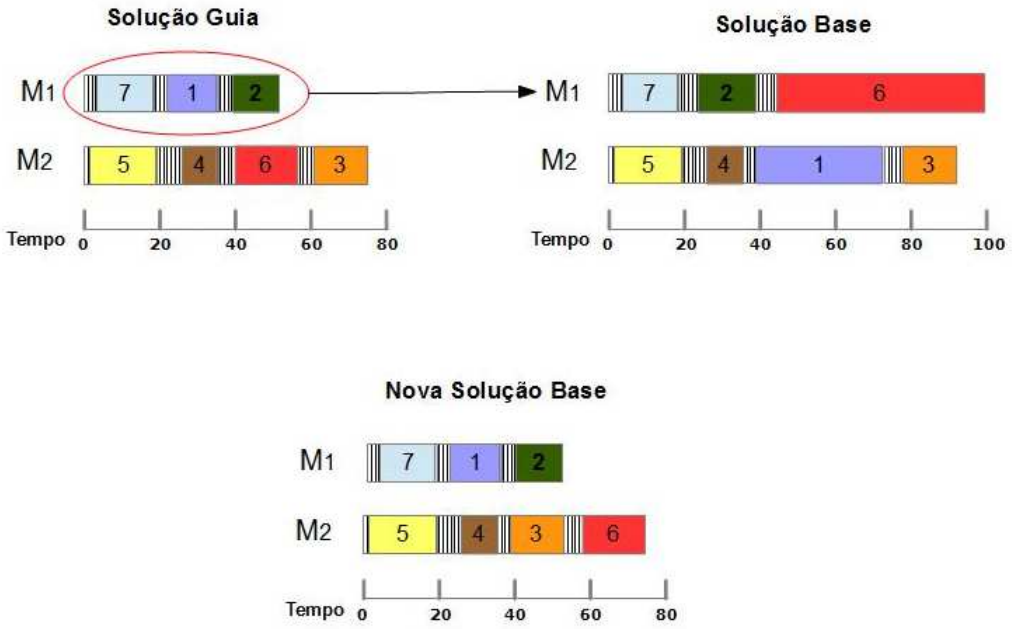



**Figura 4.5:** Exemplo de inserção de um atributo utilizando o procedimento $BkPR_{AM}$

### 4.4.13 Procedimento $\mathbf{BkPR_{AT}}$

O procedimento *Backward Path Relinking*, $BkPR_{AT}$, é similar ao apresentado na subseção 4.4.12, e também utiliza a busca local $FI^2_{MI}$. A diferença, no caso, é que o atributo escolhido para caracterizar o caminho é a posição de uma tarefa; sendo por isso, usada a nomenclatura $AT$, de Atributo Tarefa.

Inicialmente, as posições das tarefas da solução guia são inseridas em uma lista $\Phi$. A cada iteração é analisada a inserção, na solução base $s_{\text{base}}$, de um atributo (posição de uma tarefa) da solução guia $s_{\text{guia}}$. A seguir, é eliminada a tarefa repetida. Além disso, se a máquina da solução $s_{\text{base}}$ que receber esta tarefa possuir alguma tarefa diferente na mesma posição, então esta tarefa é realocada para outra posição que ainda não teve fixado seu atributo da solução $s_{\text{guia}}$.

Feitas todas as análises dos atributos da solução $s_{\text{guia}}$, é adicionado à solução base o atributo cujo custo na solução $s_{\text{base}}$ seja o menor. Este custo é dado pelo somatório dos custos de cada máquina da solução $s_{\text{base}}$. Essa solução base modificada é submetida, então, à busca local $FI^2_{MI}$, definida na subseção 4.4.6.2. Observa-se que, uma vez inserido um atributo na solução base, esse atributo não pode ser alterado.

Em seguida, este atributo (posição de uma tarefa) da solução $s_{\text{guia}}$ é retirado da lista $\Phi$. Este procedimento é repetido até que a lista $\Phi$ esteja vazia. Ao final do algoritmo a solução $s_{\text{base}}$ será uma solução idêntica à solução $s_{\text{guia}}$. É retornada a melhor solução $s$ encontrada no caminho entre a solução $s_{\text{guia}}$ e a solução $s_{\text{base}}$.

O Algoritmo 4.19 apresenta o pseudocódigo do Path Relinking $BkPR_{AT}$.

**Algoritmo 4.19**: $BkPR_{AT}$

**Entrada**: $s_{\text{base}}$, $s_{\text{guia}}$, $f(.)$
**Saída**: $s$

```
1  Seja Φ o conjunto de todos os atributos (posição de uma tarefa) de s_guia;
2  s ← s_guia;
3  enquanto (|Φ| > 0) faça
4  |  l* ← arg min{somaMaq(s_base ⊕ l) : l ∈ Φ};
5  |  Φ ← Φ \ {l*};
6  |  s_base ← s_base ⊕ l*;
7  |  s'' ← FI^2_MI(s_base) ;                    /* ver subseção 4.4.6.2 */
8  |  se f(s'') < f(s) então
9  |  |  s ← s'';
10 |  |  atualizaMelhor(s);
11 |  fim
12 fim
13 Retorne s;
```

A Figura 4.6 ilustra um exemplo de inserção de um atributo (tarefa 6 na terceira posição da máquina $M_2$) da solução guia na solução base. Na nova solução base verifica-se que a tarefa 6 é inserida na terceira posição da máquina $M_2$ e a sua ocorrência na máquina $M_1$ é eliminada.

### 4.4.14 Procedimento PLIM$_{\text{UPMSPST}}$

O procedimento $PLIM_{UPMSPST}$ consiste em uma busca local baseada na implementação da formulação de programação matemática do UPMSPST-VR da subseção 3.2.1. Essa formulação foi escolhida por apresentar, em testes preliminares, melhor desempenho que as demais relatadas na seção 3.2 para a resolução do UPMSPST.

O procedimento $PLIM_{UPMSPST}$ é aplicado somente a "duas máquinas" de uma solução. O número de máquinas é limitado a dois devido ao alto custo de resolução do UPMSPST através de programação matemática. Mesmo se tratando de duas máquinas a formulação matemática pode consumir muito tempo para a resolução do problema quando se tem

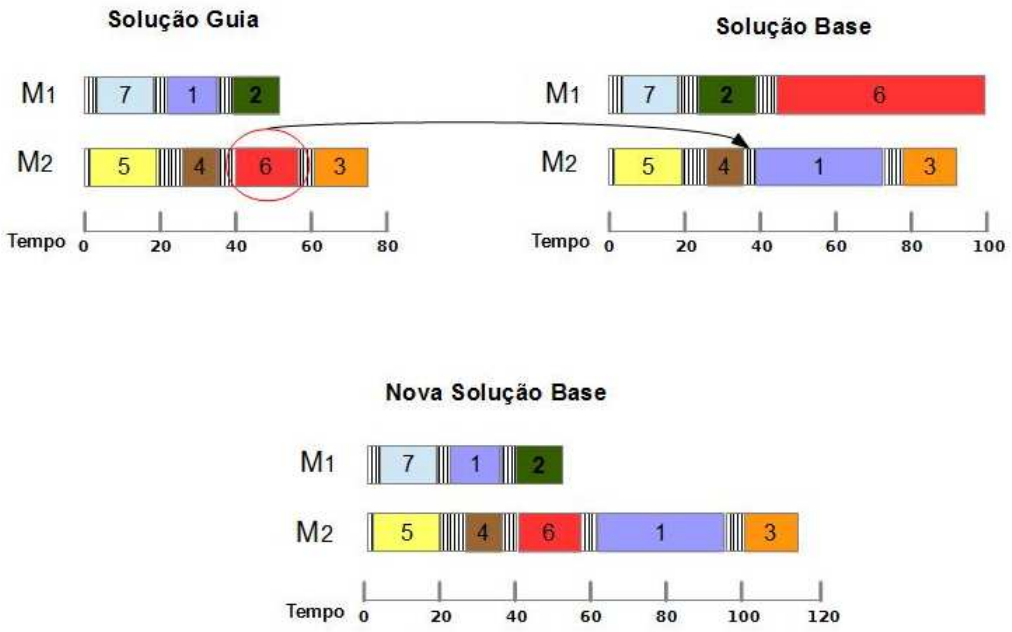



**Figura 4.6:** Exemplo de inserção de um atributo utilizando o procedimento $BkPR_{AT}$

muitas tarefas em cada uma das máquinas. Assim, é também necessário limitar a quantidade de tarefas por máquina. Para tratar essa questão, as tarefas em cada máquina são "particionadas" em conjuntos de $tamParticao$ tarefas.

A Figura 4.7 ilustra um sequenciamento para um problema-teste com 15 tarefas e 2 máquinas, com $tamParticao$ igual a 3. São criados três conjuntos de partições. O primeiro conjunto, partição 1, contém as tarefas 15, 1, 7, 5, 4 e 6; o conjunto partição 2 contém as tarefas 13, 11, 10, 3, 9 e 8; e o conjunto partição 3 contém as tarefas 2, 14 e 12. Na primeira chamada as tarefas do partição 1 são redistribuídas pelo procedimento $PLIM_{UPMSPST}$. O mesmo procedimento de redistribuição é aplicado nas demais chamadas. As soluções obtidas em cada chamada são, ao final, aglutinadas na ordem de resolução das partições, formando uma nova solução.

O pseudocódigo do procedimento $PLIM_{UPMSPST}$ é apresentado pelo Algoritmo 4.20.

O Algoritmo 4.20 recebe como parâmetros uma solução $s$, os indíces $indiceMaq1$ e $indiceMaq2$ das máquinas que serão refinadas, um $tempoParticao$ que controla o tempo limite para a formulação matemática resolver cada partição e um $tamParticao$ que define o tamanho máximo dos cortes de cada partição. A seguir, é inserida nas listas $maquina1$ e $maquina2$ as tarefas alocadas nas máquinas respectivas de índice

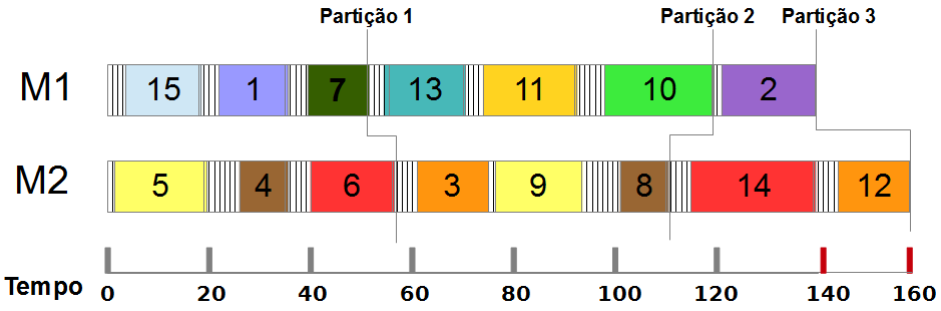



**Figura 4.7:** Exemplo de particionamento de uma solução.

**Algoritmo 4.20**: $PLIM_{UPMSPST}$

**Entrada**: $s$, $indiceMaq1$, $indiceMaq2$, $tempoParticao$, $tamParticao$
**Saída**: $s'$

$maquina1$ recebe todas tarefas alocadas na máquina $indiceMaq1$ da solução $s$;
$maquina2$ recebe todas tarefas alocadas na máquina $indiceMaq2$ da solução $s$;
$s' \leftarrow \emptyset$;
$resultMaquina1, resultMaquina2 \leftarrow \emptyset$;
$tamMaq1 \leftarrow tamanhoMaquina(maquina1)$;
$tamMaq2 \leftarrow tamanhoMaquina(maquina2)$;
$totalIter \leftarrow \max\{tamMaq1, tamMaq2\}$;
**enquanto** $(totalIter > 0)$ **faça**
    $iterMaq1 \leftarrow 0$;
    $iterMaq2 \leftarrow 0$;
    $novaMaquina1 \leftarrow \emptyset$;
    $novaMaquina2 \leftarrow \emptyset$;
    **para cada** *posição* $i1$ *de* $iterMaq1$ *até* $tamParticao$ **faça**
        Adicione a tarefa da posição $i1$ da máquina $maquina1$ na máquina $novaMaquina1$;
        $iterMaq1++$;
    **fim**
    **para cada** *posição* $i2$ *de* $iterMaq2$ *até* $tamParticao$ **faça**
        Adicione a tarefa da posição $i2$ da máquina $maquina2$ na máquina $novaMaquina2$;
        $iterMaq2++$;
    **fim**
    $PLIM(novaMaquina1, novaMaquina2, tempoParticao)$;
    Adicione as tarefas otimizadas de $novaMaquina1$ ao final de $resultMaquina1$;
    Adicione as tarefas otimizadas de $novaMaquina2$ ao final de $resultMaquina2$;
    $totalIter++$;
**fim**
Adicione as máquinas $resultMaquina1$ e $resultMaquina2$ em $s'$;
**Retorne** $s'$;

$indiceMaq1$ e $indiceMaq2$. Entre as linhas 8 e 25, é executado um laço de iterações até o tamanho máximo da maior máquina entre $maquina1$ e $maquina2$. Na linha 13 é realizado o corte na $maquina1$, definindo as tarefas da partição, sendo elas inseridas em $novaMaquina1$. De forma similiar, na linha 17 é realizado o corte na $maquina2$, e as tarefas selecionadas no corte são inseridas em $novaMaquina2$. Na linha 21 é chamado o método $PLIM$ responsável por realocar as tarefas nas máquinas $novaMaquina1$ e $novaMaquina2$ utilizando a formulação matemática UPMSPST-VR, tendo $tempoParticao$ como limite de tempo para execução. A seguir, as tarefas realocadas de $novaMaquina1$ e $novaMaquina2$ são inseridas na ordem respectiva nas máquinas $resultMaquina1$ e $resultMaquina2$. Ao final do laço, na linha 26, as máquinas $resultMaquina1$ e $resultMaquina2$ são inseridas na solução final $s'$;

### 4.4.15 Avaliação Eficiente da Função Objetivo

A avaliação total do custo de uma solução a cada movimento de inserção ou troca é muito custoso computacionalmente. Visando a redução deste custo de avaliação, e tornando os movimentos mais eficientes, utiliza-se um procedimento para otimizar este cálculo de custo. O procedimento consiste em avaliar somente as máquinas que foram alteradas. Desta forma, bastam alguns cálculos de soma e diferença para se obter o tempo de conclusão de cada máquina alterada.

Na Figura 4.8 é ilustrado um movimento de troca entre duas máquinas diferentes, relativa à Figura 2.1. A tarefa 4 que antes estava alocada na máquina $M_2$ é trocada com a tarefa 5 que estava alocada na máquina $M_1$.

O cálculo do novo tempo de conclusão da máquina $M_2$ é feito subtraindo-se o tempo de processamento da tarefa 4, $p_{24}$, os tempos de preparação relacionados, $S_{204}$ e $S_{242}$, e somando-se o tempo de processamento da tarefa 5, $p_25$, e os novos tempos de preparação relacionados, $S_{205}$ e $S_{252}$.

O novo tempo de conclusão $C_i$ da máquina $M_2$ será:

$$C_{M_2} = 97 - 17 - 2 - 7 + 43 + 1 + 8 = 123$$

Para a máquina $M_1$, deve-se somar ao tempo de conclusão, o tempo de processamento da tarefa 4, $p_{14}$, e os tempo de preparação $S_{134}$ e $S_{141}$, das novas tarefas adjacentes (3 e 1), e subtraindo-se o tempo de processamento da tarefa 5, $p_{15}$, e os tempos de preparação

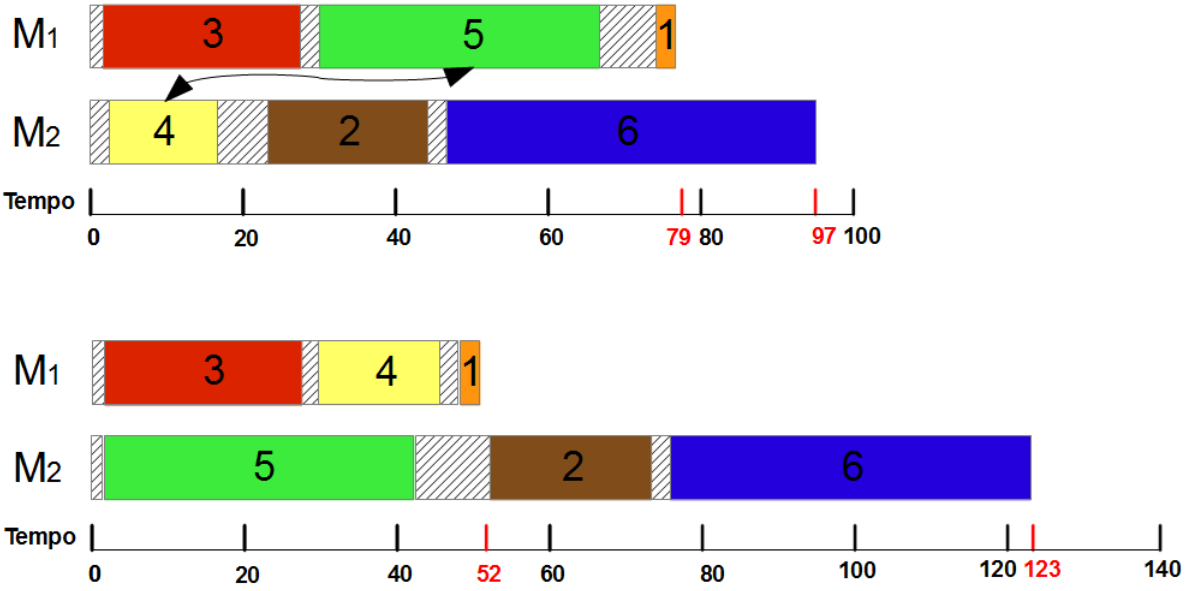



**Figura 4.8:** Avaliação em movimento de Troca entre Máquinas Diferentes

das tarefas antes adjancentes (3 e 1) da tarefa 5, $S_{135}$ e $S_{151}$.

O novo tempo de conclusão $C_i$ da máquina $M_1$ será, então, $C_{M_1}$ será:

$$C_{M_1} = 79 + 17 + 2 + 3 - 38 - 6 - 5 = 52$$

Este procedimento foi utilizado em todos os movimentos realizados pelos algoritmos implementados neste trabalho.

# Capítulo 5

# Experimentos e Resultados Computacionais

## 5.1 Problemas-teste

Nesta seção são apresentados os problemas-teste utilizados para realizar os experimentos computacionais com os algoritmos propostos.

### 5.1.1 Problemas-teste de (Vallada e Ruiz, 2011)

Os problemas-teste usados para testar os algoritmos desenvolvidos foram aqueles propostos por (Vallada e Ruiz, 2011), disponibilizados em (SOA, 2011). Esses autores classificaram o conjunto de problemas-teste em dois grandes grupos: $i$) problemas-teste pequenos; $ii$) problemas-teste grandes. Seja $n$ o número de tarefas e $m$ a quantidade de máquinas. O grupo de problemas-teste pequenos é composto por problemas que combinam $n = 6, 8, 10, 12$ tarefas e $m = 2, 3, 4, 5$ máquinas. Já o grupo de problemas-teste grandes contêm problemas que combinam $n = 50, 100, 150, 200, 250$ tarefas e $m = 10, 15, 20, 25, 30$ máquinas. Para cada combinação de $n$ e $m$ foram gerados 40 problemas-teste, gerando ao total 640 problemas-teste para o grupo de problemas-teste menores e 1000 para o grupo de problemas-teste maiores.

Os tempos de processamentos em ambos os conjuntos foram gerados pelos autores por meio de uma distribuição uniforme $U[1, 99]$. Os tempos de preparação consistem em 4 grupos diferentes, cada qual gerado por distribuição unifome, como segue: $i$) $U[0, 9]$;

*ii*) $U[0, 49]$; *iii*) $U[0, 99]$; *iv*) $U[0, 124]$.

Além dos problemas-teste, também são disponibilizadas nesse repertório, as melhores soluções encontradas, obtidas por algoritmos desenvolvidos pelos autores no trabalho (Vallada e Ruiz, 2011) e em outros métodos desenvolvidos pelos próprios autores.

### 5.1.2 Problemas-teste de (Rabadi et al., 2006)

Os problemas-teste desenvolvidos por (Rabadi et al., 2006) são disponibilizados em (SchedulingResearch, 2005). Os autores classificaram o conjunto de problemas-teste em dois grupos, denominados problemas-teste pequenos e problemas-teste grandes. Seja $n$ o número de tarefas e $m$ o número de máquinas. O grupo de problemas-teste pequenos é composto de problemas-teste que combinam $n = 6, 7, 8, 9, 10, 11$ tarefas e $m = 2, 4, 6, 8$ máquinas, totalizando 810 problemas-teste. Já o grupo de problemas-teste grandes contêm problemas-teste que combinam $n = 20, 40, 60, 80, 100, 120$ tarefas e $m = 2, 4, 6, 8, 10, 12$ máquinas, totalizando 1620 problemas-teste maiores.

As combinações de tarefas e máquinas geram 3 grupos contendo 15 problemas-teste cada. Os 3 grupos são definidos por níveis de domínio, de acordo com as seguintes características: *i*) Grupo 1: os tempos de processamento e preparação são balanceados, ou seja, eles são retirados de uma distribuição uniforme $U[50, 100]$; *ii*) Grupo 2: os tempos de processamento são dominantes, sendo extraídos da distribuição uniforme $U[125, 175]$, e os tempos de preparação são retirados da distribuição uniforme $U[50, 100]$; *iii*) Grupo 3: os tempos de preparação são dominantes, sendo extraídos da distribuição uniforme $U[125, 175]$ e os tempos de processamento são retirados da distribuição uniforme $U[50, 100]$.

Nesse repertório também são disponibilizados os melhores resultados encontrados para cada problema-teste, obtidos dos trabalhos de (Rabadi et al., 2006), (Helal et al., 2006) e (Arnaout et al., 2010). O trabalho que produziu os melhores resultados foi o de (Arnaout et al., 2010).

## 5.2 Definição dos parâmetros dos algoritmos

Esta seção apresenta os valores adotados para os parâmetros dos algoritmos apresentados na seção 4 nos experimentos computacionais.

Após uma bateria preliminar de testes, os parâmetros dos algoritmos apresentados na seção 4 foram fixados empiricamente nos seguintes valores: $\alpha = 0{,}02$; $\beta = 30\%$, isto é, apenas 30% das máquinas passam pela busca local $BI_{TMM}$ da subseção 4.4.6.3; $iterMax = 21$ e $iterAtualizaProb = 7$ (parâmetros do procedimento $ALS$ da subseção 4.4.10); $maxSemMelhora = 228$ (parâmetro utilizado no Algoritmo AIRMP da subseção 4.3.4 e responsável por acionar a busca local $PLIM_{UPMSPST}$, baseada na formulação de programação matemática da subseção 4.4.14); $tempoLimite = 0,05 \times tempoExec$, isto é, o $tempoLimite$ do procedimento $CPG_{GRASP}$ da subseção 4.4.4 representa 5% do tempo total definido pelo parâmetro $tempoExec$. Para o parâmetro $bias$ utilizado no procedimento $CPG^{HBSS}_{ASPT}$ da subseção 4.4.3, escolheu-se a função exponencial.

Os parâmetros $vezesNivel$ (usado nos algoritmos da seção 4.3) e $nivelMax$ foram fixados nos mesmos valores adotados em (Haddad et al., 2012), isto é, $vezesNivel = 15$ e $nivelMax = 4$.

O parâmetro $tempoExec$, utilizado nos algoritmos da seção 4.3, é definido de acordo com a Equação 5.1.

$$tempoExec = n \times (m/2) \times t \quad ms \tag{5.1}$$

Nesta equação, $n$ é o número total de tarefas, $m$ é o número total de máquinas e $t$ é um parâmetro testado inicialmente para três valores: 10, 30 e 50. Esses valores de $t$ são os mesmos utilizados como critério de parada nos trabalhos de (Vallada e Ruiz, 2011) e (Haddad et al., 2012).

Como para tempos de processamento menores, foi observado que o módulo de programação matemática do algoritmo AIRMP da subseção 4.3.4 não era acionado ou, então, era muito pouco acionado, o valor $t$ também foi testado para o valor igual a 500. Neste caso, o objetivo foi observar a influência da busca local baseada na formulação de programação linear inteira mista.

Mostra-se, a seguir, como foram definidos os valores dos parâmetros $tempoParticao$ e $tamParticao$ utilizados no procedimento $PLIM_{UPMSPST}$ da subseção 4.4.14. O parâmetro $tempoParticao$ é responsável por definir o limite de tempo utilizado para resolver cada corte de 2 máquinas pelo método de programação matemática $PLIM_{UPMSPST}$ da subseção 4.4.14. A ideia é utilizar o mínimo de tempo possível. O parâmetro $tamParticao$ é res-

ponsável por definir o tamanho dos cortes efetuados a cada par de máquinas.

Para definir os valores desses parâmetros foram executados experimentos utilizando-se o problema-teste I_50_10_S_9-1_1 de (SOA, 2011). Foram selecionadas 2 máquinas de maneira aleatória desse problema-teste e efetuados cortes de diferentes tamanhos. Cada corte foi resolvido 30 vezes utilizando o método de programação matemática $PLIM_{UPMSPST}$ descrito na subseção 4.4.14. A Tabela 5.1 apresenta o tempo máximo gasto para encontrar a solução ótima para cada um dos tamanhos de cortes.

**Tabela 5.1:** Tabela de calibração de *tempoParticao* e *tamParticao*

| Tamanho do corte | Tempo Máximo |
|---|---|
| Com 4 tarefas | 2 segundos |
| **Com 5 tarefas** | **5 segundos** |
| Com 6 tarefas | 2 minutos |
| Com 7 tarefas | 10 minutos |
| Com 8 tarefas | Mais de 1 hora |

Em vista desses resultados, o valor adotado para o parâmetro *tamParticao* foi igual a 5 e o do parâmetro *tamParticao* igual a 5 segundos. A justificativa da escolha pelo corte com 5 tarefas é por se tratar do maior corte que conseguiu encontrar a solução ótima em poucos segundos. O *tempoParticao* foi fixado em 5, devido este ser o tempo máximo gasto pelo método $PLIM$ para resolver um corte com 5 tarefas.

## 5.3 Resultados Obtidos

Nesta seção são apresentados os resultados obtidos nos experimentos. Todos os algoritmos propostos foram desenvolvidos na linguagem JAVA utilizando um computador Intel Core i5 3.0 GHz, com 8 GB de RAM e sistema operacional *Ubuntu 12.04*. Os parâmetros utilizados pelos algoritmos foram calibrados e definidos na seção anterior. Para executar a busca local baseada na formulação de programação matemática do algoritmo AIRMP foi utilizado o otimizador CPLEX, na versão 12.6, e sua biblioteca para JAVA, com seus parâmetros assumindo os valores padrão.

Os resultados dos experimentos computacionais são apresentados nas três próximas subseções. Na subseção 5.3.1 são apresentados os resultados da primeira bateria de

testes, na qual são realizados experimentos em um pequeno grupo de problemas-teste envolvendo três dos quatro algoritmos propostos neste trabalho. O melhor algoritmo da primeira bateria de testes é testado em um número bem maior de problemas-teste na segunda bateria (vide subseção 5.3.2); sendo ele comparado com outros algoritmos da literatura. Na terceira bateria de testes (vide subseção 5.3.3), são comparados o melhor algoritmo da primeira bateria, isto é, o AIRP, com o algoritmo AIRMP. O objetivo nesse último experimento é verificar a influência do módulo de programação matemática no melhor algoritmo da primeira bateria. Como o algoritmo AIRMP utiliza um módulo de programação linear inteira mista para realizar buscas locais e, portanto, necessita de um maior limite de tempo para ser executado, na terceira bateria é dado um tempo de processamento maior aos algoritmos comparados.

### 5.3.1 Primeira Bateria de Testes

Na primeira bateria de testes são realizados experimentos computacionais para comparar três algoritmos propostos neste trabalho, sendo eles: *i*) HIVP (vide subseção 4.3.1); *ii*) GIAP (vide subseção 4.3.2) e *iii*) AIRP (vide subseção 4.3.3).

Para realizar a comparação com estes algoritmos são utilizados alguns problemas-teste de (SOA, 2011) e de (SchedulingResearch, 2005).

Nos problemas-teste de (SOA, 2011) foram selecionados 36 problemas-teste que envolvem combinações com $n = 50, 100, 150$ tarefas e $m = 10, 15, 20$ máquinas, num total de 9 combinações. Cada combinação é subdividida em 4 grupos diferentes que utilizaram regras específicas na geração dos problemas-teste. Têm-se um total de 9 combinações com 4 grupos cada. Foi escolhido o primeiro problema-teste de cada grupo para cada combinação.

Nos problemas-teste de (SchedulingResearch, 2005) foram selecionados 18 envolvendo combinações com $n = 80, 100$ tarefas e $m = 4, 6, 10$ máquinas, num total de 6 combinações. Cada combinação é subdividida em 3 grupos diferentes que utilizaram regras específicas na geração dos problemas-teste. Os grupos são denominados *Balanced*, *P-dominant* e *S-dominat*. Ao final, têm-se 6 combinações com 3 grupos cada, sendo escolhido o primeiro problema-teste de cada grupo para cada combinação.

O critério de parada adotado é o tempo de processamento definido pela Eq. 5.1 da subseção 5.2. Adotou-se três valores de $t$, sendo eles: 10, 30 e 50.

Para comparar os três algoritmos testados nesta subseção é utilizada a métrica desvio percentual relativo $RPD$, definida pela Eq. (5.2). Nesta equação, $\bar{f}_i^{Alg}$ é o valor da solução encontrada pelo algoritmo $Alg$ para o problema-teste $i$, e $f_i^*$ é a melhor solução conhecida.

$$RPD_i = \frac{\bar{f}_i^{Alg} - f_i^*}{f_i^*} \tag{5.2}$$

Para os problemas-teste de (SOA, 2011) o valor adotado para o $f_i^*$ são as melhores soluções encontrados disponibilizadas por (Vallada e Ruiz, 2011) em (SOA, 2011). Já para os problemas-teste de (SchedulingResearch, 2005), o $f_i^*$ é limite inferior (LB) proposto por (Al-Salem, 2004) e definido pelas Eqs. (5.3), (5.4), e (5.5). Os valores dos limites inferiores para cada problema-teste do conjunto também é disponibilizado em (SchedulingResearch, 2005). O próprio autor de (Al-Salem, 2004) considera este cálculo de limite inferior relativamente fraco.

$$LB1 = \frac{1}{m}\sum_{j=1}^{n} \min[p_{jk} + S_{ijk}] \qquad \forall i \in N, k \in M \tag{5.3}$$

$$LB2 = \max\{\min[p_{jk} + S_{ijk}]\} \qquad \forall i, j \in N, k \in M \tag{5.4}$$

$$LB = \max\{LB1, LB2\} \tag{5.5}$$

Devido ao caráter estocástico, os algoritmos foram executados 30 vezes para cada problema-teste e para cada valor de $t$. A métrica usada para comparação entre os algoritmos é a média do desvio percentual relativo $RPD_i^{avg}$ dos valores $RPD_i$ encontrados.

A Tabela 5.2 compara os resultados dos algoritmos HIVP, GIAP e AIRP com relação à média do desvio percental relativo. Os problemas-teste selecionados de (SOA, 2011) são os dos grupos *Large* e os problemas-teste de (SchedulingResearch, 2005) estão separados pelos grupos *Balanced*, *P-Dominat* e *S-Dominant*. Para cada conjunto de problemas-teste são apresentados três valores de $RPD_i^{avg}$ separados por '/'. Esta separação representa os resultados dos testes para cada valor de $t$, na ordem $t = 10/30/50$. Os valores negativos indicam que os resultados alcançados pelos algoritmos superaram os melhores valores encontrados por Vallada e Ruiz (2011) em seus experimentos para os problemas-teste de (SOA, 2011).

Os melhores valores de $RPD^{avg}$ estão destacados em negrito. Analisando-se os resultados pode-se observar que o algoritmo AIRP obteve o melhor resultado na média geral para todos os valores de $t$, e também encontrou o melhor resultado na maioria dos casos, em um total de 66% dos casos. O algoritmo GIAP foi o segundo mais eficiente e encontrou o melhor resultado em 24% dos casos. Já o algoritmo HIVP ficou em último lugar encontrando o melhor resultado em 10% dos casos. Os resultados completos dos experimentos da primeira bateria de testes são disponibilizados em `http://www.decom.ufop.br/prof/marcone/projects/upmsp/Experimentos_UPMSPST_Dissert_Bateria1.ods`

A Figura 5.1 apresenta um box plot para os valores $RPD^{avg}$ dos algoritmos. Tendo por base os valores do $RPD^{avg}$, observa-se que os algoritmos AIRP e GIAP obtiveram melhores resultados que o algoritmos HIVP, sendo o algoritmo AIRP um pouco melhor que o algoritmo GIAP.

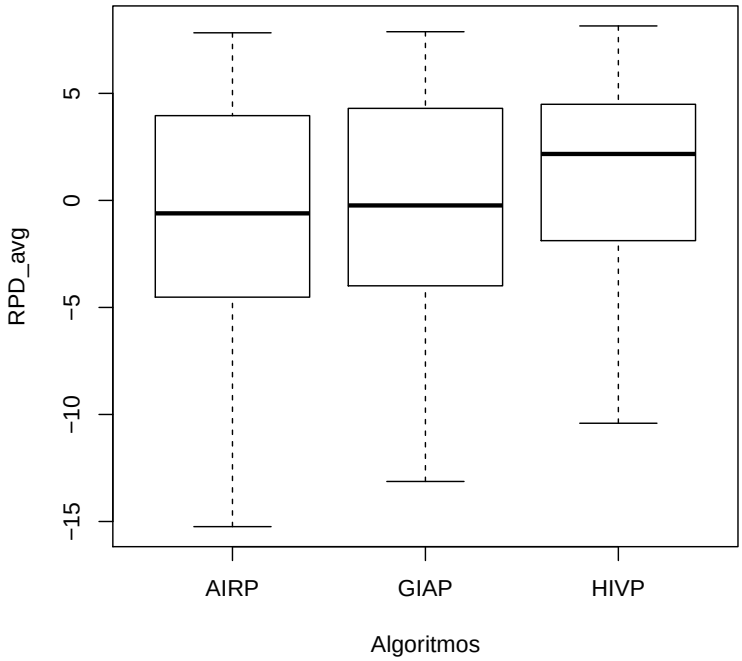



**Figura 5.1:** Box plot dos algoritmos HIVP, GIAP e AIRP.

Antes de verificar se existem diferenças estatísticas entre os algoritmos foi aplicado o teste $W$ de normalidade Shapiro-Wilk (Shapiro e Wilk, 1965) para verificar se os dados satisfazem ao teste de normalidade. Pelo teste, com nível de significância de 5% obteve-se $W = 0,9984$ e $p = 0,943$. Como $p = 0,943 > 0,05$, então podemos afirmar com

**Tabela 5.2:** Média dos *RPD*s dos algoritmos HIVP, GIAP e AIRP para $t = 10/30/50$.

| Grupos | Problemas-teste | HIVP[1] | GIAP[1] | AIRP[2] |
|---|---|---|---|---|
| Large | I_50_10_S_1-124_1 | 2,4/-1,61/-2,25 | -0,61/-3,1/-3,63 | **-2,4/-4,27/-4,56** |
| | I_50_10_S_1-49_1 | 1,64/**0,43/0,17** | 1,56/1,69/1,08 | **1,34**/1,08/0,74 |
| | I_50_10_S_1-9_1 | 2,04/0,4/-0,05 | 3,58/2,89/1,74 | **0,5/-0,1/-0,4** |
| | I_50_10_S_1-99_1 | 0/-1,84/-2,85 | -2,63/**-3,3/-3,87** | **2,74**/-3,25/-3,36 |
| | I_50_15_S_1-124_1 | 2,67/-1,64/-2,98 | -0,98/-2,93/-2,98 | **-2,4/-3,82/-4,22** |
| | I_50_15_S_1-49_1 | -1,47/-3,39/-3,16 | -3,62/-3,73/**-4,63** | **-4,12/-4,52**/-4,58 |
| | I_50_15_S_1-9_1 | 5/4,63/3,43 | 3,61/3,52/1,39 | **2,78/1,67/0,83** |
| | I_50_15_S_1-99_1 | -7,48/-8,59/-10,21 | -8,89/-10,56/-9,96 | **-9,14/-10,6/-11,41** |
| | I_50_20_S_1-124_1 | -0,58/-3,2/-4,87 | -3,91/-5,64/-5,19 | **-4,49/-5,64/-5,96** |
| | I_50_20_S_1-49_1 | -2,56/-3,16/-4,61 | -5,56/-4,53/-5,47 | **-7,18/-7,69/-7,69** |
| | I_50_20_S_1-9_1 | **5,38/4,95/2,9** | 5,59/5,48/4,95 | 6,24/5,38/4,52 |
| | I_50_20_S_1-99_1 | -8,5/-9,93/-10,41 | -11,56/-12,86/-13,13 | **-13,47/-14,69/-15,24** |
| | I_100_10_S_1-124_1 | 5,61/3,29/2,2 | **1,73**/0,92/0,21 | 1,93/**0,35/-0,8** |
| | I_100_10_S_1-49_1 | 7,55/5,16/5,09 | 5,85/4,3/3,67 | **4,78/3,56/3,29** |
| | I_100_10_S_1-9_1 | -4,2/-5,8/-6,34 | **-7,79**/-7,76/-8,07 | -7,4/**-8,01/-8,09** |
| | I_100_10_S_1-99_1 | 7,83/4,43/4,64 | 5,28/**2,99/2,06** | **4,97**/3,4/2,94 |
| | I_100_15_S_1-124_1 | 2,29/-0,45/-2,15 | -2,53/-4,51/-5,03 | **-4,14/-5,6/-6,08** |
| | I_100_15_S_1-49_1 | 2,37/0,07/-0,55 | **-0,24/-2,44/-1,68** | 0,86/-0,21/-0,65 |
| | I_100_15_S_1-9_1 | -0,49/-2,06/**-3,68** | **-1,57/-2,7**/-3,14 | -1,52/-2,65/-3,53 |
| | I_100_15_S_1-99_1 | 0,92/-0,89/-3,25 | **-2,79/-3,41/-4,88** | -0,62/-2,17/-3,2 |
| | I_100_20_S_1-124_1 | 0,5/-3,77/-4,77 | -3,53/**-6,27/-6,7** | **-4,17**/-5,97/-6,63 |
| | I_100_20_S_1-49_1 | -4,76/-6,93/-6,89 | **-6,98/-8,62/-9,73** | -5,33/-7,69/-8,4 |
| | I_100_20_S_1-9_1 | 0,14/-1,88/-2,17 | **-2,03/-3,77/-3,99** | -1,01/-2,83/-3,19 |
| | I_100_20_S_1-99_1 | -0,12/-2,8/-3,62 | -2,76/-5,49/**-7,11** | **-4,51/-6,3**/-6,91 |
| | I_150_10_S_1-124_1 | 2,72/0,48/-1,54 | **-1,24**/-3,66/-3,92 | -0,59/**-3,66/-4,6** |
| | I_150_10_S_1-49_1 | 3,67/1,74/0,83 | 2,09/0,74/-0,28 | **-0,11/-1,58/-1,83** |
| | I_150_10_S_1-9_1 | 1,39/-0,25/-0,31 | 0,09/-0,38/-0,96 | **-0,52/-1,36/-1,54** |
| | I_150_10_S_1-99_1 | 6,71/3,49/2,51 | 2,09/0/-1,32 | **1,39/-1,31/-2,39** |
| | I_150_15_S_1-124_1 | 2,43/0,32/-1,09 | -2,78/-4,32/-4,76 | **-3,48/-5/-5,98** |
| | I_150_15_S_1-49_1 | 2,14/-0,58/-1,23 | **-1/-2,91**/-2,96 | -0,42/-2,63/**-3,17** |
| | I_150_15_S_1-9_1 | -1,06/-1,99/-3,06 | **-3,44/-4,6/-4,5** | -2,71/-3,99/-4,33 |
| | I_150_15_S_1-99_1 | 4,81/0,85/0,13 | -0,23/-1,38/-3,83 | **-1,74/-3,64/-4,34** |
| | I_150_20_S_1-124_1 | -1,37/-4,56/-7,35 | **-8,34/-9,29/-11,82** | -6,9/-8,56/-9,27 |
| | I_150_20_S_1-49_1 | -6,28/-9,19/-10,29 | **-10,06/-11,62/-12,98** | -8,41/-9,64/-10,78 |
| | I_150_20_S_1-9_1 | -1,6/-4,71/-6,31 | **-6,76**/-7,11/-7,69 | -6,58/**-7,73/-8,76** |
| | I_150_20_S_1-99_1 | 4,12/0,27/-0,73 | -2,55/-4,66/-5,28 | **-5,5/-7,05/-7,7** |
| Balanced | 80on4Rp50Rs50_1 | **5,43/5,12/4,96** | 5,68/5,2/5,17 | 5,71/5,32/5,1 |
| | 80on6Rp50Rs50_1 | 7,82/7,32/7,07 | 7,6/7,24/6,95 | **7,32/6,92/6,75** |
| | 80on10Rp50Rs50_1 | 8,15/7,36/7,08 | 7,88/7,49/7,11 | **7,83/7,17/7,07** |
| | 100on4Rp50Rs50_1 | 5,35/4,86/4,85 | 5,39/5,11/5,01 | **5,29/4,85/4,67** |
| | 100on6Rp50Rs50_1 | 6,13/5,66/5,54 | 5,71/5,6/5,38 | **5,55/5,39/5,26** |
| | 100on10Rp50Rs50_1 | 8,02/7,51/7,27 | **7,52**/7,35/7,06 | 7,56/**7,2/6,96** |
| P-Dominat | 80on4Rp50Rs50_1 | **3,05/2,84/2,7** | 3,27/3,07/2,94 | 3,16/2,88/2,78 |
| | 80on6Rp50Rs50_1 | 6,26/5,95/5,68 | 6,07/5,73/5,64 | **5,89/5,67/5,51** |
| | 80on10Rp50Rs50_1 | 4,97/4,64/4,45 | 4,95/4,75/4,66 | **4,78/4,42/4,32** |
| | 100on4Rp50Rs50_1 | 2,9/**2,67/2,56** | **3,04**/2,91/2,67 | 3,1/2,77/2,63 |
| | 100on6Rp50Rs50_1 | 4,7/4,48/4,35 | 4,53/4,33/4,19 | **4,46/4,23/4,12** |
| | 100on10Rp50Rs50_1 | 4,41/4,13/4,06 | 4,37/4,15/4,02 | **3,96/3,74/3,64** |
| S-Dominat | 80on4Rp50Rs50_1 | **3,2**/2,93/2,83 | 3,44/3,06/2,93 | 3,22/**2,92/2,82** |
| | 80on6Rp50Rs50_1 | 6,18/5,89/5,72 | 6,27/5,94/5,7 | **6,12/5,81/5,64** |
| | 80on10Rp50Rs50_1 | 4,87/4,49/4,41 | 4,92/4,62/4,41 | **4,65/4,45/4,36** |
| | 100on4Rp50Rs50_1 | 2,88/2,65/**2,47** | 3,15/2,97/2,8 | **2,86/2,62**/2,5 |
| | 100on6Rp50Rs50_1 | 4,31/4,16/4,06 | 4,44/4,14/3,89 | **4,23/3,88/3,79** |
| | 100on10Rp50Rs50_1 | 4,43/3,99/3,83 | 4,4/4,06/3,98 | **4,19/3,94/3,87** |
| **Média Geral** | | 2,35/0,7/-0,02 | 0,36/-0,58/-1,11 | **0,06/-0,97/-1,4** |

[1] Executados em um Intel Core i5 3.0 GHz, 8 GB de RAM, 30 execuções para cada problema-teste

nível de significância de 5% que a amostra provém de uma população normal. Assim, podemos aplicar o ANOVA.

Para avaliar se existe diferença estatística entre algoritmos através dos valores de $RPD^{avg}$, foi aplicada a análise de variância (ANOVA) (Montgomery, 2007), com 95% de confiança ($threshold = 0,05$), tendo-se encontrado $p = 0,00332$.

Como $p < threshold$ existem diferenças estatísticas nos valores de $RPD^{avg}$. Para identificá-las, foi realizado o teste de Tukey HSD (Montgomery, 2007) com nível de confiança de 95% e $threshold = 0,05$. Na Tabela 5.3 são apresentadas as diferenças nos valores de $RPD^{avg}$ (diff), o limite inferior (lwr), o limite superior (upr) e o valor $p$ para cada par de algoritmos.

**Tabela 5.3:** Resultados para o teste Tukey HSD dos algoritmos HIVP, GIAP e AIRP.

| Algoritmos | diff | lwr | upr | p |
|---|---|---|---|---|
| GIAP-AIRP | 0,3300617 | -0,9823661 | 1,642490 | 0,8248884 |
| HIVP-AIRP | 1,7830864 | 0,4706586 | 3,095514 | 0,0042584 |
| HIVP-GIAP | 1,4530247 | 0,1405968 | 2,765453 | 0,0257615 |

De acordo com a Tabela 5.3 é possível observar que o algoritmo AIRP se diferencia estatisticamente do HIVP e que o algoritmo GIAP também diferencia estatisticamente do HIVP, porque em ambos os casos os valores de $p$ são menores que o *threshold*. O algoritmo GIAP não se diferencia estatisticamente do AIRP, pois o valor de $p$ é maior que o *threshold*.

O Gráfico 5.2 ilustra os resultados do teste Tukey HSD. Ele mostra que a maior diferença estatística é entre os algoritmos AIRP e HIVP, o algoritmo GIAP diferencia um pouco menos do HIVP. O algoritmo AIRP não se diferencia estatisticamente do GIAP, porque a reta corta o eixo das abscissas no valor 0.

Foi realizado, também, um teste de probabilidade empírica com os três algoritmos objetivando uma análise baseada no tempo gasto por cada algoritmo para encontrar um valor alvo. O teste de probabilidade empírica é importante para comparar diferentes algoritmos e tem sido amplamente utilizado (Feo et al., 1994).

Para realizar o teste de probabilidade empírica foi selecionado o problema-teste I_100_15_S_1-124_1 de (SOA, 2011), que contêm 100 tarefas e 15 máquinas. O valor alvo

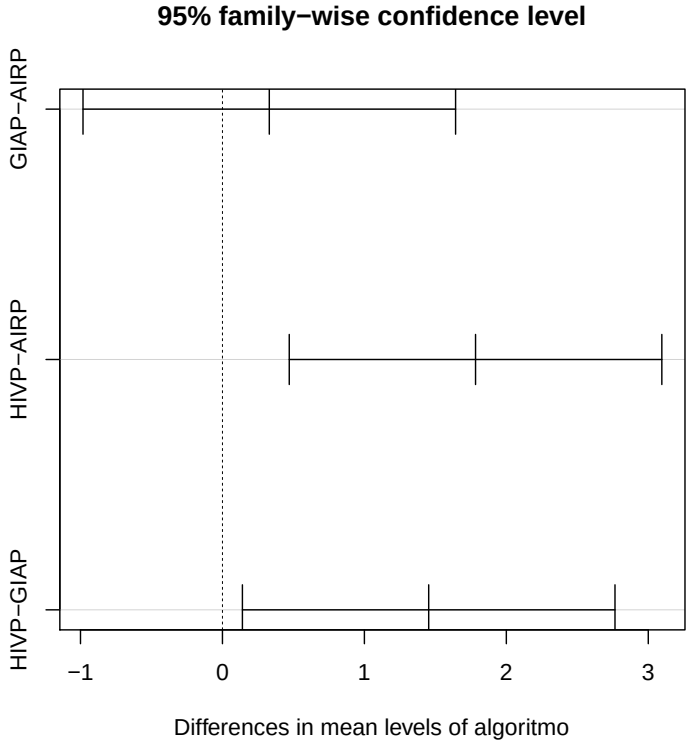



**Figura 5.2:** Gráfico de resultados para o teste Tukey HSD dos algoritmos HIVP, GIAP e AIRP.

escolhido foi de 5% do valor da melhor solução conhecida, disponibilizado por (Vallada e Ruiz, 2011). Todos os algoritmos foram executados 100 vezes, e a cada instante que o valor alvo era alcançado o tempo gasto, em segundos, foi registrado e o algoritmo interrompido. A seguir, os tempos registrados foram ordenados de forma crescente. Para cada execução $i = 1, 2, ..., 100$ existe um tempo $t_i$ e uma probabilidade empírica associada, dada por $p_i = (i - 0,5)/100$. Na Figura 5.3 são plotados os pontos $t_i \times p_i$ relativos ao comportamento de cada algoritmo. As curvas dos três algoritmos foram sobrepostas para facilitar a comparação.

A partir do gráfico 5.3 pode-se observar que o algoritmo AIRP superou os algoritmos HIVP e GIAP, encontrando o alvo mais rapidamente. O algoritmo GIAP encontrou o alvo mais rapidamente que o algoritmo HIVP na maioria das vezes.

Através dos resultados obtidos da primeira bateria de testes o algoritmo AIRP foi o que obteve os melhores resultados, com isso, ele foi considerado o melhor algoritmo entre HIVP, GIAP e AIRP.

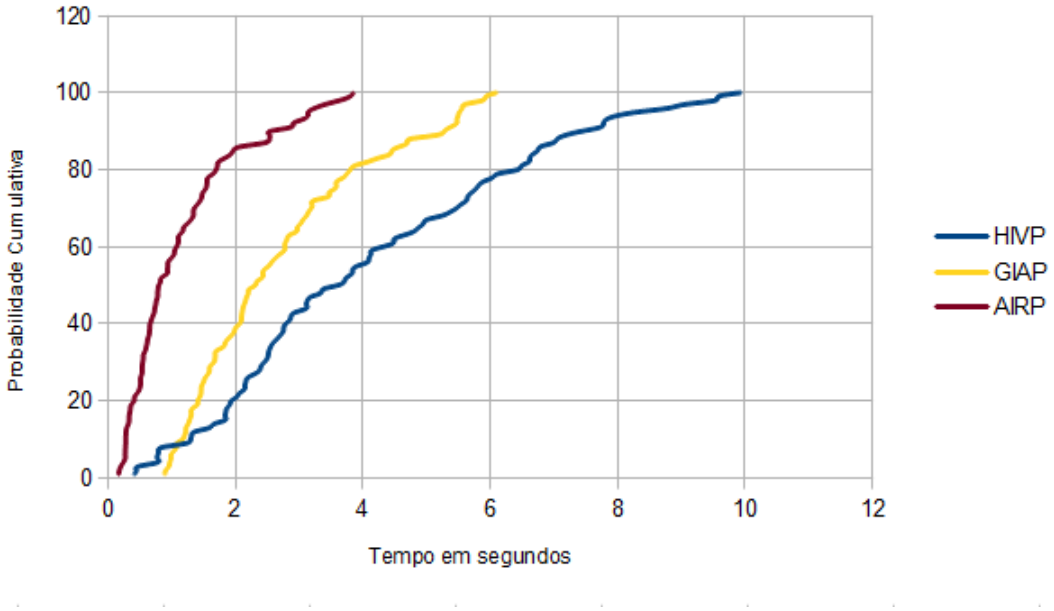



**Figura 5.3:** Probabilidade empírica - Bateria 1

### 5.3.2 Segunda Bateria de Testes

Na segunda bateria de testes foram realizados experimentos computacionais usando o algoritmo AIRP e comparando seus resultados com os de outros algoritmos da literatura. Esse algoritmo foi escolhido por obter os melhores resultados na primeira bateria de testes.

Nos experimentos computacionais foram utilizados 360 problemas-teste grandes de (SOA, 2011) e 270 problemas-teste grandes de (SchedulingResearch, 2005). Na subseção 5.3.2.1 são mostrados os resultados dos experimentos usando os problemas-teste de (SOA, 2011) e na subseção 5.3.2.2 usando os problemas-teste de (SchedulingResearch, 2005).

#### 5.3.2.1 Experimentos com os Problemas-teste de (SOA, 2011)

Nesta subseção o algoritmo AIRP é testado para 360 problemas-teste grandes de (SOA, 2011). Eles combinam 50, 100 e 150 tarefas e 10, 15 e 20 máquinas, sendo que cada combinação possui 40 problemas-teste.

Os resultados do algoritmo AIRP são comparados com os algoritmos GA2, de (Val-

lada e Ruiz, 2011), e o GIVMP, de (Haddad et al., 2012).

O critério de parada adotado é o tempo de processamento, definido pela Eq. 5.1 da subseção 5.2. Adotou-se três valores para o parâmetro $t$, sendo eles: 10, 30 e 50. Os valores de $t$ são os mesmos de (Vallada e Ruiz, 2011) e (Haddad et al., 2012).

Dado seu caráter estocástico, os algoritmos AIRP e GIVMP foram executados 30 vezes para cada problema-teste e para cada valor de $t$. Já em Vallada e Ruiz (2011), o algoritmo GA2 foi executado 5 vezes para cada problema-teste e para cada valor de $t$. A métrica usada para comparação entre os algoritmos é a média do desvio percentual relativo $RPD_i^{avg}$ dos valores $RPD_i$ encontrados. O $RPD_i$ é definido pela Eq. 5.2. Os valores adotados para o $f_i^*$ são os dos melhores resultados disponibilizados por (Vallada e Ruiz, 2011) em (SOA, 2011).

A Tabela 5.4 compara os resultados dos algoritmos AIRP, GIVMP e GA2 com relação à média do desvio percental relativo para 9 combinações de problemas-teste, com 40 cada. Os resultados estão agrupados para cada combinação de tarefas por máquinas. O valor apresentado em cada célula é a média do $RPD_i^{avg}$ para cada combinação. Para cada conjunto de problemas-teste são apresentados três valores de $RPD_i^{avg}$ separados por '/'. Esta separação representa os resultados dos testes para cada valor de $t$, na ordem $t = 10/30/50$. Os valores negativos indicam que os resultados alcançados pelos algoritmos superaram os melhores valores encontrados por (Vallada e Ruiz, 2011) nos problemas-teste de (SOA, 2011).

Os valores que representam os melhores resultados médios estão destacados em negrito. Pode-se observar que o algoritmo AIRP obteve os melhores resultados médios em todas as combinações de problemas-teste e também na média geral, superando com larga vantagem os algoritmos GA2 e GIVMP da literatura.

A Figura 5.4 apresenta um box plot para os valores $RPD^{avg}$ dos algoritmos. Tendo por base os valores do $RPD^{avg}$, observa-se que o algoritmo AIRP supera em 100% dos casos o algoritmo GA2, e na grande maioria dos casos o algoritmo GIVMP.

Antes de verificar se existem diferenças estatísticas entre os algoritmos foi aplicado o teste Shapiro-Wilk (Shapiro e Wilk, 1965) para verificar se a amostra satisfaz ao teste de normalidade. De acordo com esse teste, com nível de significância de 5%, $W = 0,985$ e $p = 0,4628$. Tem-se $p = 0,4628 > 0,05$, então podemos afirmar com intervalo de confiança de 95% que a amostra provém de uma população normal. Assim, podemos aplicar o ANOVA.

**Tabela 5.4:** Média dos *RPD*s dos algoritmos AIRP, GIVMP e GA2 para $t$ = 10/30/50.

| Combinação de Problemas-teste | AIRP[1] | GIVMP[2] | GA2[2] |
|---|---|---|---|
| 50 × 10 | **0,69/-0,51/-0,99** | 1,79/0,58/-0,11 | 7,79/6,92/6,49 |
| 50 × 15 | **-2,9/-4,09/-4,63** | -0,39/-2,10/-2,57 | 12,25/8,92/9,20 |
| 50 × 20 | **-4,14/-5,26/-5,8** | 3,92/1,43/-0,05 | 11,08/8,04/9,57 |
| 100 × 10 | **1,58/-0,02/-0,59** | 2,66/0,99/0,42 | 15,72/6,76/5,54 |
| 100 × 15 | **-1,66/-3,22/-3,94** | -0,11/-1,88/-2,74 | 22,15/8,36/7,32 |
| 100 × 20 | **-3,68/-5,51/-6,05** | -1,66/-3,65/-4,60 | 22,02/9,79/8,59 |
| 150 × 10 | **1,29/-0,46/-1,14** | 2,11/0,14/-0,70 | 18,40/5,75/5,28 |
| 150 × 15 | **-0,9/-2,61/-3,25** | 1,08/-1,03/-1,78 | 24,89/8,09/6,80 |
| 150 × 20 | **-3,95/-5,63/-6,34** | -1,91/-3,90/-4,75 | 22,63/9,53/7,40 |
| Média Geral | **-1,52/-3,03/-3,64** | 0,83/-1,05/-1,87 | 17,44/8,02/7,35 |

[1] Executados em um Intel Core i5 3.0 GHz, 8 GB de RAM

[2] Executados em um Intel Core 2 Duo 2.4 GHz, 2 GB de RAM

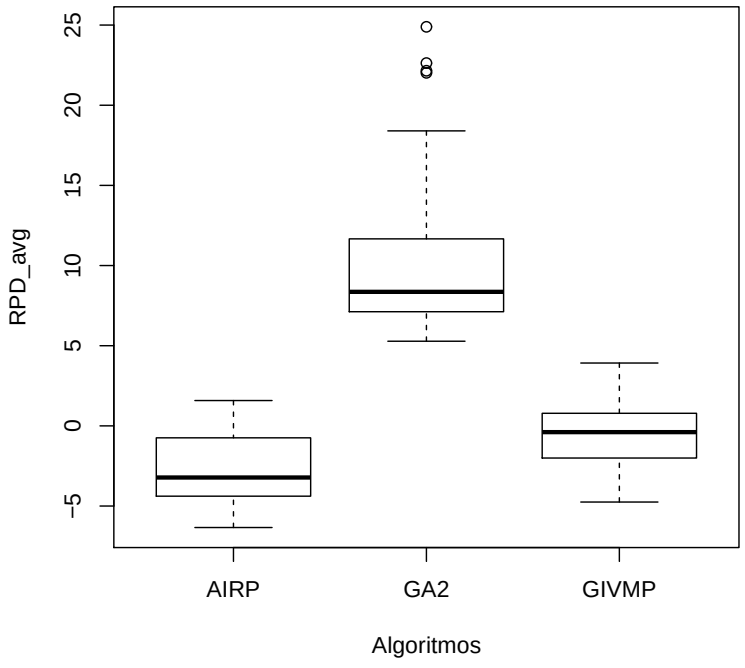



**Figura 5.4:** Box plot dos algoritmos AIRP, GIVMP e GA2.

Para avaliar se existe diferença estatística entre algoritmos através dos valores de $RPD^{avg}$ de cada combinação, foi aplicada a análise de variância (ANOVA) (Montgomery, 2007), com 95% de confiança ($threshold = 0,05$), tendo-se encontrado $p = 2 \times 10^{-16}$.

Como $p < threshold$ existem diferenças estatísticas nos valores de $RPD^{avg}$. Para identificá-las, foi realizado o teste de Tukey HSD (Montgomery, 2007) com nível de confiança de 95% e $threshold = 0,05$. Na Tabela 5.5 são apresentadas as diferenças nos valores de $RPD^{avg}$ (diff), o limite inferior (lwr), o limite superior (upr) e o valor $p$ para cada par de algoritmos.

**Tabela 5.5:** Resultados para o teste Tukey HSD dos algoritmos AIRP, GIVMP e GA2.

| Algoritmos | diff | lwr | upr | p |
|---|---|---|---|---|
| GA2-AIRP | 13,566296 | 11,035430 | 16,097163 | 0,0000000 |
| GIVMP-AIRP | 2,033333 | -0,497533 | 4,564200 | 0,1399632 |
| GIVMP-GA2 | -11,532963 | -14,063829 | -9,002097 | 0,0000000 |

Pela Tabela 5.5 é possível observar que o algoritmo AIRP se difere estatisticamente do GA2, porque o valor de $p$ é menor que o *threshold*. Assim como o algoritmo GIVMP se difere estatiscamente do algoritmo GA2. Por outro lado, o algoritmo AIRP não se difere estatisticamente do algoritmo GIVMP porque o valor de $p$ não é menor que o *threshold*.

O Gráfico 5.5 ilustra os resultados do teste Tukey HSD. Ele mostra que a maior diferença estatística existente é na comparação entre os algoritmo AIRP e GA2. E que a menor diferença é entre os algoritmos AIRP e GIVMP, e que eles não são diferentes estatisticamente porque a reta corta o eixo no ponto 0.

Os resultados completos dos experimentos computacionais realizados na segunda bateria de testes nos problemas-teste de (SOA, 2011) estão disponibilizados no endereço `http://www.decom.ufop.br/prof/marcone/projects/upmsp/Experimentos_UPMSPST_Dissert_Bateria2_SOA.ods`

Pelos resultados encontrados nos experimentos realizados nos problemas-teste de (SOA, 2011) da segunda bateria de testes pode-se constatar a supremacia, comprovada estatisticamente, do algoritmo AIRP sobre o algoritmo GA2. Por outro lado, pode-se também perceber que o algoritmo AIRP supera o algoritmo GIVMP com respeito aos

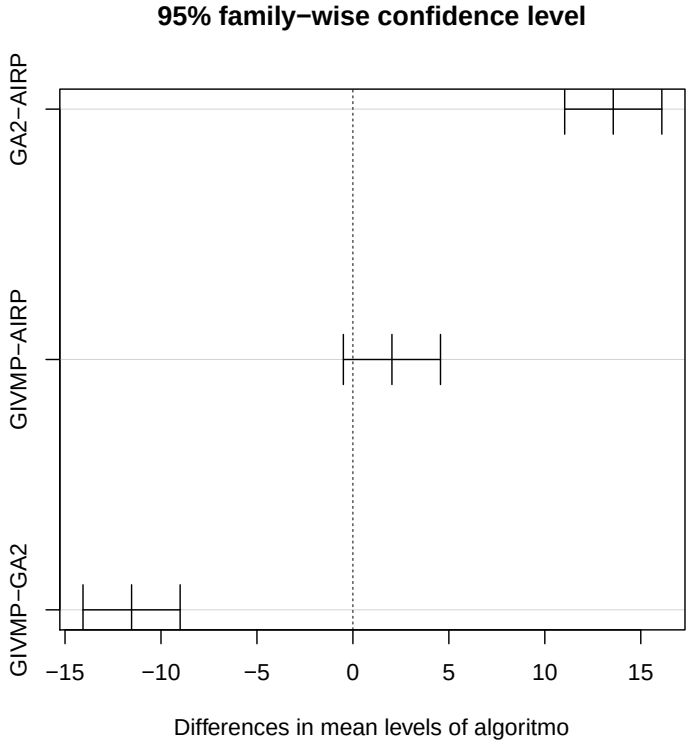



**Figura 5.5:** Gráfico de resultados para o teste Tukey HSD dos algoritmos AIRP, GIVMP e GA2.

melhores valores encontrados, assim como na média desses valores; no entanto, não há comprovação estatística da superioridade desse algoritmo em relação àquele.

#### 5.3.2.2 Experimentos com os Problemas-teste de (SchedulingResearch, 2005)

Esta subseção é utilizada para testar o algoritmo AIRP para 270 problemas-teste grandes de (SchedulingResearch, 2005). Eles combinam 80 e 100 tarefas e 4, 6 e 8, com cada combinação possuindo 45 problemas-teste separados em três grupos. Cada grupo utiliza um método diferente para gerar cada problema-teste, conforme mencionado na subseção 5.1.2.

Os resultados encontrados para o algoritmo AIRP são comparados com os dos seguintes algoritmos da literatura: *i*) Meta-RaPS de (Rabadi et al., 2006); *ii*) ACO de (Arnaout et al., 2010); *iii*) RSA de (Ying et al., 2010); *iv*) MVND de (Fleszar et al., 2011); *v*) e GIVMP de (Haddad et al., 2012)

O critério de parada adotado nesta subseção é diferente dos critérios utilizados até o momento, por não existir nenhuma padronização deste critério nos trabalhos da literatura que tratam os problemas-teste de (SchedulingResearch, 2005). Como o algoritmo AIRP utiliza o tempo de execução como critério de parada, foram utilizados os tempos médios obtidos pelo algoritmo MVND (Fleszar et al., 2011) para cada combinação de número de tarefas e número de máquinas. No trabalho de (Haddad et al., 2012), que propôs o algoritmo GIVMP, o critério de parada é também o tempo médio do MVND.

O algoritmo AIRP foi executado 30 vezes para cada problema-teste. A métrica usada para comparar os algoritmos foi o do melhor desvio percentual relativo $RPD_i^{best}$ dos melhores valores $RPD_i$ encontrados nas 30 execuções. O $RPD_i$ é definido pela Eq. (5.2). O $f_i^*$ adotado é o limite inferior (LB) definido pelas Eqs. (5.3), (5.4), e (5.5).

A Tabela 5.6 compara os resultados dos algoritmos Meta-RaPS, ACO, RSA, MVND, GIVMP e AIRP com relação ao melhor desvio percental relativo para 6 combinações de problemas-teste, com 45 cada. Os resultados estão agrupados para cada combinação de tarefas e máquinas. É apresentado o melhor desvio percentual relativo e os tempos médios, em segundos, que cada algoritmo foi executado, com exceção do Meta-RaPS que não divulgou seus tempos médios.

Os valores em negrito da Tabela 5.6 mostram os melhores resultados. É possível observar que o algoritmo MVND é o de melhor desempenho, pois seus resultados superaram os de todos os demais algoritmos para a média dos $RPD_i^{best}$. Por outro lado, o algoritmo AIRP encontrou melhores resultados que os dos algoritmos Meta-RaPS, ACO e RSA em todas as combinações. Já em comparação com o algoritmo GIVMP, o AIRP superou em 2 combinações e na média geral.

Um gráfico box plot é apresentado na Figura 5.6 para os valores $RPD_i^{best}$ da Tabela 5.6.

Pode-se verificar que o algoritmo MVND apresentou os melhores resultados superando todos os outros algoritmos. Os algoritmos AIRP e GIVMP encontraram melhores resultados que os algoritmos Meta-RaPS, ACHO e RSA mesmo utilizando um tempo pequeno para execução. O algoritmo AIRP foi um pouco melhor que o algoritmo GIVMP.

Antes de verificar se existem diferenças estatísticas entre os algoritmos foi aplicado o teste Shapiro-Wilk (Shapiro e Wilk, 1965) para avaliar se a amostra provém de uma distribuição normal. Com nível de significância de 5%, obteve-se no teste Shapiro-Wilk $W = 0,9881$ e $p = 0,9593$. Tem-se $p = 0,9593 > 0,05$, então podemos afirmar com

**Tabela 5.6:** Médias dos $RPD_{best}$ dos algoritmos Meta-RaPS, ACO, RSA, MVND e GIVMP

| | **Meta-RaPS**[1] | **ACO**[2] | | **RSA**[3] | | **MVND**[4] | | **GIVMP**[5] | | **AIRP**[6] | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Problemas-teste** | $RPD_{best}$ | $RPD_{best}$ | **Tempo (s)** | $RPD_{best}$ | **Tempo (s)** | $RPD_{best}$ | **Tempo (s)** | $RPD_{best}$ | **Tempo (s)** | $RPD_{best}$ | **Tempo (s)** |
| **80 x 4** | 4,41 | 3,26 | 311,4 | 3,16 | 68,5 | **1,98** | 8,2 | 2,70 | 8,2 | 2,83 | 8,2 |
| **80 x 6** | 6,19 | 7,29 | - | 5,95 | 67,2 | **4,66** | 13 | 5,37 | 13 | 5,28 | 13 |
| **80 x 10** | 5,89 | 8,01 | - | 5,76 | 101,1 | **3,97** | 6,5 | 5,57 | 6,5 | 5,04 | 6,5 |
| **100 x 4** | 3,39 | 2,91 | 557,4 | 2,92 | 120,9 | **1,63** | 19,2 | 2,28 | 19,2 | 2,58 | 19,2 |
| **100 x 6** | 4,79 | 5,14 | 544,2 | 4,38 | 144,7 | **2,81** | 21,4 | 3,86 | 21,4 | 4,3 | 21,4 |
| **100 x 10** | 6,43 | 6,85 | - | 5,07 | 220,3 | **3,28** | 14 | 4,88 | 14 | 4,46 | 14 |
| **Média Geral** | 5,18 | 5,58 | 471,00 | 5,58 | 120,45 | **3,06** | 13,72 | 4,11 | 13,72 | 4,08 | 13,72 |

[1]Testes realizados em um PC Pentium IV com 1.7 GHz
[2]Testes realizados em um PC Pentium IV com 2GB de RAM
[3]Testes realizados em um PC Pentium IV com 1.5 GHz e 512 MB de RAM
[4]Testes realizados em um Dell Optiplex 760, Intel Core 2 Quad Q8200 com 2.33 GHz
[5]Testes realizados em um Intel Core 2 Duo 2.4 GHz, 2 GB de RAM
[6]Testes realizados em um Intel Core i5 3.0 GHz, 8 GB de RAM

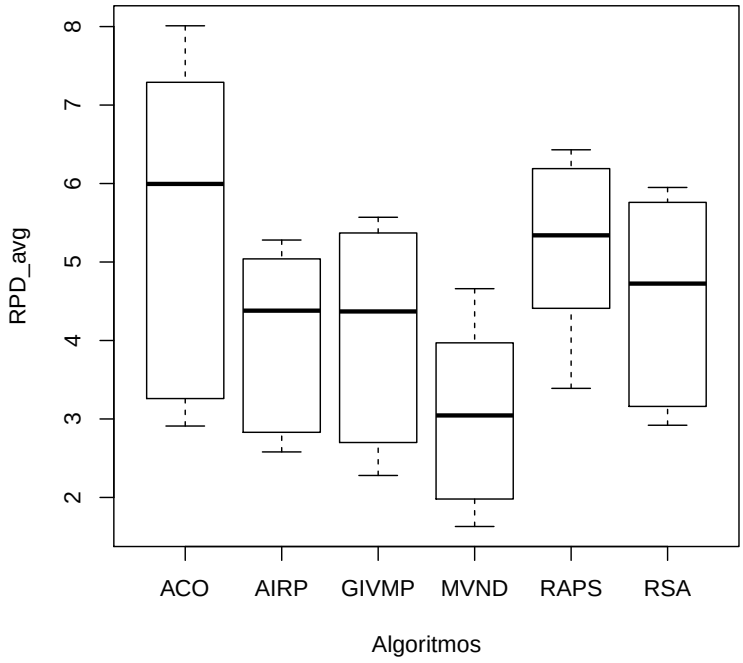



**Figura 5.6:** Box plot dos algoritmos ACO, AIRP, GIVMP, MVND, RAPS (Meta-RaPS) e RSA.

intervalo de confiança de 95% que a amostra provém de uma população normal. Assim, podemos aplicar o ANOVA.

Para verificar se existem diferenças estatísticas entre os resultados $RPD_i^{best}$ dos algoritmos, foi aplicada a análise de variância (ANOVA) (Montgomery, 2007), com 95% de confiança ($threshold = 0,05$), tendo-se encontrado $p = 0,0645$. O valor de $p$ não é menor que o $threshold$, definindo que não existe diferença estatística entre os resultados dos $RPD_i^{best}$ dos algoritmos.

Os resultados completos dos experimentos computacionais realizados na segunda bateria de testes nos problemas-teste de (SchedulingResearch, 2005) estão disponibilizados em `http://www.decom.ufop.br/prof/marcone/projects/upmsp/Experimentos_UPMSPST_Dissert_Bateria2_SCHE.ods`

### 5.3.3 Terceira Bateria de Testes

Na terceira bateria de testes o algoritmo híbrido AIRMP é testado e comparado com o melhor algoritmo da primeira bateria de testes, o AIRP. Para realizar os testes são utilizados 36 problemas-teste de (SOA, 2011) e 18 de (SchedulingResearch, 2005) usados na primeira bateria.

Na comparação entre esses dois algoritmos é adotado como critério de parada o tempo de processamento, definido pela Eq. 5.1 da subseção 5.2, com o valor de $t$ igual a 500. Foi definido um tempo bem maior porque o algoritmo AIRMP consome até 5 segundos a cada vez que é acionado um módulo de programação linear inteira mista para otimizar duas máquinas.

Cada um desses algoritmos foi executado 30 vezes para cada problema-teste. A métrica adotada para comparação entre os algoritmos é a média do desvio percentual relativo $RPD_i^{avg}$ dos valores de $RPD_i$ encontrados. O $RPD_i^{avg}$ é definido pela Eq. 5.2. Para os problemas-teste de (SOA, 2011) os $f_i^*$ adotados são as melhores soluções disponibilizadas por (Vallada e Ruiz, 2011) em (SOA, 2011). Já para os problemas-teste de (SchedulingResearch, 2005), os $f_i^*$ usados são os limites inferiores (LB) disponibilizados em (SchedulingResearch, 2005), determinados pelas Eqs. (5.3), (5.4) e (5.5) estabelecidos por (Al-Salem, 2004).

A Tabela 5.7 compara os resultados dos algoritmos AIRMP e AIRP com relação à média do desvio percental relativo. Os 36 problemas-teste selecionados de (SOA, 2011)

são todos do grupo *Large* e os 18 de (SchedulingResearch, 2005) são formados por 6 do grupo *Balanced*, 6 do grupo *P-Dominat* e 6 do grupo *S-Dominant*. Para cada conjunto de problemas-teste são apresentados os resultados encontrados pelo algoritmo AIRP para $t = 50$ e $t = 500$, e pelo AIRMP para $t = 500$. São apresentados os resultados de $t = 50$ para o AIRP visando a analisar a influência do tempo de processamento no valor do $RPD_i^{avg}$. Os valores negativos indicam que os resultados alcançados pelos algoritmos superaram os melhores valores encontrados por Vallada e Ruiz (2011) em seus experimentos para os problemas-teste de (SOA, 2011).

Na Tabela 5.7, os valores em negrito representam os melhores resultados para o $RPD_i^{avg}$. É possível observar que o algoritmo AIRP com $t = 500$ obteve os melhores resultados em 100% dos $RPD^{avg}$ e na média geral. O algoritmo AIRMP com $t = 500$ ficou em segundo lugar, e o AIRP com $t = 50$ ficou na última posição.

A Figura 5.7 apresenta um box plot para os valores $RPD^{avg}$ dos algoritmos. Tendo por base os valores do $RPD^{avg}$, observa-se que o algoritmo AIRP com $t = 500$ supera em 100% dos casos o AIRMP com $t = 500$ e o AIRP com $t = 50$. O algoritmo $AIRMP$ com $t = 500$ supera o AIRP com $t = 50$ na maioria dos casos e na média geral.

Foi também realizado o teste Shapiro-Wilk (Shapiro e Wilk, 1965) para verificar se a amostra satisfaz ao teste de normalidade, antes de analisar se existem diferenças estatísticas entre os algoritmos. Pelo teste Shapiro-Wilk, com nível de significância de 5%, obteve-se $W = 0,9261$ e $p = 0,2692$. Como $p = 0,2692 > 0,05$, então podemos afirmar com intervalo de confiança de 95% que a amostra provém de uma população normal. Assim, podemos aplicar o teste ANOVA.

Para avaliar se existem diferenças estatísticas entre os algoritmos pelos valores de $RPD^{avg}$, foi aplicada a análise de variância (ANOVA) (Montgomery, 2007), com 95% de confiança ($threshold = 0,05$), tendo-se encontrado $p = 0,03852$.

Como $p < threshold$ existem diferenças estatísticas nos valores de $RPD^{avg}$. Para encontrá-las, foi realizado o teste de Tukey HSD (Montgomery, 2007) com nível de confiança de 95% e $threshold = 0,05$. Na Tabela 5.8 são apresentadas as diferenças nos valores de $RPD^{avg}$ (diff), o limite inferior (lwr), o limite superior (upr) e o valor $p$ para cada par de algoritmos.

Pela Tabela 5.8 é possível observar que o algoritmo AIRP com $t = 500$ se difere estatisticamente do próprio AIRP com $t = 50$, porque o valor de $p$ é menor que o

**Tabela 5.7:** Média dos *RPD*s dos algoritmos AIRP e AIRMP.

| Grupos | Problemas-teste | AIRP_t_50[1] | AIRP_t_500[1] | AIRMP_t_500[1] |
|---|---|---|---|---|
| Large | I_50_10_S_1-124_1 | -4,56 | **-5,09** | -3,36 |
| | I_50_10_S_1-49_1 | 0,74 | **0,22** | 1,34 |
| | I_50_10_S_1-9_1 | -0,4 | **-0,75** | 0,75 |
| | I_50_10_S_1-99_1 | -3,36 | **-4,6** | -2,91 |
| | I_50_15_S_1-124_1 | -4,22 | **-5,29** | -4,04 |
| | I_50_15_S_1-49_1 | -4,58 | **-5,25** | -4,52 |
| | I_50_15_S_1-9_1 | 0,83 | **-1,57** | -0,18 |
| | I_50_15_S_1-99_1 | -11,41 | **-12,44** | -12,01 |
| | I_50_20_S_1-124_1 | -5,96 | **-7,56** | -6,92 |
| | I_50_20_S_1-49_1 | -7,69 | **-8,55** | -8,29 |
| | I_50_20_S_1-9_1 | 4,52 | **0,64** | 2,58 |
| | I_50_20_S_1-99_1 | -15,24 | **-16,19** | -15,65 |
| | I_100_10_S_1-124_1 | -0,8 | **-3,14** | -0,44 |
| | I_100_10_S_1-49_1 | 3,29 | **1,28** | 3,29 |
| | I_100_10_S_1-9_1 | -8,09 | **-9,52** | -8,22 |
| | I_100_10_S_1-99_1 | 2,94 | **0,33** | 2,91 |
| | I_100_15_S_1-124_1 | -6,08 | **-9,43** | -5,58 |
| | I_100_15_S_1-49_1 | -0,65 | **-3,2** | -1,1 |
| | I_100_15_S_1-9_1 | -3,53 | **-4,85** | -4,12 |
| | I_100_15_S_1-99_1 | -3,2 | **-6,29** | -3,9 |
| | I_100_20_S_1-124_1 | -6,63 | **-10,43** | -8 |
| | I_100_20_S_1-49_1 | -8,4 | **-10** | -8,44 |
| | I_100_20_S_1-9_1 | -3,19 | **-4,42** | -3,41 |
| | I_100_20_S_1-99_1 | -6,91 | **-9,51** | -7,48 |
| | I_150_10_S_1-124_1 | -4,6 | **-7,54** | -2,79 |
| | I_150_10_S_1-49_1 | -1,83 | **-3,44** | -2,27 |
| | I_150_10_S_1-9_1 | -1,54 | **-2,05** | -1,67 |
| | I_150_10_S_1-99_1 | -2,39 | **-4,51** | -2,79 |
| | I_150_15_S_1-124_1 | -5,98 | **-8,17** | -6,97 |
| | I_150_15_S_1-49_1 | -3,17 | **-4,43** | -3,68 |
| | I_150_15_S_1-9_1 | -4,33 | **-5,84** | -4,91 |
| | I_150_15_S_1-99_1 | -4,34 | **-6,82** | -5,19 |
| | I_150_20_S_1-124_1 | -9,27 | **-12,41** | -11,44 |
| | I_150_20_S_1-49_1 | -10,78 | **-12,75** | -11,62 |
| | I_150_20_S_1-9_1 | -8,76 | **-10,04** | -9,11 |
| | I_150_20_S_1-99_1 | -7,7 | **-10,08** | -8,4 |
| Balanced | 80on4Rp50Rs50_1 | 5,1 | **4,46** | 5,35 |
| | 80on6Rp50Rs50_1 | 6,75 | **6,25** | 7,13 |
| | 80on10Rp50Rs50_1 | 7,07 | **6,53** | 7,37 |
| | 100on4Rp50Rs50_1 | 4,67 | **4,11** | 4,85 |
| | 100on6Rp50Rs50_1 | 5,26 | **4,9** | 5,44 |
| | 100on10Rp50Rs50_1 | 6,96 | **6,48** | 7,33 |
| P-Dominat | 80on4Rp50Rs50_1 | 2,78 | **2,52** | 2,87 |
| | 80on6Rp50Rs50_1 | 5,51 | **5,23** | 5,63 |
| | 80on10Rp50Rs50_1 | 4,32 | **3,99** | 4,4 |
| | 100on4Rp50Rs50_1 | 2,63 | **2,22** | 2,69 |
| | 100on6Rp50Rs50_1 | 4,12 | **3,73** | 4,14 |
| | 100on10Rp50Rs50_1 | 3,64 | **3,39** | 3,75 |
| S-Dominat | 80on4Rp50Rs50_1 | 2,82 | **2,4** | 2,85 |
| | 80on6Rp50Rs50_1 | 5,64 | **5,21** | 5,68 |
| | 80on10Rp50Rs50_1 | 4,36 | **3,92** | 4,39 |
| | 100on4Rp50Rs50_1 | 2,5 | **2,16** | 2,52 |
| | 100on6Rp50Rs50_1 | 3,79 | **3,49** | 3,78 |
| | 100on10Rp50Rs50_1 | 3,87 | **3,41** | 3,93 |
| **Média Geral** | | -1,4 | **-2,76** | -1,54 |

[1] Executados em um Intel Core i5 3.0 GHz, 8 GB de RAM, 30 execuções para cada problema-teste

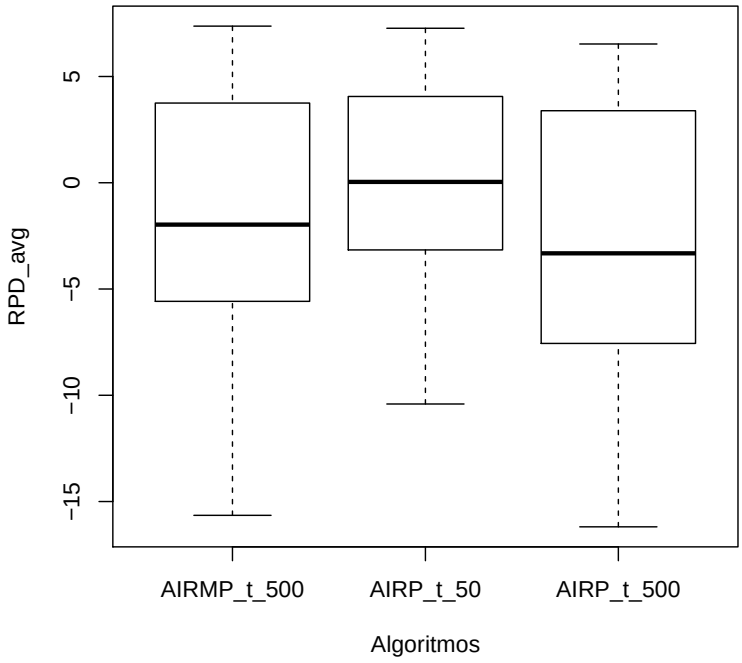



**Figura 5.7:** Box plot dos algoritmos AIRP com $t = 50$, AIRP com $t = 500$ e AIRMP com $t = 500$.

**Tabela 5.8:** Resultados para o teste Tukey HSD dos algoritmos AIRP com $t = 50$, AIRP com $t = 500$ e AIRMP com $t = 500$.

| Algoritmos | diff | lwr | upr | p |
|---|---|---|---|---|
| AIRP_t_50 - AIRMP_t_500 | 1.546481 | -0.9810457 | 4.0740086 | 0.3191567 |
| AIRP_t_500 - AIRMP_t_500 | -1.200926 | -3.7284531 | 1.3266012 | 0.5006628 |
| AIRP_t_500 - AIRP_t_50 | -2.747407 | -5.2749346 | -0.2198803 | 0.0295457 |

*threshold*. O algoritmo AIRP com $t = 50$ não se difere estatisticamente do AIRMP com $t = 500$, e o AIRP com $t = 500$ também não se difere estatisticamente do AIRMP com $t = 500$, porque em ambos os casos o valor de $p$ não é menor que o *threshold*.

O Gráfico 5.8 ilustra os resultados do teste Tukey HSD. Ele mostra que a maior diferença estatística existente é na comparação entre os algoritmos AIRP com $t = 500$ e o próprio AIRP com $t = 50$. Por outro lado, não há diferenças estatísticas entre os pares AIRP com $t = 500$ e AIRMP com $t = 500$, e AIRP com $t = 50$ e AIRMP com $t = 500$, porque a reta corta o eixo no ponto 0.

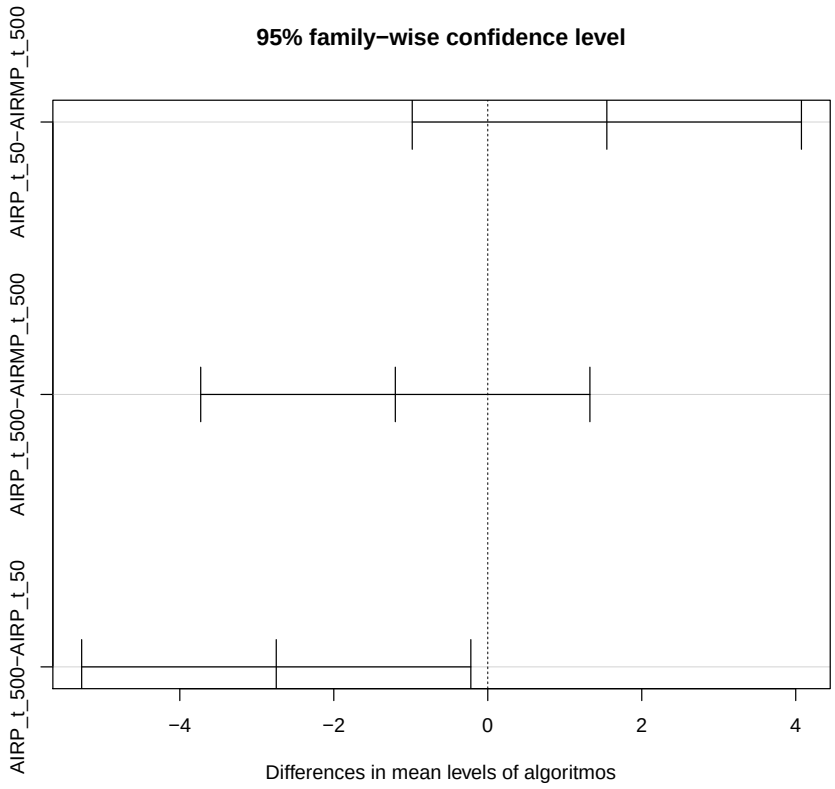



**Figura 5.8:** Gráfico de resultados para o teste Tukey HSD dos algoritmos AIRP com $t = 50$, AIRP com $t = 500$ e AIRMP com $t = 500$.

Pela terceira bateria de testes foi possível observar a supremacia do algoritmo AIRP com $t = 500$ sobre o AIRP com $t = 50$ e o AIRMP com $t = 500$ nos problemas-teste analisados. Por outro lado, foi observado que o módulo de programação matemática acoplado ao AIRMP não o conduziu a melhores resultados. Por outro lado, e como era de se imaginar, foi possível melhorar os resultados do algoritmo AIRP com $t = 50$ com um tempo maior de execução.

Os resultados completos dos experimentos da terceira bateria de testes estão disponibilizados no endereço `http://www.decom.ufop.br/prof/marcone/projects/upmsp/Experimentos_UPMSPST_Dissert_Bateria3.ods`

# Capítulo 6

# Conclusões e Trabalhos Futuros

## 6.1 Conclusões

Este trabalho tratou o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência (UPMSPST), tendo como objetivo a minimização do *makespan*.

O UPMSPST é um problema de grande importância por pertencer à classe de problemas $\mathcal{NP}$-difíceis e pelo fato de estar presente em indústrias de diferentes áreas, como téxteis e químicas.

Existe grande dificuldade na resolução do UPMSPST na otimalidade em tempos aceitáveis por meio de métodos de programação matemática, dito exatos, ficando a aplicação dessa classe de métodos restrita a problemas-teste de pequeno porte. Por isso, as heurísticas são as abordagens mais utilizadas na literatura para sua resolução.

Inicialmente, este trabalho analisou duas formulações de programação linear inteira mista existentes na literatura para o problema específico, no caso, a formulação UPMSPST-VR, de (Vallada e Ruiz, 2011), e a UPMSPST-RA, de (Rabadi et al., 2006). A partir dessa análise e de duas outras formulações para problemas similares, no caso, as formulações de (Rosa et al., 2009) e (Ravetti et al., 2007), foram propostas duas novas formulações matemáticas para tratar o UPMSPST. A primeira, denominada UPMSPST-CV, é uma modificação da formulação de UPMSPST-VR, com adição de restrições do problema do caixeiro viajante assimétrico. A segunda, denominada UPMSPST-IT, é uma formulação indexada no tempo. Nos testes realizados, observou-se que a formulação

UPMSPST-VR obteve os melhores resultados quando comparada com todas as outras.

Posteriormente, foram desenvolvidos três algoritmos heurísticos, denominados HIVP, GIAP e AIRP; bem como um algoritmo híbrido, denominado AIRMP, para resolver o UPMSPST.

O algoritmo HIVP combina os procedimentos heurísticos *Heuristic-Biased Stochastic Sampling* (HBSS), *Iterated Local Search* (ILS), *Random Variable Neighborhood Descent* (RVND) e *Path Relinking* (PR). O procedimento HBSS é usado para gerar uma solução inicial. O ILS é usado para refinar essa solução inicial e utiliza o RVND como método de busca local. O RVND utiliza três métodos de refinamento, os quais são acionados em ordem aleatória a cada chamada do procedimento. Periodicamente são realizadas intensificações e diversificações durante a exploração do espaço de busca utilizando o procedimento PR com a estratégia *Backward*. Na aplicação do PR usa-se a posição de uma tarefa como atributo.

No GIAP o funcionamento é semelhante ao do algoritmo HIVP, diferenciando-se deste na geração da solução inicial e da busca local do ILS. Diferentemente do HIVP, no GIAP a solução inicial é gerada por um procedimento construtivo baseado no *Greedy Randomized Adaptive Search* (GRASP) e o ILS utiliza o prodecimento *Adaptive Local Search* (ALS) como método de busca local. O procedimento ALS utiliza dois métodos de refinamento e um de perturbação. Esses métodos são aplicados de acordo com uma probabilidade, a qual depende do sucesso em aplicações anteriores.

O AIRP também tem o funcionamento semelhante ao do algoritmo HIVP, diferindo-se deste na geração da solução inicial, na busca local do ILS e no atributo do PR. No AIRP a solução inicial é gerada por um procedimento construtivo guloso baseado na regra *Adaptive Shortest Processing Time* (ASPT), e essa solução é refinada utilizando um ILS que tem como método de busca local o procedimento RVI. O procedimento RVI foi inspirado no RVND e ILS, e utiliza dois métodos de refinamento e um de perturbação, os quais são aplicados em ordem aleatória a cada chamada do procedimento. No procedimento PR é utilizado como atributo todas as alocações de uma máquina, na ordem em que elas aparecem.

Por fim, o algoritmo AIRMP incorpora um módulo de programação linear inteira mista ao algoritmo AIRP. Esse módulo é acionado periodicamente, a partir de um parâmetro previamente definido. Para acionar esse módulo, é selecionado um par de máquinas, tal como descrito na subseção 4.4.14, e entregue ao otimizador de programação matemática um subconjunto de tarefas dessas duas máquinas. Esse subconjunto é for-

mado por, no máximo, 5 tarefas de cada máquina. O otimizador utiliza a formulação matemática UPMSPST-VR para encontrar a melhor alocação, visto ter sido esta a formulação que gerou os melhores resultados em uma bateria preliminar de testes.

Para verificar o desempenho dos quatro algoritmos foram realizadas três baterias de testes.

Na primeira, os algoritmos heurísticos HIVP, GIAP e AIRP foram aplicados a 36 problemas-teste de (SOA, 2011) e em 18 de (SchedulingResearch, 2005). Pelos experimentos foi possível observar que os melhores resultados vieram do AIRP, seguido do GIAP. Provou-se, também, que os resultados do HIVP e do AIRP se diferenciaram estatísticamente, assim como os resultados do HIVP e do GIAP. Entretanto, os testes mostraram que não existem diferenças estatísticas entre os algoritmos GIAP e AIRP. Também foi realizado um teste de probabilidade empírica. Nesse teste foi possível observar a superioridade do algoritmo AIRP, que ficou melhor posicionado, seguido do GIAP. Em vista disso, pode-se concluir que o algoritmo AIRP foi o melhor dentre os algoritmos heurísticos desenvolvidos neste trabalho.

Na segunda bateria de testes o algoritmo AIRP foi executado usando um conjunto de 360 problemas-teste de (SOA, 2011) e 180 de (SchedulingResearch, 2005). Na primeira parte desta bateria, os resultados encontrados nos problemas-teste de (SOA, 2011) foram comparados com os dos algoritmos GA2 de (Vallada e Ruiz, 2011) e GIVMP de (Haddad et al., 2012). Foi possível observar que o algoritmo AIRP encontrou melhores resultados médios em todas as combinações de problemas-teste e na média geral. Nos testes estatísticos os resultados do AIRP se mostraram diferentes estatísticamente dos do GA2, assim como os do GIVMP também se diferenciaram estatisticamente dos do GA2. Não houve diferenças estatísticas entre os resultados médios do AIRP e do GIVMP, mas o algoritmo AIRP obteve nitidamente melhores resultados. Na segunda parte desta bateria de testes os resultados encontrados nos problemas-teste de (SchedulingResearch, 2005) foram comparados com aqueles relativos aos algoritmos Meta-RaPS de (Rabadi et al., 2006), ACO de (Arnaout et al., 2010), RSA de (Ying et al., 2010), MVND de (Fleszar et al., 2011) e GIVMP de (Haddad et al., 2012). O AIRP encontrou melhores resultados em todas as combinações de problemas-teste e na média geral que os algoritmos Meta-RaPS, ACO e RSA. Em duas combinações de problemas-teste o AIRP superou o algoritmo GIVMP, assim como na média geral. O algoritmo MVND foi o que encontrou melhores resultados que os demais em todas as combinações de problemas-teste e na média geral. Entretanto, os testes mostraram que não existem diferenças estatísticas entre todos os algoritmos comparados.

A terceira bateria de testes foi composta de experimentos envolvendo os mesmos problemas-teste utilizados na primeira bateria. Nessa bateria o algoritmo AIRP foi comparado com o algoritmo híbrido AIRMP. Como o AIRMP necessita de maior tempo de processamento para ser executado (visto que ele aciona periodicamente um módulo de programação matemática para otimizar um subconjunto de tarefas pertencentes a um par de máquinas selecionadas), os testes foram realizados utilizando-se um tempo bem maior. No entanto, mesmo com aumento no tempo, o algoritmo AIRP foi o que obteve os melhores resultados. Desta forma, fica evidenciado que a inserção do módulo de programação matemática, pelo menos no tempo destinado para processamento e usando a estrutura de composição dos subconjuntos de tarefas nas duas máquinas, não foi capaz de melhorar o desempenho do algoritmo AIRMP.

## 6.2 Trabalhos Futuros

Como trabalhos futuros são propostas as seguintes atividades:

- Desenvolver técnicas que explorem a redução do espaço de busca. Essas técnicas consistiriam em limitar os movimentos realizados no espaço de busca, de maneira a evitar movimentos que não sejam promissores;
- Avaliar a inclusão de novas técnicas de perturbação;
- Implementar um algoritmo baseado no *Iterated Greedy Search* – IGS. A justificativa é que, de certa forma, os algoritmos desenvolvidos neste trabalho têm analogias com o IGS;
- Executar o algoritmo AIRP em todos os problemas-teste de (SOA, 2011) e (SchedulingResearch, 2005);
- Analisar o algoritmo AIRMP para outros valores maiores de tempo de processamento e em um conjunto maior de problemas-teste;
- Estudar e implementar outras formas de distribuir as tarefas do par de máquinas selecionadas para uso no módulo de programação matemática do algoritmo AIRMP, assim como estudar e implementar outras formas de compor a solução resultante nesse procedimento. É importante destacar que na proposta implementada cada subconjunto de tarefas escolhidas é otimizada separadamente, sem ser influenciada pelo resultado da alocação anterior.

- Realizar experimentos computacionais específicos para uma melhor comparação entre as formulações matemáticas UPMSPST-VR, UPMSPST-RA, UPMSPST-CV e UPMSPST-IT.

# Apêndice A

# Publicações

A seguir são listados os trabalhos oriundos desta dissertação que foram aceitos para apresentação em eventos científicos e publicados nos seus anais:

1. **Título**: AIV: A Heuristic Algorithm based on Iterated Local Search and Variable Neighborhood Descent for solving the Unrelated Parallel Machine Scheduling Problem with Setup Times
   **Autores**: Matheus Nohra Haddad, Luciano Perdigão Cota, Marcone Jamilson Freitas Souza e Nelson Maculan
   **Evento**: 16th International Conference on Enterprise Information Systems – ICEIS 2014
   **Local**: Lisboa, Portugal
   **Período**: 27 a 30 de abril de 2014
   **Observação**: Trabalho premiado como melhor artigo de estudante na área Inteligência Artificial e Sistema de Apoio à Decisão

2. **Título**: AIRP: A heuristic algorithm for solving the unrelated parallel machine scheduling problem
   **Autores**: Luciano Perdigão Cota, Matheus Nohra Haddad, Marcone Jamilson Freitas Souza e Vitor Nazário Coelho
   **Evento**: IEEE Congress on Evolutionary Computation 2014 – IEEE CEC 2014
   **Local**: Beijing, China
   **Período**: 06 a 11 de julho de 2014

3. **Título**: Um algoritmo heurístico para resolver o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência
   **Autores**: Luciano Perdigão Cota, Matheus Nohra Haddad, Marcone Jamilson Freitas Souza e Alexandre Xavier Martins
   **Evento**: 35th Congresso Nacional de Matemática Aplicada e Computacional – CNMAC 2014
   **Local**: Natal, Brasil
   **Período**: 08 a 12 de setembro de 2014

# Referências Bibliográficas


Al-Salem, A.: 2004, Scheduling to minimize makespan on unrelated parallel machines with sequence dependent setup times, *Engineering Journal of the University of Qatar* **17**, 177–187.

Arnaout, J., Rabadi, G. and Musa, R.: 2010, A two-stage ant colony optimization algorithm to minimize the makespan on unrelated parallel machines with sequence-dependent setup times, *Journal of Intelligent Manufacturing* **21**(6), 693–701.

Baker, K. R.: 1974, *Introduction to Sequencing and Scheduling*, John Wiley & Sons.

Bresina, J. L.: 1996, Heusistic-biased stochastic sampling, *Proceedings of the thirteenth national conference on Artificial intelligence* **1**, 271–278.

Chang, P. and Chen, S.: 2011, Integrating dominance properties with genetic algorithms for parallel machine scheduling problems with setup times, *Applied Soft Computing* **11**(1), 1263–1274.

de Paula, M. R., Ravetti, M. G., Mateus, G. R. and Pardalos, P. M.: 2007, Solving parallel machines scheduling problems with sequence-dependent setup times using variable neighbourhood search, *IMA Journal of Management Mathematics* **18**, 101–115.

Feo, T. and Resende, M.: 1995, Greedy randomized search procedures, *Journal of Global Optimization* **6**, 109–133.

Feo, T., Resende, M. and Smith, S.: 1994, A greedy randomized adaptive search procedure for maximum independent set, *Operations Research* **42**, 860–878.

Fleszar, K., Charalambous, C. and Hindi, K.: 2011, A variable neighborhood descent heuristic for the problem of makespan minimisation on unrelated parallel machines with setup times, *Journal of Intelligent Manufacturing* . doi:10.1007/s10845-011-0522-8.

Garey, M. and Johnson, D.: 1979, Computers and intractability: A guide to the theory of np-completeness, *WH Freeman & Co., San Francisco* **174**.

Glover, F.: 1996, Tabu search and adaptive memory programming - advances, applications and challenges, *in* R. S. Barr, R. V. Helgason and J. L. Kennington (eds), *Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research*, Kluwer Academic Publishers, pp. 1–75.

Goldberg, D. E.: 1989, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Berkeley.

Graham, R., Lawler, E., Lenstra, J. and Kan, A.: 1979, Optimization and approximation in deterministic sequencing and scheduling: a survey, *Annals of discrete Mathematics* **5**(2), 287–326.

Guinet, A.: 1991, Textile production systems: a succession of non-identical parallel processors shops, *Journal of Operational Research Society* **42**(8), 655–671.

Guinet, A.: 1993, Scheduling sequence-dependent jobs on identical parallel machines to minimize completion time criteria, *International Journal of Production Research* **31**(7), 1579–1594.

Haddad, M. N.: 2012, *Algoritmos heurísticos híbridos para o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência*, Dissertação de mestrado, Universidade Federal de Ouro Preto, Ouro Preto/MG , Brasil.

Haddad, M. N., Souza, M. J. F. and Santos, H. G.: 2011, Algoritmos genéticos para o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência, *Proceedings of the 63th Simpósio Brasileiro de Pesquisa Operacional*, Ubatuba/SP, Brasil.

Haddad, M. N., Souza, M. J. F., Santos, H. G. and Martins, A. X.: 2012, Algoritmos heurísticos híbridos para o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência, *Proceedings of the 34th Congresso Nacional de Matemática Aplicada e Computacional*, Águas de Lindóia/SP, Brasil.

Hansen, P., Mladenovic, N. and Pérez, J. A. M.: 2008, Variable neighborhood search: methods and applications, *4OR: Quartely journal of the Belgian, French and Italian operations research societies* **6**, 319–360.

Helal, M., Rabadi, G. and Al-Salem, A.: 2006, A tabu search algorithm to minimize the makespan for the unrelated parallel machines scheduling problem with setup times, *International Journal of Operations Research* **3**(3), 182–192.

Holland, J. H.: 1975, *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press.

Karp, R. M.: 1972, Reducibility among combinatorial problems, *Complexity of Computer Computations* **40**(4), 85–103.

Kim, D. W., Kim, K. H., Jang, W. and Frank Chen, F.: 2002, Unrelated parallel machine scheduling with setup times using simulated annealing, *Robotics and Computer-Integrated Manufacturing* **18**, 223–231.

Kim, D. W., Na, D. G. and Frank Chen, F.: 2003, Unrelated parallel machine scheduling with setup times and a total weighted tardiness objective, *Robotics and Computer-Integrated Manufacturing* **19**, 173–181.

Logendran, R., McDonell, B. and Smucker, B.: 2007, Scheduling unrelated parallel machines with sequence-dependent setups, *Computers & Operations research* **34**(11), 3420–3438.

Lourenço, H. R., Martin, O. and Stützle, T.: 2003, Iterated local search, *in* F. Glover and G. Kochenberger (eds), *Handbook of Metaheuristics*, Vol. 57 of *International Series in Operations Research & Management Science*, Kluwer Academic Publishers, Norwell, MA, pp. 321–353.

Montgomery, D.: 2007, *Design and Analysis of Experiments*, fifth edn, John Wiley & Sons, New York, NY.

Pereira Lopes, M. J. and de Carvalho, J. M.: 2007, A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup times, *European Journal of Operational Research* **176**, 1508–1527.

Peydro, L. F. and Ruiz, R.: 2010, Iterated greedy local search methods for unrelated parallel machine scheduling, *European Journal of Operation Research* **207**, 55–69.

Peydro, L. F. and Ruiz, R.: 2011, Size-reduction heuristics for the unrelated parallel machines, *Computers & Operations Research* **38**, 301–309.

Rabadi, G., Moraga, R. J. and Al-Salem, A.: 2006, Heuristics for the unrelated parallel machine scheduling problem with setup times, *Journal of Intelligent Manufacturing* **17**, 85–97.

Randall, P. A. and Kurz, M. E.: 2007, Effectiveness of Adaptive Crossover Procedures for a Genetic Algorithm to Schedule Unrelated Parallel Machines with Setups, *International Journal of Operational Research* **4**, 1–10.

Ravetti, M. G., Mateus, G. R., Rocha, P. L. and Pardalos, P. M.: 2007, A scheduling problem with unrelated parallel machines and sequence dependent setups, *International Journal of Operational Research* **2**, 380–399.

Rocha, L. P., Ravetti, M. G., Matheus, G. R. and Pardalos, P. M.: 2006, Exact algorithms for a scheduling problem with unrelated parallel, *Computers & Operations Research* **35**, 1250–1264.

Rosa, B. F., Souza, S. R. and Souza, M. J. F.: 2009, Formulações de programação matemática para o problema de sequenciamento em uma máquina com janelas distintas e tempos de preparação dependentes da sequência de produção, *Proceedings of the 32th*

*Congresso Nacional de Matemática Aplicada e Computacional*, Cuiabá/MT, Brasil. **URL:** *http://www.sbmac.org.br/cnmac2009/*

Rossetti, I. C. M.: 2003, *Heurísticas para o problema de síntese de redes a 2 caminhos*, Tese de doutorado, Programa de Pós-Graduação em Informática, PUC-RJ, Rio de Janeiro, Brasil.

Sarin, S., Sherali, H. and Bhootra, A.: 2005, New tighter polynomial length formulations for the asymmetric traveling salesman problem with and without precedence constraints, *Operations research letters* **33**(1), 62–70.

SchedulingResearch: 2005, Scheduling research virtual center. A web site that includes benchmark problem data sets and solutions for scheduling problems. Disponível em `http://www.schedulingresearch.com`. Acesso em 01 de julho de 2013.

Shapiro, S. S. and Wilk, M. B.: 1965, An analysis of variance test for normality (complete samples), *Biometrika* **52**, 591–611.

SOA: 2011, Sistemas de optimización aplicada. Página que contém problemas-teste para o problema de sequenciamento em máquinas paralelas não-relacionadas com tempos de preparação dependentes da sequência. Disponível em `http://soa.iti.es/problem-instances`. Acesso em 01 de julho de 2013.

Souza, M., Coelho, I., Ribas, S., Santos, H. and Merschmann, L.: 2010, A hybrid heuristic algorithm for the open-pit-mining operational planning problem, *European Journal of Operational Research* **207**(2), 1041–1051.

Vallada, E. and Ruiz, R.: 2011, A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times, *European Journal of Operational Research* **211**(3), 612–622.

Weng, M. X., Lu, J. and Ren, H.: 2001, Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective, *International Journal of Production Economics* **70**, 215–226.

Ying, K.-C., Lee, Z.-J. and Lin, S.-W.: 2010, Makespan minimisation for scheduling unrelated parallel machines with setup times, *Journal of Intelligent Manufacturing* . doi:10.1007/s10845-010-0483-3.