UMA HEURISTICA GRASP-VND PARA O PROBLEMA DE SEQUENCIAMENTO DE TAREFAS NUM AMBIENTE ASSEMBLY FLOWSHOP COM TRÊS ESTÁGIOS E TEMPOS DE SETUP DEPENDENTES DA SEQUÊNCIA

Saulo Cunha Campos

José Elias C. Arroyo

Luciana Brugiolo Gonçalves

Departamento de Informática, Universidade Federal de Viçosa (UFV)
Viçosa – Minas Gerais – MG – Brasil – 36.570-000
saulo.campos@ufv.br, jarroyo@dpi.ufv.br, lbrugiolo@ufv.br

RESUMO

Neste artigo é considerado o problema de programação de *n* tarefas em um ambiente de produção *flowshop* com três estágios. O primeiro estágio consiste na fabricação em paralelo das *m* partes dos produtos (tarefas) utilizando, respectivamente, *m* máquinas paralelas independentes. No segundo estágio as peças fabricadas são transportadas para a linha de montagem e no terceiro estágio elas são montadas obtendo os produtos finais. No primeiro estágio são considerados tempos de *setup* das máquinas dependentes da sequência. O objetivo do problema é determinar o sequenciamento das tarefas de forma a minimizar o tempo de fluxo médio e o atraso máximo das tarefas com relação a suas datas de entrega. Neste trabalho é desenvolvida uma heurística híbrida baseada na metaheurística GRASP que utiliza a heurística VND na fase da busca local. Na heurística VND são usadas quatro estruturas de vizinhança que são exploradas em ordem aleatória. Cada vizinhança é associada a uma probabilidade de seleção proporcional à quantidade de soluções melhores obtidas. Assim, as vizinhanças que produzem melhores soluções possuem maiores probabilidades de serem escolhidas. São analisadas duas versões da heurística proposta e os resultados obtidos são comparados com os resultados de um algoritmo *Simulated Annealing* proposto na literatura para o mesmo problema. Os experimentos computacionais e análises estatísticas mostram que a heurística proposta possui um excelente desempenho.

PALAVRAS CHAVE. Sequenciamento de tarefas, *flowshop*, Tempos de setup dependentes da sequencia, metaheurísticas, GRASP, VND.

ABSTRACT

In this paper we consider an assembly flowshop scheduling problem with *n* jobs and three stages. In the first stage different production operations are done in parallel by using *m* parallel machines. In the second stage the manufactured parts are collected and transferred to the assembly line and in the third stage these parts are assembled into final products. In the first stage are considered sequence dependent setup times of the machines. The goal of the problem is to determine the sequence of jobs in order to minimize the mean flow time and maximum tardiness of the jobs. In this work we developed a hybrid heuristic based on metaheuristic GRASP that uses VND heuristic in the local search phase. The VND heuristic uses four neighborhood structures that are explored at random order. Each neighborhood is associated with a probability of selection proportional to the number of best solutions obtained. Thus, a neighborhood that produces better solutions has a higher probability of selection. Are analyzed two versions of the heuristic, and the obtained results are compared with the results of a Simulated Annealing algorithm proposed in the literature for the same problem. The computational experiments and statistical analyzes show that the proposed heuristic has excellent performance.

KEYWORDS. *Assembly flowshop scheduling, Sequence-dependent setup times*, metaheuristics, GRASP, VND.

1. Introdução

Os problemas de programação (*scheduling*) da produção são amplamente estudados na literatura devido a sua importância prática e teórica (Allahverdi *et al*. 2008). Estes problemas possuem varias aplicações em diversos setores das indústrias de manufatura. A maioria desses problemas pertencem à classe NP-Difícil e é necessário utilizar algoritmos eficientes para resolvê-los.

De acordo com Pinedo (2005) e Lustosa *et al*. (2008) a programação da produção evolui devido a alta concorrência de mercado e a globalização que pressiona as empresas a melhorarem a qualidade de seus produtos e reduzirem seus custos. Assim a programação da produção constitui um importante processo decisório para alocação eficiente de recursos sobre um determinado ambiente de produção com várias tarefas (SHEN *et al*, 2006).

Neste trabalho é abordado o problema conhecido como "*Assembly Flowshop Scheduling* – AFS", que é um ambiente de produção onde diversas partes do produto (peças e componentes), são fabricadas de forma independente em diferentes linhas de produção, e, em seguida, são transferidas para um local onde é realizada a montagem do produto, que geralmente é feito em uma única máquina. Pinedo (2005) apresenta um exemplo deste tipo de ambiente de produção em indústrias que fabricam placas de circuitos, onde diversos componentes são fabricados de forma independente por máquinas paralelas em um primeiro estágio, e depois, em um segundo estágio, são agrupados e soldados às placas, formando o produto final. Já Potts *et al*. (1995) exemplificou o problema através da indústria de computadores pessoais que fabricam microprocessadores, discos rígidos e outros componentes em fábricas diferentes (primeiro estágio) e depois são reunidos em um dado local, para montagem do computador (segundo estágio). Koulamas e Kyparisis (2001) afirma que este tipo de ambiente de produção vem crescendo em resposta a pressão do mercado sobre as empresas para que ofereçam uma variedade maior de produtos e customizações.

Na literatura, o problema AFS é abordado considerando dois ou três estágios. Alguns trabalhos consideram apenas estágios de fabricação e montagem, enquanto outros consideram também um estágio de transporte. O problema com três estágios é considerado mais realista e é tratado como uma evolução do modelo com dois estágios. Potts *et al*. (1995) foram uns dos primeiros a abordar o problema AFS, onde consideraram um modelo com dois estágios, sendo o primeiro estágio para fabricação dos componentes dos produtos em um ambiente de máquinas paralelas e independentes e o segundo estágio, destinado à montagem do produto final por uma única máquina. Potts *et al*. (1995) mostraram as propriedades do problema e provaram que o problema é NP-Difícil. Além disso, propuseram métodos heurísticos para minimização do *makespan*.

Outros autores também abordaram o problema AFS com dois estágios, como a abordagem feita por Tozkapan *et al*. (2001) que trataram a minimização do tempo total de fluxo através de uma técnica *branch-and-bound* com critérios de dominância e procedimentos heurísticos para determinação de *upper-bounds*. Allahverdi e Al-Anzi (2006) consideraram o problema em sistemas de banco de dados distribuídos e fabricação de computadores, com o objetivo de minimizar o atraso máximo. Eles criaram regras de dominância para o problema e fizeram experimentos computacionais comparativos entre as heurísticas: *Earliest Due Date* (EDD), *Particle Swarm Optimization* (PSO) e Busca Tabu. Mais a frente Al-Anzi e Allahverdi (2007) abordaram o problema com dois critérios na função objetivo: minimização do *makespan* e do tempo médio de fluxo. A abordagem foi feita através das metaheurísticas *Simulated Annealing* (SA), Colônia de Formigas e algoritmos evolutivos adaptativos (*SDE - Self-adaptive differential evolution*). Em 2009 os mesmos autores (Al-Anzi e Allahverdi) utilizaram as metaheurísticas Busca Tabu, Nuvem de Partículas e Algoritmos evolutivos para resolver o problema, porém consideraram a minimização do atraso máximo e do *makespan*.

Apesar de haver muitos trabalhos na literatura que consideram o AFS com dois estágios, este modelo não é tão realístico, pois as maiorias dos ambientes de produção gastam um determinado tempo para coletar e transportar as partes fabricadas para o estágio de montagem. Este modelo se torna ainda menos real quando as peças veem de diferentes locais de produção ou

diferentes fábricas. Por este fato, Koulamas e Kyparisis (2001) propuseram a introdução de um estágio de transporte, onde as peças são coletadas e transferidas da fase de fabricação para a montagem, tornando a representação do problema mais real. Neste ambiente produtivo o primeiro estágio possui *m* máquinas paralelas e os estágios seguintes, apenas uma máquina em cada um. Koulamas e Kyparisis (2001) apresentaram as propriedades do problema AFS com três estágios e propuseram heurísticas para minimização do *makespan*. Outros autores também exploraram o AFS com três estágios e com máquinas paralelas no primeiro estágio. Hatami *et al.* (2010) abordou o problema com tempos de preparação nas máquinas do primeiro estágio dependentes da sequencia das tarefas. Eles propuseram um modelo matemático baseado em programação linear inteira mista para a resolução exata de instâncias pequenas do problema, além de abordagens heurísticas, baseadas em *Simulated Anealing* e Busca Tabu, para a resolução aproximada de instâncias com até 70 tarefas, com o intuito é de minimizar o tempo médio de fluxo e atraso máximo das tarefas.

Recentemente, Andrés e Hatami (2011) propuseram um modelo matemático para o mesmo problema tratado em Hatami *et al.* (2010), no entanto é considerado a minimização do tempo de fluxo total. Dalfard *et al.* (2011) aplicou um algoritmo memético para minimização multiobjectivo do *makespan*, média de atrasos e média de antecipação das tarefas.

Neste trabalho trata do mesmo problema abordado por Hatami *et al.* (2010). Sabendo que o problema é NP-Difícil, propõe-se a aplicação de uma heurística baseada nas metaheurísticas GRASP e VND. O GRASP é uma metaheurística simples que iterativamente constrói diversas soluções diferentes para serem melhoradas aplicando busca local. Neste trabalho, a busca local do GRASP é realizada através da metaheurística VND, que opera explorando diversas estruturas de vizinhanças diferentes.

O artigo está estruturado na seguinte forma: a seção 2 apresenta a definição formal do problema abordado, a seção 3 apresenta a solução heurística desenvolvida neste trabalho, a seção 4 descreve os experimentos computacionais realizados, e, por fim, na seção 5, as conclusões.

2. Definição do problema

O problema investigado neste trabalho é o sequenciamento de tarefas em um ambiente *Assembly Flowshop* com três estágios. Este problema consiste em processar (ou fabricar) um conjunto de *n* tarefas (produtos), sendo que cada tarefa possui *m* componentes (ou peças) que são fabricados de forma independente. No primeiro estágio, os componentes das tarefas são fabricados em um ambiente de *m* máquinas paralelas. No segundo estágio, as peças de uma tarefa são transportadas para uma máquina onde é feita a montagem do produto, constituindo o terceiro estágio da produção. Os estágios 2 e 3 são realizadas por máquinas diferentes. Sendo assim, todas as máquinas processam apenas uma tarefa por vez e não podem ser interrompidas durante seu processamento. Note que para cada tarefa existem *m*+2 operações distintas, sendo que *m* operações independentes são feitas no primeiro estágio e as outras duas operações são realizadas nos dois últimos estágios, respectivamente.

Os três estágios caracterizam o problema como um ambiente de produção *flowshop*. Ou seja, uma tarefa *j* só pode iniciar seu processamento no estágio 2 quando todas suas peças são finalizadas no estágio 1, e a montagem (estágio 3) só poderá ser iniciada após a tarefa *j* ser totalmente finalizada no estágio 2.

$t_{[k,j]}$ denota o tempo de processamento da tarefa *j* na máquina *k*, ou seja, o tempo gasto para fabricar a peça *j* na máquina *k*. $tt_{[j]}$ e $ta_{[j]}$ são, respectivamente, os tempos gastos para fazer o transporte das peças e montagem da tarefa *j*. No primeiro estágio existem tempos de preparação (*setup*) das máquinas que dependem da sequência das tarefas, ou seja, na máquina *k*, entre duas tarefas consecutivas *j*-1 e *j* é gasto um tempo $S_{[k,j-1,j]}$, *j*=1,...*n*. Todas as tarefas estão disponíveis para serem processadas no tempo zero e cada tarefa *j* possui uma data de entrega $d_{[j]}$.

O sequenciamento das *n* tarefas nas máquinas é representado por uma permutação simples, que denota a ordem de execução das tarefas nas máquinas e nos estágios. O objetivo do problema é determinar o sequenciamento das tarefas que minimize o tempo médio de fluxo das

tarefas ($\bar{F}$) e atraso máximo (T_{max}) de entrega. Como no trabalho de Hatami *et al.* (2010), o problema de otimização biobjetivo é reduzido a um problema mono-objetivo considerando a seguinte função ponderada:

$$f = w\bar{F} + (1-w)T_{max}$$

onde *w* e 1–*w* ($0 \leq w \leq 1$) são os pesos atribuídos aos objetivos $\bar{F}$ e T_{max}. Note que, para valores de *w* próximos de zero, maior prioridade é dada ao atraso máximo, e consequentemente menor prioridade é dada ao tempo de fluxo.

O tempo fluxo médio e o atraso máximo são calculados pelas seguintes equações:

$$\bar{F} = \frac{\sum_{j=1}^{n} C_{3[j]}}{n}$$

$$T_{max} = \max_{j=1,\ldots,n}\left\{C_{3[j]} - d_{[j]}\right\}$$

onde, $C_{3[j]}$ é o tempo de conclusão, no terceiro estágio, da tarefa que está na posição *j*. $C_{3[j]}$ é calculado através da seguinte equação:

$$C_{3[j]} = \max\left\{C_{2[j]},\ C_{3[j-1]}\right\} + ta_{[j]}\ ,\quad \forall_{j=1,\ldots,n}$$

Onde $C_{1[j]}$ e $C_{2[j]}$ são, respectivamente, os tempos de conclusão no primeiro e segundo estágio da tarefa que está na posição *j*. $C_{1[j]}$ e $C_{2[j]}$ são calculadas usando as equações seguintes:

$$C_{1[j]} = max_{k=1,\ldots,m}\left\{\sum_{i=1}^{j} S_{[k,i-1,i]} + t_{[k,j]}\right\},\qquad \forall_{j=1,\ldots,n}$$

$$C_{2[j]} = \max\left\{C_{1[j]},\ C_{2[j-1]}\right\} + tt_{[j]}\ ,\qquad \forall_{j=1,\ldots,n}$$

A seguir é apresentado um exemplo do cálculo da função objetivo.

Exemplo: considere uma instância com *n*=6 tarefas e *m*=2 máquinas paralelas no estágio 1. Na Tabela 1 estão os tempos de processamento das tarefas para os três estágios e as datas de entrega das tarefas. Nas Tabelas 2 (a) e (b) estão os tempos de preparação das máquinas 1 e 2, respectivamente. A Figura 1 mostra o gráfico de *Gantt* correspondente à sequência das tarefas *S* = {4, 5, 6, 1, 2, 3}. Neste gráfico, as partes pretas correspondem aos tempos de preparação das máquinas do primeiro estágio. Nesta Figura, também é mostrado os tempos de conclusão das tarefas (no terceiro estágio) e os atrasos gerados. Para este exemplo o fluxo médio é (13+20+25+31+39+41)/6 = 28,17, e o atraso máximo é max(0, 1, 4, 17, 22) = 22. Se *w*=0,6, o valor da função objetivo é *f* = 0,6 × 28,17 + (1 – 0,6) × 22 = 25,702.

Tabela 1: Tempos de processamento nos estágios 1, 2 e 3 e as datas de entrega das tarefas

		j1	*j2*	*j3*	*j4*	*j5*	*j6*
$t_{[k,j]}$	**M1**	3	5	6	2	3	6
	M2	1	4	5	4	6	2
$tt_{[j]}$		5	1	2	4	3	2
$ta_{[j]}$		4	8	2	3	4	5
$d_{[j]}$		14	17	45	14	16	28

Tabela 2: Tempo de preparação nas máquinas M1 e M2

(a) Máquina **M1**

	j1	*j2*	*j3*	*j4*	*j5*	*j6*
j0	4	1	2	3	2	1
j1	0	2	5	2	3	1
J2	1	0	1	2	3	2
J3	2	1	0	3	2	4
J4	1	3	4	0	1	2
J5	2	3	1	4	0	1
J6	3	4	3	1	3	0

(b) Máquina **M2**

	j1	*j2*	*j3*	*j4*	*j5*	*j6*
j0	1	3	2	2	4	5
j1	0	2	1	1	3	4
J2	2	0	3	2	3	1
J3	3	2	0	2	4	3
J4	2	3	2	0	1	2
J5	2	3	1	2	0	3
J6	2	2	1	1	2	0

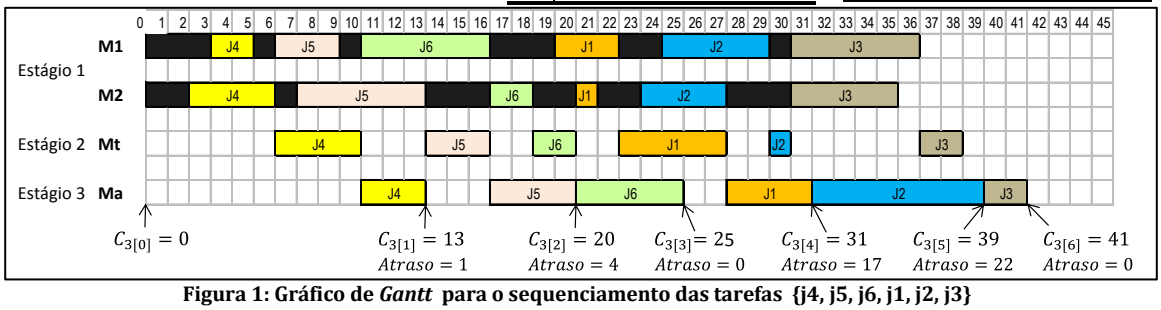

Figura 1: Gráfico de *Gantt* para o sequenciamento das tarefas {j4, j5, j6, j1, j2, j3}

3. Aplicação de metaheurísticas

Para problemas de otimização combinatória da classe NP-Difícil não se conhecem algoritmos de tempo polinomial para a obtenção de soluções ótimas. Os melhores algoritmos, baseados em métodos exatos, para esses problemas possuem tempo exponencial em função do tamanho da entrada, o que torna inviável a utilização desses algoritmos (LEVITIN, 2003). Como alternativas aos métodos exatos são utilizadas as Metaheurísticas, que são métodos aproximados genéricos para geração de soluções de alta qualidade (próxima da solução ótima) em um tempo computacional polinomial (TALBI, 2009).

Neste trabalho propõe-se uma heurística que combina as metaheurísticas GRASP (FEO e RESENDE, 1995) e VND (HANSEN e MLADENOVIĆ, 2003). O GRASP é uma metaheurística que iterativamente executa dois métodos: um método de construção (de uma solução) e um método de melhoria ou busca local. Já o VND é uma metaheurística baseada na execução sucessiva de buscas locais em diferentes estruturas de vizinhanças.

Na heurística proposta, denominada como GRASP_VND, utiliza-se o VND como método de melhoria do GRASP. O pseudocódigo da heurística GRASP_VND é apresentado no Algoritmo 1. A heurística recebe como entrada dois parâmetros: o parâmetro de aleatoriedade α, utilizado no método de construção do GRASP e o parâmetro *TempoMaximo* que define o tempo máximo de execução do algoritmo (critério de parada). Observa-se que o Algoritmo GRASP_VND, executa iterativamente os métodos **Construção** e **VND**, e a solução S'' retornada pelo VND é utilizada para atualizar a melhor solução S a ser retornada pelo algoritmo.

Nas subseções seguintes são descritos os métodos **Construção** e **VND**.

Algoritmo1: GRASP_VND (α, *TempoMaximo*)

```
f(S) ← M; //um valor suficientemente grande
Tempo ← 0;
Enquanto (Tempo < TempoMaximo) faça
        S' ← Construção( α ); //Construção Gulosa Aleatória
        S'' ← VND( S'); //Busca Local
        Se ( f(S'') < f(S) ) Então
                S ← S'';
        AtualizaTempo(Tempo);
Fim_Enquanto
Retorna S;
```

3.1. Método de Construção

A fase de construção de uma solução é realizada através da heurística NEH (NAWAZ *et al.*, 1983) que é adaptada para a minimização da função $f = \alpha\bar{F} + (1-\alpha)T_{max}$.

Nesta heurística primeiramente as tarefas são arranjadas de acordo a uma regra de prioridade (ou função gulosa) formando uma lista de tarefas candidatas $J = \{j_1, j_2, ..., j_n\}$. A primeira tarefa j_1 é usada para formar uma sequencia parcial S ($S = \{j_1\}$). Em seguida, escolhe-se a próxima tarefa e é feita a inserção em cada uma das posições de S, obtendo duas sequências parciais $\{j_2, j_1\}$ e $\{j_1, j_2\}$. Estas sequências são avaliadas (calcula-se o valor da função objetivo) e determina-se a melhor. Assim sucessivamente, a próxima tarefa j_i da lista é inserida em cada uma das posições da melhor sequencia, obtendo i sequências parciais. Destas i sequências parciais determina-se a melhor. Para $i = n$, o método finaliza e é obtida a melhor sequencia completa (com n tarefas). Ressalta-se que, cada vez que uma tarefa j_i é escolhida de J, ela é removida da lista: $J = J - \{j_i\}$, assim, o método finaliza quando $J = \varnothing$.

No algoritmo GRASP é necessário obter uma solução diferente a cada iteração. Para isto, ao invés de selecionar a primeira tarefa da lista J, a tarefa é selecionada aleatoriamente dentre as t primeiras tarefas da lista. t define o tamanho da lista restrita de candidatos que é calculada como $t = \max\{1, \alpha^* |J|\}$. Note que se $\alpha = 0$, sempre selecionada a primeira tarefa da lista (escolha gulosa). Já se $\alpha = 1$, a escolha será totalmente aleatória.

3.2. Método de Busca Local com VND

A fase de busca local do algoritmo GRASP consiste em aplicar a metaheurística VND.

O pseudocódigo do método VND é mostrado no Algoritmo 2. O algoritmo VND possui dois parâmetros de entrada: a solução S a ser melhorada e *nv* o número de estruturas de vizinhança a serem usadas. Inicialmente as *nv* vizinhanças $\{V_1,V_2,...,V_{nv}\}$ são ordenadas de acordo a algum critério. Iniciando com a primeira (k=1) vizinhança da solução corrente S, denotada por $V_k(S)$, procura-se a melhor solução S' nessa vizinhança. Se S' é melhor que a solução corrente (f(S') < f(S)), S' substitui S e o processo reinicia a partir do uso da vizinhança principal V_1, ou seja, k=1. Caso contrário, troca-se de vizinhança (explora-se a próxima vizinhança da lista: V_{k+1}). Se for explorada a ultima vizinhança (V_{nv}) e a solução corrente não for melhorada, então a execução do VND finaliza retornando a melhor solução S (ótima local).

Algoritmo 2: VND (S, *nv*)

```
Ordenar as estruturas de vizinhanças: V ← {V1,V2,...,Vnv};
k ← 1;
Enquanto (k ≤ nv) faça
    S' ← Encontrar o melhor vizinho na vizinhança Vk(S);
    Se ( f(S') < f(S) ) Então
            S ← S';
            k ← 1; //reinício
    Senão
            k ← k + 1;
Fim_Enquanto
Retorna S;
```

No método VND são utilizadas 4 estruturas de vizinhança (*nv* = 4) definidas a seguir.

- V_1 – *Inserção*: Para uma dada sequência $S = \{j_1,..,\ j_{x-1},\ \mathbf{j_x},\ j_{x+1},...,\ j_{y-1},\ j_y,\ j_{y+1},...,\ j_n\}$, esta vizinhança gera uma solução vizinha inserindo uma tarefa j_x (que está na posição x) em outra posição y tal que $y \neq x$ e $y \neq (x-1)$. Fazendo esse movimento obtém-se: $S' = \{j_1,..,\ j_{x-1}, j_{x+1},...,\ j_{y-1},\ \mathbf{j_x}, j_y, j_{y+1},...,\ j_n\}$. A exploração desta vizinhança gera $(n-1)^2$ vizinhos.
- V_2 – *Troca entre 2 tarefas*: Para uma dada sequência $S = \{j_1,..,\ j_{x-1},\ \mathbf{j_x},\ j_{x+1},...,\ j_{y-1},\ \mathbf{j_y},\ j_{y+1},...,\ j_n\}$, esta vizinhança gera uma solução vizinha trocando as posições de duas tarefas diferentes j_x e j_y, ou seja, obtém-se $S' = \{j_1,..,\ j_{x-1},\ \mathbf{j_y},\ j_{x+1},...,\ j_{y-1},\ \mathbf{j_x},\ j_{y+1},...,\ j_n\}$. A exploração desta vizinhança gera $\frac{n(n-1)}{2}$ vizinhos.
- V_3 – *Troca entre 2 tarefas sucessivas*: Dada uma sequência $S = \{j_1,..,\ \mathbf{j_{x-1}}, \mathbf{j_x},\ j_{x+1},...,\ \mathbf{j_{y-1}}, \mathbf{j_y}, j_{y+1},...,\ j_n\}$, o movimento de troca entre duas tarefas sucessivas consiste em trocar j_{x-1} e j_x, respectivamente, com j_{y-1} e j_y, assim obtém-se uma nova sequência $S' = \{j_1,..,\ \mathbf{j_{y-1}}, \mathbf{j_y},\ j_{x+1},...,\ \mathbf{j_{x-1}}, \mathbf{j_x},\ j_{y+1},...,\ j_n\}$. A exploração desta vizinhança gera $\frac{(n-3)(n-2)}{2}$ vizinhos.
- V_4 – *Troca xyz*: Esta vizinhança foi proposta por Eksioglu *et al*. (2008) para o problema de *flowshop* permutacional. Para uma dada sequência $S = \{j_1,..,\ j_{x-1},\ \mathbf{j_x},\ j_{x+1},...,\ j_{y-1},\ \mathbf{j_y},\ j_{y+1},...,\ j_{z-1},\ \mathbf{j_z},\ j_{z+1},...,\ j_n\}$, esta vizinhança gera uma solução vizinha inserindo j_x na posição y, e j_y na posição z, em seguida, inserindo j_z na posição x, de forma que $1 \leq x < y < z \leq n$. Fazendo estes movimentos, obtêm-se a nova sequencia: $S' = \{j_1,..,\ j_{x-1},\ \mathbf{j_z}, j_{x+1},...,\ j_{y-1},\ \mathbf{j_x},\ j_{y+1},...,\ j_{z-1},\ \mathbf{j_y}, j_{z+1},...,\ j_n\}$. A exploração desta vizinhança gera $\frac{n(n-1)(n-2)}{3}$ vizinhos, porém para reduzir o tamanho da vizinhança, evitando o consumo de tempo de CPU, a posição y é um valor fixo escolhido aleatoriamente, então o número de vizinhos gerados é no máximo $\frac{n(n-2)}{2}$.

As estruturas de vizinhanças foram ordenadas de acordo ao potencial que possuem na obtenção de soluções melhores. Uma análise da qualidade das vizinhanças é apresentada na seção de resultados computacionais.

3.2. VND com Ordenação Aleatória das Vizinhanças (RVND)

Neste trabalho é proposta uma variante do VND na qual as estruturas de vizinhanças são arranjadas de forma aleatória e a cada iteração do algoritmo uma vizinhança é escolhida aleatoriamente. Esta variante, denominada RVND (*Random Variable Neighborhood Descent*), é similar à estratégia usada em Subramanian *et al.* (2010). Neste trabalho propõe-se uso de uma probabilidade de seleção para cada vizinhança. Esta probabilidade é proporcional à frequência de melhoras obtidas com a respectiva vizinhança. Então, as melhores vizinhanças terão maior probabilidade de serem escolhidas. A seleção da vizinhança é feita como no método da roleta (*roulette wheel*), onde uma vizinhança V_i é sorteada baseado na sua probabilidade $Prob_i$.

No Algoritmo 3, é apresentado o pseudocódigo do método RVND. Neste algoritmo, inicialmente todas as vizinhanças possuem a mesma probabilidade, $Prob_k \leftarrow 1/nv$, $k = 1,...,nv$, e cada iteração, a probabilidade $Prob_k$ aumenta caso a vizinhança k melhore a solução. É usada uma lista L para armazenar as vizinhanças que não produzem melhoria da solução corrente. As vizinhanças em L são proibidas de serem escolhidas, assim o método finaliza quando todas as nv vizinhanças forem exploradas e nenhuma produzir melhoria da solução, ou seja quando L = V (conjunto das vizinhanças). Caso uma vizinhança gere melhoria da solução, então a lista L é reinicializada (L = ∅), ou seja, reinicia-se o processo utilizando todas as vizinhanças. Note que no Algoritmo VND o reinicio consiste em retornar ao uso da vizinhança principal V_1.

Algoritmo 3: RVND (S, *nv*)

```
V ← {V1,V2,...,Vnv};    //Conjunto de estrutura de vizinhança
Probk ← 1/ nv, ∀k=1,..., nv  //Probabilidade das vizinhanças
k ← 1;    L ← ∅;
Enquanto (k ≤ nv) faça
        i ← SeleçãoVizinhanca(Prob, V – L); //seleciona uma vizinhança de V–L
        S' ← Encontrar o melhor vizinho na vizinhança Vi(S);
        Se ( f(S') < f(S) ) Então
                S ← S';
                Atualizar a probabilidade da vizinhança i: Probi;
                L ← ∅; //todas as vizinhanças poderão ser escolhidas
                k ← 1; //reinício
        Senão
                L ← L ∪ {i}; //a Vi não poderá ser escolhida na próxima iteração
                k ← k + 1;
Fim_Enquanto
Retorna S;
```

4. Experimentos Computacionais

Neste trabalho testa-se o desempenho das heurísticas GRASP_VND e GRASP_RVND, ou seja, no algoritmo GRASP são testados dois métodos de busca local, o VND e RVND. Os algoritmos foram implementados em linguagem C++ e todos os experimentos foram realizados em um computador com processador Intel Core 2 Quad Q6600, com velocidade de 2,40 GHz, memória RAM de 3.0 GB e sistema operacional Windows 7 32 bits.

Para realização dos experimentos foram geradas um total de 576 instancias do problema, de diferentes tamanhos, de acordo com o trabalho de Hatami *et al*. (2010). O tamanho das instâncias varia de acordo à quantidade de tarefas (n = {20, 30, 40, 50, 60, 70}) e quantidade de máquinas paralelas no primeiro estágio (m = {2, 4, 6, 8}). Os tempos de processamento, datas de entrega e tempos de setup são gerados aleatoriamente com distribuição uniforme. Os tempos de processamento das tarefas nas máquinas do primeiro, segundo e terceiro estágio são gerados nos intervalos [1, 100], [1, 10] e [1,10], respectivamente. As datas de entrega pertencem ao intervalo $[LB\left(1 - T - {}^{R}/_{2}\right), LB\left(1 - T + {}^{R}/_{2}\right)]$, onde LB é um *lower bound* para o tempo de conclusão de todas as tarefas proposto por Hatami *et al*. (2010), os parâmetros T e R definem, respectivamente, o fator de atraso das tarefas e a faixa de dispersão das datas de entrega. Para

estes parâmetros foram usados os seguintes valores: $T \in \{0.5, 0.8\}$ e $R \in \{0.2, 0.4, 0.6\}$. Os tempos de preparação das máquinas do primeiro estágio estão no intervalo [1, 20].

Na avaliação da função objetivo foram usados os seguintes pesos w= 0.2, 0.4, 0.6, e 0.8.

A métrica utilizada para avaliar os resultados dos experimentos é o Desvio Percentual Relativo que é denotado por RPD e obtido pela seguinte fórmula:

$$\text{RPD} = \frac{f_{method} - f_{best}}{f_{best}} \; x \; 100$$

onde f_{method} corresponde ao valor da função objetivo obtido por um dado algoritmo e f_{best} correspondente à melhor solução encontrada a partir da execução de todos os algoritmos comparados.

Foram feitos experimentos para calibração dos métodos do algoritmo GRASP_VND, com o intuito de gerar soluções com maior qualidade. Também foi feita uma calibração do parâmetro de aleatoriedade (α) usado no método de construção do GRASP. Em seguida foi realizado um experimento para comparar o desempenho das heurísticas GRASP_VND e GRASP_RVND. As heurísticas aqui propostas são comparadas com uma heurística literatura.

4.1. Regras de Prioridade para a Geração da Solução Inicial

Inicialmente foram testadas diferentes regras de prioridade para ordenar as tarefas e aplicar a heurística de construção NEH (NAWAZ *et al*., 1983). As regras testadas foram:

- EDD – *Earliest due date*: Ordena as tarefas em ordem crescente das suas datas de entrega.
- LPT – *largest processing time*: Ordena as tarefas em ordem decrescente dos tempos totais de processamento: $(\max_{k=1,\dots,m} t_{[j,k]}) + tt_{[j]} + ta_{[j]}$, $\forall_{j=1,\dots,n}$.
- SPT – *Shortest processing time*: Ordena as tarefas em ordem crescente de seus tempos totais de processamento.
- TLB – *tardiness lower bound*: Ordena as tarefas em ordem decrescente do limitante inferior do atraso total: $d_{[j]} - \{(\max_{k=1,\dots,m} t_{[j,k]}) + tt_{[j]} + ta_{[j]}\}, \forall_{j=1,\dots,n}$.
- HATAMI: Uma sequência (solução) inicial também é gerada utilizando a heurística proposta por Hatami *et al*. (2010).

As regras EDD e TLB são muito utilizadas na literatura para minimização de atrasos. As regras LPT e SPT são usadas para minimização do tempo máximo de conclusão e tempo de fluxo, respectivamente.

A heurística NEH é testada com cada uma das regras definidas acima. Ou seja, utilizando uma das regras determina-se uma sequência (ordenação) de tarefas em seguida aplica-se a heurística NEH. O desempenho das regras é analisado estatisticamente utilizando a medida do RPD. A Figura 2 mostra o gráfico de médias resultante do teste Tukey da Diferença Honestamente Significativa HSD (*Honestly Signicant Dierence*) com nível de confiança de 95% para as heurísticas testadas. O Teste mostra que a heurística NEH com a regra EDD gera resultados estatisticamente melhores. Portanto, no método de construção do algoritmo GRASP foi utilizado a heurística NEH com a regra EDD.

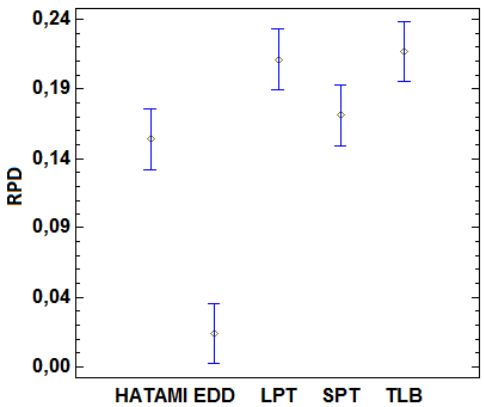

Figura 2: Gráfico das médias e intervalos HSD de Tukey com nível de confiança de 95% para as regras de ordenação na heurística NEH.

4.2. Qualidade das Estruturas de Vizinhança

Para definir a ordem de exploração das estruturas de vizinhanças pelo VND, foi feito um experimento computacional que consiste em aplicar uma busca local em descida utilizando cada uma das vizinhanças. A busca local inicia a partir de uma solução gerada aleatoriamente (solução corrente), explora-se a vizinhança (avalia-se todas as soluções vizinhas) e seleciona-se a melhor solução. Se a solução corrente é melhorada, continua-se explorando a vizinhança da melhor solução, caso contrário a busca local é finalizada e retorna a solução corrente. Para cada uma das 4 vizinhanças testadas, a busca local foi executada em 100 soluções geradas aleatoriamente para cada valor de n (n={20, 30, 40, 50, 60, 70}), totalizando 700 soluções. A Figura 3 mostra o gráfico de médias e intervalos HSD de Tukey com nível de confiança de 95% resultante das comparações das vizinhanças V_1, V_2, V_3 e V_4. Note que as vizinhanças V_1 e V_4 apresentaram os melhores e piores resultados, respectivamente. Com base nestes resultados, foi definida a seguinte ordem de exploração da vizinhança no VND: V_1, V_2, V_3 e V_4.

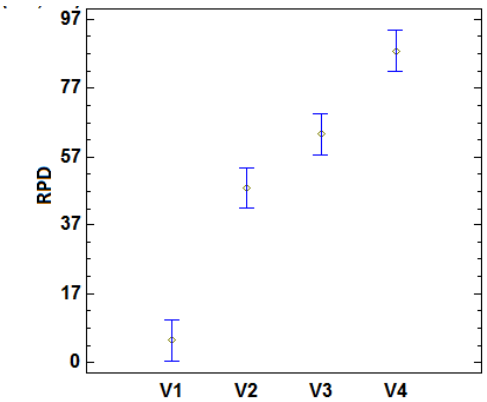

Figura 3: Gráfico das médias e intervalos HSD de Tukey com nível de confiança de 95% para as buscas locais aplicadas sobre as vizinhanças V_1, V_2, V_3 e V_4.

4.3. Calibração do parâmetro de aleatoriedade usado no método de construção

O algoritmo GRASP_VND foi testado utilizando diferentes valores do parâmetro α, utilizado no método de construção. Os valores testados foram: α = 0,1, 0,2, 0,3, 0,4, 0,5, 0,6, 0,7, 0,8 e 0,9. O algoritmo, para cada valor de α, foi executado 5 vezes para resolver cada uma das 576 instâncias e no cálculo do RPD foi considerada a média do valor da função objetivo.

A Figura 4 mostra o gráfico de médias e intervalos HSD de Tukey com nível de confiança de 95% para a comparação do GRASP_VND com os diferentes valores do parâmetro α. Os testes mostraram que não há diferenças estatísticas significativas para o uso dos diferentes valores de α, mas é possível observar que, para α=0,5, obtém-se a menor média. Portanto, nos algoritmos GRASP_VND e GRASP_RVND é usado esse valor para o α.

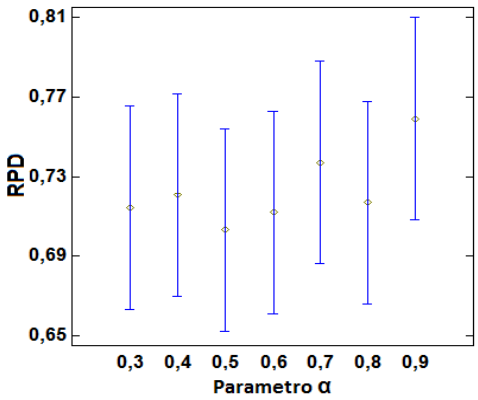

Figura 4: Gráfico das médias e intervalos HSD de Tukey com nível de confiança de 95% para a calibração do parâmetro α.

4.4. Comparação das heurísticas GRASP_VND e GRASP_RVND

Para testar o desempenho dos algoritmos GRASP_VND e GRASP_RVND, estes foram executados nas 576 instâncias do problema. Cada algoritmo foi executado 5 vezes para resolver cada instância e no cálculo do Desvio Percentual Relativo foi considerado o resultado médio (ou seja, a média do valor da função objetivo). Para que a comparação seja justa, ambos os algoritmos foram executados dentro do mesmo período de tempo de CPU (critério de parada), variando de acordo com o tamanho da instância. O tempo considerado é *TempoMaximo* = $500 \times n \times m$ milissegundos.

Para validar os resultados obtidos pelos dois algoritmos e verificar se há diferenças estatísticas significantes, foi realizado um teste-t (MONTGOMERY, 2006) para a comparação das médias dos dois algoritmos. O nível de confiança utilizado foi de 95%. Foi observado que o *P-value* computado foi menor que 0,05 o que mostra que há uma diferença significativa entre os algoritmos. O teste Tukey da Diferença HSD com nível de confiança de 95% mostrado na Figura 5 também reforça o resultado. Note que o algoritmo GRASP_RVND apresenta um desempenho melhor que o GRASP_VND uma vez que não há sobreposição dos intervalos.

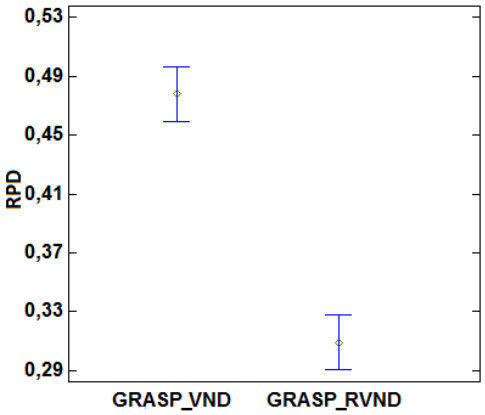

Figura 5: Gráfico das médias e intervalos HSD de Tukey com nível de confiança de 95% para o GRASP_VND e GRASP_RNVD

4.5. Comparação da heurística GRASP_RVND com um algoritmo da Literatura

Tabela 3: RPD médio para os algoritmos SA e GRASP_RVND

N	*m*	Peso (w)	SA	GRASP_RVND	*n*	*m*	Peso (w)	SA	GRASP_RVND
40	4	0,2	39,859	0,309	60	4	0,2	121,003	0,342
		0,4	36,746	0,383			0,4	56,701	0,249
		0,6	43,445	0,313			0,6	43,768	0,215
		0,8	25,255	0,207			0,8	27,423	0,342
	8	0,2	57,607	0,331		8	0,2	35,116	0,181
		0,4	29,430	0,267			0,4	30,587	0,251
		0,6	24,695	0,348			0,6	20,728	0,274
		0,8	18,825	0,277			0,8	19,788	0,285
50	4	0,2	145,121	0,257	70	4	0,2	115,013	0,512
		0,4	91,307	0,404			0,4	49,706	0,431
		0,6	43,997	0,302			0,6	46,153	0,325
		0,8	31,793	0,307			0,8	30,037	0,400
	8	0,2	82,756	0,239		8	0,2	22,254	0,313
		0,4	29,828	0,221			0,4	29,854	0,380
		0,6	31,058	0,262			0,6	32,496	0,273
		0,8	22,126	0,243			0,8	25,186	0,375

Os resultados obtidos pela heurística GRASP_RVND foram comparados com os resultados da heurística *Simulated Annealing* (SA) proposta por Hatami *et al.* (2010). O

algoritmo SA foi reimplementado na linguagem C++ seguindo o artigo original. No algoritmo SA também foi utilizado a condição de parada usada no GRASP_RVND (tempo de execução = $500 \times n \times m$ milissegundos). Para cada uma das 576 instâncias o SA foi rodado 5 vezes, e nas comparações foram consideradas as médias dos valores da função objetivo.

A Tabela 3 mostra o resultado médio do RPD para as instâncias com 40, 50, 60 e 70 tarefas, com 4 e 8 máquinas paralelas no estágio 1 e para cada um pesos *w* usados na função objetivo. Claramente é possível perceber que os valores de RPD para o GRASP_RVND são menores que 0,5%, isto mostra que ele obteve resultados bem melhores em comparação com o algoritmo SA.

Os resultados são validados aplicando um teste-t para a comparação das médias do RPD. Na Figura 6 é mostrado são mostrados as médias e os intervalos HSD de Tukey com nível de confiança de 95% para os algoritmos comparados. Como não há sobreposição entre os intervalos, conclui-se que o GRASP_RVND é estatisticamente melhor que o SA.

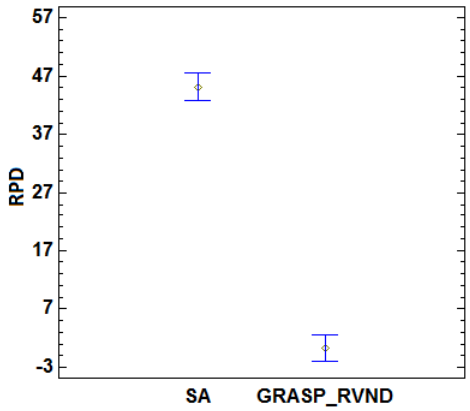

Figura 6: Comparação entre os algoritmos SA e GRASP_RVND através das médias e intervalos HSD de Tukey com nível de confiança de 95%

5. Conclusão

Neste trabalho foi abordado o problema de programação de tarefas em um ambiente de produção *assembly flowshop* com três estágios: fabricação de componentes, transporte e montagem. A proposta de estudo foi examinar a aplicabilidade de novas abordagens heurísticas, além das apresentadas por Hatami *et al.* (2010). Para isso foi proposto uma heurística híbrida fundamentada nas metaheurísticas GRASP e VND. Foi proposto um novo método, denominado RVND, que utiliza a estratégia *roulette wheel*, para a escolha das vizinhanças. Nesta estratégia cada vizinhança possui uma probabilidade de seleção proporcional à quantidade de soluções melhores obtidas.

Foram realizados diversos experimentos computacionais e análises estatísticas para ajustar cada uma das componentes e parâmetros das heurísticas a fim de obter resultados com maior qualidade. Os experimentos mostraram que o GRASP com RVND apresentou os melhores resultados que o GRASP com o VND simples.

A heurística proposta GRASP_RVND foi comparada com a heurística SA proposta por Hatami *et al* (2010). As heurísticas foram executadas utilizando a mesma condição de parada (critério de tempo). Depois de uma extensiva análise computacional e estatística conclui-se que a heurística proposta apresentou um desempenho superior que o SA.

Agradecimentos: Os autores agradecem ao CNPq e à FAPEMIG pelo apoio financeiro ao desenvolvimento deste trabalho.

Referências

Allahverdi, A., Ng, C. T., Cheng, T. C. E. e Kovalyov, M. Y. (2008), A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, p. 985-1032.

Andrés, C., Hatami, S. (2011). The three stage assembly permutation flowshop scheduling problem. *5th International Conference on Industrial Engineering and Industrial Management*

Al-Anzi F.S., Allahverdi, A. (2007). A self-adaptive differential evolution heuristic for two stage assembly scheduling problem to minimize maximum lateness with setup times. *Europe Journal Operation Research* 182:80–94

Al-Anzi F.S., Allahverdi A. (2009). Heuristics for a two-stage assembly flowshop with bicriteria of maximum lateness and makespan. *Computer & Operation Research* 36:2682–2689

Allahverdi A., Al-Anzi F.S. (2006). A PSO and a tabu search heuristics for the assembly scheduling problem of the two-stage distributed database application. *Computer & Operation Research* 33(4):1056–1080

Dalfard, V. M, Ardakani, A., Banihashemi, T. N. (2011) hybrid genetic algorithm for assembly flow-shop scheduling problem with sequence-dependent setup and transportation times. *Tehni ki vjesnik* 18, 4 pag. 497-504

Eksioglu, B., Eksioglu S., Jain P. (2008) Tabu search algorithm for the flowshop scheduling problem with changing neighborhoods. *Computers & Industrial Engineering* 54 1–11

Feo, T., Resende, M. (1995) Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133.

Glover F. (1989). Tabu search—part I. *ORSA J Comput* 1(3):190–20

Hansen, P., Mladenović, N. (2003). Variable Neighborhood Search. *Handbook of Metaheuristics, International Series in Operations Research & Management Science*, 2003, Volume 57, 145-184, DOI: 10.1007/0-306-48056-5_6

Hatami S., Ebrahimnejad S., Tavakkoli-Moghaddam R., Maboudian Y. (2010). Two meta-heuristics for three-stage assembly flowshop scheduling with sequence-dependent setup times. *Int J Adv Manuf Technol* 50:1153–1164

Koulamas, C., Kyparisis, G. (2001). The three-stage assembly flowshop scheduling problem. *Computer & Operation Research*, 28:689–704

Levitin, A. (2003). Introduction to the design & analysis of algorithms. *Addison-Wesley Reading*.

Loukil T., Teghem J., Tuyttens D. (2005). Solving multi-objective production scheduling problems using metaheuristics. *Europe Journal Operation Research* 161:42–61

Lustosa, L., Mesquita, M., e Oliveira, R. (2008). Planejamento e controle da produção. *Elsevier*

Montgomery D. C. (2006) Design and analysis of experiments (7th ed.). *New York: Wiley*;

Nawaz, M., Enscore, E., Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

Pinedo, M. (2005). Planning and scheduling in manufacturing and services. *Springer Verlag*.

Potts, C. N., Sevast'janov, V. A., Strusevich, V. A., Van Wassenhove, L. N., E Zwaneveld, C. M. (1995) The Two-Stage Assembly Scheduling Problem: Complexity and Approximation. *Operations Research*, vol. 43 no. 2 346-355.

Shen, W., Wang, L., e Hao, Q. (2006). Agent-based distributed manufacturing process planning and scheduling: a state-of-the-art survey. Systems, Man, and Cybernetics, Part C: Applications and Reviews. *IEEE Transactions on*, 36(4):563–577.

Subramanian, A., Drummond, L. M. A., Bentes, C., Ochi, L. S., Farias, R. (2010) A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery. *Computers & Operations Research* 37 (11), 1899–1911.

Talbi, E-G. (2009). Metaheuristics: from design to implementation. *Jonh Wiley and Sons Inc*.

Tozkapan A., Kirca O., Chung CS. (2003). A branch and bound algorithm to minimize the total weighted flowtime for the two-stage assembly scheduling problem. *Computer & Operation Research* 30:309–320.