Busca Local Iterativa para a Minimização do Atraso Total no Problema de Seqüenciamento de Tarefas em uma Máquina com Setup Times

[1]Gilberto Vinícius P. Nunes, [2]José E. C. Arroyo, [3]Edmar Hell Kamke

Departamento de Informática, Universidade Federal de Viçosa
Campus Universitário da UFV, 36570-00 Centro Viçosa, MG, Brasil
[1]gilbertovinicius@dpi.ufv.br, [2]jarroyo@dpi.ufv.br, [3]edmar.kamke@ufv.br

Abstract. This paper considers the problem of scheduling jobs in a single machine with sequence dependent setup times with the objective of minimizing the total tardiness with respect to the due dates. To solve the boarded problem, this work proposes an efficient Iterative Local Search (ILS) algorithm. The ILS algorithm is compared with two algorithms from the literature, a GRASP algorithm and an Ant Colony Optimization algorithm. Computational tests, on a set of test problems involving up to 85 jobs, show that our ILS heuristic is very competitive.

Keywords: Job scheduling, heuristics, local search, combinatorial optimization.

1 Introdução

Neste artigo é considerado o problema de Seqüenciamento de Tarefas em uma Máquina simples com tempos de preparação (*Setup Time*) dependentes da seqüência. Este problema é denotado por STMST. O objetivo do problema é minimizar o atraso das tarefas com relação a suas datas de entrega. Neste problema, a partir de um tempo zero, um conjunto de n tarefas devem ser processadas em uma máquina disponível continuamente. Cada tarefa j possui um tempo de processamento p_j e uma data de entrega d_j. Se duas tarefas i e j são processadas consecutivamente, existe um tempo de preparação (*setup time*) S_{ij} entre o tempo de finalização da tarefa i e o inicio da tarefa j. Em geral, $S_{ij} \neq S_{ji}$ (*setup time* dependente da seqüência de tarefas). Quando uma tarefa j é a primeira a ser processada, um tempo de preparação inicial S_{0j} é considerado, ou seja, uma tarefa fictícia 0 é incluída. Considere uma seqüência de tarefas (1, 2, …, i, …., n), onde i denota a tarefa na posição i. O problema STMST, abordado neste trabalho, consiste em determinar a permutação de n tarefas que minimize o atraso total definido por:

$T = \sum_{i=1}^{n} T_i$, onde $T_i = \max_{k=1}^{n}\{C_k - d_k, 0\}$ é o atraso da tarefa i e $C_i = \sum_{k=1}^{i}(S_{k-1,k} + p_k)$ representa o tempo de finalização da tarefa i.

A maioria dos trabalhos sobre problemas de programação de tarefas em máquinas não consideram tempos de preparação (*setup times*) dependentes da seqüência. Revisões bibliográficas indicam que tempos de preparação são muito importantes na maioria das situações práticas e devem ser considerados no desenvolvimento de algoritmos para problemas de programação de tarefas [1] [2].

No problema STMST tem sido considerado vários critérios de decisão a serem otimizados. Os principais critérios são: i) minimização do tempo de finalização do processamento da todas as tarefas (*makespan*), ii) minimização do tempo de permanência das tarefas no sistema (tempo de fluxo), iii) minimização de atrasos na execução das tarefas com relação a suas datas de entrega (por exemplo, atraso máximo ou atraso total). O critério i) esta relacionado com a utilização eficiente das máquinas. O critério ii) está associado ao nível de estoque em processamento (*work - in - process*). O critério iii) estabelece uma política da programação da produção que procura atender a ordem do cliente na data de entrega determinada. Critérios de decisão envolvendo datas de entrega são de grande importância nos sistemas de manufatura, pois pode existir uma série de custos quando uma tarefa é entregue com atraso. Dentre estes custos podem ser citados: penalidades contratuais, perda de credibilidade e danos na reputação da empresa que poderão afastar outros clientes [14].

O problema STMST com minimização do atraso total é um problema NP-Difícil [5] e é abordado em vários trabalhos da literatura [2]. Em [12] é proposto um algoritmo exato baseado na técnica *Branch-and-Bound* (B&B). Tan e Narashimha [16] propuseram uma heurística *Simulated Annealing* (SA) que mostrou ser mais eficiente que outras heurísticas propostas anteriormente. Diferentes versões de algoritmos genéticos foram propostos por Rubin e Ragatz [13], Armentano e Mazzini [3] e França [6]. Rubin e Ragatz [13], também, propuseram uma heurística de busca local com múltiplos reinícios (*Random Multi Start Local Search - RMSLS*) e geraram um conjunto de 32 problemas testes com 15, 25, 35 e 45 tarefas. Em [17], utilizando os problemas testes de Rubin e Ragatz, é analisado desempenho de quatro métodos diferentes: B&B de Ragatz [12], SA de Tan e Narashimha [16] e algoritmo genético (AG) e RMSLS de Rubin e Ragatz [13]. As conclusões apresentadas em [17] indica que as heurísticas SA e RMSLS produzem os melhores resultados, especialmente para problemas envolvendo maior número de tarefas. Vale ressaltar que o algoritmo exato B&B de Ragatz [12] gera soluções aproximadas (limites superiores) com relação às soluções ótimas, devido à execução do algoritmo ser limitada a um número máximo de operações (exploração de dois milhões de nós).

Em [7] é desenvolvido uma heurística baseada em otimização por colônia de formigas (*Ant Colony Optimization*), denotada por ACO. A heurística ACO foi comparada com os melhores resultados obtidos por Tan [17] e mostrou ser bastante competitiva e rápida computacionalmente. Uma vantagem da heurística ACO é que ela apresentou soluções de melhor qualidade para problemas com maior número de tarefas (35 e 45 tarefas). Em [7] foram gerados outro conjunto de 32 problemas de maior dimensão divididos em grupos de 55, 65, 75 e 85 tarefas. Os resultados gerados pela heurística ACO, para este novo conjunto de problemas, são disponibilizados.

Gupta e Smith [8] propuseram duas heurísticas diferentes: GRASP e PSLS (*Problem Space-based Local Search*). No algoritmo GRASP de Gupta e Smith é usado um procedimento de pós-otimização baseado na técnica *path-relinking* [9] que

consiste em explorar trajetórias que conectem duas soluções de elite, a primeira denominada "*origem*" e a segunda denominada "*guia*". Gupta e Smith [8] mostraram que a heurística PSLS é bastante competitiva com a heurística ACO de Gagne [7], além disso, a heurística PSLS mostrou ser mais rápida computacionalmente que a heurística ACO. Eles também mostraram que a heurística GRASP é bastante superior a heurística ACO. A desvantagem da heurística GRASP é que consome um tempo computacional alto em comparação à heurística ACO.

Recentemente, Ching e Hsiao [4] desenvolveram outro algoritmo ACO para o problema STMST na qual são minimizados, separadamente, dois critérios: atraso total das tarefas e o atraso total ponderado. Este algoritmo, aqui, é denotado por ACO_{CH}. Para o critério de atraso total, o algoritmo ACO_{CH} apresentou soluções muito melhores que o algoritmo ACO de Gagne [7].

Neste artigo é proposto um algoritmo de busca baseado na heurística *Iterated Local Search* – ILS [11] para gerar soluções aproximadas do problema STMST. O desempenho do algoritmo ILS é avaliado através da comparação com o algoritmo GRASP proposto por Gupta e Smith [8] e com o algoritmo ACO_{CH}.

O algoritmo desenvolvido neste trabalho é testado utilizando os conjuntos de problemas gerados por Rubin-Ragatz [13] e Gagne [7]. Estes problemas são disponibilizados na Internet e foram usadas por vários pesquisadores na avaliação de algoritmos para o problema STMST.

Este artigo é organizado da seguinte maneira. Na Seção 2 são apresentados os detalhes da implementação do algoritmo ILS proposto. Os testes realizados e resultados computacionais são mostrados na Seção 3. A Seção 4 apresenta as conclusões do trabalho.

2 Busca Local Iterativo (*Iterated Local Search* - ILS)

Iterated Local Search (ILS) é uma algoritmo heurístico inicialmente proposto por Lourenço, Martin e Stützle [11], é baseada na ideia de que um procedimento de busca local pode ser melhorado, gerando-se novas soluções de partida, as quais são obtidas por meio de perturbações numa solução ótima local. A perturbação deve permitir que a busca local explore diferentes soluções e, além disso, deve evitar um reinício aleatório. O método ILS é, portanto, um método de busca local que procura focar a busca não no espaço completo de soluções, mas em um pequeno subespaço, definido por soluções que são ótimas locais [11].

O algoritmo ILS possui 4 etapas principais: Obtenção de uma solução ótima local inicial s^*, perturbação da solução s^* obtendo uma solução s', melhoria da solução s' obtendo $s^{*'}$ (busca local) e um critério de aceitação da solução atual. Na Figura 1 ilustra-se o funcionamento do algoritmo ILS. As três últimas etapas são executadas iterativamente durante um número n_{ILS} de vezes e algoritmo retorna a melhor solução obtida durante toda sua a execução. O pseudocódigo da Figura 2 mostra os passos da heurística ILS. A heurística ILS é fortemente dependente da solução inicial ótima local. Neste trabalho, para a geração da solução inicial, é utilizado um algoritmo guloso construtivo seguido de um algoritmo de busca local.

Nas subseções seguintes são detalhadas cada uma das etapas do algoritmo ILS aplicado ao problema STMST onde é minimizado o atraso total das tarefas com relação a suas datas de entrega.

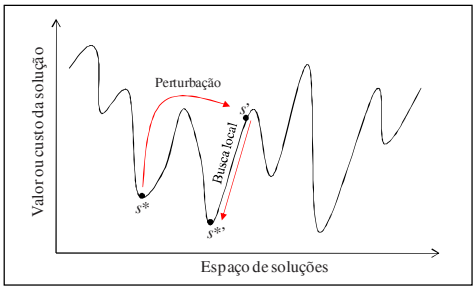

Fig. 1. Representação gráfica do algoritmo ILS.

Procedimento ILS (n_{ILS}) //n_{ILS} =*número de iterações*
1. s^* ← **ObtemSolucaoInicial** ( );
2. **Para** i ← 1 até n_{ILS} **faça**
3. s' ← **Perturbação**(s^*);
4. $s^{*'}$ ← **BuscaLocal**(s');
5. **Se** $f(s^{*'}) < f(s^*)$ **então**//*critério de aceitação*
6. $s^* \leftarrow s^{*'}$;
7. **fim Se**.
8. **fim para**.
9. Retorne s^*;

fim ILS

Fig. 2. Pseudocódigo do algoritmo ILS.

2.1 Obtenção da Solução Inicial

Nesta etapa é construída uma solução inicial (seqüência de todas as tarefas) tentando minimizar o atraso total. Iniciando com uma seqüência vazia, passo a passo é inserida uma tarefa que é escolhida de forma gulosa. Como estratégia gulosa é utilizada a regra de despacho proposta por Gupta e Smith [8] para minimizar o atraso das tarefas no problema STMST. Esta regra consiste em ordenar as tarefas não inseridas na seqüência de acordo com a seguinte função de custo:

$$Custo(j) = \{d_j - (p_j + C_k)\} (S_{kj} + p_j),$$

onde, d_j e p_j são, respectivamente, a data de entrega e o tempo de processamento da tarefa j candidata para ser inserida na seqüência. C_k é o tempo de finalização da última tarefa k da seqüência e S_{kj} é o *setup time* produzido pelas tarefas k e j.

Nesta função, $d_j - (p_j + C_k)$ representa o tempo de folga de uma tarefa j com relação a sua data de entrega, $(S_{kj} + p_j)$ é o tempo de finalização (*completion time*) da tarefa j processada logo após a tarefa k. Se uma tarefa tiver bastante tempo folga, ela pode ser inserida na seqüência (ou seja, processada) mais tarde, não afetando o valor do atraso total. Se uma tarefa tiver um tempo baixo de folga ou um valor negativo da mesma, a tarefa deve ser seqüenciada imediatamente. Se duas tarefas tiverem o

mesmo tempo de folga, então a tarefa j com menor tempo de finalização ($S_{kj} + p_j$) será escolhida para ser inserida na seqüência.

Gupta e Smith [8] observaram que a função *Custo* proposta por eles apresentou um melhor desempenho que a função ATCS (*Apparent Tardiness Cost with Setups*) proposta em [10], em termos de qualidade de soluções obtidas e tempo computacional. A regra ATCS já foi considerada uma das melhores regras para a minimização de atrasos no problema STMST.

O algoritmo de construção pode ser resumido nos seguintes passos:

Seja *LC* o conjunto de todas as tarefas e $s^* = \varnothing$ a seqüência a ser construída. Selecionar a tarefa j que possua o menor valor de *Custo* para ser inserida em s. Remove-se a tarefa j de *LC*, reordenam-se as tarefas (utilizando a função de custo) e processo é repetido escolhendo uma nova tarefa j. A construção finaliza quando é obtida uma seqüência s contendo todas n tarefas.

A solução (seqüência s^*) obtida pelo algoritmo de construção é melhorada pelo algoritmo de busca local detalhado na subseção 2.3. Vale lembrar que a solução obtida será um ótimo local, pois o processo da busca local finaliza quando não seja possível melhorar mais a solução (isto significa que a busca fica presa em uma região do espaço de soluções do problema).

2.2 Perturbação da Solução

O objetivo da perturbação é escapar do atual ótimo local e explorar outra região promissora de soluções. Para isto, a partir da solução atual s^* é gerada outra solução s' fazendo uma perturbação. Como uma solução é formada por uma seqüência de n tarefas, a perturbação consiste em fazer duas trocas aleatórias de tarefas. A primeira troca é feita entre duas tarefas adjacentes e a segunda entre duas tarefas que se encontram a d posições distantes, onde $d = max\{n/3, 15\}$. Esta forma de perturbação foi também usada por Stützle [15], na solução de um problema de programação de tarefas.

A solução s' obtida pela perturbação deve ser melhorada pela busca local para obter outro um ótimo local $s^{*'}$.

2.3 Busca Local

A busca local é um procedimento iterativo que consiste em melhorar uma solução s' procurando novas soluções vizinhas dela. Estas soluções vizinhas são obtidas realizando alterações (*movimentos*) na estrutura da solução atual s'. Sempre escolhe-se um vizinho s'' dentre todos os vizinhos de s'. Se o vizinho escolhido s'' é melhor que s', a busca continua a partir de s'' (ou seja, $s' \leftarrow s''$). O procedimento finaliza quando não seja possível melhorar a solução atual s', ou seja, quando s' é um ótimo local.

Neste trabalho foram utilizados dois tipos de movimentos para a obtenção de soluções vizinhas:

- *Movimento de inserção*: um vizinho de *s*' é gerado inserindo uma tarefa que está na posição *i*, em outra posição *j* da seqüência, $1 \leq i, j \leq n$ com $j \neq i$. Usando este movimento é possível gerar $(n\text{-}1)^2$ vizinhos.
- *Movimento de troca*: uma solução vizinha de *s*' é gerada fazendo a troca de duas tarefas *i* e *j* da seqüência, $1 \leq i, j \leq n$ com $i \neq j$. O número de vizinhos gerados com este movimento é $n(n\text{-}1)/2$.

No algoritmo implementado, inicialmente é executada uma busca local que utiliza o movimento de *inserção*. A solução obtida após dessa busca local é ainda melhorada usando uma busca local que utiliza o movimento de *troca*.

O critério utilizado para escolher um vizinho, é baseado no "*primeiro melhor*", ou seja, a primeira solução vizinha que melhore a solução atual (*s*') é escolhida.

2.3 Critério de Aceitação e Critério de Parada

A solução ótima local *s**' obtida pela busca local é aceito se ela melhorar a melhor solução atual *s**, caso contrário ela é descartada (veja passos 6 e 7 do pseudocódigo da Figura 2) .

No algoritmo ILS podem ser aceitos, com uma pequena probabilidade, soluções que pioram a melhor solução. Neste trabalho, foi testado o aceite de soluções piores, no entanto os resultados obtidos não foram satisfatórios.

O critério de parada do algoritmo é baseado número de iterações (n_{ILS}) executados. Na avaliação do desempenho do algoritmo ILS foram testados diferentes valores para o parâmetro n_{ILS}. O valor que gerou bons resultados foi $n_{ILS} = 2000$.

3 Testes Computacionais

O algoritmo ILS proposto neste trabalho foi programado na linguagem C++ e os testes computacionais foram realizados no ambiente do sistema operacional Linux (kernel 2.6.17) utilizando um computador com processador Pentium IV de 3.0 GHz e com 1 GB de RAM.

O desempenho do algoritmo ILS é avaliado através da comparação com os algoritmos GRASP de Gupta e Smith [8] e ACO_{CH} de Ching e Hsiao [4], ambos propostos para o mesmo problema abordado neste trabalho. Vale ressaltar que os dois algoritmos, GRASP e ACO_{CH} apresentaram um desempenho superior (em termos de qualidade de soluções) com relação ao algoritmo ACO de Gagne [7].

Os testes computacionais do algoritmo proposto são realizados utilizando dois conjuntos de problemas teste disponíveis na literatura:

Conjunto 1: composto por 32 problemas teste gerados por Rubin e Ragatz [13]. Neste conjunto, para cada $n \in \{15, 25, 35 \text{ e } 45\}$ existem 8 problemas.

Conjunto 2: formado também por 32 problemas que foram gerados por Gagne [7]. Para cada $n \in \{55, 65, 75 \text{ e } 85\}$ existem 8 problemas.

Para os problemas do Conjunto 1, os algoritmos GRASP e ILS foram executados 20 vezes. Já o algoritmo ACO_{CH} foi executado somente 10 vezes [4]. Para os problemas do Conjunto 2, os três algoritmos foram executados 20 vezes. Neste

trabalho, para cada problema, comparam-se as melhores soluções obtidas pelos três algoritmos (ILS, GRASP e ACO_{CH}) em todas as execuções.

Para o Conjunto 1 de problemas (com número de tarefas variando entre 15 e 45) as melhores soluções obtidas pelos algoritmos ILS, GRASP e ACO_{CH} são comparadas com as soluções determinadas pelo algoritmo *Branch and Boud* (B&B) de Ragatz [12]. Vale lembrar que as soluções geradas pelo algoritmo B&B não necessariamente são ótimas, pois a execução deste algoritmo foi limitada à exploração de dois milhões de nós da árvore de busca.

A eficiência dos algoritmos ILS, GRASP e ACO_{CH} são medidas calculando o erro relativo porcentual com relação ao algoritmo B&B:

$$ER\% = \frac{T - T_{B\&B}}{T_{B\&B}} \times 100\%,$$

onde T é o valor do atraso total obtido por cada algoritmo heurístico (ILS, GRASP ou ACO_{CH}) e $T_{B\&B}$ é o valor do atraso total obtido pelo algoritmo B&B. Vale destacar que o valor de $ER\%$ é considerado zero se $T_{B\&B} = 0$ e $T = 0$. Note que, se $T < T_{B\&B}$, o valor de $ER\%$ será negativo. Neste caso, $-ER\%$ expressa o porcentual de melhoria de um algoritmo heurístico com relação ao método B&B.

Na Tabela 1 são apresentados os resultados da comparação das melhores soluções obtidas pelos algoritmos ILS, GRASP e ACO_{CH}, para os 32 problemas do Conjunto 1. Nesta Tabela pode-se observar que, dos 32 problemas, o algoritmo proposto ILS é melhor que os algoritmos GRASP e ACO_{CH} em 5 problemas. O GRASP é melhor que o ILS e ACO_{CH} em 1 problema. Em 26 problemas os três algoritmos encontraram a mesma solução.

Na Tabela 1 também são apresentados os tempos de CPU em segundos (*Time*) gastos pelos algoritmos ILS e GRASP. Os tempos computacionais do algoritmo ACO_{CH} não são apresentados por Ching e Hsiao [4]. Mesmo que os algoritmos ILS e GRASP tenham sido executados em computadores com processadores diferentes (ILS: Pentium IV 3.0 e GRASP: Pentium IV 2.4), os tempos do algoritmo ILS são bem menores que os tempos do GRASP. Nos 32 problemas do Conjunto 1, as médias de tempo gasto pelos algoritmos ILS e GRASP são 2,23 e 54,84 segundos, respectivamente.

Na Tabela 2 mostram-se os melhores valores do atraso total (T) obtidas pelos algoritmos ILS, GRASP e ACO_{CH} para os 32 problemas do Conjunto 2. Note que o algoritmo ILS obteve o menor valor do atraso total em 13 problemas, o algoritmo ACO_{CH} obteve a melhor solução em 6 problemas. Em nenhum problema o GRASP encontrou uma solução melhor. Em 10 problemas, onde o atraso total é zero, os três algoritmos encontraram as mesmas soluções.

A Tabela 2 também apresenta os tempos computacionais gasto pelos algoritmos ILS e GRASP. Note que na maioria dos problemas os tempos do algoritmo ILS são bem menores que os tempos do algoritmo GRASP. Por exemplo, para o problema p858 com 85 tarefas, o algoritmo GRASP gastou 3714,4 segundos e o algoritmo proposto ILS gastou apenas 52,7 segundos. Os tempos médios gasto, para resolver cada problema do Conjunto 2, pelos algoritmos ILS e GRASP são 20,9 e 982,9 segundos, respectivamente. Ching e Hsiao [4] mencionam que o tempo médio gasto pelo algoritmo ACO_{CH} para resolver cada problema é 24.17 segundos (usando um computador Pentium IV 2.8GHz).

Tabela 1. Resultados para o Conjunto 1 de problemas: comparação dos valores do erro relativo com relação ao algoritmo B&B.

Problema	*n*	B&B	ILS		GRASP		ACO_{CH}
		Atraso total	*ER%*	*Tempo*[a]	*ER%*	*Tempo*[b]	*ER%*
p401	15*	90	0,0	0,3	0,0	4,0	0,0
p402	15*	0	0,0	0,0	0,0	0,0	0,0
p403	15*	3418	0,0	0,2	0,0	3,7	0,0
p404	15*	1067	0,0	0,1	0,0	2,5	0,0
p405	15*	0	0,0	0,0	0,0	0,0	0,0
p406	15*	0	0,0	0,0	0,0	0,0	0,0
p407	15*	1861	0,0	0,2	0,0	3,9	0,0
p408	15*	5660	0,0	0,2	0,0	3,2	0,0
p501	25	264	**-1,4**	1,0	-1,1	14,0	-0,4
p502	25*	0	0,0	0,0	0,0	0,0	0,0
p503	25	3511	**-0,4**	1,2	**-0,4**	18,5	**-0,4**
p504	25*	0	0,0	0,0	0,0	0,0	0,0
p505	25*	0	0,0	0,0	0,0	0,0	0,0
p506	25*	0	0,0	0,0	0,0	0,0	0,0
p507	25*	7225	0,0	1,3	0,0	24,4	0,0
p508	25	2067	**-7,4**	1,6	**-7,4**	23,3	**-7,4**
p601	35	30	**-53,3**	3,8	-46,7	53,3	**-53,3**
p602	35*	0	0,0	0,0	0,0	0,0	0,0
p603	35	17774	-0,95	3,5	**-1,1**	94,4	-0,7
p604	35	19277	**-1,0**	3,1	-0,9	88,8	**-1,0**
p605	35	291	**-21,6**	3,6	-16,5	59,0	-17,5
p606	35*	0	0,0	0,0	0,0	0,0	0,0
p607	35	13274	**-2,3**	3,7	**-2,3**	88,0	-2,0
p608	35	6704	**-29,4**	4,2	**-29,4**	83,5	**-29,4**
p701	45	116	-10,4	7,2	**-11,2**	22,5	**-11,2**
p702	45*	0	0,0	0,0	0,0	0,0	0,0
p703	45	27097	**-2,1**	7,2	-1,8	216,4	-2,0
p704	45	15941	**-4,6**	6,6	**-4,6**	201,3	-3,3
p705	45	234	**-9,4**	6,9	-5,1	129,9	-6,4
p706	45*	0	0,0	0,0	0,0	0,0	0,0
p707	45	25070	**-5,1**	6,8	-5,0	253,3	-4,5
p708	45	24123	**-5,5**	7,7	**-5,5**	267,0	-4,5
Average			**-5,2**	2,23	-4,5	54,84	-4,3

*Problems with optimal solutions. [a] Pentium IV 3.0GHz, 1GB RAM. [b] Pentium IV 2.4 GHz, 1GB RAM.

Tabela 2. Resultados para o Conjunto 2 de problemas: comparação dos melhores valores do atraso total obtidas pelos algoritmos.

Problema	*n*	ILS		GRASP		ACO_{CH}
		T	*Tempo*[a]	*T*	*Ttempo*[b]	*T*
p551	55	209	12,4	242	258,2	**185**
p552	55	**0**	0,01	**0**	0,0	**0**
p553	55	**40600**	12,7	40678	511,8	40676
p554	55	**14653**	9,2	**14653**	497,1	14684
p555	55	**0**	4,6	**0**	219,8	**0**
p556	55	**0**	0,01	**0**	0,0	**0**
p557	55	**35842**	13,2	35883	557,0	36420

p558	55	**19871**	15,3	**19871**	541,8	19888
p651	65	301	20,01	333	460,8	**268**
p652	65	**0**	0,01	**0**	0,0	**0**
p653	65	57586	22,6	57880	1023,0	**57584**
p654	65	**34302**	21,9	34410	939,7	34306
p655	65	16	23,8	30	521,2	**7**
p656	65	**0**	0,01	**0**	0,0	**0**
p657	65	**54913**	23,1	55355	1143,4	55389
p658	65	**27114**	25,4	**27114**	1029,9	27208
p751	75	273	30,4	317	811,1	**241**
p752	75	**0**	0,02	**0**	0,05	**0**
p753	75	**77629**	32,0	78211	1875	77663
p754	75	**35239**	33,0	35323	1671,1	35630
p755	75	**0**	0,03	**0**	0,0	**0**
p756	75	**0**	0,02	**0**	0,0	**0**
p757	75	**59928**	35,2	60217	1848,9	60108
p758	75	**38339**	32,7	38368	2000,3	38704
p851	85	**438**	44,4	531	1355,5	455
p852	85	**0**	0,02	**0**	0,0	**0**
p853	85	**97930**	49,4	98794	3022,4	98443
p854	85	**79272**	50,6	80338	2832,4	79553
p855	85	340	52,9	393	1400,8	**324**
p856	85	**0**	0,02	**0**	0,05	**0**
p857	85	**87265**	52,0	88089	3217,0	87504
p858	85	**74818**	52,7	75217	3714,4	75506
Average		**26152,4**	20,9	26320,2	982,9	26273,3

[a]Pentium IV 3.0GHz, 1GB RAM. [b]Pentium IV 2.4GHz, 1GB RAM.

4 Conclusões

Neste artigo foi proposto um algoritmo ILS para resolver o problema de seqüenciamento de tarefas em uma máquina com tempos de preparação (*setup time*) dependentes da seqüência. O objetivo do problema é minimizar o atraso total das tarefas com relação a suas datas de entrega. É um problema bastante importante que é encontrado em varias situações praticas no planejamento da produção.

O algoritmo proposto usa uma estratégia gulosa para a construção da solução inicial. A solução construída é melhorada por um procedimento de busca local e em seguida submetida ás etapas iterativas do método *Iterated Local Search* (ILS).

Neste trabalho, o algoritmo ILS é comparado com dois algoritmos da literatura, GRASP e ACO_{CH}. Dos testes computacionais realizados podemos concluir que o algoritmo proposto ILS, em termos de qualidade de solução e tempo computacional, é bastante eficiente quando comparado com os algoritmos GRASP e ACO_{CH}. Do total de 64 problemas resolvidos (conjunto 1 e 2 de problemas), o ILS obteve a melhor solução em 18 problemas. Os algoritmos GRASP e ACO_{CH} encontraram a melhor solução em somente 1 e 6 problemas, respectivamente. Os três algoritmos encontraram as mesmas soluções em 28 problemas.

Agredecimentos. Este trabalho foi financiado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq e pela Fundação de Amparo à Pesquisa do Estado de Minas Gerais – FAPEMIG.

Referências

1. Allahverdi, A., Gupta, J.N.D. e Aldowaisan, T.: A review of scheduling research involving setup considerations, OMEGA, 27: 219–239, (1999).
2. Allahverdi, A., Ng, C.T., Cheng, T.C.E. e Kovalyov, M.Y.: A survey of scheduling problems with setup times or costs, European Journal of Operational Research, 187:985–1032, (2008).
3. Armentano, V.A. e Mazzini, R.: A genetic algorithm for scheduling on a single machine with set-up times and due dates. Production Planning and Control, 11: 713–720, (2000).
4. Ching L.J., and Hsiao J.C.: An ant colony optimization for single-machine tardiness scheduling with sequence-dependent setups, Computers & Operations Research, 34 1899–1909, (2007).
5. Du, J. e Leung, J.Y.T.: Minimizing total tardiness on one machine is NP-hard. Mathematics of Operations Research, 15, 483-495, (1990).
6. França, P.M., Mendes, A. e Moscato, P.: A memetic algorithm for the total tardiness single machine scheduling problem. European Journal of Operational Research, 132: 224–242, (2001).
7. Gagne, C., Price, W.L. e Gravel, M.: Comparing an ACO algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times. Journal of the Operational Research Society, 53: 895–906. (2002).
8. Gupta, S.R. e Smith, J.S.: Algorithms for single machine total tardiness scheduling with sequence dependent setups. European Journal of Operational Research, 175: 722–739, (2006).
9. Laguna M. e Martí, R.: GRASP and path relinking for 2-layer straight line crossing minimization. INFORMS Journal on Computing, 11, 44-52. (1999).
10. Lee, Y.H., Bhaskaran, K. e Pinedo, M.: A Heuristic to Minimize the Total Weighted Tardiness with Sequence-Dependent Setups, IIE Transactions, 29, 45–52, (1997).
11. Lourenço, H. R., Martin, O. C. e Stützle, T.: Iterated local search. In F. Glover & G. A. Kochenberger (Eds.), Handbook of metaheuristics (321–353). Boston: Kluwer Academic. (2003).
12. Ragatz G.L.: A branch-and-bound method for minimum tardiness sequencing on a single processor with sequence dependent setup times, In: Proceedings: twenty-fourth annual meeting of the Decision Sciences Institute, 1375–1377, (1993).
13. Rubin P.A. e Ragatz G.L.: Scheduling in a sequence dependent setup environment with genetic search. Computers and Operations Research, 22(1):85–99, (1995).
14. Sen, T. e Gupta, S.K.: A State-of-Art Survey of Static Scheduling Research Involving Due Date, OMEGA, 12: 63-76, (1984).
15. Stützle, T.: Applying iterated local search to the permutation flow shop problem. AIDA-98-04, FG Intellektik, TU Darmstadt, (1998).
16. Tan, K.C. e Narasimhan, R.: Minimizing tardiness on a single processor with sequence-dependent setup times: A simulated annealing approach. OMEGA, 25: 619–634, (1997).
17. Tan, K.C., Narasimhan, R., Rubin, P.A. e Ragatz, G.L.: A comparison of four methods for minimizing total tardiness on a single processor with sequence dependent setup times. OMEGA, 28: 313–326, (2000).