Algoritmo Genético e Busca Local para o problema Just-in-Time Job-Shop Scheduling

Rodolfo P. Araujo, André G. dos Santos, José E. C. Arroyo

Departamento de Informática – Universidade Federal de Viçosa (UFV)
Av. P. H. Rolfs, s/n, DPI – Campus UFV – 36.570-000 – Viçosa, MG – Brasil

rodolfo.araujo@ufv.br, {andre, jarroyo}@dpi.ufv.br

Abstract. *This paper describes a successful combination of genetic algorithm and local search procedure for a just-in-time job-shop scheduling problem with earliness and tardiness penalties. For each job is given a specific order of machines in which its operations must be processed, and each operation has a due date, a processing time, and earliness and tardiness penalties, which are paid if the operation is completed before or after its due date. The problem is very hard to solve to optimality even for small instances, but the proposed combination of genetic algorithm and local search found good solutions for the instances tested. The quality of the solutions is evaluated and compared to a set of instances from the literature, with up to 20 jobs and 10 machines. The proposed algorithm improved the solution value for several of the instances.*

Resumo. *Este artigo descreve uma combinação bem sucedida de algoritmo genético e busca local para o problema de just-in-time job-shop scheduling com penalidades por atraso e adiantamento. As operações de cada tarefa têm uma ordem específica de processamento nas máquinas e cada operação tem um tempo de processamento e penalidades por atraso e adiantamento que são pagos se a tarefa é finalizada depois ou antes da data determinada. Soluções exatas são difíceis até para pequenas instâncias, mas a combinação do algoritmo genético com uma proposta de busca local se mostrou eficiente. A qualidade das soluções é avaliada e comparada com um conjunto de instâncias da literatura com até 20 tarefas e 10 máquinas. O método proposto melhorou o valor da solução para várias instâncias da literatura.*

1. Introdução

Problemas de Just-In-Time Scheduling são muito comuns na indústria. Em um ambiente de just-in-time (JIT) scheduling cada tarefa deve ser finalizada tão próxima da data de entrega quanto possível. Um adiantamento no término de uma tarefa resulta na necessidade de manter o bem produzido temporariamente em estoque, o que resulta em custos de armazenamento. Por outro lado, um atraso no término da tarefa resulta em multas e penalidades, e pode comprometer a credibilidade com os consumidores, ferindo a reputação da indústria.

Um grande esforço tem sido destinado para este tipo de problema, considerando tanto o adiantamento quanto o atraso [Baker e Scudder, 2003], entretanto, dada sua intratabilidade, a maioria dos esforços têm sido dedicados para o problema com uma única máquina [Sourd e Kedad-Sidhoum, 2003], ou, no caso de muitas máquinas, só o atraso é considerado [Mattfeld e Bierwirth, 2004]. Existem poucos trabalhos considerando múltiplas máquinas e penalidades por atraso e adiantamento. Em [Beck e

Refalo, 2003] e [Kelbel e Hanzálek, 2007], por exemplo, todas as operações, exceto a última de cada tarefa possuem penalidades de atraso e adiantamento zero. Assim, não são considerados custos de estocagem das operações intermediárias: as tarefas podem ser terminadas na hora, mas nada se pode dizer quanto às operações intermediárias, que podem demandar custos de estocagem entre as operações

A definição do problema de just-in-time job shop scheduling (JITJSSP) considerada neste artigo é a mesma apresentada por Baptiste, Flamini e Sourd (2008a). Existe um grupo de n tarefas, $J = \{J_1, J_2, \dots, J_n)$ e um grupo de m máquinas $M = \{M_1, M_2, \dots, M_m)$. Cada tarefa J_i possui uma sequência ordenada de operações $O_i = \{o_i^1, o_i^2, \dots, o_i^m)$, onde o_i^k é a k-ésima operação da tarefa J_i. Cada operação o_i^k tem uma data de entrega d_i^k, um tempo de processamento p_i^k, e uma máquina específica $M(o_i^k)$ onde deve ser processada. Para se avaliar uma solução, cada operação o_i^k tem dois coeficientes de penalidade, α_i^k para o adiantamento e β_i^k para o atraso. Sendo C_i^k o tempo de termino da operação o_i^k e $E_i^k = max\ (0, d_i^k, - C_i^k)$ e $T_i^k = max\ (0, C_i^k, - D_i^k)$ seus adiantamento e atraso, ela está adiantada se $E_i^k > 0$ ou atrasada se $T_i^k > 0$.

O objetivo é encontrar um escalonamento factível, isto é, determinar o tempo de término C_i^k de cada operação minimizando a penalidade total por atraso e adiantamento:

$$min \sum_{i=1}^{n} \sum_{k=1}^{m} (\alpha_i^k E_i^k + \beta_i^k T_i^k) \quad (1)$$

Uma solução factível deve obedecer aos critérios de precedência: para cada par de operações consecutivas o_i^{k-1} e o_i^k da mesma tarefa, a operação o_i^k não pode começar antes da operação o_i^{k-1} ser completada, isto é, para $i = 1, \dots, n$ e $k = 1, \dots, m$,

$$C_i^k \geq C_i^{k-1} + p_i^k \quad (2)$$

Considera-se $C_i^0 = 0$ para $i = 1, \dots, n$ para a restrição valer para a 1ª operação.

A solução deve obedecer também às restrições de recurso: duas operações de tarefas distintas o_i^k e o_j^h que precisam ser processadas na mesma máquina não podem ser processadas simultaneamente, isto é, para $i,\ j = 1, \dots, n$, $i \neq j$ e para $k, h = 1, \dots, m$ com $M(o_i^k) = M(o_j^h)$,

$$C_i^k \geq C_j^h + p_i^k \ \ ou \ \ C_j^h \geq C_i^k + p_j^h \quad (3)$$

O job shop scheduling (JSSP) é um dos problemas de sequenciamento mais bem conhecidos, e está entre os problemas mais difíceis de otimização combinatória. O JITJSSP abordado neste artigo é uma extensão do JSSP, que é classificado como NP-difícil [Du e Leung, 1990]. Baptiste, Flamini e Sourd (2008a) obtiveram bons limites inferiores para várias instâncias diferentes do JITJSSP utilizando dois tipos diferentes de relaxações Lagrangianas de um modelo de programação linear mista baseado em (1)-(3). Também calcularam limites superiores com heurísticas sobre as soluções das relaxações e também pelo ILOG CPLEX.

Nos últimos anos, tem aumentado o interesse pelos algoritmos genéticos (AG) na solução de problemas de otimização combinatória e estes tem mostrado resultados bastante promissores em experimentos em várias áreas da engenharia, sendo então propostas várias abordagens para o JSSP baseadas em AGs [Jain e Meeran, 1999].

Assim como em outras meta-heurísticas para o JSSP, processos de busca local para melhorar o resultado das soluções são largamente utilizados em AGs [Gonçalves, Mendes e Resende, 2005][Gao, Sun e Gen, 2007]. Isto é especialmente importante nos AGs para problemas de seqüenciamento, pois dificilmente se consegue incorporar nos operadores genéticos alguns refinamentos que são simples para as buscas locais.

Neste artigo apresenta-se um algoritmo genético com um processo de busca local eficiente para encontrar boas soluções para o JITJSSP. O método foi testado e a qualidade das soluções avaliada para um grupo de instâncias com até 20 tarefas e 10 máquinas, propostas por Baptiste, Flamini e Sourd (2008a).

Este artigo é organizado da seguinte forma: na Seção 2 os detalhes do algoritmo genético proposto são apresentados; a seção 3 descreve a busca local; os testes computacionais com o algoritmo genético e a busca local são apresentados na Seção 4; finalmente a seção 5 resume as conclusões do trabalho.

2. Algoritmo Genético

A seguir são descritas as características do algoritmo genético usado neste trabalho.

2.1. Representação do Cromossomo

No algoritmo genético proposto cada cromossomo representa uma solução factível, codificada num vetor de $n \times m$ inteiros que contêm os números de 1 a n (representando as tarefas), cada um aparece m vezes (representando as operações de cada tarefa). Estes números representam a ordem em que as tarefas devem ser escalonadas nas máquinas. Cada operação é escalonada tão cedo quanto possível, obedecendo as restrições de precedência e recursos. Considere por exemplo uma possível solução para um problema de 4 tarefas e 2 máquinas representada na Fig. 1. Como o primeiro gene tem o valor 1, a primeira operação da tarefa 1 é escalonada na máquina $M(o_1^1)$ começando no instante 0.

1	2	4	1	4	3	2	3

Figura 1: Um cromossomo codificando uma solução

O segundo gene tem o valor 2, então a primeira operação da tarefa 2 é escalonada no instante 0 se a máquina $M(o_2^1)$ estiver vazia, ou no instante p_1^1 se esta máquina está ocupada, isto é, se esta é processada na mesma máquina que a tarefa 1 (que estará livre no instante p_1^1). O terceiro gene tem o valor 4, e será escalonado na máquina $M(o_4^1)$ para começar no instante 0, p_1^1, p_2^1, ou $p_1^1 + p_2^1$, dependendo de quando esta máquina estará livre. Isto depende de quais das operações foram escalonadas nesta máquina: nenhuma, uma ou as duas operações das tarefas anteriores. Agora o quarto gene também é 1, o que significa que a segunda operação da tarefa 1 é a próxima a ser escalonada (pois este é o segundo gene 1). Esta será escalonada tão cedo quanto for possível na máquina $M(o_1^2)$: no instante p_1^1 (por causa das restrições de precedência) ou quando a operação atual nesta máquina for completada (por causa das restrições de recurso), sendo escolhido aquele que for mais tarde.

A mesma idéia é aplicada para todas as outras operações, na ordem 4, 3, 2 e 3. Se um gene tem um valor i, e este é a k-ésima ocorrência deste valor, a operação o_i^k será escalonada na máquina $M(o_i^k)$ no instante $max(C_i^{k-1}, C_j^h)$, onde C_j^h é o tempo de termino da última operação na máquina $M(o_i^k)$ ou 0 se ainda não há tarefas na máquina.

Por exemplo, seja a seguinte designação pré-definida das operações às máquinas

$$M(o_1^1) = M(o_2^2) = M(o_3^1) = M(o_4^2) = 1$$

$$M(o_1^2) = M(o_2^1) = M(o_3^2) = M(o_4^1) = 2$$

A Fig.2 mostra o escalonamento dessas operações de acordo com o cromossomo da Fig. 1. O tamanho do retângulo indica o tempo de processamento. Note que apesar da máquina 1 estar livre no instante t_a após a operação o_1^1 de J_1 ser completada, a operação o_4^2 de J_4 não pode iniciar antes do instante t_b pois a operação o_4^1 ainda está sendo processada na máquina 2 (restrição de precedência). A primeira ocorrência do valor 3 no cromossomo indica a operação o_3^1 de J_3, mas esta não pode ser processada antes do instante t_c porque a máquina está ocupada processando o_4^2 (restrição de recurso). No instante t_d a operação o_3^1 está completa na máquina 1 então o_3^2 começa na máquina 2.

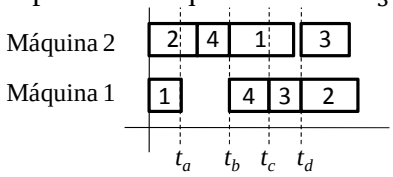

Fig. 2. Escalonamento das operações na ordem da Fig. 1.

Há um total de $(n.m)!/(m!^n)$ possíveis cromossomos, todos representando possíveis soluções. Este é um número muito grande, até mesmo para poucas máquinas. Como as datas de entrega são em geral muito "apertadas", quase não permitindo nenhum tempo intermediário, pode-se considerar que qualquer seqüência possível pode ser representada por uma destas seqüências.

O fitness de um cromossomo é a penalidade total da solução representada, e pode ser calculado enquanto as operações são escalonadas, ou seja, transformando a representação do cromossomo em um escalonamento válido, como apresentado acima.

2.2. População Inicial

A população inicial contém 200 cromossomos gerados aleatoriamente. A seção 4 mostra que resultados muito bons foram obtidos mesmo com essa população inicial aleatória, então por enquanto não foi usado nenhum outro esquema mais elaborado.

2.3. Fase de Seleção

Em cada iteração toda a população é substituída por uma nova de mesmo tamanho, preservando apenas o melhor cromossomo. O cromossomo com melhor valor de fitness é sempre mantido na próxima população para evitar a perda de uma boa solução (elitismo). Cada um dos outros cromossomos da nova população são selecionados por um torneio ternário: três cromossomos são escolhidos aleatoriamente e aquele com melhor fitness (menor valor de penalidade total) é selecionado.

2.4. Operador de Crossover

Os cromossomos selecionados da população são combinados por um operador de crossover. Apenas o melhor dos cromossomos é diretamente copiado para a nova população, todos os outros são gerados pelo operador de crossover.

Dois cromossomos são selecionados aleatoriamente; seus genes são combinados resultando dois novos cromossomos, e só um sobrevive para a próxima iteração, o que tem o melhor valor de fitness. O operador de crossover usado para combinar os genes foi proposto por Zhang, Rao e Li (2008): cria-se um conjunto J com $p\%$ das n tarefas escolhidas aleatoriamente. Todos os genes do cromossomo 1 que pertencem a J são copiados na mesma posição para o filho 1, preenchendo-se então $p\%$ do cromossomo. Os demais genes desse filho são preenchidos com os genes que não pertencem a J, na ordem em que aparecem no cromossomo 2. O outro filho é construído da mesma forma, invertendo-se o papel dos cromossomos 1 e 2.

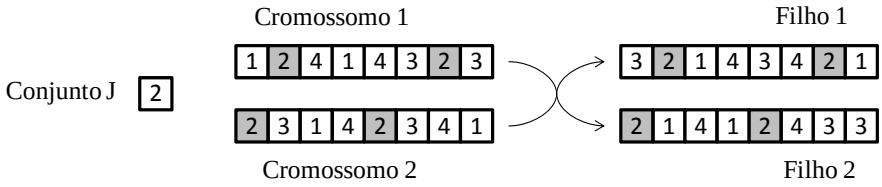

Fig. 3. Operador de crossover usando um conjunto escolhido aleatoriamente

Experimentalmente, foi fixado o tamanho do conjunto J em 20% das tarefas.

2.5. Operador de mutação

O operador de mutação é usado para diversificar a população e consiste em trocar aleatoriamente dois genes num cromossomo, isto é, trocar a ordem de duas operações na seqüência. O melhor cromossomo é copiado sem sofrer mutação, e cada um dos outros tem uma probabilidade p_m= 5% de sofrer mutação, valor este valor obtido experimentalmente.

2.6. Critério de parada

As populações seguem iterativamente pelas fases de seleção, crossover e mutação. Este processo se repete $niter$ vezes, e a melhor solução encontrada pela última população é retornada, conforme pseudo-código mostrado na Fig. 4.

Os resultados apresentados na seção 4 foram obtidos com $niter$ = 100. Experimentalmente notou-se que além desse valor a solução raramente é melhorada, e como o algoritmo genético termina em poucos segundos, não é necessário diminuir este valor.

Algoritmo Genético – AG
1. $P \leftarrow$ População aleatória
2. **repetir** $niter$ **vezes**
3. $P' \leftarrow$ Seleção(P)
4. $P'' \leftarrow$ Crossover(P')
5. $P \leftarrow$ Mutação(P'')
6. **fim repetir**
7. **retornar** Melhor Solução de P

fim AG

Fig 4. Pseudo-código do Algoritmo Genético

3. Busca Local

A melhor solução encontrada pelo algoritmo genético é submetida a um processo de busca local que explora a vizinhança por trocas de inteiros consecutivos do vetor solução (genes no cromossomo). Como a solução contém $n \times m$ inteiros, cada solução tem $n \times m - 1$ vizinhos. Foram avaliadas algumas estratégias comuns: expandir toda a vizinhança e recomeçar a busca local com o melhor vizinho se este for melhor que o atual; recomeçar a busca local tão logo uma solução melhor fosse encontrada. Em ambos os casos, a busca local terminava quando nenhum vizinho é melhor que a solução atual.

Entretanto, também é proposta uma nova estratégia de busca local, que obteve melhores resultados: ao invés de manter apenas uma solução há uma lista L de tamanho limitado $MaxL$ de potencias soluções. A lista sempre é mantida ordenada pelo valor da solução, e a melhor solução é expandida primeiro. Uma vez que a solução é expandida, a lista é atualizada mantendo as melhores soluções entre as soluções que já estavam na lista e os vizinhos da solução expandida. A busca local termina quando todos os vizinhos das soluções na lista foram considerados, ou seja, quando todas as soluções na lista já foram expandidas. O processo termina, uma vez que um vizinho é incluído na lista apenas se seu valor é melhor que todos os presentes na lista. Como o valor da solução ótima é limitado, em algum momento a lista não é mais atualizada.

A busca local é apresentada na Fig. 5 em forma de pseudo-código. O operador de união ∪ "junta" as duas listas, retornando uma lista ordenada contendo os melhores $MaxL$ elementos das duas listas. Pode ocorrer de nenhuma solução ser adicionada a lista (quando todos os vizinhos são piores que as soluções da lista), ou a lista ser completamente atualizada (quando todos os vizinhos são melhores que as soluções da lista) ou ser apenas parcialmente atualizada. É bom notar também que a lista pode conter soluções já expandidas e soluções ainda não expandidas, e a solução que está sendo expandida pode continuar na lista, se ainda estiver entre as $MaxL$ melhores já geradas. A grande vantagem é que mesmo se nenhum vizinho for melhor que a solução expandida, eles ainda podem ser adicionados à lista e dependendo das soluções na lista, eles podem ser expandidos em iterações futuras, evitando alguns mínimos locais.

Busca Local – BL

1. $L \leftarrow$ solução retornada pelo AG
2. **enquanto** há solução em L não expandida **faça**
3. $x \leftarrow$ Primeira solução em L não expandida
4. Marcar x como expandida
5. $N \leftarrow$ lista de vizinhos de x
6. $L \leftarrow L \cup N$
7. **fim enquanto**
8. **retornar** Primeira solução em L

fim BL

Fig. 5. Pseudo-código do processo de busca local

No processo de busca local foi usada uma lista com tamanho $MaxL$ = 10. Se a lista for menor, a busca local torna-se muito gulosa e perde a habilidade de escapar de alguns mínimos locais. Para listas maiores, o tempo computacional aumenta bastante sem, contudo, melhorar significativamente os resultados.

4. Resultados Experimentais

Esta seção mostra os resultados experimentais do algoritmo genético com a busca local.

4.1. Instâncias

As instâncias foram geradas por Baptiste, Flamini e Sourd (2008b). Existem oito instâncias para cada combinação de $(n, m) \in (\{10,15,20\} \times \{2,5,10\})$. Em todas as instâncias as tarefas são processadas exatamente uma vez em cada máquina, e a máquina onde cada operação deve ser processada é escolhida aleatoriamente, com tempos de processamento escolhidos no intervalo [10,30]. As instâncias foram nomeadas por `I-n-m-DD-W-ID`, sendo

- `DD` é a distância entre duas datas de entrega de operações consecutivas, que pode ser o tempo de processamento da última operação (`DD = tight`), ou esse valor acrescido de um valor aleatório no intervalo [0,10] (`DD = loose`).
- `W` é o valor relativo dos custos de adiantamento e atraso (α, β), em que ambos podem ser escolhidos aleatoriamente em [0.1,1] (`W = equal`) ou α em [0.1,0.3] e β em [0.1,1] (`W = tard`).
- Duas instâncias foram geradas para cada combinação dos parâmetros (`ID = 1`) e (`ID = 2`).

  Isto leva a um total de 72 instâncias, todas disponíveis on-line.

4.2. Resultados

Os algoritmos foram implementados na linguagem de programação C++, e os resultados foram obtidos em um computador desktop Intel Pentium IV 3.0 GHz, com 2GB RAM. Os resultados são comparados com os de Baptiste, Flamini e Sourd (2008a), que criaram as instâncias. Apesar do objetivo principal do referido artigo ser comparar limites inferiores, foram também encontrados limites superiores heuristicamente, baseados nas soluções da relaxação lagrangeana. Como o método proposto neste artigo produz soluções viáveis, a comparação será feita com as soluções viáveis do artigo citado, ou seja, com seus limites superiores. A qualidade da solução é avaliada pelo percentual de melhoria do limite superior. O critério de comparação é então:

$$\%melhoria = \frac{UB - S_{AG+BL}}{UB} \times 100 \qquad (4)$$

onde S_{AG+BL} é a solução obtida pelo Algoritmo Genético com a Busca Local, e UB é o limite superior fornecido por Baptiste et al. (2008a). Os resultados para as instâncias com 15 e 20 tarefas são listados nas Tabelas 1 e 2, que contêm os seguintes dados.

- LB e UB: melhor limite inferior (lower bound) e limite superior (upper bound) publicados em [Baptiste, Flamini e Sourd, 2008a], calculados por relaxação lagrangeana das restrições de precedências e de máquinas e pelo ILOG CPLEX.
- S_{AG+BL}: valor da solução encontrada pelo método proposto, algoritmo genético com busca local. Esse valor é a média de três execuções do método.
- %melhoria: a melhoria percentual, calculada por (4).
- CPU(s): tempo, em segundos, gastos pelo algoritmo de Baptiste, Flamini e Sourd (2008) que encontrou o melhor limite inferior, e pelo o algoritmo genético com a busca local. Um tempo 0 indica solução obtida em menos de 1 segundo.

Em cada tabela, as instâncias são agrupadas em quatro grupos, de acordo com a distância entre as datas de entrega (*tight* ou *loose*) e com o valor relativo dos custos (*equal* ou *tard*). Os valores em negrito na coluna *%melhoria* evidenciam as instâncias em que o método proposto melhorou o limite superior conhecido.

Tabela 1: Resultados para instâncias com 15 tarefas

Instância	LB	UB	CPU (s)	S_{AG+BL}	%melhoria	CPU (s)
I-15-2-tight-equal-1	3316	3559	143	3441,8	**3,3**	0
I-15-2-tight-equal-2	1449	1579	76	1491,1	**5,6**	0
I-15-5-tight-equal-1	1052	1663	260	1484,1	**10,8**	3
I-15-5-tight-equal-2	1992	2989	355	3023,8	-1,2	3
I-15-10-tight-equal-1	4389	8381	555	7270,0	**13,3**	48
I-15-10-tight-equal-2	3539	7039	825	6100,5	**13,3**	37
I-15-2-loose-equal-1	1032	1142	33	1053,0	**7,8**	0
I-15-2-loose-equal-2	490	520	10	506,3	**2,6**	0
I-15-5-loose-equal-1	2763	4408	1569	3541,2	**19,7**	2
I-15-5-loose-equal-2	2818	4023	323	3529,1	**12,3**	6
I-15-10-loose-equal-1	758	1109	267	1520,4	-37,1	50
I-15-10-loose-equal-2	1242	2256	395	3050,1	-35,2	43
I-15-2-tight-tard-1	786	913	741	794,2	**13,0**	0
I-15-2-tight-tard-2	886	956	72	915,6	**4,2**	0
I-15-5-tight-tard-1	1014	1538	556	1478,3	**3,9**	3
I-15-5-tight-tard-2	626	843	34	771,8	**8,4**	3
I-15-10-tight-tard-1	649	972	77	1250,9	-28,7	37
I-15-10-tight-tard-2	955	1656	268	1692,9	-2,2	48
I-15-2-loose-tard-1	650	730	62	660,6	**9,5**	0
I-15-2-loose-tard-2	278	310	13	312,2	-0,7	0
I-15-5-loose-tard-1	1098	1723	110	1400,2	**18,7**	4
I-15-5-loose-tard-2	314	374	29	480,7	-28,5	5
I-15-10-loose-tard-1	258	312	106	605,3	-94,0	30
I-15-10-loose-tard-2	476	855	243	837,2	**2,1**	47

Os resultados para instâncias com 10 tarefas não são mostrados por questão de espaço. Para essas instâncias o método proposto encontrou soluções melhores para apenas 5 das 24 instâncias, porém com tempo computacional inferior a 1 segundo.

Aumentando-se o número de tarefas para 15, já foi capaz de melhorar a solução em 16 das 24 instâncias, e em 7 dessas a melhoria foi de mais de 10%, sendo que em todas as instâncias o tempo computacional foi menos de 1 minuto, um tempo muito curto, quando comparado ao tempo de execução dos outros métodos, que gastam vários mintuos para obter resultados semelhantes. Por exemplo, uma solução quase 20% melhor foi encontrada para a instância `I-15-5-loose-equal-1` em apenas 2 segundos, contra quase meia hora de execução do outro método.

Para problemas ainda maiores, com 20 tarefas, o método proposto melhorou a solução anteriormente conhecida em 20 dos 24 casos, conforme relatado na Tabela 2. E nos 4 casos remanescentes a solução encontrada foi bem próxima da conhecida, mesmo tendo empregado um tempo computacional bem mais curto. Isso indica que, apesar de haver mais tarefas nestas instâncias, aumentando substancialmente o número de

soluções factíveis, e com isso o espaço factível, o AG com Busca Local ainda trabalha muito bem, e em poucos segundos consegue alcançar, em média, resultados muito melhores em relação ao CPLEX ou às heurísticas baseadas em relaxações lagrangeanas.

Tabela 2: Resultados para instâncias com 20 tarefas

Instância	LB	UB	CPU (s)	S_{AG+BL}	%melhoria	CPU (s)
I-20-2-tight-equal-1	1901	2008	1730	1958,2	**2,5**	0
I-20-2-tight-equal-2	912	1014	37	964,5	**4,9**	0
I-20-5-tight-equal-1	2506	3090	152	3198,8	-3,5	8
I-20-5-tight-equal-2	5817	7537	7585	7109,6	**5,7**	11
I-20-10-tight-equal-1	6708	12951	1970	11017,8	**14,9**	123
I-20-10-tight-equal-2	5705	9435	1190	7698,8	**18,4**	167
I-20-2-loose-equal-1	2546	2708	1250	2607,3	**3,7**	0
I-20-2-loose-equal-2	3013	3318	351	3089,7	**6,9**	1
I-20-5-loose-equal-1	6697	9697	1042	8394,3	**13,4**	16
I-20-5-loose-equal-2	6017	8152	839	7307,2	**10,4**	14
I-20-10-loose-equal-1	3538	6732	1243	7176,0	-6,6	200
I-20-10-loose-equal-2	1344	2516	279	2434,8	**3,2**	111
I-20-2-tight-tard-1	1515	1913	26	1734,2	**9,3**	0
I-20-2-tight-tard-2	1375	1594	31	1452,9	**8,9**	1
I-20-5-tight-tard-1	3244	4147	735	3809,5	**8,1**	9
I-20-5-tight-tard-2	1633	1916	154	1853,6	**3,3**	8
I-20-10-tight-tard-1	3003	5968	623	5123,8	**14,1**	109
I-20-10-tight-tard-2	2740	3788	865	3566,0	**5,9**	171
I-20-2-loose-tard-1	1194	1271	22	1225,4	**3,6**	0
I-20-2-loose-tard-2	735	857	73	784,6	**8,4**	1
I-20-5-loose-tard-1	2524	3377	2707	3542,1	-4,9	10
I-20-5-loose-tard-2	3060	5014	892	4305,2	**14,1**	9
I-20-10-loose-tard-1	2462	6237	678	5389,7	**13,6**	188
I-20-10-loose-tard-2	1226	1830	392	2095,0	-14,5	117

5. Conclusões

Neste artigo é apresentado um algoritmo genético para resolver o JITJSSP. As instâncias usadas consideram penalidades para o adiantamento e atraso para todas as operações, e não só para as tarefas. O AG é combinado com um processo de busca local.

O método foi testado com 72 instâncias e os resultados comparados com os melhores limites superiores conhecidos. O método AG com a Busca Local proposta se mostrou muito eficiente, tendo encontrado melhores soluções para a maioria das instâncias, com um tempo de execução significativamente menor que o das heurísticas baseadas na relaxação lagrangiana e programação inteira.

Uma observação interessante é que a melhoria do método proposto torna-se mais significativa nas instâncias que possuem mais tarefas, justamente as mais difíceis de serem resolvidas. Para o último grupo, com 20 tarefas e 2, 5 e 10 máquinas, encontrou-

se resultados melhores em 20 das 24 instâncias. Isto indica que, apesar do desempenho da programação inteira e dos métodos de relaxação lagrangiana tenderem a diminuir com o aumento do tamanho do problema, o desempenho do método proposto não diminui consideravelmente, e por isso obtém resultados relativamente bem melhores, e sem aumentar significativamente o tempo computacional.

Como continuação deste trabalho pretende-se utilizar e propor outros operadores de crossover e mutação, e aplicar processos de intensificação da solução.

Agradecimentos

Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) pelo financiamento do projeto APQ3768-6.01/07.

Referências

Baker, K. R. e Scudder, G. D. (1990) "Sequencing with earliness and tardiness penalties: A review", In *Operations Research*, vol. 38(1), pp.22–36.

Baptiste, P., Flamini M. e Sourd F. (2008a) "Lagrangian bounds for just-intime job-shop scheduling", In *Computers & Operations Research*, vol. 35, pp.906–915.

Baptiste, P., Flamini, M. e Sourd, F. (2008b) "Job-shop scheduling with earliness-tardiness penalties", http://wwwpoleia.lip6.fr/~sourd/project/etjssp/, Abril

Beck, J. C. e Refalo, P. (2003) "A hybrid approach to scheduling with earliness and tardiness costs", In *Annals of Operations Research*, vol. 118, pp.49–71.

Du, J. e Leung, Y. T. (1990) "Minimizing total tardiness on one machine is NP-hard", In *Mathematics of Operations Research*, vol. 15, pp.483–495.

Gao, J., Sun, L. e Gen M. (2007) "A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems", In *Computers & Operations Research*, vol. 35(9), pp.2892– 2907.

Gonçalves, J. F., Mendes, J. J. M. e Resende, M. G. C. (2005) "A hybrid genetic algorithm for the job shop scheduling problem", In *European Journal of Operational Research*, vol. 167, pp.77–95.

Jain, A. S. e Meeran, S. (1999) "A state-of-the-art review of job-shop scheduling techniques", In *European Journal of Operational Research*, vol. 113, pp.390–434.

Kelbel, J. e Hanzálek. Z. (2007) "Constraint Programming Search Procedure for Earliness/Tardiness Job Shop Scheduling Problem", In *Proc. of the 26th Workshop of the UK Planning and Scheduling Special Interest Group*, pp. 67–70.

Mattfeld, D. C. e Bierwirth, C. (2004) "An efficient genetic algorithm for job shop scheduling with tardiness objectives", In *European Journal of Operational Research*, vol. 155, pp.616–630.

Sourd, F. e Kedad-Sidhoum, S. (2003) "The one machine problem with earliness and tardiness penalties", In *Journal of Scheduling*, vol. 6, pp.533–49.

Zhang, C., Rao, Y. e Li, P. (2008) "An effective hybrid genetic algorithm for the job shop scheduling problem", In *The International Journal of Advanced Manufacturing Technology*, vol. 39, pp.965–974.