Desenvolvimento de um algoritmo *Hill-Climbing* para minimização do *makespan* em Problemas de Sequenciamento em *Flow Shops*

Ana Carolina Colnago, Ana Paula Barbosa de Morais, Mariana dos Santos Cardoso, Gabriel Otávio Botelho, Rafael Henrique Palma Lima

Resumo: Este artigo propõe um algoritmo *Hill-Climbing* para o problema de sequenciamento em *flow shops* com soluções permutacionais, e tem por objetivo a minimização do tempo total gasto pelo fluxo de atividades em diferentes máquinas, podendo ser aplicado a diversos casos de problemas Flow Shop, independentemente do número de máquinas e atividades. No algoritmo proposto, inicialmente há a geração de uma solução aleatória, que consiste em um vetor com a permutação entre os Jobs da instância selecionada e, partindo disso, o *Hill-Climbing* gera a estrutura de vizinhança utilizando o método da inserção e rodando a quantidade de iterações desejadas até encontrar o ótimo global para a instância escolhida. Neste documento iremos demonstrar a implementação, execução e comparação com soluções obtidas anteriormente por outros autores. Os resultados dos experimentos demonstram que o método escolhido obteve uma melhora significativa quando comparadas a soluções aleatórias, os GAPs obtidos foram baixíssimos sendo o maior deles de 3,0% na 40ª instância.
Palavras chave: Flow Shop Permutacional, *Hill-Climbing*; Minimização do *Makespan*

Development of a Hill-Climbing algorithm to minimize makespan in flow shops scheduling problems

Abstract: This article proposes a Hill-Climbing algorithm for the flow shop sequencing problem with permutational solutions and aims to minimize the total time spent by the flow of activities on different machines, and can be applied to several cases of Flow Shop problems, independently. of the number of machines and activities. In the proposed algorithm, initially there is the generation of a random solution, which consists of a vector with the permutation between the Jobs of the selected instance and from that Hill-Climbing generates the neighborhood structure using the insertion method and rotating the number of iterations. desired until you find the optimal global for the instance you choose. In this document we will demonstrate the implementation, execution and comparison with solutions previously obtained by other authors. The results of the experiments show that the chosen method obtained a significant improvement when compared to random solutions, the obtained GAPs were very low and the highest of them was 3.0% in the 40th instance.
Key-words: Permutational Flow Shop, Hill-Climbing, Makespan Minimization

1. Introdução

Diante da globalização das economias e da ampla difusão da internet as formas de produção e suas estratégias passaram por algumas modificações. As indústrias passaram a ter que lidar com clientes de todas as partes mundo, e para se manterem competitivas muitas das pequenas e médias empresas começaram a planejar sua produção de forma a atender melhor as necessidades individuais de cada consumidor.

Segundo Lopez e Roubellat (2013), o investimento em programação da produção acabou por se tornar uma necessidade para as empresas permanecerem competitivas no mercado, uma vez que, ao programar as tarefas de forma adequada os recursos disponíveis começam a ser

utilizados de uma maneira mais eficiente o que reduz os custos e atende melhor todos os prazos estipulados com os clientes.

Os problemas de programação da produção (tarefas) geralmente são classificados de acordo com o fluxo de operações nas máquinas: máquina única, máquinas paralelas, flow shop, open shop, job shop, flow shop permutacional etc.

Este trabalho trata de um do tipo Flow Shop que consiste na programação do fluxo de *n* tarefas de uma determinada produção em *m* máquinas distintas, na mesma sequência. O somatório do tempo que cada tarefa demora para ser executada em cada máquina é chamado de *makespan*. Todas as tarefas precisam necessariamente passar por todas as máquinas. A otimização do fluxo de tarefas vem sendo alvo de estudos de engenheiros e interessados mundo a fora, já que a otimização desse processo impacta positivamente na produção. Sendo assim, pesquisadores já desenvolveram muitas ferramentas que permitem otimizar o fluxo e minimizar o *makespan*.

Para otimizar esse fluxo e com objetivo de minimizar o *makespan*, existem uma série de ferramentas que se adequam de maneira estratégica em qual será a ordem das tarefas executadas em cada máquina. Para a aplicação do Flow Shop, foram utilizadas 40 instâncias contendo diferentes quantidades de máquinas e tarefas. No primeiro momento o *makespan* foi calculado para cada instância a partir de diferentes ordenações aleatórias das tarefas, resultando em um *makespan* elevado. Para a otimização do fluxo de tarefas, foi escolhida o método de inserção, com a aplicação de *Hill-Climbing*.

Nesse contexto, o objetivo deste trabalho é desenvolver um algoritmo de busca local do tipo *Hill-Climbing* empregando uma vizinhança consolidada na literatura que é a inserção, a qual consiste em gerar uma nova sequência de Jobs (nova vizinhança) a partir de uma sequência atual, transferindo uma tarefa de sua posição inicial e alocando-a em outra posição. O algoritmo desenvolvido foi testado usando 40 instâncias, variando entre elas, o número de máquinas, Jobs e tempo dos processos.

O restante do artigo está organizado da seguinte maneira. A Seção 2 diz respeito ao embasamento e referencial teórico necessário para o entendimento dos problemas de Flow Shop e a formação das estruturas de vizinhança. A Seção 3 traz a implementação e execução do algoritmo proposto, bem como os componentes do algoritmo e alguns exemplos reais com as instâncias utilizadas. A Seção 4 mostra os resultados obtidos através do *Hill-Climbing* e uma breve análise dos mesmos. A Seção 5 conclui este documento com uma contextualização do cenário global e uma maior análise dos resultados obtidos.

2. Referencial Teórico

O problema de programação, de modo geral, pode ser definido como a alocação de recursos no tempo de forma a executar um conjunto de tarefas (MACCARTHY & LIU, 1993). Para French (1982), problemas do tipo flow shop são expressos considerando *n* difrentes Jobs que devem ser processados em *m* máquinas na mesma ordem, cada Job tem uma operação a ser realizada em cada máquina e o tempo de processamento do Job *i* na máquina *j* é igual a *pij.* Baker em 1997 propôs uma ideia muito similar a de French relacionado aos problemas de FSP, porém acrescenta que em um *flow shop* cada tarefa tem sua sequência de processamento com um fluxo linear unidirecional, o que quer dizer que não há retorno no fluxo.

Neste trabalho inicialmente foram apresentadas 40 instâncias diferentes, cada uma com um número de Jobs e máquinas distintos onde o que se propõe é que, a partir da instância selecionada aplica-se o *Hill-Climbing*.

O *Hill-Climbing* é um método de busca por soluções que decide qual será o próximo passo baseado na análise das soluções existentes na vizinhança da solução atual. Dada uma solução *S*, a estrutura de vizinhança *V(S)* é o conjunto de soluções que se encontram próximo a S segundo algum critério e geralmente é encontrada fazendo apenas pequenas mudanças na estrutura da solução inicial.

Neste método, em cada iteração inicialmente parte-se da solução *S* atual e então avalia-se todas as soluções que pertencem a sua vizinhança *V(S*). Se *S'* é a melhor solução da vizinhança *V(S)*, então *S'* será atribuída a *S* e assim tem inicio uma nova iteração.

Há várias formas de se gerar uma estrutura de vizinhança. Uma das mais conhecidas é o Algoritmo Genético, que funciona da seguinte maneira, a partir de um par soluções iniciais ("pais"), é aplicado o operador de cruzamento do Algoritmo Genético puro, o qual seleciona dois melhores (menores) valores de *makespan*, dentre as quatro soluções encontradas ("pais e filhos"). (MOTA, 1996).

Outra forma muito estabelecida na literatura para a geração da estrutura de vizinhança é o método da Busca Tabu, primeiramente apresentada por Glover (1989) mas que teve sua aplicação para o FSP feita por Widmer e Hertz (1989). Muito útil para resolver problemas de otimização combinatória, consiste em dada uma solução inicial factível *S* e sua vizinhança *N(S)*, em cada iteração é feito um movimento de *S* para um vizinho qualquer *S** pertencente *N(S)*, que permite assim que *S** seja pior que *S* com relação a função objetivo. Tem a finalidade de evitar que ciclos repetitivos ocorram associados aos movimentos, para isso se define uma lista tabu de movimentos proibidos na iteração corrente. Essa lista é composta pelos movimentos que são excluídos da iteração pois a levariam de volta a soluções anteriores, evitando que o procedimento entre em ciclo.

Neste trabalho o método escolhido para gerar a estrutura de vizinhança foi o método da inserção, que será explicado na Seção 3.

3. Algoritmo Proposto

Neste trabalho, o algoritmo proposto tem por objetivo a minimização do *makespan*, que nada mais é que a redução do tempo de conclusão das tarefas, resultando em uma utilização mais eficiente dos recursos, enquanto a otimização do tempo de fluxo reduz o número médio das tarefas que estão na fila de espera (PASUPATHY et al., 2006).

Como mencionado anteriormente, o método escolhido para a geração da estrutura de vizinhança foi o método da inserção. Esse método consiste em, dadas as posições do vetor solução, supondo que sejam p(1) e p(2), se p(1) é menor que p(2) então, na estrutura das posições p(2) é inserido logo após p(1) e em seguida a estrutura segue normalmente, e se p(1) for maior que p(2) este é inserido logo após p(2) e o resto do código continua o mesmo. Como o próprio nome sugere, os pontos são apenas inseridos na nova posição, dessa forma o número não se repete e o vetor continua tendo o mesmo tamanho, com a diferença na alocação dos valores. A Figura 1 a seguir mostra de maneira simplificada como esse método funciona.

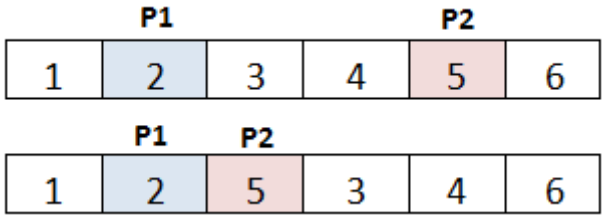

Figura 1 – Exemplo de funcionamento da estrutura com base na inserção

A seguir, a Tabela 1 apresenta as informações da instância na forma tabelada e de melhor visualização.

Matriz Tempos	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
Job 1	57	35	58	72	43	9	84	81	35	31
Job 2	76	96	16	51	59	89	79	52	63	90
Job 3	39	15	80	67	16	27	18	62	81	30
Job 4	52	92	35	41	58	62	23	98	66	48
Job 5	40	22	53	54	74	71	92	52	82	60
Job 6	10	51	95	55	17	64	14	57	21	51
Job 7	24	57	41	80	45	6	65	32	9	49
Job 8	38	7	4	1	64	52	54	73	61	7
Job 9	9	80	98	9	62	76	12	41	37	6
Job 10	74	96	87	72	62	85	14	58	53	34
Job 11	96	39	36	23	91	84	32	79	33	46
Job 12	43	25	49	58	84	25	87	12	71	14
Job 13	89	78	55	73	79	92	37	99	46	8
Job 14	9	88	38	8	84	7	94	24	39	44
Job 15	92	85	54	73	72	18	37	34	73	52
Job 16	90	39	3	19	5	74	7	42	25	13
Job 17	25	46	32	91	34	48	97	15	34	98
Job 18	35	76	18	93	47	12	23	69	39	36
Job 19	70	47	52	22	32	98	28	15	78	53
Job 20	26	8	6	50	39	4	37	15	78	50

Tabela 1- Instância na forma tabelada

Logo após a instância ser aberta há a necessidade de definir a quantidade de iterações que o *Hill-Climbing* irá rodar, as quais podem ser determinadas colocando o número desejado na célula “QTD Itrç”. Em seguida, com a criação do botão “Resolver” o método começa a ser desenvolvido e o primeiro passo do *Hill-Climbing* é a geração do vetor solução aleatorizando suas posições.

Em primeiro momento, o *makespan* foi calculado a partir de uma solução aleatória, para que posteriormente se pudesse comparar com a solução otimizada. Para gerar essa solução aleatória, foi criada uma função chamada GeraSolucaoAleatoria, onde suas variáveis eram: o tamanho do Vetor (V), o índice escolhido (F), o número sorteado aleatoriamente (Num_Sort) e a posição na qual o vetor seria colocado (LIN).

A função *GeraSolucaoAleatoria* (Figura 2) age contando a posição dos Jobs de cada instância (sendo a quantidade de Jobs o tamanho do vetor solução), sorteando um valor aleatório e escolhendo uma posição para colocar o valor sorteado, gerando assim um vetor de mesmo tamanho com os Jobs posicionados aleatoriamente. A função era então repetida até que todas as posições do vetor que continham os Jobs fossem aleatorizadas.

```
Função GeraSolucaoAleatoria(TamanhoVetor)
Para i = 1 até TamanhoVetor faça
        NUM_SORT = Int(Rnd * 50)
        REP = 0
        Para j = 1 até TamanhoVetor faça
          Se NUM_SORT = VetorAleat(j) ou NUM_SORT > TamanhoVetor então
                REP = 1
          Fim
        Fim
        Se REP = 1 então
                Vá para REPETE
        Senão
                VetorAleat(i) = (NUM_SORT)
        Fim
Fim
Retorna (VetorAleat)
```

Figura 2- Pseudo Código da função que gera a Solução Aleatória

Nesta função existem algumas restrições que precisaram ser codificadas para que não ocorresse repetição de posições no mesmo vetor ou que algum valor que não pertencesse ao tamanho do vetor inicial fosse sorteado. Para isso, foi criada uma restrição que sorteasse novamente caso o valor sorteado já houvesse sido selecionado anteriormente ou caso esse valor não fosse válido para o tamanho do vetor solução inicial referente a determinada instância.

Com todos os valores válidos e computacionalmente aleatorizados, o vetor que continha a solução aleatória é assim gerado, podendo a partir dele ser calculado o *makespan*. Na sequência, o *Hill-Climbing* usa o vetor aleatorizado para gerar sua estrutura de vizinhança. A Figura 3 a seguir, mostra o pseudocódigo do método escolhido para a geração da vizinhança do problema, que é o método da inserção:

```
Função insercao(VetorSolucao, p1, p2, qtdJobs)
VetorNovaSol = VetorSolucao
Se p1 = p2 então (Sair da função)
Se p1 < p2 então
        o = p1
        VetorNovaSol (o) = VetorNovaSol (p2)
        Para k = 1 até qtdJobs faça
                Se k = o e k < p2 então
                        VetorNovaSol (o+1) = VetorSolucao(o)
                        o = o + 1
                Fim
        Fim
Fim
Se p1 > p2 então
        o = p2
        VetorNovaSol (o) = VetorNovaSol (p1)
        Para k = 1 até qtdJobs faça
                Se k = o e k < p1 então
                        VetorNovaSol (o+1) = VetorSolucao(o)
                        o = o + 1
                Fim
        Fim
Fim
Retorna (VetorNovaSol)
```

Figura 3- Pseudo Código que gera a estrutura de vizinhança pleo método da inserção

Juntamente com o resultado obtido através do *Hill-Climbing* houve a geração de um relatório em que pode se observar iteração a iteração como foi o progresso do método até encontrar o ótimo global da instância selecionada. Na Tabela 2 abaixo se observa como o relatório é gerado tendo como base 20 iterações rodadas:

Iteração	Sol. Inicial	Ótimo Local	Melhor Global
1	1861	1649	1649
2	1875	1636	1636
3	1867	1618	1618
4	1898	1618	1618
5	1870	1624	1618
6	2006	1604	1604
7	1895	1580	1580
8	1913	1570	1570
9	1753	1631	1570
10	1993	1626	1570
11	1858	1628	1570
12	1964	1608	1570
13	1849	1608	1570
14	2037	1666	1570
15	1908	1670	1570
16	1999	1618	1570
17	1974	1613	1570

18	1946	1593	1570
19	1807	1635	1570
20	2013	1703	1570

Tabela 2-Exemplo de relatório gerado depois de 20 iterações

Para problemas do tipo FSP a representação da solução é dada por um vetor de tamanho *n* formado pela permutação da sequência dos Jobs em cada máquina (sequência em que as tarefas serão executadas). A avaliação da solução é feita pela análise do resultado do *makespan* final observando se houve redução significativa no tempo quando comparado com os resultados obtidos na solução inicial e com outros dados já conhecidos de melhor solução ótima.

4. Resultados

Primeiramente determinamos a quantidade de iterações necessárias para que o algoritmo rodasse aproximadamente 1 minuto em cada instância, variando assim, o número de iterações em cada instância selecionada.

A partir da Tabela 1 já apresentada, o vetor foi aleatorizado com a sequência dos Jobs e como retorno obteve-se o vetor final, o seu *makespan* já calculado e informações sobre o tempo que o algoritmo demorou para rodar o número de iterações. Tendo como exemplo a 40ª intância, a solução aleatória inicial do *makespan* era igual a 1861. Após a aplicação do algoritmo, e rodando 100 iterações em 57 segundos, o resultado otimizado do *makespan* baixou para 1534.

Por fim, a Tabela 3 a seguir mostra a comparação entre o *makespan* da solução aleatória e o ótimo global (*makespan*) de todas as instâncias após a aplicação do *Hill-Climbing* para a otimização das soluções iniciais bem como a quantidade de iterações realizadas e o tempo para que estas estivessem completas, além da quantidade de máquinas e Jobs contidos em cada uma:

Instâncias	Máquinas x Jobs	Ótimo Global	Qtd Iterações	Tempo(s)
1	5 x 10	695	2000	70
2	5 x 10	698	2000	60
3	5 x 10	728	2000	56
4	5 x 10	697	2200	70
5	5 x 10	713	2200	97
6	5 x 10	748	2000	78
7	5 x 10	728	2000	61
8	5 x 10	683	2000	45
9	5 x 10	761	2000	74
10	5 x 10	664	2000	73
11	10 x 10	1097	2000	117
12	10 x 10	1146	2000	108
13	10 x 10	1124	1800	119
14	10 x 10	1038	1800	129

15	10 x 10	1093	1000	58
16	10 x 10	1085	1000	59
17	10 x 10	115	1000	53
18	10 x 10	1113	1000	60
19	10 x 10	1045	1000	65
20	10 x 10	1099	1000	63
21	5 x 20	1195	1000	405
22	5 x 20	1280	500	167
23	5 x 20	1325	350	121
24	5 x 20	1139	250	131
25	5 x 20	1339	250	95
26	5 x 20	1081	200	96
27	5 x 20	1158	150	63
28	5 x 20	1110	150	61
29	5 x 20	1325	150	57
30	5 x 20	1252	150	61
31	10 x 20	1552	150	130
32	10 x 20	1574	100	94
33	10 x 20	1636	80	67
34	10 x 20	1492	80	76
35	10 x 20	1628	50	38
36	10 x 20	1085	1000	61
37	10 x 20	1632	60	60
38	10 x 20	1580	80	64
39	10 x 20	1567	80	79
40	10 x 20	1534	80	71

Tabela 3- Comparação entre o Ótimo Global e a Solução Aleatória

Como se observa na Tabela 3 na coluna de ótimos globais (melhores resultados obtidos ao final de todas as iterações), após a aplicação do *Hill-Climbing*, todas as instâncias obtiveram melhora significativa sobre a solução aleatória gerada inicialmente variando apenas o número de iterações realizadas em cada instância devido as diferentes quantidades de Jobs e máquinas que essas possuíam. A seguir, a Tabela 4 traz o resultado da melhor solução encontrada pelo algoritmo comparado ao melhor resultado conhecido já calculando o GAP (diferença) entre os dois:

INSTÂNCIA	Melhor Conhecida	Valor Objetivo	Gap (%)	Iterações	Tempo (s)
VFR10_10_1_Gap.txt	1097	1097	0,0%	1800	64,48
VFR10_10_10_Gap.txt	1099	1099	0,0%	1800	69,4
VFR10_10_2_Gap.txt	1146	1146	0,0%	1800	60,33
VFR10_10_3_Gap.txt	1124	1124	0,0%	1800	73,49
VFR10_10_4_Gap.txt	1038	1038	0,0%	1800	80,52
VFR10_10_5_Gap.txt	1093	1093	0,0%	1800	67,21
VFR10_10_6_Gap.txt	1085	1085	0,0%	1800	69,5
VFR10_10_7_Gap.txt	1115	1115	0,0%	1800	60,03
VFR10_10_8_Gap.txt	1113	1113	0,0%	1800	67,46
VFR10_10_9_Gap.txt	1045	1045	0,0%	1800	78,97
VFR10_5_1_Gap.txt	695	695	0,0%	2000	49,27
VFR10_5_10_Gap.txt	664	664	0,0%	1800	45,94
VFR10_5_2_Gap.txt	698	698	0,0%	2000	41,25
VFR10_5_3_Gap.txt	728	728	0,0%	2000	38,9
VFR10_5_4_Gap.txt	697	697	0,0%	2000	56,71
VFR10_5_5_Gap.txt	713	713	0,0%	2000	91,57
VFR10_5_6_Gap.txt	748	748	0,0%	1500	38,59
VFR10_5_7_Gap.txt	728	728	0,0%	1500	34,53
VFR10_5_8_Gap.txt	683	683	0,0%	1800	40,77
VFR10_5_9_Gap.txt	761	761	0,0%	1800	52,96
VFR20_10_1_Gap.txt	1532	1550	1,2%	100	48,21
VFR20_10_10_Gap.txt	1489	1534	3,0%	100	56,69
VFR20_10_2_Gap.txt	1525	1568	2,8%	100	51,75
VFR20_10_3_Gap.txt	1592	1632	2,5%	100	60,64
VFR20_10_4_Gap.txt	1442	1474	2,2%	100	58,76
VFR20_10_5_Gap.txt	1604	1627	1,4%	100	51,57
VFR20_10_6_Gap.txt	1576	1617	2,6%	100	57,78
VFR20_10_7_Gap.txt	1591	1611	1,3%	100	56,34
VFR20_10_8_Gap.txt	1574	1580	0,4%	100	53,46
VFR20_10_9_Gap.txt	1530	1564	2,2%	100	58,2
VFR20_5_1_Gap.txt	1192	1196	0,3%	500	117,61
VFR20_5_10_Gap.txt	1243	1243	0,0%	250	62,2
VFR20_5_2_Gap.txt	1275	1280	0,4%	500	103,24
VFR20_5_3_Gap.txt	1323	1325	0,2%	350	75,15
VFR20_5_4_Gap.txt	1127	1140	1,2%	350	116,09
VFR20_5_5_Gap.txt	1339	1339	0,0%	250	61,08
VFR20_5_6_Gap.txt	1066	1075	0,8%	250	76,12
VFR20_5_7_Gap.txt	1154	1155	0,1%	250	68,66
VFR20_5_8_Gap.txt	1102	1102	0,0%	250	63,95
VFR20_5_9_Gap.txt	1317	1317	0,0%	250	61,25

Tabela 4- Comparativo entre os resultados conhecidos e os obtidos

5. Conclusões

Atualmente, em meio a grandes produções o grande foco tem sido em economias, reduzir custos direta ou indiretamente tem sido a meta de engenheiros e administradores de todo o mundo. Ao reduzir o *makespan*, aumenta-se a produtividade e economiza-se tempo. Utilizar de ferramentas de programação resulta em soluções nas quais dificilmente chegaríamos utilizando outros meios, ou ainda, gastaríamos tanto tempo que seria inviável.

Dentre as dificuldades encontradas na execução deste artigo, podemos citar que o domínio da linguagem de programação e dos fatores que envolveram o problema do Flow Shop, além de seu funcionamento, foram e são fundamentais para que se chegue a soluções melhores e aja a otimização do problema. Em muitos momentos foi necessário recorrer a artigos e pesquisas para que pudéssemos criar uma base capaz de gerar um pensamento lógico para pensar em métodos de solução.

Os resultados obtidos foram muito satisfatórios, uma vez que todas as instâncias obtiveram melhora significativa no *makespan* quando comparados aos melhores valores conhecidos. Os GAPs obtidos tiveram valores muito baixos, muitas das instâncias inclusive alcançaram o mesmo ou um valor menor do que o conhecido o que leva a um GAP de 0%. A instância com maior diferença entre o ótimo global encontrado e o utilizado como referência foi a 40ª instância, que obteve 1534 como melhor solução, enquanto o valor referência era de 1489, gerando assim um GAP de 3,0%. Ainda assim a maioria dos GAPs se manteve abaixo de 1%, o que mostra a qualidade dos resultados encontrados com a execução do algoritmo e como estes ficaram próximos dos resultados esperados.

Apesar de todas as dificuldades, foi possível gerar uma otimização capaz de resolver instâncias de quaisquer tamanho, mesmo sabendo que existem muitas variáveis que não foram consideradas, sabe-se que existe um caminho que precisa ser estudado mais a fundo para que se atinja soluções ainda melhores e cada vez mais próximas da solução ótima. Existem métodos talvez mais complexos, porém mais eficientes de se resolver o problema, que resultem em resultados melhores, ou inda, métodos que sejam resolvidos computacionalmente mais rapidamente do que o escolhido.

A importância desse projeto está não apenas na resolução do problema, mas sim na apresentação de novas ferramentas que ao serem estudadas permitiram que engenheiros e estudiosos fossem capazes de resolver problemas que manualmente seriam imensuráveis e podendo sempre adaptar a cada situação. Esse novo conhecimento demonstra uma grande oportunidade de fazer a diferença dentro do mercado de trabalho.

Referências

BAKER, K.R. **Introduction to Sequencing and Scheduling**, Wiley, New York, 1974.

DEROUSSI, L., GOURGAND, M., NORRE, S. **New effective neighborhoods for the permutation flow shop problem**. LIMOS, p. 1-18, 12 mar. 2012.

FRENCH, S. **Sequencing and Scheduling: An introduction to the Managemente Science**, 1997.

HILLIER, Frederick S.; LIEBERMAN, Gerald J. **Introdução a Pesquisa Operacional**. [*S. l.*]: Saraiva, 2012.

HORDONES, P., CAMARGO, V., FUCHIGAMI, H. Programação da produção em flow shop permutacional envolvendo medidas de atraso: uma constribuição bibliométrica. **Simpósio Brasileiro de Pesquisa Operacional**, Vitória, ES, p. 1-14, 30 set. 2016.

KIESKOSKI, A. Um estudo do problema de flow shop permutacional. Uma proposta de solução através da metaheurística colônia de formigas. **Acervo UFPR**, Universidade Federal do Paraná, p. 1-111, 4 out. 2016.

KOMESU, A. Heurística Evolutiva para a minimização do atraso total em ambiente de produção flow shop com buffer zero. **Pesquisa Operacional**, Universidade de São Paulo, p. 1-84, 10 abr. 2015.

MACCARTHY B., LIU J. Addressing The Gap In Scheduling Research: A Review Of Optimization And Heuristic Methods In Production Scheduling. **International Journal of Production Research**. v. 31, n. 1, p. 59-79. 1993.

SILVA, Nathália Cristina Ortiz. **Aplicação de simulação para análise do makespan devido á inserção e/ou desistência de tarefas no problema de sequeciamento de produção em uma máquina**. 2016. Dissertação (Mestrado) - Graduada, Uni, 2016.

WIDMER, M., HERTZ, A. A New Method For The Flow Sequencing Problem. **European Journal of Operational Research**. v. 41, p. 186-193. 1989.

XAVIOR, A. **Optimization of Makespan in Job and Machine Priority Environment.** Disponívelem: <https://www.sciencedirect.com/science/article/pii/S1877705814032792> Acesso em: 05 jun, 2019.