Universidade Federal de Minas Gerais

Computação Evolucionária

Job Shop Scheduling Problem

Nome: Daniela Caroline Lucas dos Santos - 2011021337

17 de Novembro de 2015

Introdução

O problema *Job Shop Scheduling* (JSSP) é um problema de otimização combinatorial e sua formulação consiste em: é passado um conjunto de n tarefas (jobs) ,J_1, J_2,J_3, ... J_n, as quais possuem diferentes tempos de duração, e um conjunto de m máquinas, M_1,M_2...,M_m, que são responsáveis por executar as tarefas. Cada tarefa é composta por um conjunto de operação cuja ordem de execução é pré-definida e para cada operação sabe-se em qual máquina ela irá executar e o tempo total de execução.

O JSSP tem como objetivo encontrar o menor tempo possível de realização de todas as tarefas (*makespan*) atendendo todas as restrições possíveis. No caso do trabalho proposto as restrições foram as seguintes:

- uma mesma máquina não pode realizar duas tarefas simultaneamente;
- uma mesma tarefa não pode ser realizada por duas máquinas simultaneamente, ou seja, a tarefa só deve ser iniciada na máquina Mm+1 após sua conclusão na máquina Mm;
- as tarefas devem ser processadas sequencialmente nas máquinas.

Devido a estas restrições é possível concluir que este é um caso especifico de *Flow Shop* Permutacional (Figura 1). Nele, um determinado conjunto de tarefas *n* devem ser processadas em um determinado conjunto de máquinas *m* ordenadas, com a ordem de processamento igual para todas as máquinas, a fim de obter um produto final. Partindo desse principio o programa foi implementado com o objetivo de obter uma sequência das tarefas que otimiza uma determinada medida de desempenho. Nos modelos para solução do problema, as medidas usuais referem-se à minimização da duração total da programação (*makespan*).

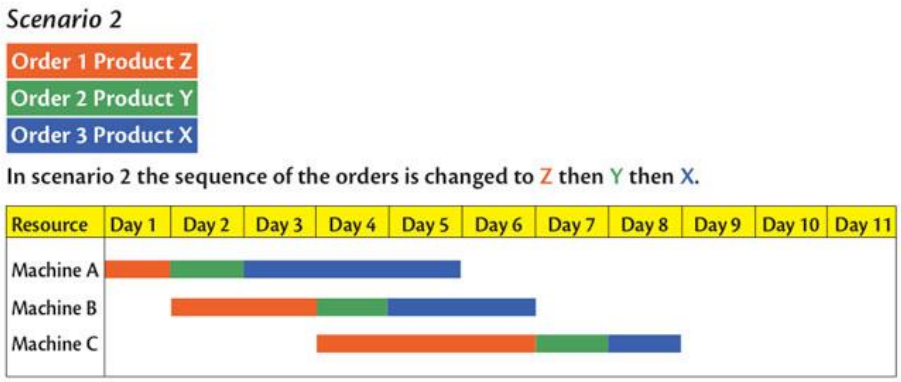

Figure 1-Job Shop

Decisões de implementação

1. Forma de representação dos indivíduos

Os indivíduos representam a ordem em que as tarefas serão executas nas máquinas. Logo usando a funções *randperm()* os indivíduos foram criados, onde cada indivíduo é um vetor que ordenado de forma aleatório de 1 ao número total de tarefas.

pop =[2 3 1

3 2 1

1 3 2

1 3 2

3 1 2]

Na matriz acima cada linha representa um indivíduo o qual compõem uma população de um total de 5 indivíduos.

2. Algoritmo evolucionário

O algoritmo foi baseado em um algoritmo genético simples. Inicialmente uma população é gerada e o *makespan* de cada indivíduo é calculado. Através do vetor *fit* retornado é possível calcular a média do valor de *makespan* e o melhor valor. Em seguida é selecionado os pais, procurado sempre encontrar os melhores indivíduos do meio da população.

Após a seleção dos pais é feito um cruzamento entre eles o qual gerará 2 filho, depois esses filhos podem sofrer mutação ou não. A população no decorrer do programa tem sempre o mesmo tamanho inicial, então é realizada a seleção dos sobreviventes, onde os indivíduos com os maiores *makespan* são. Assim, uma nova população será gerada e poderá ser avaliada da mesma forma no próximo loop.

3. Tipo de operador de seleção utilizado

Seleção dos Pais

Para selecionar os pais, 10 indivíduos são escolhidos de forma aleatória do meio da população, o valor de fitness desses indivíduos é comparado com o fitness do vetor chamado *paisinit*, o qual inicializa o valor dos pais selecionados sempre como um vetor ordenado de 1 até o valor total de tarefas. Assim, é sempre escolhido os 2 melhores entre os 10 indivíduos para realizar o cruzamento.

Seleção dos Sobreviventes

Para selecionar os sobreviventes, os filhos gerados são incluídos dentro da matriz populacional e em seguida todos os indivíduos são ordenados (raqueados) de acordo com o seu valor de *makespan*, mas a população tem sempre de manter o tamanho inicial, então os dois indivíduos com os piores fitness, ou seja, com o maior *makespan* são retirados. Assim, uma nova população será gerada e poderá ser avaliada da mesma forma no próximo loop.

4. Tipo de operador de cruzamento utilizado

Na implementação do cruzamento foi utilizado o crossover com 1 ponto de corte. Sendo o mesmo algoritmo fornecido pelo professor na tarefa 1 (N-Rainhas) "CutAndCrossfill_CrossOver.m". Inicialmente, um número é sorteado entre 1 e o número total de tarefas, a partir do número sorteado é definido o ponto de corte nos cromossomos pais e então é gerado os filhos a partir das caudas dos pais. Caso não haja cruzamento, os filhos gerados são iguais aos pais.

5. Tipo de operador de mutação utilizado

Como o indivíduo é representado por um vetor de número inteiro e esses números não podem se repetir no vetor, a mutação é realizada apenas trocando duas posições de forma

aleatória de dentro do vetor. Para isso duas posições do cromossomo são selecionadas aleatoriamente em seguida é feito a troca dessas posições (permutação).

6. Critério de parada utilizado

O único critério de parada utilizado é quando o programa atinge o número máximo de gerações. Optou-se por não usar o valor de fitness como critério de parada, pois muitas vezes o algoritmo parece estar estabilizado por diversas gerações, mas devido a alta probabilidade de mutação volta a convergir para um menor valor de *makespan*.

7. Valores de alguns parâmetros do algoritmo

Os parâmetros escolhidos foram obtidos de forma experimental, visando encontrar sempre o melhor resultado:

- Tamanho da população $\rightarrow$ 20 indivíduos;
- Probabilidade de mutação $\rightarrow$ 0.9;
- Probabilidade de cruzamento $\rightarrow$ 0.9;
- Número de gerações $\rightarrow$ 200;
- r = 10 $\rightarrow$ escolha aleatoriamente dez indivíduos na população para seleção dos pais.

8. Fitness

A função fitness é responsável por calcular o *makespan* de cada indivíduo da população e retornar um vetor contendo todos esses valores.

Para isso, ela recebe a matriz A, a qual contém o arquivo de entrada e a ordena de acordo com o indivíduo avaliado, o resultado é uma matriz como a representada na figura 1. Em seguida, são construídos vetores para cada máquina, os locais em branco foram representados com o número 0 no programa, como mostrado na figura 2. O valor do *makespan* retornado para este indivíduo é igual ao tamanho do vetor da máquina 3.

Tarefas	Máquinas			
		1	2	3
	1	3	2	1
	2	3	3	2
	3	4	3	1

Figure 2 - Tarefas ordenadas

- *Makespan*
- Seqüência de tarefas: 2 - 1 - 3

makespan

	Tempo													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1		2	2	2	1	1	1	3	3	3	3				
2					2	2	2	1	1			3	3	3	
3								2	2	1					3

Figure 3- Vetores contendo as tarefas executas

Resultados

Os seguintes resultados foram obtidos:

- **Entrada: entrada_3.txt**

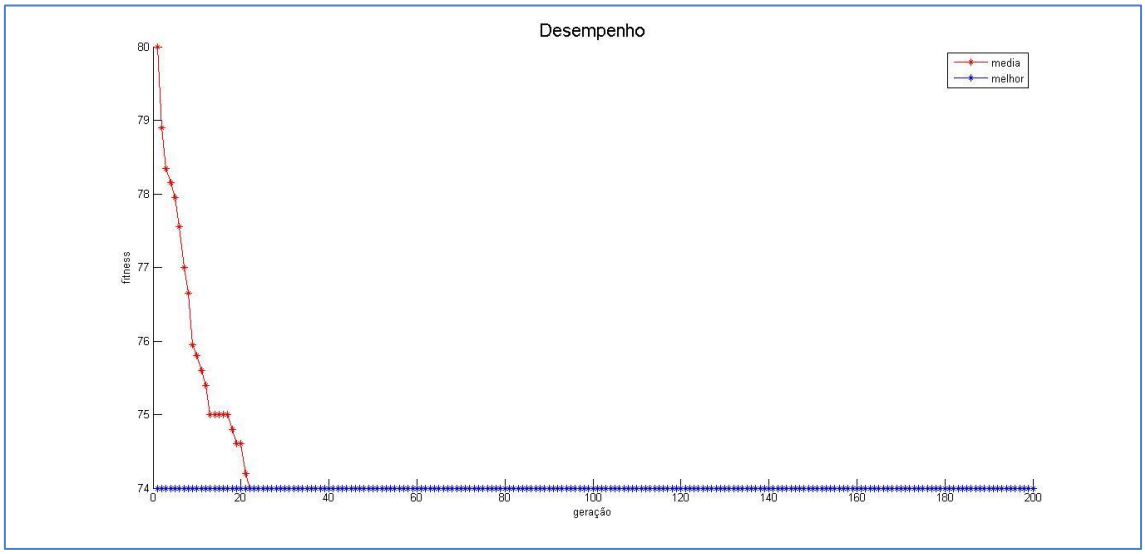

Figure 4 - Gráfico de Desempenho – entrada_3 (1ª Simulação)

Makespan = 74;

Sequência = [1 2 3];

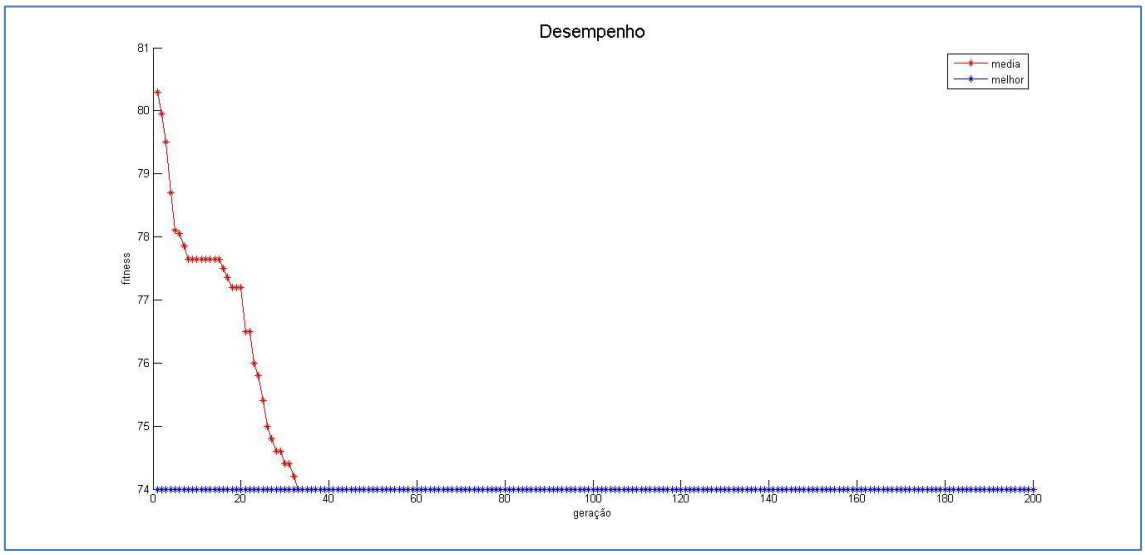

Figure 5- Gráfico de Desempenho – entrada_3 (2ª Simulação)

Makespan = 74;

Sequência = [1 2 3];

- **Entrada: entrada_10.txt**

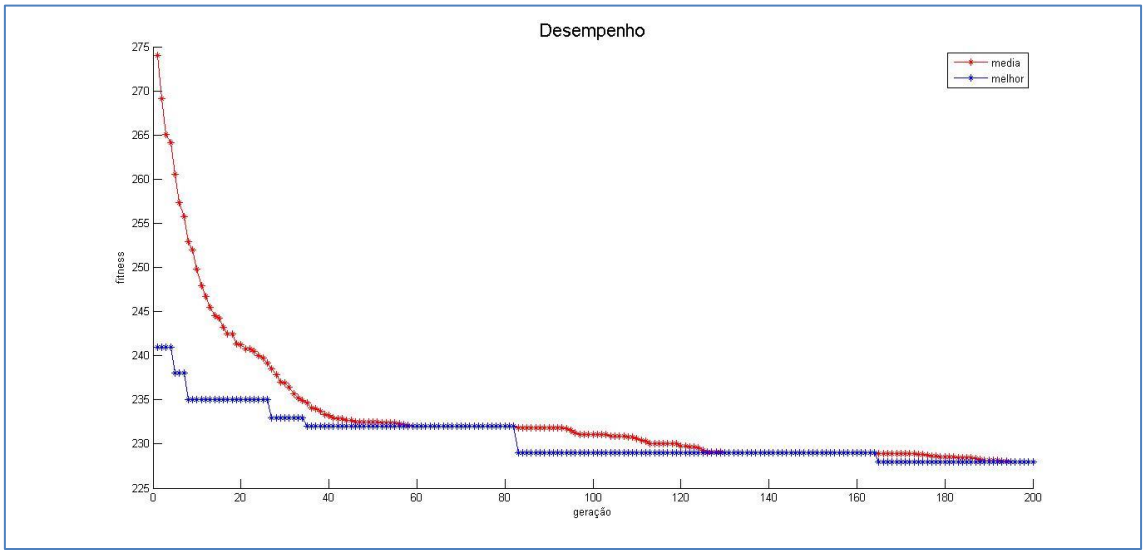

Figure 6- Gráfico de Desempenho - entrada_10 (1ª Simulação)

Makespan =228;

Sequência =[10 6 4 1 5 8 2 3 7 9];

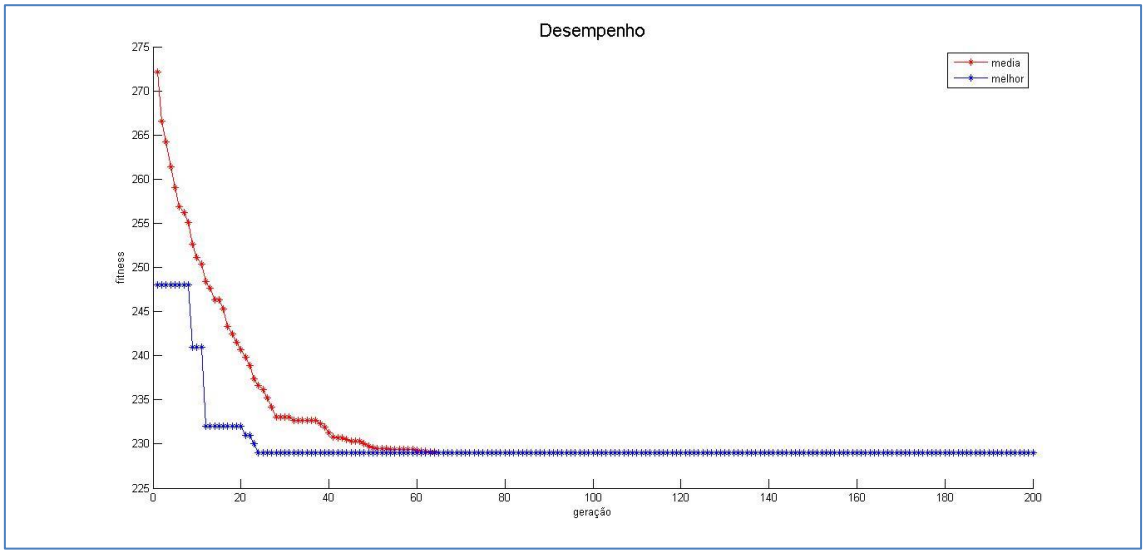

Figure 7 - Gráfico de Desempenho - entrada_10 (2ª Simulação)

Makespan =229;

Sequência =[10 1 3 6 4 7 5 8 2 9]

- **Entrada: entrada_25.txt**

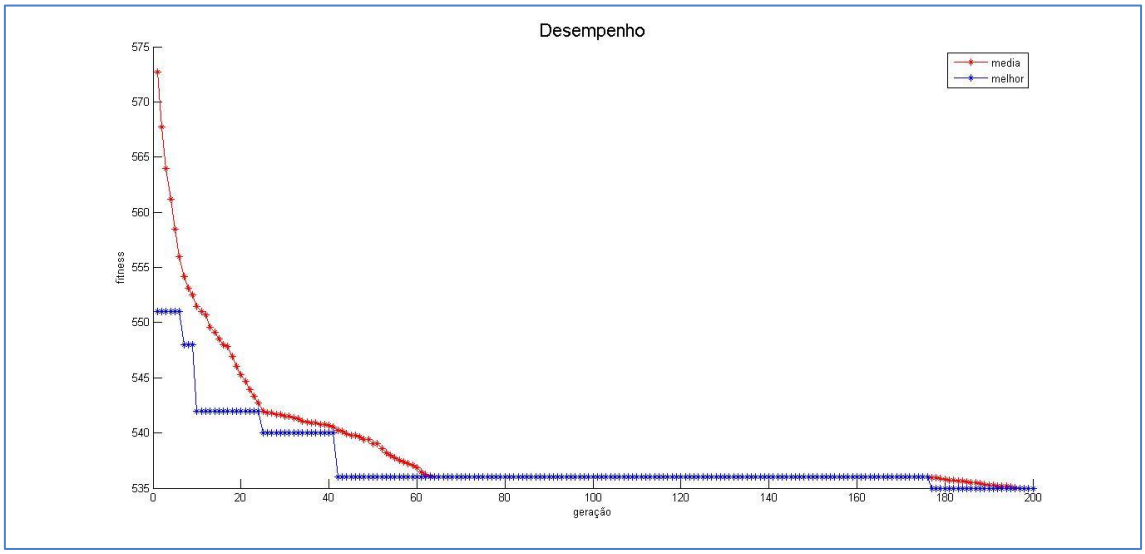

Figure 8- Gráfico de Desempenho - entrada_25 (1ª Simulação)

Makespan = 535;

Sequência = [4 7 3 11 17 9 8 23 12 25 13 22 2 16 21 14 15 10 6 19 5 1 18 24 20];

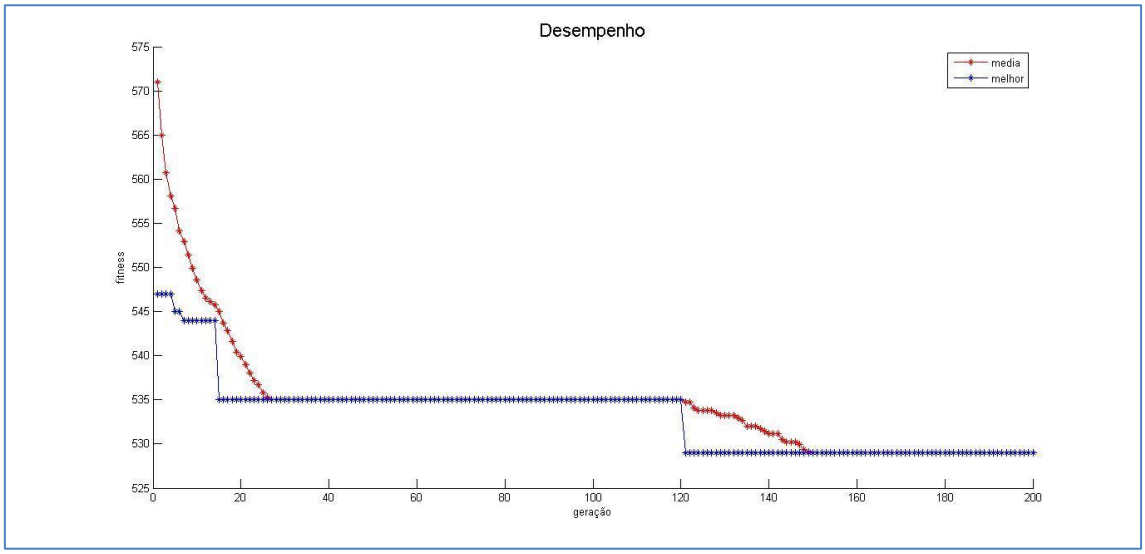

Figure 9 - Gráfico de Desempenho - entrada_25 (2ª Simulação)

Makespan = 529;

Sequência = [7 13 18 16 19 11 3 23 4 15 6 2 25 1 21 8 12 10 5 17 24 14 9 22 20]

Conclusão

O algoritmo genético implementado realiza exatamente o que foi pedido, retornando sempre:

- *Makespan*: o fitness da melhor solução encontrada.
- *Sequence*: sequência de pedidos a ser inserida na linha que acarreta no *makespan*.
- *avg_fit*: vetor contendo em cada posição o *makespan* médio a cada geração.
- *best_fit*: Vetor contendo em cada posição o melhor *makespan* a cada geração.

Os vetores *avg_fit* e *best_fit* possuem o mesmo tamanho.

O algoritmo genético usado para a otimização do problema de *Job Shop Scheduling* funciona melhor para entradas menores, como pode ser visualizado nas figuras 4 e 5 o gráfico converge rapidamente e todas as vezes que foi simulado o valor de melhor *makespan* foi sempre o mesmo.

Quanto maior a entrada, mas demorado é o tempo de execução e menor a chance da função convergir para o ponto ótimo global (mínimo global), muitas vezes o algoritmo fica preso em um mínimo local e não converge para o ponto ótimo, como pode ser visto nas figuras 7 e 8.

No programa optou-se por uma alta probabilidade de mutação e muitas vezes isso auxilia o algoritmo sair de um mínimo local. Como o critério de parada é apenas o número de gerações é possível notar pelos gráficos que algumas vezes o algoritmo converge apenas bem ao final (figura 8), foram feitas simulações em que o algoritmo convergiu nas últimas 10 gerações, por isso pode- se acreditar que caso aumentasse o número de gerações a chance de se sair de mínimos locais aumentariam, pois poderiam ocorrer mutações que levariam ao mínimo global.

Em relação a implementação do trabalho tudo transcorreu normalmente, a parte mais problemática para ser implementada foi a função fitness, mas os problemas foram devidamente corrigidos.

Bibliografia

[1] Disponível online em: http://homepages.dcc.ufmg.br/~glpappa/compnat/TP1.pdf - [Novembro 2015].

[2] Slides de Computação evolucionária- Professor Cristiano

[3] Disponível Online em: https://en.wikipedia.org/wiki/Job_shop_scheduling -[Novembro 2015].

[4] Disponível Online em: http://www.lac.inpe.br/~lorena/nagano/CS-flowshop.pdf - [Novembro 2015]

[5] Disponível Online em: http://slideplayer.com.br/slide/356559/- [Novembro 2015]

Anexos

- JSSP.m
- fitness.m
- SelecionaPais.m
- Cruzamento.m
- Mutacao.m
- SelecionaSobrevimentos.m