UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LEANDRO DE OLIVEIRA PEREIRA
PEDRO TRINDADE
VALÉRIA SOLDERA GIRELLI

Permutational Flowshop Scheduling Problem

Orientador: Prof. Luciana Salate Buriol

Porto Alegre
2019

1 INTRODUÇÃO

Este trabalho apresenta a formulação matemática em programação linear inteira do problema *Permutational Flowshop Scheduling Problem* (PFSP). O PFSP é um problema NP-Completo, e sua formulação matemática foi realizada utilizando o *GNU Linear Programming Kit* (GLPK). O arquivo de formulação é separado em um arquivo de modelo (arquivo *.mod*) e em arquivos de dados (arquivo *.dat*). As intâncias utilizadas e modeladas foram as disponibilizadas no repositório referente à especificação do trabalho[1]. Essas instâncias foram convertidas por um *script* em *Python* para gerar os arquivos de dados de acordo com os parâmetros definidos.

Ademais, também foi implementada uma meta-heurística para solução do problema. A meta-heurística escolhida foi o algoritmo *Simulated Annealing* (SA)(BERTSIMAS; TSITSIKLIS, 1993). Todos os códigos desenvolvidos para esse trabalho estão disponíveis no repositório do trabalho no GitHub[2].

2 PROBLEMA

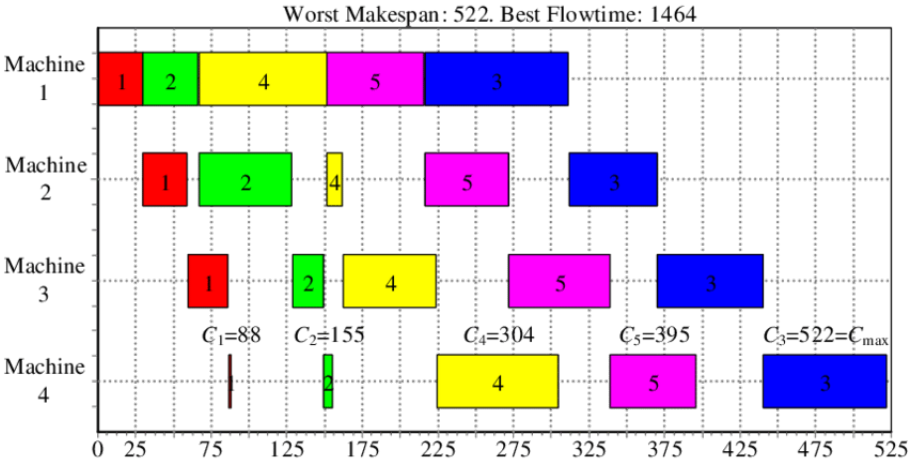
O problema *Permutational Flowshop Scheduling Problem* é composto por um conjunto de tarefas $N = \{1, ..., n\}$ e um conjunto de máquinas $M = \{1, ..., m\}$. Todas as tarefas devem ser processadas em todas as máquinas, e todas as máquinas devem processar as tarefas em uma mesma sequência. As tarefas requerem um tempo específico para serem processadas por cada máquina, que é definida pelo parâmetro $T_{ir} \geq 0$, para cada tarefa $i \in N$ e máquina $r \in M$. As tarefas devem ser processadas sequencialmente no conjunto das máquinas, ou seja, uma tarefa $i \in N$ deve ser processada na máquina 1, depois na máquina 2, e assim por diante até última máquina.

O objetivo do problema é encontrar uma ordenação de tarefas que minimize o *makespan*, que é o tempo de término da última tarefa escolhida na última máquina. Esse problema inicialmente começou com Manne (1960) que propôs para um modelo geral de *job shop problem* e depois foi adaptado para permutação com o método *big M* por Tseng (2002).

A Figura 2.1 mostra um exemplo de instância do PFSP com *makespan* igual a 522. Além disso, demonstra a ordenação das tarefas em cada máquina e as restrições sendo respeitadas.

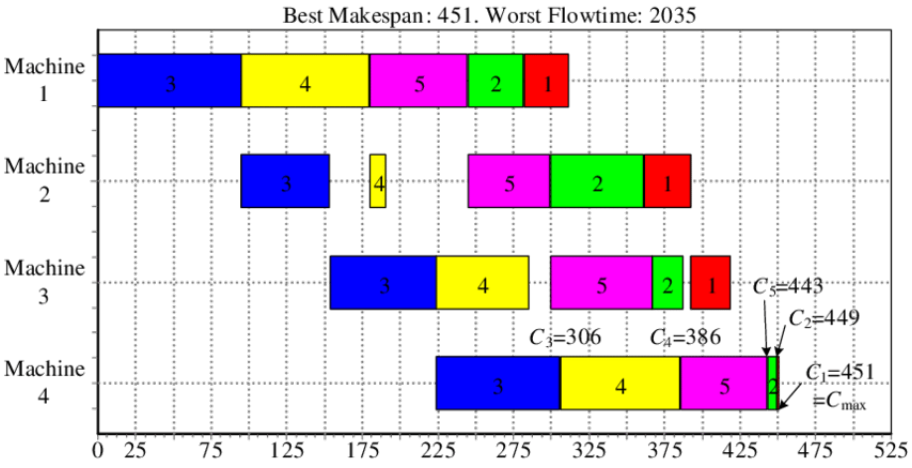
[1] <https://github.com/afkummer/ufrgs-inf05010-2019-1>
[2] <https://github.com/vsgirelli/pfsp_SA>

Figura 2.1: Exemplo de instância com *makespan* 522.



A Figura 2.2 mostra que com uma diferente ordem de execução das tarefas obtém-se uma melhor solução para o problema, com a mesma instância do PFSP, que resulta em C_{max} igual a 449.

Figura 2.2: Exemplo de instância com *makespan* melhor.



2.1 VARIÁVEIS

As variáveis do problema são descritas a seguir:

1. $C_{ri} \geq 0$: Variáveis reais que indicam o tempo em que a tarefa $i \in N$ termina de ser processada na máquina $r \in M$;

2. $C_{max} \geq 0$: Variável real que indicam o tempo final de processamento da última máquina, m;

3. $D_{ik} \in \{0,1\}$: Variáveis binárias que indicam se a tarefa $i \in N$ precede a tarefa $k \in N$, e só considera $i < k$.

4. P: número extremamente grande, chamada de técnica *big M*.

$$min\ C_{max} \tag{2.1}$$

Sujeito a:

$$C_{1i} \geq T_{1i} \quad 1 \leq i \leq n \tag{2.2}$$

$$C_{ri} - C_{r-1,i} \geq T_{ri} \quad 1 \leq i \leq n, 2 \leq r \leq m \tag{2.3}$$

$$C_{ri} - C_{r,k} + P * D_{ik} \geq T_{ri} \quad 1 \leq i < k \leq n, 1 \leq r \leq m \tag{2.4}$$

$$C_{ri} - C_{r,k} + P * D_{ik} \leq P - T_{ri} \quad 1 \leq i < k \leq n, 1 \leq r \leq m \tag{2.5}$$

$$C_{max} \geq C_{mi} \quad 1 \leq i \leq n \tag{2.6}$$

$$C_{ri} \geq 0 \quad 1 \leq r \leq m, 0 \leq i \leq n \tag{2.7}$$

$$D_{ik} \in \{0,1\} \quad 1 \leq i < k \leq n \tag{2.8}$$

Cada restrição tem um importante impacto na solução final, como pode ser visto a seguir:

1. Restrição 2.2: Assegura que o tempo de completude de uma tarefa na máquina 1 será pelo menos igual ao seu tempo para execução.

2. Restrição 2.3: Assegura que uma tarefa não vá rodar na próxima máquina antes de estar completa na máquina anterior

3. Restrições 2.4 e 2.5: Ambas asseguram que uma mesma tarefa vai ser executada antes ou após a outra, ou seja, a ordem das tarefas nas máquinas.

4. Restrição 2.6: Assegura que o tempo de completude da tarefa vai ser igual ao *makespan*, isto é, tempo final de processamento da última tarefa na última máquina.

5. Restrições 2.7 e 2.8: Ambas asseguram o domínio das variáveis de decisão do problema.

3 META-HEURÍSTICA

Frequentemente, encontrar a solução ótima para um problema de programação inteira pode levar a tempos de execução exponenciais. Desse modo, heurísticas podem ser

desenvolvidas de modo que se obtenha soluções boas o suficiente em tempos de execução inferiores. No entanto, ao utilizarmos heurísticas e meta-heurísticas para solucionar problemas, uma série de pontos precisam ser considerados. Por exemplo, não temos garantia de convergência, ou seja, não temos garantia de que obteremos uma solução aceitável. Além disso, dependendo do problema e da instância, não há nem como garantir que há uma solução ótima.

Para buscar solucionar o PFSP, a meta-heurística escolhida foi o *Simulated Annealing* (SA)(BERTSIMAS; TSITSIKLIS, 1993). O SA pode ser considerado uma evolução do método Hill Climbing [1], que analisa os vizinhos mais próximos em busca de uma solução melhor. A diferença principal entre os dois métodos é o fato de o SA permitir que alguns vizinhos piores sejam escolhidos ao longo das iterações, com o objetivo de fugir de ótimos locais.

3.1 Temperatura e Resfriamento

Para alcançar o objetivo de fugir de ótimos locais, o SA possui um parâmetro de temperatura. A temperatura controla a frequência na qual uma solução pior pode ser aceita e é decrementada ao longo do tempo com base em uma função de resfriamento. Temperaturas altas favorecem a escolha de vizinhos piores, enquanto que com o decremento da temperatura, soluções piores serão cada vez menos aceitas. Definir adequadamente esses dois aspectos é essencial no desempenho do algoritmo.

Os valores de temperatura testados foram: 100, 200, 300 e 400, sendo que o valor que gerou os melhores resultados foi o valor 100. Desenvolvemos as seguintes funções de resfriamento[2]:

$$T_i = T_0 - iter * \frac{T_0 - T_N}{max\_iter} \tag{3.1}$$

$$T_i = T_0 * \frac{T_N}{T_0}^{\frac{iter}{max\_iter}} \tag{3.2}$$

$$A = \frac{(t_0 - T_N) * (max\_iter + 1))}{(max\_iter)} + \frac{max\_iter}{2} \quad T_i = \frac{A}{iter + 1} \tag{3.3}$$

Os melhores resultados foram obtidos com a equação 3.3. Acreditamos que isso seja devido ao comportamento da função 3.3 de decrementar rapidamente a temperatura

[1] <https://www.sciencedirect.com/topics/computer-science/hill-climbing>

[2] Funções de resfriamento obtidas em <http://www.btluke.com/simanf1.html>

nas primeiras iterações. Com isso, o algoritmo passa menos tempo aceitando resultados piores, e se mantém com uma taxa baixa e quase constante de aceitação no restante das iterações.

3.2 Geração de Soluções

A geração da primeira solução é feita de forma randômica de acordo com a *seed* informada pelo usuário ou com as *seeds* variando de 1 a 10 (valores padrão do algoritmo desenvolvido). Já para a geração das demais soluções vizinhas, a primeira ideia desenvolvida foi com base no tempo médio de execução das tarefas. A cada iteração do algoritmo, buscava-se ordenar de forma crescente uma nova tarefa. À vista disso, o número de iterações estava limitado pelo número de tarefas, o que muito provavelmente contribuiu para o baixo desempenho da solução. Ao final da execução, tinha-se as tarefas ordenadas de forma crescente pelo seu tempo médio de execução nas máquinas. Como eram realizadas poucas iterações e, portanto, gerava-se pouca variabilidade, as soluções encontradas representavam apenas uma pequena parte do total de soluções possíveis.

O segundo método de geração de vizinhos foi desenvolvido com base na troca aleatória de quaisquer duas tarefas. A partir desse método, geramos outros 3 métodos dinâmicos que variavam de acordo com o número de tarefas, com o número de iterações e totalmente randômico.

O método de geração aleatória com base no número de tarefas (*newNeighboorRandTasks*) aumenta o número de trocas entre duas tarefas quaisquer a cada aumento de 100 tarefas no tamanho da entrada do problema. Desse modo, até 100 tarefas, tem-se uma troca entre duas tarefas quaisquer. Se o problema possuir entre 100 e 200 tarefas, tem-se duas trocas entre duas tarefas quaisquer, e assim por diante.

A geração aleatória baseada no número de iterações (*newNeighboorRandIter*) também faz trocas de duas tarefas quaisquer. No entanto, o número de trocas aumenta a cada 1000 iterações do algoritmo. Durante as 1000 primeiras iterações, realiza-se apenas uma troca entre duas tarefas quaisquer. Entre 2000 e 3000 iterações, realiza-se duas trocas entre duas tarefas quaisquer, e assim por diante. Já o método totalmente randômico (*newNeighboorRandRand*) realiza um número randômico de trocas entre duas tarefas quaisquer a cada iteração do algoritmo.

Os melhores resultados foram obtidos com o método de geração de vizinhos com base no número de iterações (*newNeighboorRandIter*).

3.3 Número de Iterações

Inicialmente, o número de iterações era limitado pelo número de tarefas devido ao método de geração de vizinhos. Depois, escolhemos 100, 5.000 e 50.000 iterações. Com 50.000 iterações, obtivemos os resultados apresentados neste trabalho. Notamos que para chegar a soluções suficientemente boas era necessário um número alto de iterações, pois dessa forma seria explorado um maior número de soluções vizinhas. Assim sendo, o número de iterações é o critério de parada do algoritmo.

4 RESULTADOS

Na Tabela 4.1 é possível observar uma comparação entre a melhor solução conhecida para cada instância e os resultados obtidos com a formulação em GLPK, assim como seus tempos de processamento. Visto que o problema a ser resolvido possui complexidade quadrática, a execução com GLPK se torna extremamente lenta e ineficiente devido ao aumento exponencial de restrições.

Além disso, na Tabela 4.2 são observados os resultados com a implementação da meta-heurística *Simulated Annealing*, também em comparação com a melhor solução conhecida, e os tempos de execução observados. Comparando os resultados obtidos com o GLPK e com o SA para as instâncias VFR20_10_3 e VFR20_20_1, percebe-se que o GLPK encontra soluções piores que as obtidas com meta-heurística em um tempo de execução inferior. Quanto às demais instâncias, a ferramenta GLPK não conseguiu encontrar nenhuma solução no tempo definido de uma hora.

A coluna Desvio(%) indica um desvio percentual entre a solução obtida e a melhor solução conhecida para a instância.

Tabela 4.1: Tabela de resultados da formulação em GLPK.

ID	BKS	Resultado	Tempo(s)	Desvio(%)
VFR10_15_1	1.307	1.307	978	0
VFR20_10_3	1.592	1.795	3600	11,30
VFR20_20_1	2.270	2.611	3600	13,06
VFR60_5_10	3.663	-	3.600	-
VFR60_10_3_Gap	3.423	-	3.600	-
VFR100_60_1_Gap	9.395	-	3.600	-
VFR500_40_1_Gap	28.548	-	3.600	-
VFR500_60_3_Gap	31.125	-	3.600	-
VFR600_20_1_Gap	31.433	-	3.600	-
VFR700_20_10_Gap	36.417	-	3.600	-

Tabela 4.2: Tabela de resultados com 10 execuções da meta-heurística.

ID	Sol. inicial	Melhor Sol.	Desvio Padrão	Tempo(s)	Desvio%
VFR10_15_1	1.455	1.309	1,41	02,81	0,15
VFR20_10_3	1.895	1.734	100,40	03,59	8,18
VFR20_20_1	2.836	2.449	126,57	08,94	7,95
VFR60_5_10	3.823	3.674	7,77	06,55	0,29
VFR60_10_3_Gap	4.292	3.898	335,87	12,30	12,18
VFR100_60_1_Gap	11.389	10.703	924,89	117,95	12,22
VFR500_40_1_Gap	33.560	32.704	2.938,73	421,33	12,70
VFR500_60_3_Gap	36.530	35.534	3.117,63	623,37	12,40
VFR600_20_1_Gap	36.145	34.930	2.472,75	260,17	9,90
VFR700_20_10_Gap	41.046	40.067	2.580,93	306,01	10,52

5 CONCLUSÃO

A disciplina nos mostrou o a eficácia do método *Simplex* para resolução de problemas. Porém, o GLPK não se mostrou eficiente para resolver o PFSP para a maioria das instâncias, como visto na Tabela 4.1. Em comparação com os resultados do GLPK, a implementação do *Simulated Annealing* se mostrou superior. Como obtivemos poucos resultados com o GLPK, não podemos realizar uma análise geral. No entanto, é possível observar que conforme a entrada aumenta, a diferença entre as soluções encontradas pela meta-heurística e GLPK também aumenta.

Além da otimalidade dos resultados, também é importante observar a grande diferença nos tempos de execução obtidos. Mesmo não obtendo resultados ótimos, as execuções realizadas com o *Simulated Annealing* foram melhores, com exceção da primeira instância, e mais rápidas do que as execuções com o GLPK.

REFERÊNCIAS

BERTSIMAS, D.; TSITSIKLIS, J. Simulated annealing. **Statist. Sci.**, The Institute of Mathematical Statistics, v. 8, n. 1, p. 10–15, 02 1993. Available from Internet: <https://doi.org/10.1214/ss/1177011077>.

MANNE, A. S. **On the job-shop scheduling problem**. [S.l.], 1960. Available from Internet: <https://pubsonline.informs.org/doi/abs/10.1287/opre.8.2.219>.

TSENG, F. T. **An empirical analysis of integer programming formulations for the permutation owshop**. [S.l.], 2002. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S030504830300152X>.