Otimização Combinatória - 2019/1

Permutational Flowshop Scheduling Problem(PFSP)

Bernardo Hummes Flores - 287689

Maria Cecília Matos Corrêa- 287703

- **Introdução**

Neste trabalho apresentamos os resultados da implementação de um métodos de aproximação de resultados do Permutational Flowshop Scheduling Problem (PFSP). Para a busca de uma solução aproximada do problema, com ênfase no tempo de execução, é utilizada a meta-heurística GRASP, *Greedy Randomized Adaptive Search Procedure*, com modificações para melhorar o desempenho, melhorando a velocidade sacrificando os requisitos de memória.

- **O problema**

O Permutational Flowshop Problem se trata de um problema de agendamento, onde um número M de máquinas devem executar sequencialmente um número N de tarefas de modo que cada máquina só comece uma tarefa depois que ela tenha sido terminada na máquina anterior. Uma máquina *r* leva para executar a tarefa *i* um tempo T[*r*,*i*], e o objetivo do PFSP é encontrar a ordem de realização de tarefas que o tempo total para que todas as máquinas executem todas as tarefas, chamado de *makespan*, seja mínimo. Para isso, criamos a variável Cmax, que guarda o maior tempo encontrado para a execução de alguma ordem de tarefas nas máquinas, e buscamos minimizá-lo na função objetivo, como vemos no código abaixo:

```
# makespan
var Cmax >= 0;
# funcao objetivo
minimize PFSP: Cmax;
```

No problema, temos também que todas as máquinas precisam executar todas as tarefas. Criamos, então, a variável inteira C[*r*,*i*] para guardar o tempo em que a tarefa *i* demorou para ser completada na máquina *r* a partir do início das execuções, ou seja, é a soma das execuções de tarefas anteriores a ela mais os possíveis tempos que cada tarefa espera que a máquina anterior termine de executá-la para começar a ser executada na seguinte, que é propagado adiante. Precisamos forçar que a primeira máquina comece a realizar as tarefas, caso contrário Cmax poderia ser zero, que é um valor mínimo mas que não nos interessa.

Depois disso, garantimos que o valor total do tempo para a ordem escolhida seja maior ou igual que Cmax para que ele sempre seja o menor.

```
s.t. comecarExecucao{ i in N }: C[1,i] >= T[1,i];
s.t. naoComecarAntesDaAnterior{ r in M, i in N: r > 1 }: C[r,i] - C[r-1,i] >= T[r,i];
s.t. makespan{ i in N }: Cmax >= C[|M|,i];
```

Por fim, precisamos fixar a ordem testada para execução das tarefas. A variável binária D[*i*,*k*], quando em 1, indica que a tarefa *i* foi agendada antes da tarefa *k*, e é 0 caso contrário. No código abaixo, *P* indica um número suficientemente grande, utilizado como *Big-M*. Quando ativado, ele garante que o tempo de execução de uma tarefa *k* na máquina *r* não é adicionado ao tempo de completar uma tarefa *i* que é agendada antes de *k*, e garante o contrário quando desativada. Com isso, terminamos a modelagem do Permutational Flowshop Problem com as duas últimas restrições:

```
s.t. fixa1{r in M, i in N, k in N: k > i and k > 1}: C[r,i] - C[r,k] + P*D[i,k] >=
T[r,i];
s.t. fixa2{r in M, i in N, k in N: k > i and k > 1}: C[r,i] - C[r,k] + P*D[i,k] <= P -
T[r,k];
```

- **Descrição do algoritmo**

Na implementação do GRASP, representamos uma possível solução como um arranjo de valores de 1 ao número de tarefas, com cada número representando uma tarefa, o objetivo foi encontrar a permutação que representa a organização com menor *makespan*. Ao usar um arranjo para representar cada possível solução, temos uma estrutura simples de dados, que ocupa espaço mínimo contendo toda a informação necessária, além de tornar a geração de vizinhos fácil, que é feito com a troca de dois valores de posições aleatórias.

Usamos uma matriz de duas dimensões para representar a matriz t, em que cada linha representa uma tarefa, com as colunas associadas sendo o tempo de execução dela em cada máquina, assim como no arquivo de texto das instâncias fornecidas. A representação foi mantida pela facilidade do cálculo do *makespan*, em que bastava construir uma matriz análoga, alterando para que cada item represente o tempo de término da tarefa *i* na máquina *j*; dessa maneira o *makespan* se encontrava disponível na célula do tempo de término da última tarefa na última máquina.

Outra estrutura de dados importante foi o dicionário de *makespans*, usado para evitar a computação repetida de soluções já consultadas, sendo o primeiro teste da função para esse cálculo a existência de um *makespan* já calculado na tabela, calculando apenas caso negativo. O uso dessa estrutura, apesar de reduzir o tempo de execução, aumentou consideravelmente a memória necessária para a execução do algoritmo, porém como ele tinha como objetivo um tempo limitado a 5 minutos para encontrar a melhor solução possível, optamos por usá-la.

Na parte de realizar a construção de uma solução, através de um método guloso randomizado, utilizamos uma versão modificada, em que ao invés de construirmos uma solução tarefa por tarefa, geramos um número arbitrário de soluções candidatas e, dentre as alfa% melhores, escolhemos uma aleatoriamente. Tomamos essa decisão pois a implementação original se demonstrou demasiadamente computacionalmente custosa, levando dezenas de vezes mais tempo para a construção de uma solução inicial do que para executar toda a busca local. Tentamos reduzir o tempo preservando a última linha das solução candidata sendo construída, à medida que era construída, para poupar cálculos, porém dessa maneira nosso algoritmo se tornou inviável em termos de memória.

Para a geração da solução inicial modificada, apenas criamos uma permutação aleatória dos arranjo de índices das tarefas. O mesmo é feito para a geração do conjunto de soluções candidatas, no início da Construção Gulosa-Randomizada. Com esse método, a construção de soluções iniciais não é direcionada de qualquer maneira para uma relativamente boa, porém assim é poupado o custo computacional disso e com uma quantidade considerável de soluções candidatas aleatórias, o espaço de soluções é efetivamente explorado, fugindo de casos em que pouca otimização é possível.

No quesito dos vizinhos, foi também optado por um processo simples, em que 2 valores aleatórios de 1 ao número de tarefas são sorteados e eles representam os índices do arranjo a serem trocados. Assim, a vizinhança, apesar de grande, representa uma porção consideravelmente pequena do universo de soluções. Optamos por não reaproveitar o cálculo da solução candidata que gerou um vizinho, que seria o reuso da matriz do tempo de término de cada tarefa em cada máquina, até o primeiro ponto de alteração, porque seria demasiadamente custoso para a memória, por ter que armazenar essa matriz para todas as soluções testadas.

Para a nossa implementação, temos como parâmetros o alfa, representando a porcentagem das soluções ótimas ordenadas que vai ser considerada para o sorteio da seguinte, o tamanho do conjunto de soluções iniciais, o número de iterações sem alteração do *Hill Climbing* e o número de iterações para a parada do GRASP. O alfa foi escolhido como 0,1 por cooperar com o número de soluções candidatas geradas - 270 -, levando ao sorteio de um indivíduo entre os 28 melhores , quantidade que em instâncias grandes proporciona uma boa forma de evitar soluções candidatas não promissoras. Já o número de soluções candidatas foi escolhido por funcionar bem com o valor do critério de parada, considerando que ele afeta menos a qualidade dos resultados, além de aumentar desnecessariamente o custo computacional do todo.

O *Hill Climbing* encerra com 50 iterações sem melhora, sendo o valor inicial desse parâmetro decidido como o menor possível que levasse a menor instância a sempre chegar à melhor solução conhecida. Ele foi ajustado após diversos testes ser o que levava à um mínimo local com frequência arbitrária aceitável, sacrificando minimamente o desempenho.

Por fim, o critério de parada do algoritmo foi a quantidade de iterações, escolhida de maneira a balancear com os outros valores a fim de limitar ao tempo de execução de 5 minutos.

- **Tabela**

Instância	|N|	|M|	BKS	Melhor Solução do GLPK	Solução com Relaxação	Tempo de Execução (s)	Melhor Solução com GRASP	Média dos Valores Iniciais	Média das Melhores Soluções Encontradas	Desvio Padrão das Melhores Soluções Encontradas	Média do Tempo de Execução (s)	Desvio Percentual das Soluções
VFR10_15_1_Gap	10	15	1307	1307	880	1975	1307	1550,5	1307,3	0,483	8,665	0,02295332823
VFR20_10_3_Gap	20	10	1592	1850	687	3600,6	1653	1987	1664,4	6,381	13,67	4,547738693
VFR20_20_1_Gap	20	20	2270	2764	1391	3600,8	2342	2723,4	2361,9	8,879	15,647	4,04845815
VFR60_5_10_Gap	60	5	3663	0	382	3636,7	3663	3963,1	3664,8	0,632	20,1667	0,049140049 14
VFR60_10_3_Gap	60	10	3423	0	720	3664,5	3724	4414,1	3753,9	15,581	25,518	9,666958808
VFR100_60_1_Gap	100	60	9395	0	0	3684,2	10430	11351,6	10470,2	34,829	122,578	11,44438531
VFR500_40_1_Gap	500	40	28548	0	0	3602,2	32054	33520,8	32125,1	48,685	264,109	12,5301247
VFR500_60_3_Gap	500	60	31125	0	0	3629,6	34868	36463,4	34949,9	41,698	321,552	12,28883534
VFR600_20_1_Gap	600	20	31433	0	0	3610,0	34332	36095,6	34393,4	61,664	234,933	9,418127446
VFR700_20_10_Gap	700	20	36417	0	0	3647,2	39336	4125,8	39432,2	45,841	270,811	8,279649614

Obs: melhor solução como zero significa que não houve valores inteiros definidos.

- **Análise dos resultados**

A partir das instâncias com 60 tarefas, uma hora não foi suficiente para que o GLPK encontrasse alguma solução inteira e, com 100 tarefas, nem mesmo fazendo relaxação. Isso nos mostra a relevância dos métodos heurísticos. Com base na tabela, pelas colunas de melhor solução encontrada e média das soluções iniciais, notamos que as melhores ordens de tarefas geradas pela nossa implementação são, de modo geral, bastante influenciadas pela ordem inicial, e tem uma melhora na solução com crescimento bastante consistente entre si.

Outro ponto interessante é perceptível nos desvios percentuais e médias das melhores soluções de uma instância, que mostram o quão próximas as melhores soluções para uma instância são uma da outra; como a busca local empregada não faz troca entre soluções igualmente boas e não tem nenhum mecanismo para evitar mínimos locais, é possível que a heurística precisasse de muitas iterações sem melhora a mais antes de conseguir uma ordem de tarefas com makespan menor.

- **Conclusão**

Com base na análise de dados, a procura por um makespan mínimo para o problema é consideravelmente impactada pelo uso de uma busca local combinada com uma boa solução inicial. No caso da nossa implementação do GRASP, apesar de possuir um método que tenta construir boas soluções, poderia ser mais criteriosa com o custo de memória e tempo de processamento, mas provavelmente obteria resultados melhores. Outra alternativa também seria uma busca local diferente, como o Simulated Annealing, que usa o artífício da temperatura para escapar de mínimos locais, como vimos ser a tendência do nosso algoritmo.

Além disso, no que tange a formulação matemática, existem modelos diferentes que são mais eficientes e poderiam ser usados, talvez conseguindo resultados inteiros, ao menos com a relaxação, no tempo estipulado. Ainda assim, os resultados inteiros encontrados nas instâncias menores não foram melhores que o encontrado pela heurística.

- **Referências**

[1] Luciana Buriol, Marcus Ritt, INF05010 – Otimização combinatória, Notas de aula

[2] T.A. Feo, M.G.C. Resende, and S.H. Smith, A greedy randomized adaptive search procedure for maximum independent set, Operations Research, vol. 42, pp. 860-878, 1994

[3] Fan TTseng, Edward F Stafford Jr., Jatinder N.D Gupta, An empirical analysis of integer programming formulations for the permutation flowshop, Omega, Volume 32, Issue 4, August 2004, Pages 285-293