INSTITUTO DE COMPUTAÇÃO - UNICAMP
MC658 - Análise de Algoritmos III

Docente: Prof. Cid Carvalho de Souza
PED: Natanael Ramos

1º Trabalho Prático

Grupo 12:

José Ribeiro - RA : 176665
Guilherme Henrique Higa Santan - RA : 160162 (Sem contribuição)

IC/UNICAMP
2019

1 Introdução

1.1 Problema Flowshop Schedule

O Flowshop é um clássico problema de agendamento, onde N tarefas devem ser processadas sequencialmente por M máquinas distintas, isto é, dada uma tarefa $i \in [1, N]$ a mesma poderá ser executada na máquina $j \in [2, M]$ somente quando acabar de ser processada pela máquina $j-1$, sendo que cada máquina executa uma única tarefa por vez. Para cada execução, temos um tempo de duração d_{ji}, sendo i referente a i-th tarefa e j a j-th máquina. Além disso, associado a esta duração, temos também o tempo de término de cada tarefa i na máquina j, dado por f_{ji}. Com estas definições, podemos descrever uma das vertentes do Flowshop Schedule, o problema da soma dos tempos de término (da sigla em inglês *SFTP*), consistindo em encontrar a ordem em que as N tarefas devem ser executadas em cada máquina de forma a minimizar f:

$$f = \sum_{i=1}^{N} f_{Mi} \tag{1}$$

Ou melhor, o *SFTP* busca a ordem de execução das N tarefas na M-th máquina, dado que conhecemos o seguinte teorema, provado em [1, pg. 81]:

Teorema 1 *Há uma permutação ótima para o SFTP no qual todas as M máquinas processam as N tarefas na mesma ordem, sem que haja tempo ocioso desnecessário entre execuções. A este fato, dá-se o nome de "Permutation Schedules".*

Para resolução deste problema, um primeiro enfoque (ingênuo) seria encontrar, dentre todas as $N!$ permutações, a que produz o menor valor f. Contudo, para $N > 13$, este algoritmo se torna impraticável. Em contrapartida, uma solução mais elegante consiste em explorar apenas regiões específicas da árvore de permutações, sem que haja o risco de inexploração de uma região contendo uma solução ótima. Para tanto, uma possível abordagem consiste no uso do paradigma de design de algoritmos Branch and Bound (B&B) [*W*] associado com uma possível estratégia de poda, capaz de impedir a exploração das regiões citadas. Com este algoritmo, é esperado que seja possível resolver o problema para instâncias com $N > 13$.

Desta forma, neste trabalho buscamos estudar o B&B aplicado na resolução do *SFTP*, bem como implementá-lo em *C++*. Com ele, pretendemos en-

contrar limitantes (duais e primais) que nos possibilite excluir ou postergar a exploração de possíveis permutações da entrada (conjunto de tarefas) que são garantidamente não-ótimas. Além disso, assumimos nesta implementação que o número de máquinas (M) é fixado em 2, como dado por [2, pg. 444-448].

Nas próxima seção 2 explicamos o paradigma de B&B, bem como citamos as técnicas utilizadas para encontrar os limitantes citados. Na seção 3, descrevemos a relação de dominância utilizada para eliminar ainda mais permutações desnecessárias. Por fim, na seção 4, comentamos as possíveis diferentes implementações realizadas, além de apresentarmos os resultados obtidos para a execução do algoritmo em 27 instâncias distintas do problema.

2 Branch and Bound para o SFTP

Branch and Bound (B&B) é um paradigma de design de algoritmo que pode ser utilizado para a resolução de problemas de Otimização combinatória, tal como o *SFTP*. O algoritmo utilizado na resolução deste problema consiste em explorar ramos da árvore de permutações, que por sua vez representam subconjuntos do conjunto de soluções S. Na exploração de cada ramo (ou nodo) E, é aplicado a operação de Branch, onde é calculado o melhor custo f_{E_j} que pode ser obtido ao explorar cada descendente E_j de E. A este custo, chamamos de limitante dual (ou limitante inferior no nosso problema). Para mais informações sobre seu cálculo, consultar [2, pg. 444-448]. Em seguida, é realizado o passo de bound, de forma que cada custo f_{E_j} é comparado com o custo da melhor solução factível encontrada até o momento, também chamada por limitante primal (ou limitante superior no nosso problema). Caso f_{E_j} seja menor do que o custo do limitante primal, então E_j é inserido na árvore de nodos ativos, tendo o seu respectivo limitante dual f_{E_j} utilizado como chave identificadora (e de comparação) nesta árvore. Do contrário, este nodo E_j é descartado, e a execução do algoritmo prossegue. Além disso, existem diversas estratégias para a escolha do nodo E. Para este relatório, utilizamos a de best-bound, onde o nodo com a melhor chave identificadora f_E é escolhido primeiro.

3 Relação de dominância do SFTP

3.1 Descrição teórica

Tomemos dois nodos da árvore de permutação: t e u. Assumimos que ambos tenham o mesmo conjunto de tarefas agendadas. Neste caso, se tivermos

$$f_{2t_r} \leq f_{2u_r}, \tag{2}$$

(sendo t_r o índice da última tarefa agendada em t e u_r o índice da última tarefa agendada em u) e caso

$$\sum_{i \in A} F_{2i} | schedule\ t \leq \sum_{i \in B} F_{2i} | schedule\ u, \tag{3}$$

(sendo A o conjunto de indíces de tarefas já agendadas em t e B o conjunto de indíces de tarefas já agendadas em u) então temos que a melhor solução obtida explorando os descendentes de t é no mínimo tão boa quanto a melhor solução obtida explorando os descendentes de u, e portanto dizemos que t domina u.

3.2 Implementação

Para a implementação desta relação de dominância, associamos um número inteiro (32 bits), chamado 'tasks', a cada nodo t da nossa estrutura de nodos ativos. Este número simplesmente representa o conjunto de tarefas agendadas até o momento da exploração do nodo t. Isto é, se o i-th bit de 't.tasks' for 0, isto indica que a i-th tarefa ainda deverá ser agendada por algum descendente de t. Caso contrário, a i-th tarefa já foi agendada.

Assim, verificar se um nodo u é passível de ser dominado por um outro nodo t se torna uma tarefa simples, visto que basta encontrarmos os nodos que possuem os mesmos bits setados em 'u.tasks', ou seja, que possuem o mesmo valor (em decimal) que 'u.tasks' (a relação entre o número na base binária é 1:1 com relação a decimal). Para realizarmos esta verificação, utilizamos uma tabela de dominância hash (ou um 'unordered_map' no caso de C++), onde a chave é dada por 't.tasks' e o conteúdo relacionado a chave dado por uma tupla '(t.f2, t.sumF2)', sendo 't.f2' o mesmo valor f_{2t_r} descrito na equação 2 e sendo 't.sumF2' o mesmo somatório descrito na equação 3.

Por simplicidade das implementações e restrições de tempo, uma única tupla foi mantida para cada valor de 'tasks'. Desta forma, a cada novo nó u sendo removido da nossa árvore de nós ativos, buscávamos na tabela de dominância pela existência de 'u.tasks'. Caso não encontrássemos, 'u.tasks' junto a ('u.f2', 'u.sumF2') eram insiridos na tabela e a etapa de branch dos descendentes de 'u' prosseguia normalmente. Do contrário, verificávamos se 'u.f2' e 'u.sumF2' eram equivalentes às equações 2 e 3, sendo 't' a tupla já presente na tabela. Caso de fato fosse equivalente, concluíamos que o nodo 'u' tinha sido dominado pelo nodo 't', e portanto o branch dos descendentes de 'u' eram evitados. Caso fosse incompatível, então a tupla referente a 't' era substituída pela tupla ('u.f2', 'u.sumF2').

4 Resultados do Experimento

4.1 Dados da máquina

O experimento foi realizado em uma máquina com 15.5 GiB de RAM e Intel® Core^{TM} i7-7700HQ CPU @ 2.80GHz × 8, rodando Ubuntu 18.04.2 LTS. O programa foi compilado com o g++ em sua versão 7.4.0, estando ativadas as flags '-std=c++11' e '-O3'.

4.2 Comentários sobre a estrutura de nodos ativos

Para a estrutura de nodos ativos, consideramos duas possibilidades: utilizar um heap de prioridades (*priority_queue* em *C++*) ou uma árvore de busca balanceada (*map* em *C++*, que implementa um algoritmo de red-black tree). Para ambas, temos um custo de $\mathcal{O}(log(n))$ na inserção e na remoção. Apesar das contantes iguais, após a implementação, notamos uma certa diferença de eficiência, de forma a obtermos melhores tempos com a segunda em comparação aos da primeira. Por exemplo, usando o heap de prioridades, o custo estimado do nodo como chave para a ordenação, e tomando a instância 30 como teste, obtivemos um tempo para a parada por otimalidade igual a $21.24s$ (um tempo $1.41x$ maior do que o apresentado na tabela 2, que utilizou um *map*). O mesmo resultado se repetiu na instância 28, onde obtivemos um tempo igual a $11.39s$ para parada por otimalidade (aumento de $1.36x$).

Neste caso, estamos assumindo uma comparação injusta. Isto pois, na implementação da árvore de busca (segunda estrutura de dados), utilizamos como chave o custo estimado para o nodo, e como conteúdo a pilha contendo todos os nodos que compartilhavam este mesmo tempo estimado. Assim, ao extrairmos um elemento da estrutura de nodos ativos, e consequentemente o primeiro nodo da pilha respectiva, podíamos comparar seu valor de custo estimado com o custo do melhor limitante primal encontrado até momento. Caso o valor deste limitante fosse menor do que o custo estimado do nodo, certamente não precisaríamos realizar o branch dos descendentes deste nodo (ou de qualquer outro nodo presente na pilha citada), visto que o melhor custo a ser obtido não poderia ser melhor do que um já encontrado. Esta diferença de implementação fica evidente ao compararmos a quantidade de nodos explorados pela primeira estrutura (heap) e pela segunda (árvore balanceada). Por exemplo, para a mesma instância 30 citada anteriormente, exploramos 53586854 nodos quando a estrutura de heap foi utilizada. Já, quando a árvore balanceada foi considerada, uma quantidade igual a 52774300 nodos foram exploradas. Ou seja, no primeiro caso, 812554 nodos foram explorados a mais em comparação ao segundo. Reflexo este direto da poda realizada sobre a pilha citada acima.

4.3 Comentários sobre o uso da relação de dominância

A utilização da relação de dominância foi uma das estratégias que mais permitiu podas. Além disso, a simplicidade de sua implementação permitiu com que a sobrecarga referente ao uso do "unordered_map" não fosse relevante perante o ganho em tempo de execução até a parada do programa por otimalidade, justificando assim o seu uso. Ou melhor, não apenas justificando, como também necessitando, visto que o uso de memória RAM quando a estratégia de dominância estava inativa facilmente extrapolava os 10Gb para o caso da instância i20 da tabela 2. Neste caso, uma quantidade de nodos igual a 172275208 são explorados quando definimos um tempo limite de $35s$. Em contrapartida, quando a relação estava ativa, aproximadamente $0.05Gb$ de memória foi utilizada, e apenas 4461426 nodos foram explorados (como mostra a tabela 2), ou seja, no minimo uma diminuição de 38 vezes em comparação aos 172275208 nodos visitados.

Uma outra questão que nos deparamos foi escolher entre manter uma única tupla associada a uma dada chave do hash, ou mantermos várias tuplas para uma mesma chave hash (assumindo aqui que uma tupla não dominaria

a outra). Por simplicidade de implementação, optamos pela primeira opção, estando a segunda aberta para exploração. Contudo, mesmo no âmbito da primeira, pudemos realizar alguns testes, tal como a substituição de tuplas somente quando o valor de $f2$ da tupla antiga fosse maior que o valor de $f2$ da tupla mais nova, ou talvez quando $sumF2$ da antiga fosse maior que $sumF2$ da tupla mais nova. Neste caso, obtivemos bons e maus resultados com relação a primeira alternativa (substituição das tuplas apenas quando $f2$ da mais antiga fosse maior do que $f2$ da mais nova). Ou seja, tivemos um ganho de $1s$ para a instância 30 da tabela 2, porém perdemos em torno de $0.1s$ na instância 29. Em contrapartida, utilizando a segunda estratégia (substituição apenas quando $sumF2$ da tupla mais antiga fosse maior do que $sumF2$ da tupla mais nova), resultados péssimos foram obtidos, onde para a mesma instância 29, $35s$ não foram suficientes para encontrar a permutação ótima, tendo um total de 133823323 nodos explorados (ou seja, um aumento de no mínimo 3.75 em comparação ao citado na tabela 2). Para esta mesma estratégia, e para a instância 30, um acréscimo de aproximadamente $1.4s$ foi obtido. Portanto, dado a disparidade de ambas as estratégias, preferimos manter a simplicidade, ou seja, sempre que um nodo não era dominado, o mesmo tomava o lugar da tupla presente na tabela. Possivelmente não é a melhor estratégia, contudo possibilitou bons resultados.

Vale-se também de ressalva que em um dado momento consideramos o uso de uma árvore binária balanceada (*map* em *C++*, que implementa um algoritmo de red-black tree) ao invés de uma tabela hash (*unordered_map* em *C++*). Para tanto, o tempo necessário para parada por otimalidade foi ligeiramente maior para todas as instância em comparação aos obtidos com o uso do hash. Em especial citamos a instância i30, onde o tempo necessário até a parada foi de $18.33s$, ou seja, um aumento de cerca de $1.22x$ em comparação aos $15.03s$ descrito na tabela 2. Esta diferença se justifica devido ao comportamento assintótico das operações de inserção e busca sobre ambas as estruturas, sendo $\mathcal{O}(log(n))$ para o *map* e $\mathcal{O}(1)$ para o *unordered_map*. Desta forma, acabamos por continuar com o hash.

References

[1] Conway, R. W., W. L. Maxwell, and L. W. Miller, Theory of Scheduling. Reading, Mass.: Addison-Wesley Publishing Co., Inc., 1967

[2] Papadimitriou, C. H., Steiglitz, K., Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Inc., 1982

Table 1: Dados referentes as instâncias 08 - 18

Instância	**Nós explorados**	**Lim. Primal**	**Tempo Lim. Primal (s)**	**Lim. Dual**	**Tempo Lim. Dual (s)**	**Tempo Total**	**Melhor Solução**
i08 (N=8)	302	212	0.00	182	0.00	0.00	2-4-3-5-7-8-6-1
i08b (N=8)	722	186	0.00	156	0.00	0.00	1-7-3-5-4-6-2-8
i09 (N=9)	69	197	0.00	170	0.00	0.00	5-7-2-3- 1-4-6-9-8
i10 (N=10)	2029	2177	0.00	1320	0.00	0.00	6-7-5-4-1- 10-3-2-9-8
i10b (N=10)	3994	156	0.00	134	0.00	0.00	10-2-8-5- 6-3-9-1-4-7
i11 (N=11)	11	2965	0.00	971	0.00	0.00	1-8-10-2-3- 4-6-5-11-9-7
i12 (N=12)	1384	3017	0.00	1632	0.00	0.00	11-10-9-8-12- 5-7-6-4-3-2-1
i12b (N=12)	22568	302	0.01	245	0.00	0.01	6-12-7-2-9- 10-4-3-8-1-5-11
i13 (N=13)	33175	24619	0.01	11665	0.00	0.01	2-12-4-6-7-11- 3-9-8-10-5-13-1
i13b (N=13)	67275	692	0.02	546	0.00	0.03	13-2-11-4-9-6- 7-8-5-10-3-12-1
i14 (N=14)	120796	458	0.04	364	0.00	0.05	7-14-8-2-10- 4-11-12-5- 3-9-1-6-13
i15 (N=15)	262838	266	0.01	226	0.00	0.11	11-1-15-13-7- 5-3-2-4-6- 8-10-12-14-9
i16 (N=16)	29706	729	0.00	622	0.00	0.01	2-4-13-14-1- 3-10-12-5-6- 16-15-11-7-9-8
i17 (N=17)	95024	834	0.01	630	0.00	0.03	2-8-13-5-9- 11-16-15-3-17-10- 14-1-12-6-7-4
i18 (N=18)	196772	974	0.02	832	0.00	0.05	1-5-4-6-17- 15-12-16- 13-14-10-9-11- 7-2-3-8-18

Table 2: Dados referentes as instâncias 19 - 30

Instância	**Nós explorados**	**Lim. Primal**	**Tempo Lim. Primal (s)**	**Lim. Dual**	**Tempo Lim. Dual (s)**	**Tempo Total**	**Melhor Solução**
i19 (N=19)	28851	1009	0.00	755	0.00	0.01	4-8-13-3-2-9-6-5-11-7-17-18-19-14-10-16-1-12-15
i20 (N=20)	4461426	1128	0.70	888	0.00	1.39	2-15-16-11-19-4-5-13-3-17-18-10-1-14-12-8-9-6-20-7
i21 (N=21)	1330968	1089	0.19	901	0.00	0.37	15-10-16-2-14-11-3-9-17-6-7-20-19-5-1-4-8-18-21-13-12
i22 (N=22)	3300448	1183	0.56	1000	0.00	0.88	9-7-11-17-13-4-15-19-12-3-22-21-5-14-1-6-8-10-18-20-16-2
i23 (N=23)	6520308	1267	1.08	1058	0.00	1.77	6-18-19-14-17-12-2-16-22-8-23-9-13-1-21-15-4-3-7-11-10-20-5
i24 (N=24)	3510609	1146	0.65	996	0.00	0.86	5-4-24-17-23-19-22-21-14-16-13-12-10-2-20-3-6-18-11-1-15-9-8-7
i25 (N=25)	1490685	1810	0.28	1485	0.00	0.42	18-9-10-14-4-7-20-12-5-17-21-11-24-2-22-15-3-25-19-13-8-6-23-16-1
i26 (N=26)	859730	1525	0.14	1147	0.00	0.23	24-18-26-5-11-23-3-6-25-4-8-10-12-13-17-22-14-16-1-20-9-2-15-21-19-7
i27 (N=27)	3270508	2050	0.36	1626	0.00	0.95	9-20-7-21-22-3-2-15-12-24-8-4-27-1-17-10-13-5-25-23-11-18-19-14-6-16-26
i28 (N=28)	24804057	1970	6.84	1557	0.00	8.34	22-10-11-3-19-20-7-21-17-1-4-28-9-5-26-8-12-24-6-23-14-2-18-15-27-16-13-25
i29 (N=29)	35639943	2250	7.07	1786	0.00	12.67	16-18-29-28-20-1-15-14-23-2-10-5-21-3-4-26-22-8-11-17-9-7-19-6-27-25-13-24-12
i30 (N=30)	52774300	2126	13.61	1842	0.00	15.03	28-21-14-9-15-13-24-7-3-1-5-17-19-4-23-2-27-30-16-10-20-25-26-22-12-6-11-8-29-18