Universidade Federal
do Espírito Santo

Centro Tecnológico

Departamento de Informática

Job Scheduler

Autor:
Vinicius Arruda

Professora:
Mariella Berger

10 de outubro de 2015

Resumo

Trabalho da disciplina de Estrutura de Dados II, que consiste na implementação de dois algoritmos para solucionar o problema de escalonamento de tarefas (do inglês: *Job Scheduler*). Os algoritmos são o Beam Search e o Branch and Bound.

1 Introdução

Um escalonador de tarefas é um programa que consiste em determinar uma atribuição de tarefas para um processador de forma a otimizar o tempo de execução total destas tarefas. O problema de um escalonador de tarefas é da classe dos problemas NP-completo (do inglês: *NP-complete*) e pode ser convertido para o problema da mochila (do inglês: *knapsack problem*) e do caixeiro viajante (do inglês: *travelling salesman problem*).

A proposta deste trabalho é mostrar duas implementações para encontrar o melhor escalonamento de tarefas. O Beam Search é uma das implementações, e por se tratar de uma heurística, o resultado não é necessariamente o melhor. A outra implementação é o Branch and Bound, que é um algoritmo exato e encontra o melhor escalonamento.

2 Objetivo

Aplicar o conhecimento adquirido na disciplina de Estruturas de Dados II e reconhecer a complexidade dos problemas NP-completo aplicando algoritmos de aproximação e outras técnicas para contornar a dificuldade do problema em encontrar a solução ótima.

3 Ferramentas

Os algoritmos foram implementados na linguagem de programação C. Para a compilação foi utilizado o compilador GCC versão 4.8.4 em uma máquina com o sistema operacional Ubuntu 14.04.3. O código foi escrito utilizando o editor de texto Sublime Text. Para a depuração do programa, foi utilizado a ferramenta Valgrind versão 3.10.

4 Beam Search

Beam Search é uma heurística de busca que explora um grafo expandindo o caminho mais promissor. No caso deste trabalho, o caminho mais promissor é aquele de menor *Upper Bound*[1]. O Beam Search explora o grafo em largura (do inglês: *Breadth-first Search*) montando uma árvore. A cada nível da árvore

[1] *Upper Bound*: Cálculo feito a priori do custo máximo que um caminho pode obter. A medida que o caminho vai sendo percorrido, este valor pode decrescer.

montada, o algoritmo gera os filhos dos n nós mais promissores, onde n é o *Beam-width*[2] e todos os outros nós são descartados. Escolher o nó de menor *Upper Bound* não garante que este produzirá a melhor solução, por isso, se trata de uma heurística.

4.1 Metodologia

Inicialmente, uma implementação do Beam Search baseada em busca primeiro em profundidade (do inglês: *Depth-first Search*) foi desenvolvida. Após algumas pesquisas sobre o Beam Search para a elaboração deste relatório, foi constatado que este deveria ser implementado baseado em busca primeiro em largura. Sendo assim, uma outra versão foi implementada baseado na busca primeiro em largura.

No arquivo *beamSearch.c* na pasta *BeamSearch*, encontra-se a implementação das duas versões com as assinaturas *beamSearchBreadth(...)* e *beamSearchDepth(...)* para a versão em busca primeiro em largura e busca primeiro em profundidade, respectivamente. A versão utilizada ao executar *./trab3 # bs* é a baseada em busca primeiro em largura, pois de acordo com a pesquisa feita[3] e com os testes realizados, o resultado é obtido mais rápido e com melhor precisão.

[2] *Beam-width*: Número de nós promissores a serem desenvolvidos.

[3] Ver referências no final do relátório.

Algorithm 1 Função de auxiliar ao Beam Search

```
1: function HANDLER(PriorityQueue, E, s, BeamWidth)
2:     for each b ∈ E do
3:         CalculatesBoundaries(b) ▷ Calculates the lower and upper bound of b.
4:         if b.LowerBound < s.Cost then
5:             if b.UpperBound < s.Cost then
6:                 s.Cost ← b.UpperBound
7:             end if
8:
9:             if b.LowerBound = b.UpperBound then
10:                if b.UpperBound = s.Cost then
11:                    s ← b
12:                end if
13:            else
14:                PriorityQueue.Enqueue(b)
15:            end if
16:        end if
17:    end for
18:    PriorityQueue.Reduce(BeamWidth)          ▷ Reduces the Queue to
   BeamWidth elements.
19: end function
```

O algoritmo 1 mostra uma função que é utilizada no código do Beam Search para manipular o *Lower Bound* e o *Upper Bound* de cada nó e a partir de seus valores, decidir se será incluso na fila de prioridade ou se sera excluído. A exclusão de um nó se ocorre quando o custo do caminho chegando até este for maior que o custo do melhor caminho encontrado ou quando a fila já possuir *Beam-width* elementos de maior prioridade.

A seguir, o algoritmo 2 representa a primeira implementação do Beam Search, baseado em busca primeiro em profundidade e logo após, o algoritmo 3 que é a implementação atual, baseada em busca primeiro em largura.

Algorithm 2 Algoritmo Beam Search baseado em Depth-first Search

```
1: procedure BeamSearchDepth(Jobs, BeamWidth)
2:     s ← ∞                                              ▷ Best solution.
3:     C ← ∅                    ▷ Set of nodes generated by a promising node.
4:     PriorityQueue ← ∅
5:     J ← Jobs
6:     Handler(PriorityQueue, J, s, BeamWidth)
7:     while PriorityQueue ≠ ∅ do
8:         j ← PriorityQueue.Dequeue()
9:         C ← Explore(j)                    ▷ Return the children nodes of j.
10:        Handler(PriorityQueue, C, s, BeamWidth)
11:    end while
12: end procedure
```

Algorithm 3 Algoritmo Beam Search baseado em Breadth-first Search

```
1: procedure BeamSearchBreadth(Jobs, BeamWidth)
2:     s ← ∞                                              ▷ Best solution.
3:     C ← ∅                    ▷ Set of nodes generated by a promising node.
4:     PrevPQ ← ∅                          ▷ Previous level Priority Queue.
5:     NextPQ ← ∅                              ▷ Next level Priority Queue.
6:     J ← Jobs
7:     Handler(PrevPQ, J, s, BeamWidth)
8:     while PrevPQ ≠ ∅ do
9:         while PrevPQ ≠ ∅ do
10:            j ← PriorityQueue.Dequeue()
11:            C ← Explore(j)                ▷ Return the children nodes of j.
12:            Handler(NextPQ, C, s, BeamWidth)
13:        end while
14:        PrevPQ ← NextPQ
15:        NextPQ ← ∅
16:    end while
17: end procedure
```

4.2 Resultados

Para analisar os resultados, foi elaborado um script para criar Jobs aleatórios. Os Jobs foram gerados com números aleatórios de 1 a 1000, sendo que o valor da *deadline* é sempre maior ou igual ao valor do tempo de processamento.

Foram gerados um total de 100 arquivos de Jobs, cada um com um número de Jobs distinto, indo de 10 à 1000, de 10 em 10. Para a análise do *Beam Width*, foi executado para cada arquivo de Jobs o algoritmo Beam Search nas duas versões, com o *Beam Width* indo de 1 à 15 de 2 em 2, e de 20 à 50 de 5 em 5, somando um total de 15 valores de *Beam Width* distintos para cada arquivo de Jobs.

O resultado final da base de testes para as configurações dispostas acima gerou uma tabela de 1500 linhas por 7 colunas, e a partir dela foi gerada uma outra tabela filtrando apenas os resultados dos melhores custos para cada arquivo de Jobs. O resultado filtrado se encontra na tabela 1 a seguir.

Tabela 1: Resultados do algoritmo Beam Search.

	Breadth-first Search			Depth-first Search		
# Jobs	**Beam Width**	**Custo**	**Tempo (s)**	**Beam Width**	**Custo**	**Tempo (s)**
10	1	2116	0	1	2116	0
20	30	5407	0	35	5159	0.006
30	5	14688	0	35	14688	0
40	25	18222	0	45	18222	0.029
50	25	23541	0.003	30	23998	0.015
60	20	24547	0.004	50	25421	0.599
70	9	29805	0.003	40	32563	0.028
80	25	39013	0.008	30	39537	0.023
90	30	37638	0.015	1	40072	0
100	25	43635	0.018	35	45216	2.231
110	30	50427	0.028	1	53666	0
120	20	59167	0.019	20	59167	0.063
130	35	55347	0.033	30	56458	0.026
140	5	64308	0.007	1	68389	0
150	45	65546	0.091	1	72327	0
160	35	71099	0.093	5	76897	0
170	20	73427	0.036	1	76482	0
180	50	81370	0.103	35	85193	0.149
190	45	88590	0.105	45	90713	0.43
200	25	98563	0.06	45	103045	1.798
210	20	94139	0.06	1	97490	0.004
220	20	105507	0.062	45	107596	2.842
230	1	107203	0.004	1	107203	0.004
240	50	113709	0.172	50	116441	2.825
250	30	116825	0.136	1	123111	0.003
260	50	121388	0.25	1	122990	0.005
270	35	119963	0.195	35	124032	0.26
280	15	125289	0.126	1	131814	0.006
290	15	137340	0.088	1	143029	0.004
300	3	144936	0.019	1	146246	0.006
310	13	144254	0.133	25	148236	0.286
320	50	156292	0.339	1	159100	0.008
330	15	154338	0.121	1	161481	0.005
340	50	163944	0.397	1	168758	0.008
350	7	162199	0.067	1	165264	0.009
360	30	184675	0.246	20	187477	0.333
370	7	174647	0.070	20	175874	0.705
380	25	176947	0.236	1	180560	0.008
390	20	185904	0.292	1	194711	0.005
400	3	193068	0.026	1	195792	0.011
410	15	183211	0.278	30	191137	2.355
420	20	217921	0.252	1	221013	0.012
430	50	199521	1.232	1	206763	0.01
440	5	219350	0.075	1	222770	0.011
450	7	205101	0.135	1	211827	0.013
460	1	231464	0.017	1	231464	0.017
470	11	224489	0.22	1	229975	0.012
480	35	224341	0.942	1	231341	0.011
490	25	235094	0.593	1	240679	0.013
500	25	239052	0.581	1	243407	0.015
510	25	239110	0.484	1	248254	0.014

Continua na próxima página.

Tabela 1 – *Continuação - Resultados do algoritmo Beam Search.*

	Breadth-first Search			Depth-first Search		
# Jobs	**Beam Width**	**Custo**	**Tempo (s)**	**Beam Width**	**Custo**	**Tempo (s)**
520	11	254474	0.236	1	259434	0.017
530	25	249127	0.59	1	255166	0.012
540	40	251451	0.863	1	254037	0.024
550	25	268296	0.508	1	274419	0.01
560	5	273821	0.125	1	275817	0.028
570	25	273539	0.819	1	277508	0.024
580	7	284812	0.196	1	288250	0.021
590	20	281137	0.57	1	289789	0.012
600	11	289618	0.39	1	298223	0.017
610	25	294442	0.88	1	304456	0.014
620	15	291908	0.71	1	297302	0.031
630	7	306224	0.292	1	312330	0.014
640	50	314449	2.041	30	320320	2.323
650	9	320060	0.34	1	322588	0.033
660	40	337446	1.727	1	342969	0.027
670	40	331950	1.648	1	333809	0.032
680	35	332758	2.477	1	338038	0.043
690	7	333156	0.254	1	335149	0.032
700	30	331872	1.611	1	337818	0.028
710	9	346765	0.49	1	349552	0.036
720	40	356180	2.006	1	365181	0.023
730	20	361947	0.993	1	365545	0.048
740	7	377962	0.326	1	379822	0.035
750	40	381025	2.646	1	386714	0.039
760	45	364211	3.501	1	366620	0.054
770	50	377252	2.24	1	385433	0.017
780	40	374846	2.247	3	380221	0.056
790	40	388331	3.363	1	395538	0.035
800	20	392043	1.091	1	396963	0.043
810	20	389193	1.539	1	399469	0.027
820	45	406499	2.754	1	414759	0.048
830	40	409492	2.084	1	412812	0.034
840	40	405288	2.823	1	413555	0.025
850	40	410465	3.308	1	416220	0.069
860	25	420828	1.794	1	427606	0.079
870	20	438434	1.799	1	441639	0.143
880	9	435297	0.831	1	441932	0.092
890	20	432000	2.164	1	439794	0.071
900	50	442968	3.669	35	447016	14.005
910	25	453157	2.235	1	457580	0.111
920	30	446557	2.944	1	452133	0.088
930	45	452940	5.184	1	459833	0.144
940	3	467678	0.273	1	470033	0.081
950	20	473669	2.251	1	482473	0.067
960	40	458785	5.531	1	465562	0.115
970	25	456365	2.627	1	465185	0.071
980	7	487925	0.625	1	494015	0.121
990	40	474197	5.629	1	485967	0.115
1000	11	477882	1.097	1	484289	0.051

Podemos observar pela tabela 1 a quantidade de melhores soluções que cada versão obteve, e segue na tabela 2.

Tabela 2: Quantidade de melhor solução para as duas versões.

Breadth-first Search	Depth-first Search	Ambas as versões
93	1	6

Ainda pela tabela 1, obtemos o número de soluções encontradas para cada *Beam Width* da tabela e segue o resultado na tabela 3.

Tabela 3: Número de melhores soluções para cada Beam Width.

Beam Width	Número de soluções
1	3
3	3
5	4
7	8
9	4
11	4
13	1
15	5
20	14
25	16
30	6
35	6
40	12
45	5
50	9

4.3 Conclusão

A partir dos resultados obtidos, configuramos como *Beam Width* do algoritmo Beam Search o valor 25, pois pela tabela 3, com este valor o algoritmo Beam Search obteve melhores soluções.

Não podemos, de forma alguma generalizar o *Beam Width* 25 como melhor *Beam Width* possível, pois pela natureza do algoritmo, ele é sensível a entrada. Mas podemos admitir que 25 é um bom valor de *Beam Width*.

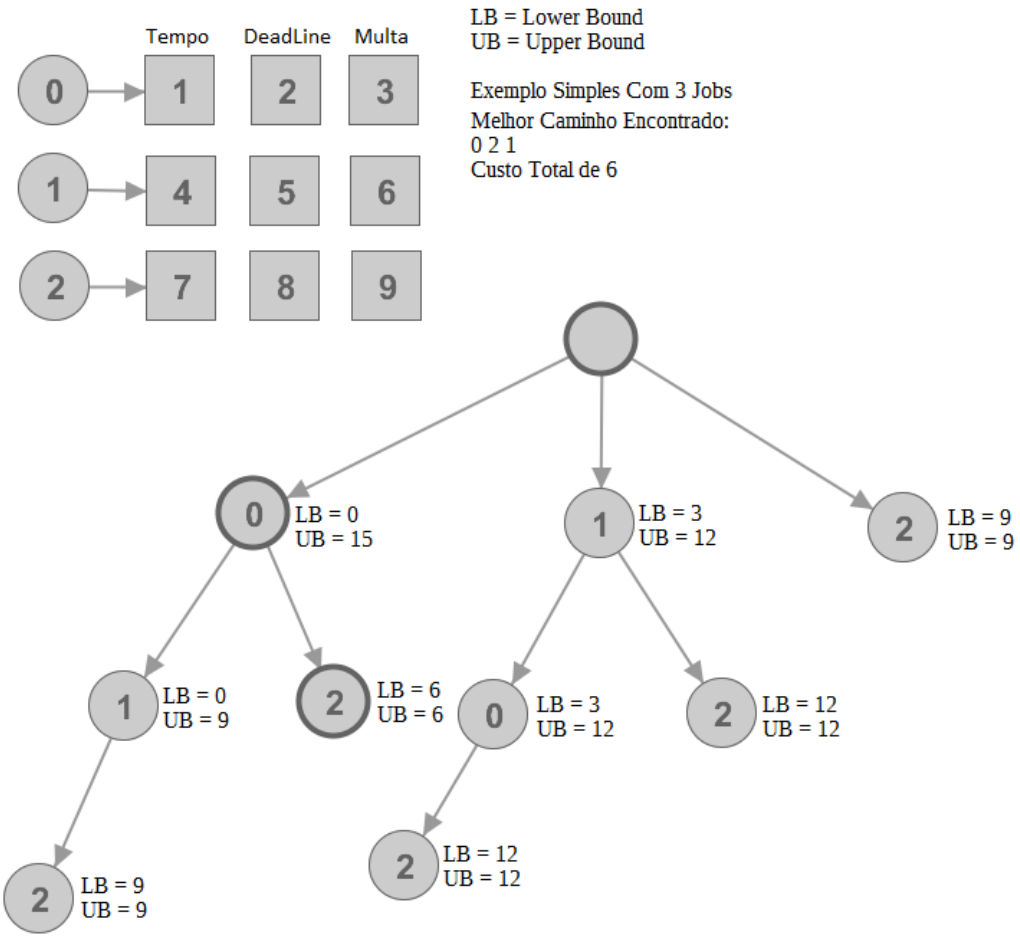
Sem dúvidas o Beam search baseado em busca primeiro em largura se mostrou muito mais eficiente do que o baseado em busca primeiro em profundidade. O lado negativo do baseado em busca primeiro em largura é um maior consumo de memória para um *Beam Width* muito grande, mas para valores como 100, a memória gasta é irrelevante.

5 Branch and Bound

Branch and bound é um algoritmo que busca o valor exato, ou seja, o melhor valor possível para o problema. É mais lento que o Beam Search por precisar percorrer varias possibilidades. O Branch and Bound consiste em uma enumeração sistemática de todos os candidatos à solução, com a eliminação de um candidato parcial à solução quando puder ser afirmado que esta não será a melhor escolha. Dependendo do tamanho da entrada, torna-se inviável nos piores casos utilizar o Branch and Bound. O desempenho do algoritmo Branch and Bound está fortemente relacionado a qualidade dos seus limitantes inferiores e superiores (Lower Bound e Upper Bound), quanto mais precisos forem estes limitantes, menos soluções parciais serão consideradas e mais rápido o algoritmo encontrará a solução ótima.

A forma encontrada para não cair no pior caso é utilizar uma heurística antes e a partir do resultado dela realizar o algoritmo, assim melhorando significantemente o tempo de execução do programa gerando soluções razoavelmente rápidas nos casos médios. Para este trabalho, o algoritmo Beam Search será a heurística que fornecerá uma possível solução para o problema do escalonamento de tarefas, e a partir desta solução, o algoritmo Branch and Bound irá procurar e encontrar a melhor solução possível.

A figura abaixo mostra um pequeno exemplo de 3 Jobs. Caso o Branch and Bound tenha como solução inicial fornecido por uma heurística o valor 9, as ramificações serão como o da figura. Podemos garantir que o custo de qualquer escalonamento que comece com os Jobs 0 depois 2 tenha o custo 6, então é só preencher o escalonamento com os Jobs que restam, que no caso é o Job 1. Então o escalonamento de menor custo encontrado é: 0 2 1, com custo 6.



5.1 Metodologia

A meodologia para o Branch and Bound é semelhante ao Beam Search, porém se fosse possível, seria equivalente a utilizar o Beam Search com o valor infinito para o *Beam Width*. Para o Branch and Bound, apesar de ter sido implementado, se manteve apenas a implementação baseada em busca primeiro em profundidade, pois para o baseado em busca primeiro em largura, o consumo de memória cresce a medida que as ramificações crescem, e para os casos médios a ruins, isso pode ser até fatorial. Utilizando uma base de testes de 40 Jobs e uma implementação do Branch and Bound baseado em busca primeiro em largura a execução consumiu em menos de um minuto 5 GigaBytes de memória.

Uma observação importante a se fazer é ter uma boa implementação de uma fila de prioridade. Esta fila seria útil para o Beam Search e principalmente para o Branch and Bound. Neste trabalho, foi utilizado uma implementação ineficiente para a fila de prioridade, uma lista encadeada ordenada.

Durante a execução da base de testes, se mostrou bastante necessário uma boa fila de prioridade, mas, mesmo se o problema cair em um caso médio à ruim, uma boa fila de prioridade não terá grande diferença, pois sua complexidade pode ser fatorial para o pior caso.

5.2 Resultados

Infelizmente, várias base de testes caíram em casos ruíns, onde a solução do Beam Search não foi muito boa ou a entrada de dados era muito ruim, pois pode haver casos em que uma grande quantidade de ramificações pode ser gerada. As bases de testes foram executadas de 3 em 3, pois a máquina em que foi executado o algoritmo possue 4 núcleos, e essas 3 bases ocupariam 3 núcleos. As bases de 60 Jobs e de 100 Jobs por exemplo, executaram por mais de 12 horas, e não concluíram o processamento até a entrega deste relatório. A base de 80 Job executou por mais de 5 horas e também não concluiu o processamento.

O resultado obtido pode ser visto na tabela 4, onde faz um comparativo do valor encontrado pela heurística e o valor encontrado pelo Branch and Bound. Podemos observar também, que em alguns casos a heurística Beam Search pode encontrar a solução exata do problema.

Tabela 4: Resultado Branch and Bound

# Jobs	Solução Beam Search	Solução Branch and Bound	tempo (s)
10	2116	2116	0.000
20	5407	5159	0.069
30	14688	14688	0.006
40	18222	17786	0.459
50	23541	22926	3.434

6 Referências Bibliográficas

1. http://www.ic.unicamp.br/~zanoni/mc102/2013-1s/aulas/aula22.pdf
2. https://pt.wikipedia.org/wiki/Branch_and_bound
3. https://en.wikipedia.org/wiki/Beam_search
4. https://en.wikibooks.org/wiki/Artificial_Intelligence/Search/Heuristic_search/Beam_search
5. http://www.win.tue.nl/~awijs/articles/beam.pdf
6. https://www.aaai.org/Papers/AAAI/1998/AAAI98-060.pdf