Universidade Federal do Espírito Santo

André Barreto e Igor Ventorim

Escalonamento de *Jobs*

Trabalho 3 de Estrutura de Dados II

Vitória
2015

1 Introdução

Este trabalho tem como objetivo solucionar o problema de sequenciamento de *jobs* ou processos visando obter a melhor sequência possível. *Jobs* são considerados tarefas a serem realizadas na qual cada tarefa possui um tempo para realizá-la, um tempo limite no qual ela precisa ser concluída e uma penalidade final caso o *job* não seja realizado dentro do tempo limite.

Para a realização desse escalonamento de foram utilizados dois algoritmos onde ambos tem o objetivo de obter um menor custo, porém com abordagens de solução diferentes e, consequentemente, resultados distintos. Com isto, deve ser analisado qual a prioridade desejada para poder ser tomada a escolha de qual algoritmo ser adotado.

De modo geral, podemos observar que o escalonamento de *jobs* pode ter diversas aplicações, desde selecionador de processos de um CPU [1] até encontrar melhores caminhos para, por exemplo, um problema de Caixeiro Viajante.

2 Algoritmos considerados

Para a solução do problema de escalonamento de *jobs*, foram utilizados dois algoritmos: *Beam Search* e *Branch and Bound*.

2.1 *Beam Search*

Em Ciência da Computação, *beam search* é um algoritmo de busca heurístico no qual explora um grafo, expandindo os W nós mais promissores de um nível de uma árvore de decisões, onde eles são guardados na memória para serem visitados e expandidos, enquanto os demais nós são descartados permanentemente [2]. Sendo ele um método para resolver problemas de otimização combinatória muito eficaz, devido a uma grande parte dos nós serem descartados, isto é, somente alguns poucos nós são selecionados para serem analisados, o tempo de execução do método é polinomial com relação ao tamanho do problema, e garantindo boas soluções devido aos W nós mais promissores selecionados.

Em resumo pode ser dizer que o algoritmo *beam search* é uma técnica de busca em árvores, na qual em cada nível da árvore é analisado um número fixo de nós e, por consequência um número fixo de possíveis soluções, lembrando que quanto maior for o seu W, menos nós serão cortados, e consequentemente mais análises serão feitas, o que levará a mais tempo de processamento. O número de nós analisados em cada nível é chamado de

largura de busca e é denotado de W. Caso seja procurado a melhor solução possível, não se é recomendado o algoritmo do tipo *beam search*, pois não fornece garantia de que a solução encontrada é a melhor.

2.2 *Branch and Bound*

Branch and bound é um algoritmo para encontrar soluções ótimas para vários problemas de otimização [3], especialmente em otimização combinatório como, por exemplo, o problema do Caixeiro Viajante. O algoritmo *branch and bound* consiste em uma enumeração sistemática de todos os candidatos em uma árvore de decisões e é passado uma solução aleatória para ele com limite superior e inferior, o algoritmo explora ramos desta árvore, que representam subconjuntos do conjunto solução.

Antes de enumerar as possíveis soluções geradas por um nó, os nós filhos são verificados em relação ao limite superior e inferior estimados até o momento pelo algoritmo, e é descartado caso não possa produzir uma solução melhor do que o melhor encontrado até o momento. Desta forma podemos observar que quanto melhor for a solução inicial passada como parâmetro mais cortes os algoritmo irá realizar, conseguindo encontrar a solução exata em um tempo aceitável.

No caso do problema do caixeiro viajante, caso seja utilizado o algoritmo, dependendo do caminho passado como parâmetro para ele, resultará em um resultado imediato e não fatorial. Entretanto dependendo da entrada passada para este algoritmo ele pode chegar a tempo fatorial, devido a não realizar poucos ou nenhum corte.

3 Implementação

Nesta seção serão apresentadas as implementações dos algoritmos descritos, estruturas utilizadas e principais funções em linguagem C.

3.1 Estruturas utilizadas

Estas são as principais estruturas utilizadas em ambos os algoritmos que auxiliaram na solução do problema.

3.1.1 *Job*

A estrutura *Job* é responsável por armazenar as informações de entrada para facilitar a manipulação e recuperação destes dados básicos.

```c
typedef struct job {
    int id;             // Identificador do Job
    int tempo;          // Tempo de processamento
    int deadline;       // Tempo limite sem penalidade
    int penalidade;     // Penalidade ao exceder o deadline
} Job;
```

Listing 1: Estrutura *Job*

3.1.2 *Path*

Path é uma *struct* que representa um "caminho" realizado até determinado momento. Por exemplo, caso tenham sido realizados os processos 0, 4 e 7, nesta sequência, uma estrutura *path* respectiva teria armazenado: o custo deste caminho; o pior caso possível em questão de penalidade (no caso do *beam search* será sempre 0); o tempo gasto desta sequência, uma lista de inteiros: 0, 4, 7; e o último *job* da sequência.

```c
typedef struct path {
    int pathCost;       // Custo ou Lower Bound
    int ub;             // Upper Bound
    int tempoGasto;
    typeList *caminho;
    Job *job;
} Path;
```

Listing 2: Estrutura *Path*

3.1.3 *Skew Heap*

Para simular a árvore de possibilidades de ambos algoritmos, utilizamos a estrutura *Skew Heap*. Em cada nível, são computados os possíveis próximos *jobs* a serem realizados, criam-se *paths* de acordo e estes são armazenados em uma *skew heap*, que permite rapidamente recuperar o melhor caminho a se seguir [4], onde o "melhor" é computado de forma diferente dependendo de qual dos algoritmos está sendo utilizado.

```c
typedef struct skewHeap
{
    Path *path;             // Path armazenado
    struct skewHeap *lc;    // LeftChild
    struct skewHeap *rc;    // RightChild
} SkewHeap;
```

Listing 3: Estrutura *SkewHeap*

3.2 *Beam Search*

O algoritmo *beam search* tem como função principal *beamSearch* que retorna uma estrutura *Path*, onde existe um caminho de solução boa. Após a declaração de variáveis utilizadas, na linha 8 criamos o primeiro caso, gerando todas as possibilidades possíveis no primeiro momento, e são inseridas dentro da *skew heap*.

Na linha 19, executaremos o *loop* do algoritmo N-1 vezes, que é o nível da arvore de possibilidades. Percorreremos um laço enquanto não removermos os W melhores caminhos da *skew heap* ou enquanto a skew heap não estiver vazia, onde em cada rodada retiraremos o menor *Path* de dentro da *skew heap* e colocaremos todas as possibilidades que estes caminhos geraram mesclados dentro de uma *skew heap*. Após N-1 iterações, teremos uma *skew heap* com os W melhores custos encontrados de forma gulosa até o momento. Na linha 24 removeremos o melhor caminho dentro destes W, no qual será o caminho bom encontrado pelo *Beam Search*.

Desta forma será retornado esta estrutura *Path*, com um caminho bom, que pode até mesmo ser encontrado o caminho ótimo, tudo dependendo de sua entrada e do parâmetro W fornecido para a função.

3.2.1 Principais funções

```
1  Path *beamSearch(Job **jobs, const int N, const int W)
2  {
3      int i, j;
4      SkewHeap *sh = NULL, *shAtual;
5      Path *path, *minPath;
6
7      // Primeiro caso
8      path = criaPath(NULL, 0, 0, 0, createList());
9      shAtual = mergeSkewHeap(NULL, geraPossibilidades(jobs, path,
       N));
10     sh = shAtual;
11
12     // Demais casos
13     for(i = 1; i < N; i++) {
14         shAtual = NULL;
15         j = 0;
16         while(j < W && sh != NULL) {
17             path = removeMin(&sh);
18             shAtual = mergeSkewHeap(shAtual, geraPossibilidades(
       jobs, path, N));
19             j++;
20         }
21         freeSkewHeap(sh);
22         sh = shAtual;
23     }
```

```
24      minPath = removeMin(&sh);
25      freeSkewHeap(sh);
26      return(minPath);
27  }
```

Listing 4: Função beamSearch

```
1  SkewHeap *geraPossibilidades(Job** jobs, Path *p, const int N)
2  {
3      int i, custo, tempo;
4      SkewHeap *sh = NULL;
5      Path *newPath;
6      typeList *newList;
7
8      for(i = 0; i < N; i++) {
9          if(!isElementList(p->caminho, i)) {
10             custo = p->pathCost + calculaLB(jobs,p,N,i);
11             tempo = p->tempoGasto + jobs[i]->tempo;
12             newList = dupList(p->caminho);
13             insertList(newList, i);
14             newPath = criaPath(jobs[i], custo, 0, tempo, newList
       );
15             sh = mergeSkewHeap(sh, criaSkewHeap(newPath));
16         }
17     }
18     freePath(p);
19     return sh;
20 }
```

Listing 5: Função geraPossibilidades

3.3 *Branch and Bound*

O algoritmo *branch and bound*, tem como função principal a *branch-Bound* que retorna uma estrutura *Path*, onde temos o caminho de solução ótima.

Após a declaração de variáveis utilizadas, na linha 7 executamos o algoritmo *beamSearch*, e guardamos o caminho que este nos fornece como o menor caminho até o momento e o menor custo. Também calculamos a maior multa que pode ser paga no início e a menor multa que se pode ter desde o começo. Após armazenado os dados iniciais necessários, encontraremos as primeiras possibilidades que podem ser geradas, iniciando um *Path* vazio, e o passando para a função *geraPossibilidades*, que retorna uma *skew heap* com todas as possibilidades de sequência a partir deste primeiro momento com seus devidos *lower bound*, *upper bound* e caminho atualizados.

Na linha 19, executaremos o laço do algoritmo N-1 vezes, que é o nível da árvore de possibilidades, em que se retira o caminho de menor custo. Após isso será analisado este caminho, verificando se o *lower bound* do *Path* atual é menor que o menor caminho que temos armazenado, caso não for,

podemos descartar, porque ele nunca diminuirá, caso seja verificamos se seu *upper bound* é igual seu *lower bound*. Caso seja, todas as possibilidades terão o mesmo custo, então podemos preencher o caminho, e substituiremos o menor caminho por este. Caso o *upper bound* seja diferente do *lower bound* então colocaremos na *skew heap* todas as possibilidades que *geraPossibilidades* pode gerar a partir daquele caminho e removemos o melhor deles e fazemos o mesmo processo.

Se no final dos *loops* não for encontrado nenhum outro melhor caminho então será retornado o caminho exato encontrado pelo *beam search*, caso contrário será encontrado o caminho exato e o retornará.

3.3.1 Principal função

```
1  Path *branchBound(Job **jobs, const int N, const int W)
2  {
3      int i, j, minCost, ubMax,lbMin;
4      SkewHeap *sh = NULL, *shAtual;
5      Path *path, *minPath, *menorCaminhoBS;
6
7      menorCaminhoBS = beamSearch(jobs, N, W);
8      minPath = menorCaminhoBS;
9      minCost = menorCaminhoBS->pathCost;
10     ubMax = penalidadeTotal(jobs, N);
11     lbMin = penalidadeMinima(jobs, N);
12
13     // Primeiro caso
14     path = criaPath(NULL, lbMin, ubMax, 0, createList());
15     shAtual = geraPossibilidadesBB(jobs, path, N);
16     sh = shAtual;
17
18     // Demais casos
19     for(i = 1; i < N; i++) {
20         shAtual = NULL;
21         path = removeMin(&sh); // Caminho a ser avaliado
22
23     while(path != NULL && path->pathCost < minCost) {
24             if(path->pathCost == path->ub) {
25                 for(j = 0; j < N; j++) {
26                     // Preenchendo o caminho
27                     if(!isElementList(path->caminho, j))
28             insertList(path->caminho, j);
29                 }
30                 //Atualizando menor caminho
31                 freePath(minPath);
32                 minPath = path;
33                 minCost = path->ub;
34             }
35             else
36                 shAtual = mergeSkewHeap(shAtual,
    geraPossibilidadesBB(jobs, path, N));
```

```
37
38                  path = removeMin(&sh);
39              }
40              if(path != NULL) freePath(path);
41              freeSkewHeap(sh);
42              sh = shAtual;
43          }
44          return(minPath);
45      }
```

Listing 6: Função branchBound

4 Análise

Com o programa funcional, podemos verificar os aspectos técnicos da solução obtida. Para isto, foram realizados estudos e uma série de testes presentes nesta seção.

4.1 Estudo sobre o *beam width*

O *beam width* é o parâmetro *W* como já referenciado neste documento em seções anteriores. É fundamental um estudo sobre este valor para a resolução do problema, pois implica diretamente na resposta final do *beam search*. A saída do *beam search* é redirecionada como entrada para o *branch and bound*, afetando diretamente em seu tempo de execução. Por isto foram feitos diversos testes em relação ao W para que fosse obtido o melhor custo benefício (Tempo X Qualidade da resposta) para alcançar as soluções esperadas.

Em primeiro momento o algoritmo *beam search* realizava um cálculo de multa verificando somente se o tempo gasto era menor do que o *deadline* do *job* a ser observado. Caso não fosse era acrescentado a multa do *job*, no custo do caminho, onde o algoritmo selecionava os W melhores custos naquele momento. Haviam vários casos de empate de *lower bound* em estágios iniciais, sucedendo em formas diferentes em que a *skew heap* era montada de acordo com o tamanho do W.

Devido à situação descrita, o aumento de W em vários casos não significava alcançar resultados melhores, pois para cada W tínhamos uma *skew heap* diferente. Desta forma o *beam width* era um parâmetro inconsistente, podendo gerar resultados melhores ou piores, não sendo uma regra, quanto maior o W melhor o resultado, como é o esperado.

No entanto, uma solução observada também devido a retirar esse enorme número de empates de custos a serem pagos, foi realizando um custo mais específico, não sendo verificado somente o *job* observado, mas sim todos aqueles ao qual ultrapassaram o seu deadline com o tempo já gasto,

desta forma eliminamos o caso de vários empates dentro de nossa *skew heap*, sendo assim um critério certo de que quanto maior o nosso W melhor será o resultado de nosso *beam search*, porém com isso foi pago o preço de verificar se os *jobs* que ainda não foram visitados, já pagaram multa naquele instante. Porém devido a mais cálculos afetou diretamente ao tempo gasto quanto maior for o W.

Uma solução encontrada foi realizando um custo mais específico, não sendo verificado somente o *job* observado, mas sim todos aqueles ao qual ultrapassaram o seu *deadline* com o tempo já gasto. Assim eliminando o caso de vários empates dentro da *skew heap*, resultando em um critério certo de que quanto maior for W, melhor será o resultado do *beam search*.

Veja na tabela 1 as alterações devidas à variação do parâmetro *W*. Esta análise foi feita executando o arquivo de entrada *in100* (ver Seção 4.2) variando somente o valor de *W*.

W	**Resultado**	**Tempo de execução**
2	522	0.168s
10	506	0.592s
50	506	2.823s
100	506	6.086s
200	506	13.094s

Tabela 1: Parâmetro W

Após os relatos, podemos verificar que quanto maior o W, mais precisa será a solução do *beam search* comparado com o caminho exato, porém não necessariamente o resultado melhora, como pode ser visto na tabela. Com o aumento do parâmetro W, temos garantia que a solução está melhor ou igual. Além da possível melhoria da solução do algoritmo, temos um aumento sempre crescente de tempo de execução, de tal forma que, em certo momento, não vale a pena utilizar um W grande, pois o tempo gasto cresce muito mas a solução melhora muito pouco ou permanece estática.

Por meio dos testes, foi decidido o valor *30* para *W*, pois é um valor que garante uma boa solução se comparado com valores inferiores e ainda tem um tempo de execução quase instantâneo.

É importante utilizar um valor de W adequado pois quanto mais próxima da ótima for a solução obtida, mais cortes o *branch and bound* poderá realizar e menos cálculos serão feitos para encontrar a solução ótima. O que reduz significantemente o tempo necessário pelo algoritmo exato.

4.2 Arquivos de testes

Para verificar a integridade dos algoritmos implementados, foram realizados diversos testes com entradas de tamanhos e formatos variados.

Os arquivos de teste foram criados automaticamente por um programa gerador, cujo código é o seguinte:

```
1  void createFileJobs(int qtdJobs)
2  {
3    int tempo,penalidade,multa;
4    FILE *arq = fopen("jobs.txt","w");
5
6    int i;
7    srand( (unsigned)time(NULL) );
8
9    for(i = 0; i < qtdJobs; i++)
10   {
11     tempo = rand()%qtdJobs+1;
12     penalidade = tempo + rand()%qtdJobs + 1;
13     multa = rand()%10+1;
14     fprintf(arq,"%d %d %d\n",tempo,penalidade,multa);
15   }
16
17   fclose(arq);
18 }
```

Listing 7: Gerador de Jobs

As entradas utilizadas nos testes da seção 4.3 estão caracterizadas na tabela 2. Devido ao tamanho dos arquivos, foram mostrados apenas os somatórios de cada um dos parâmetros. Ou seja, a soma de todos os tempos de execução, todos os *deadlines* e a soma das penalidades.

Os arquivos são identificados como *in* seguido pela *quantidade de jobs*. Por exemplo, *in50* seria um arquivo de entrada com 50 *jobs* a serem processados.

Arquivo	**Tempo**	***Deadline***	**Penalidade**
in20	168	366	101
in35	633	1202	184
in40	776	1527	227
in100	5130	10170	580
in200	20420	40219	1072
in500	122983	251422	2700

Tabela 2: Arquivos de entrada

4.3 Comparação entre os algoritmos

Veja na tabela 3 os resultados da execução dos testes de entradas desiguais apresentadas na seção 4.2. Cada teste foi realizado utilizando o *beam width* $W = 30$, como definido e explicado na seção 4.1.

Algoritmo	**Arquivo**	**Resultado**	**Tempo de execução**
Beam Search	in20	52	0.014s
	in35	147	0.066s
	in40	192	0.099s
	in100	506	1.679s
	in200	993	27.661s
	in500	2526	1139.692s
Branch and Bound	in20	52	0.0850s
	in35	132	11.191s
	in40	171	49.529s
	in50	230	29.317s
	in51	244	27.459s

Tabela 3: Resultados e comparação

Com os dados apresentados, podemos claramente observar a diferença entre os algoritmos considerados.

O *Beam Search* é significantemente mais rápido que o outro. Isto é esperado, uma vez que é caracterizado como uma heurística, ou seja, tem como objetivo se aproximar da solução exata, perdendo a precisão mas ganhando em tempo de execução.

O *Branch and Bound*, por sua vez, garante a melhor resposta possível, deixando em segundo plano a velocidade de resposta. Mesmo assim, este algoritmo é consideravelmente rápido se comparado com um procedimento de permutação simples, pois utiliza o resultado do *beam search* para prever e eliminar sequências ruins que existem durante o decorrer do algoritmo.

Uma observação importante é que a velocidade em que o algoritmo *branch and bound* encontra a solução depende muito da entrada. Como pode ser visto pelos testes *in40*, *in50* e *in51*, que, nesta ordem, reduziram o tempo de processamento. Assertivamente, nestes testes com maior número de *jobs*, ocorreram mais situações em que pode ser eliminado diversas possibilidades, ganhando assim em tempo de execução.

5 Conclusão

Ambos os algoritmos implementados são eficientes dependendo da aplicação. Caso o objetivo não requira uma exatidão e/ou a quantidade de dados a serem processadas é vasta, seria mais apropriado a utilização do *Beam Search*. Por outro lado, se for necessário uma solução ótima para o problema, então deve-se preparar e executar o *Branch and Bound*.

Referências

[1] D. Novato, *Sistemas Operacionais - O que é Escalonamento de Processos?*, May 2014. 1

[2] Wikipedia, *Beam search*, July 2015. 1

[3] Wikipedia, *Branch and bound*, Sept. 2015. 2

[4] D. D. Sleator, *Self Adjusting Heaps*, Feb. 1986. 3