UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA
CIÊNCIA DA COMPUTAÇÃO
ESTRUTURAS DE DADOS II

ESCALONAMENTO DE JOBS

Alunos: Daniel Favoreto e Rafael Athaydes

Professora: Mariella Berger

Vitória
10 de Novembro de 2015

Introdução

Este trabalho tem como finalidade a implementação de sequenciamento de jobs visando o menor custo, baseado nos algoritmos Beam Search (BS) e Branch and Bound (BB).

O programa lê argumentos da linha de comando,sendo um inteiro n especificando o número de jobs a sequenciar e em seguida a especificação do algoritmo, BB caso Branch and Bound ou BS caso Beam Search.

Em seguida o usuário deve digitar linha por linha valores inteiros separados por espaços para as seguintes colunas: Tempo de processamento, Deadline e Multa. Por final, o programa deve gerar na linha de comando o sequenciamento de menor custo de acordo com o algoritmo especificado. Mais especificamente, o programa imprime na tela o menor custo seguido de ":"e o sequenciamento obtido.

De antemão podemos dizer que o algoritmo Branch and Bound foi mais eficiente no sentido de resultados mais refinados, porém, obtivemos um tempo de execução muito maior em relaçao ao Beam search.

Implementação

O trabalho é implementado na linguagem C e compilado no GCC. A figura 1 representa as estruturas de dados usadas:

```
typedef struct sequencia{
        int *caminho, *visitados, tempo, multa,nVisitados,lb,ub;
        struct sequencia *prox;
}Sequencia;
```

Figura 1: Estruturas de dados utilizadas

As estruturas utilizadas assim como a assinatura das funções estão no arquivo **Functions-Trab.h**.

Para a simplificação do uso convencionou-se criar o tipo string. A estrutura `Sequencia` utiliza um ponteiro para o caminho que está sendo percorrido (`*caminho`),ponteiro para a próxima sequência (`*prox`), ponteiro para os nós visitados (jobs) (`*visitados`), campo para o tempo (`tempo`), campo para a multa do job (`multa`), campo para o número de nós (jobs) visitados (`nVisitados`) assim como campos para (`lb`) (lower bound) e (`ub`) (uper bound).

No arquivo **trab3.c** está a função **main** principal que recebe os parâmetros de entrada e faz a verificação se o número de argumentos corresponde com a especificação. Após isso a função verifica qual foi o algoritmo escolhido podendo ser beam search (bs) ou branch and bound (bb),assim como faz a leitura do número de jobs e chama as funções **bs** ou **bb** respectivamente.

Ao final, a função **liberarPonteiros** faz a desalocação de memória para os jobs criados dando os respectivos "frees".

Quando é especificada a opção por Branch and Bound, o algoritmo executa a função bs com o segundo argumento setado para 0 indicando que não é necessário imprimir o caminho gerado por bs, mas apenas utilizar como entrada para o algoritmo de Branch and Bound executar. Se o segundo argumento for 1, a função bs imprime o caminho gerado com o menor custo.

Funções utilizadas:

Beam Search: As funções principais para este algoritmo estão representadas na figura abaixo.

```
void bs(int n, int imprimir);
void addMelhores(Sequencia s[], Sequencia nova,int w);
Sequencia createSeq(Sequencia s, int pos, int n,int multa, int tempo);
void liberarPonteiros(int n);
```

A função **createSeq** tem o papel de retornar uma nova sequência de caminho a partir da atual,inserindo um novo job e atualizando os jobs visitados, número de visitados, multa e tempo. A função tem como argumentos a sequência atual, o job a ser adicionado, número de jobs da entrada, tempo e multa do job.

A função **addMelhores** adiciona uma nova melhor sequência na matriz de melhores sequências, recebendo a matriz de melhores sequências, uma sequência a ser comparada e a variável w.

A função **bs** recebe como argumento o número de jobs e a opção de impressão. Ela é responsável

inicialmente por fazer a leitura das 3 colunas job a job e alocar e armazenar em uma matriz jobs n x 3 os valores de tempo, deadLine e multa.

Esta função também tem por principal papel alocar um vetor da sequência de n caminhos e uma matriz n x w dos melhores caminhos que encontrar ao longo do algoritmo. Além disso há um loop **for** iniciado na linha 62 que calculará a multa e atualizará os tempos a cada job visitado e chamará a função **createSeq** para criar uma sequência nova a partir de caminhos[i]. Em seguida usando a função **addMelhores** vai adicionando essa sequência à matriz de melhores sequências.

Ao final, verifica se o argumento de imprimir está setado para 1, o que significa que não será utilizada na função bb e pode imprimir o resultado usando a matriz melhores. Após, dá free nas estruturas alocadas e retorna à função **main**.

Caso seja imprimir = 0, aloca uma matriz menorsequencia para armazenar a sequência encontrada, assim como cria uma variável para armazenar o custo encontrado. Depois disso, desaloca a memória alocada para as estruturas criadas e como a matriz menorsequencia e valorMenorSeq são variáveis globais, a função pode retornar à main, para então menorsequencia e valorMenorSeq serem utilizadas pela função **bb**.

Branch and Bound: As funções principais para este algoritmo estão representadas na figura abaixo.

```
void addMelhoresBb(int posM,Sequencia s, int pos, int n, int multa, int tempo, int lb,int ub);
void bb(int n);
```

A função **addMelhoresBb** a cada iteração adiciona os melhores jobs (filhos) à matriz melhores (que por sua vez é global) e retorna a nova matriz melhores.

A função **bb** aloca e inicializa o vetor de caminhos e em seguida vai escolhendo os filhos que possuem o lower Bound menor em relação ao encontrado pelo bs. Ou seja, esses filhos então são adicionados na matriz dos melhores (n x w) e a cada job (filho) soma o lower Bound e compara em relação ao bs. Para tanto, o algoritmo usa a função `addMelhoresBb`.

De maneira que ao final, no vetor p tem-se o melhor caminho e é impresso na tela o resultado com o custo e menor caminho.

Análise

Beam width:

Número jobs (n)	W <= n Resultado	W > n Resultado
3	6	6
6	51	42
10	151	126
15	206	190
20	255	242
25	377	366
30	458	447
50	793	771
75	1283	1247
100	2431	2330
150	3666	3590
200	4833	4720

Acima podemos ver uma tabela com vários tamanhos de entradas diferentes (de 3 a 200 jobs) para o algoritmo Beam search. Assim como há duas colunas que especificam o tamanho do Beam width escolhido, variando de 2 até 3*n.

O tamanho de w foi testado em todos os casos como sendo: para w <= n, w = 2 e w = n; para w > n, w = 2*n. Entretanto, os dados fornecidos na coluna w <= n são em sua grande maioria executados com w = 2, enquanto na coluna w > n são todos w = 2 * n, onde n é o número de jobs.

Porém, é importante destacar que a escolha do w = 2 para a coluna w <= n se deve ao fato de que houve uma diferença pequena em alguns casos utilizando w = n e w = 2 * n, e mostrar uma tabela com diferenças tão pequenas de custos seria injusta frente ao impacto que a escolha do beam width propicia no resultado.

É importante destacar que quanto maior o valor de w, obteve-se uma piora de desempenho por parte do bs.

Beam search:

O algoritmo de Beam search implementado obteve uma eficiência em questão de rapidez, porém, pecou no sentido de otimização do custo.

Observou-se que o resultado estava estritamente ligado à entrada e ao tamanho do beam width que por sinal setamos o default para 3 caso n seja menor que 10 e 7 caso n seja maior que 10.

Branch and Bound: O algoritmo de Branch and Bound obteve um refinamento maior em relação ao Beam Search como pode-se observar na tabela acima, porém um desempenho muito pior em questão de tempo.

Importante observar que utilizamos w = 2 para Beam search.

Número jobs (n)	BS Resultado	BB Resultado
10	151	151
15	206	192
20	255	245
25	377	358
30	458	432
40	556	540
50	793	739
75	1283	1226
100	2431	2323
150	3666	3547

Conclusão

A dupla pode concluir que este trabalho nos deu um entendimento melhor a respeito desses algoritmos e entedemos sua importância no meio computacional. Foi necessário um certo nível de abstração para podermos entender os conceitos de beam width e lower bound assim como uper bound.

Tivemos alguns problemas com core dump mas pudemos resolver ao longo da implementação.

Como demonstramos nas análises dos algoritmos

Bibliografia

Sites da web:

stackoverflow.com

en.wikipedia.org

Livros:

ZIVIANI, N. Projeto de Algoritmos, Cengage Learning.