TRABALHO PRÁTICO 2

Arvores Binárias de Busca Ótimas

Wagner Vinícius Guimarães Lopes

Departamento de Ciência da Computacão – Universidade Federal de Minas Gerais (UFMG)
wagnerlopes.bh@gmail.com
wagnerl@dcc.ufmg.br

***Resumo**: Este documento descreve um sistema que contém três estratégias de construção de arvores binárias de pesquisa. A primeira é uma árvore balanceada que é construída ignorando-se o peso de cada um dos termos de um dicionário de entrada. O segundo algoritmo utiliza uma estratégia de força bruta para construir uma árvore de pesquisa ótima. E a terceira estratégia implementa uma heurística que encontra uma solução aproximada para o problema da arvore de pesquisa ótima.*

1 INTRODUÇÃO

Neste documento esta descrito a implementação de um sistema que, implementa três estratégias de construção de árvores binárias de pesquisa:

Estratégia para construção de uma árvore balanceada

Estratégia para construção de uma árvore de pesquisa ótima utilizando força bruta

Estratégia para construção de uma árvore de pesquisa quase ótima utilizando uma heurística

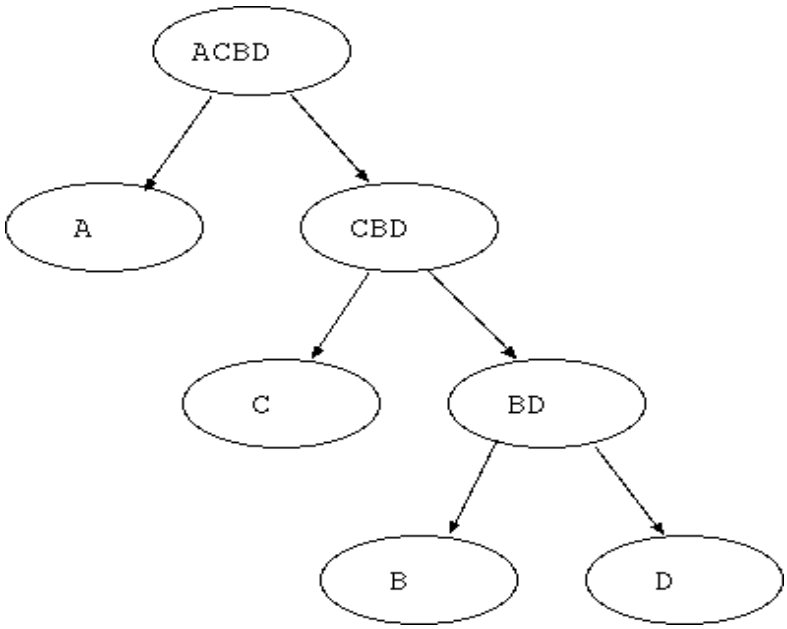
Como descrito na especificação do trabalho, o termo deve ser utilizado como chave para o vertice, ou seja, nenhuma outra informação deve ser armazenada nos nós da arvore além do termo lido do arquivo. Além disso, cada termo possui um peso que é igual à quantidade de vezes que este aparece no arquivo de entrada.

2 ANALISE DO PROBLEMA

1. Objetivo:

O objetivo desse trabalho é realizar a implementação de três árvores binárias de pesquisa, como descrito na seção anterior deste documento.

2. Formato da árvore



Na imágem acima, temos uma árvore binária não balançeada, semelhante às que vão ser implementadas. No exemplo, cada nó possui somente um termo que representa a chave e dois apontadores, um para o filho da direita e outro para o filho da esquerda.

3. Arvore binária balanceada

Esse tipo de árvore possui a sub-árvore da esquerda com, no máximo, um nível de diferença de altura da sub-árvore da direita.

4. Árvore de pesquisa ótima

Esse tipo de árvore possui custo total mínimo, considerando-se os pesos dos termos, ou seja, é necessário organizar a árvore de uma forma que os termos mais pesquisados fiquem mais próximos da raiz. Dessa forma, o custo médio de acesso aos termos, será mínimo, considerando a probabilidade de acesso a cada termo.

3 SOLUÇÃO PROPOSTA

1 Preparação inicial

1. O arquivo é lido para se identificar a quantidade de linhas existentes
2. O vetor de termos é alocado dinâmicamente
3. O arquivo é lido novamente e o vetor de termos é preenchido
4. O vetor de termos é ordenado alfabeticamente
5. É gerado um segundo vetor, chamado vetorCondensado, onde cada termo aparece uma única vez acompanhado de seu respectivo peso e de uma variável que armazenará a profundidade do termo na árvore.
    1. Dado que o vetor já está ordenado, todos os termos repetidos estão em sequência. Sendo assim, foi criado um loop para descobrir a quantidade de vezes que o mesmo termo aparece e o armazena no segundo vetor junto com seu peso (quantidade de ocorrências).

2 Algoritmo da árvore balanceada:

Uma maneira de se criar uma árvore binária balanceada é inserir os termos em uma ordem correta tal que a árvore já seja construída dessa forma, sem a necessidade de alterações posteriores. O problema maior é encontrar essa ordem correta de inserção que possibilite a geração dessa árvore.

Encontrei uma solução que utiliza uma estratégia de particionamento de vetores parecida com a utilizada no algoritmo do quicksort que funciona da seguinte maneira:

1. O vetor deve ser ordenado
2. O ítem localizado no centro do vetor deve ser inserido como raiz da árvore. Dessa forma, garante-se que todos os termos à esquerda no vetor sejam inseridos no lado esquerdo da árvore e os termos do lado direito do vetor sejam inseridos no lado direito da árvore.
3. A função é chamada, recursivamente, passando o subvetor à esquerda do meio.
4. A função é chamada, recursivamente, passando o subvetor à direita do meio.

Dessa forma, a árvore será construída de forma balanceada.

3 Algorítmo da árvore de pesquisa ótima utilizando força bruta

Utilizando uma algorítmo de força bruta, uma forma de se encontrar a árvore ótima é gerar todas as possibilidades de árvores para os dados de entrada e depois compara-las. Sendo assim, decidi implementar o seguinte algorítmo:

1. Para cada N! possibilidade de ordenação do vetor fazer
    1. Gerar a árvore inserindo os termos na ordem em que eles aparecem no vetor
    2. Calcular o custo total da árvore
    3. Se o custo da árvore for menor que o custo da árvore armazenada na estrutura, fazer:

        Armazenar a árvore junto com seu custo dentro de uma estrutura da forma:

        EstruturaArvore {

Vertice* raiz;

int custo;

}

Fim Se

2. Retorna ao passo 1;

4 Heurística da árvore de pesquisa quase ótima

A heurística tem o objetido de encontrar uma solução aproximada para o problema, já que a o tempo de processamento para se encontrar uma solução ótima para arquivos de entradas maiores é muito grande. Sendo assim, optei pelo seguinte algoritmo solução:

1. O vetor lido do arquivo é condensado, isto é, o vetor original é transformado em um vetor que armazenará as informações dentro de uma estrutura contendo o termo e a quantidade de vezes que ele aparece no arquivo de entrada. Dessa forma, cada termo aparecerá uma única vez no vetor acompanhado do seu respectivo peso.
2. O vetor é ordenado de forma decrescente por peso
3. Os termos de maior peso são inseridos primeiro na árvore. Dessa forma, os termos de maior peso tendem a ocupar posições superiores na árvore facilitando seu acesso.

No final desse processamento, uma árvore quase ótima será gerada

5 Calculo dos custos de cada termo:

1. Para cada termo no vetorCondensado fazer:
    1. Descobrir custo de acesso ao termo fazendo busca na árvore
    2. Armazenar peso junto com seu respectivo termo no vetorCondensado

6 Divisão dos módulos

Foram criados três modulos para o problema:

main: Módulo de controle

arvore: Implementa a TAD árvore

util: Armazena algumas funções úteis como a função de ordenação e as funções de acesso à disco

7 Lista de funções do programa:

7.1 arvore.criaArvore

7.2 arvore.calculaCustoDaArvore

7.3 arvore.calculaProfundidade

7.4 arvore.calculaCustoTermo

7.5 arvore.calculaProfundidade

7.6 arvore.insere

7.7 arvore.criaArvoreBalanceada

7.8 arvore.ordenaDecrescentePorPeso

7.9 arvore.heuristicaArvorePesquisaOtima

7.10 arvore.condensaVetorOrdenado

7.11 arvore.criaArvorePesquisaOtimaForcaBruta

4 Implementação

1. **Compilação**
   1. Na pasta Debug digite:

      make
2. **Execução**
   1. Na pasta Debug digite:

      ./ArvoreBinaria -t <Tipo de algoritmo> -i <Arquivode entrada> -s <Arquivo de saida>
3. **Formado da saída**

```
Quantidade de termos:   7
Custo da arvore:  36
Altura da arvore:       3

baby beef
      1 3
ciencia idade media
      2 2
lazer
      1 3
policia
      1 1
redes e cabeamento estruturado
      1 3
sonhos
      2 2
turismo
      6 3
```

No arquivo acima, os números após os termos são, respectivamente, o peso do termo e sua profundidade na árvore.

5 ANALISE DE COMPLEXIDADE

1. Para a preparação inicial descrita em Solução Proposta temos os seguintes algoritmos e suas complexidades:

   Leitura do arquivo: O(N)

   Ordenação do arquivo (algoritmo de seleção): $O(N^2)$
2. Algoritmo da árvore balanceada:
   1. Esse algoritmo simplesmente insere os N termos na árvore a um custo de log M, sendo M o número de termos já inseridos na árvore, para cada termo. Sendo que, na média teremos N/2 termos na árvore, então temos a seguinte complexidade: N*log(N/2). Sendo assim, a função de complexidade é: $2N^2$ +N*log(N/2) e a ordem de complexidade é

$O(N^2)$.

3. Heurística da árvore de pesquisa ótima
    1. Para implementação dessa heurística, foi necessário ordenar novamente o vetor por órdem decrescente de peso utilizando o algorítmo de seleção que é $O(N^2)$. Após esse processamento, o algoritmo insere os temos na ordem em que eles aparecem no vetor a um custo N*log(N/2) igual ao algorítmo da árvore balanceada. Sendo assim, a função de complexidade é: $2N^2$ +N*log(N/2) e a ordem de complexidade é $O(N^2)$.
4. O algoritmo de força bruta testa cada uma das N! possibilidades de ordenação do vetor, sendo assim, esse algorítmo é O(N!)

6 DIFICULDADES DE IMPLEMENTAÇÃO

A maior dificuldade do projeto foi encontrar um algorítmo que gerasse todas as permutações do vetor de termos sem que haja repetição de vetores. Por esse motivo, o algoritmo de força bruta que foi especificado na seção 3 deixou de ser implementado.

No mais, a implementação dos outros 2 algoritmos foi relativamente simples, uma vez que as árvores binárias já tinham sido estudadas nas aulas de AEDs II.

7 AVALIAÇÃO EXPERIMENTAL

Foi realizada uma comparação entre o algorítmo da árvore balanceada e a heurística. Nos testes realizados com algumas massas de dados, a arvore balanceada teve um custo médio 30% maior que o custo da árvore gerada pela heurística. Vale à pena ressaltar que, no pior caso, a heurística irá gerar uma árvore totalmente desbalanceada e com um custo muito maior que o custo da árvore balanceada. Isso pode acontecer, por exemplo, se a ordem de peso coincidir com a ordem alfabética dos termos do vetor. Por outro lado, a probabilidade dessa situação ocorrer em uma aplicação real é próxima de 0, o que torna a heurística implementada viável na prática.

8 CONCLUSÃO

Durante a implementação dos algoritmos foi possível assimilar o funcionamento da árvore binária de pesquisa de uma maneira eficiente. Foi possível, também, realizar uma boa análise de performance dos algoritmos. Por exemplo, foi possível perceber que a implementação de um algoritmo de força bruta como o especificado para se encontrar a árvore ótima é totalmente inviável para situações onde se tem um número maior de termos a serem analisados. Essa impossibilidade se dá por causa da ordem de complexidade fatorial do problema. Sendo assim, encontrar uma solução utilizando força bruta para esse problema demandaria um tempo um tempo muito grande. Como possibilidade de solução, encontrei alguns algorítmos baseados em programação dinâmica que, apesar de ter uma complexidade maior que a da heurística, apresentariam resultados melhores, uma vez que a solução encontrada seria, garantidamente, ótima. Um slide com a explicação da solução por programação dinâmica pode ser encontrada aqui. Sobre a heurística, foi possível notar que ela apresentou resultados médio satisfatórios comparados com a árvore balanceada. E a árvore balanceada, por sua vez, não demonstrou ser uma estratégia aconselhável para situações onde o peso dos termos varia. Nesses casos é melhor organizar a árvore de forma que os termos de maior peso sejam acessados a um custo menor.

9 REFERÊNCIAS

Livro: Título: Projeto de algoritmos: Com implementações em Pascal e C
Autor: Ziviane, Nivio.
Publicação: São Paulo - 1993