Centro de Informática - UFPE
Thaisa Mirely (tmbs2@cin.ufpe)
February 17, 2016

Relatório
Implementação do projeto IPMT

Visão Geral

O propósito do desenvolvimento do projeto foi o desenvolvimento de uma aplicação que realizasse a indexação de um arquivo de texto e busca utilizando a indexação criada.

A aplicação foi desenvolvida utilizando os algoritmos de array de sufixo - indexação; Huffman - codificação e descodificação; e busca binária - para realização da busca de padrões em um arquivo de index.

O objetivo deste relatório é apresentar as instruções de uso da aplicação(Sessão 1), sua forma de implementação(Sessão 2) e testes realizados para medição de performance(Sessão 3). Ao final são apresentadas as referências descritas no decorrer do texto.

1. Instruções de uso.

A interface da aplicação utiliza três comandos específicos listados abaixo:

$ ipmt index textfile

O comando acima recebe um arquivo de texto, indexa-o e comprimi-o gerando um arquivo no mesmo diretório do arquivo de texto com extensão .idx. Para indexação do arquivo foi utilizado o algoritmo array de sufixo e para codificação/descodificação foi utilizado o algoritmo Huffman, como base.

- ***textfile*** - equivale a um arquivo de texto com extensão .txt. Ex.: $ ipmt index file.txt

$ ipmt search [options] pattern indexfile

O comando acima remete a busca de arquivos. O método de busca utilizado é a busca binária. O comando recebe como entrada um arquivo de padrão ou uma *strings* simples de padrão e irá realizar a busca em um arquivo de index(com extensão .idx);

- ***pattern :*** strings contendo padrões para busca.
- ***indexfile :*** arquivo para busca contexto texto e index. O arquivo deverá ser gerado previamente através do comando *$ ipmt index textfile.*
- ***options***
  - ***-p, —pattern:*** arquivo de padrão no formato de texto com extensão txt, onde em cada linha é composta de um padrão para busca. Ex.: $ ipmt search -p pattern_file.txt index_file.idx
  - ***-c, —count:*** exibe a quantidade de ocorrências encontradas ao final da exibição das linhas onde a ocorrência do padrão foi vizualizada.

$ ipmt help

Comando para exibição de texto de ajuda contendo o detalhamento do comandos necessário para execução da aplicação.

O comando pode ser inserido de duas formas:

- ***$ ipmt -h***
- ***$ ipmt —help***

2. Implementação

Visão Geral

A aplicação foi desenvolvida utilizando a linguagem C++ e estruturada utilizando o padrão de projeto *factory method* como forma de selecionar o comando adequado de acordo com o tipo de argumento de entrada. O padrão escolhido ajuda em que outros tipos de comandos possam ser adicionados sem gerar alta grau de impacto a estrutura atualmente. O mesmo ocorreu com relação as técnicas de codificação e indexação, onde há uma generalização afim de proporcionar adição sem impacto de nossos algoritmos. O projeto é composto de classes de manipulação de arquivo, algoritmos(busca, indexação e compressão) e um arquivo Makefile, para compilação do projeto via terminal.

Indexação

Para indexação foi utilizado como padrão o algoritmo de array de sufixo, presentes nos arquivos *suffix_array_algorithm.cpp* e *suffix_array_algorithm.hpp*. No decorrer do desenvolvimento foram realizados testes com duas abordagens de carregamento de arquivo:

A. Carregamento completo do arquivo de texto em memória: O arquivo de texto teria seu conteúdo armazenado em uma strings(*ManipulationFile::GetFileLines*) e seguiria para o método *SuffixArrayAlgorithm::BuildSuffixArray* onde seria realizada a leitura para criação dos indexes.
B. Carregamento do arquivo linha por linha: O arquivo de texto seria lido linha por linha e armazenado em um vetor de string(*ManipulationFile::FileRead*). A indexação do arquivo seria realizada de forma parcial.

Como abordagem escolhida para a versão atual da aplicação fui utilizada a versão A, visto que mesmo com o consumo relativo de memória, o arquivo era indexado linearmente e de forma coesa, pois os testes realizados utilizando a opção B geraram indexes incoerentes.

Com a finalidade de criar uma estrutura para compor o sufixo foi implementada a struct Suffix, composta pelo index e posições relativas do sufixo. Os sufixos são ordenados de acordo com o primeiro caractere, segundo, quarto e assim por diante. Foi utilizado o método *SuffixArrayAlgorithm::ComparationSuffixPair* para realizar a comparação das posições(atual e próxima) e ordenar o index.

Incialmente todos os sufixos são ordenados de acordo com o primeiro caracter de entrada, seguindo com o segundo caracter, quarto e assim por diante. Foi utilizado o algoritmo de *merge sort* nativo para ordenação com tempo . Em seguida é realizado um ranqueamento comparando o sufixo anterior e o atual.

Como ferramenta de comparação e análise de corretude do index gerado, foi utilizada a implementação de array de sufixo disponível para fins de execução em [1]

Com relação ao arquivo .idx gerado por meio da indexação, foi utilizada a abordagem de criação de um arquivo temporário, visto que foi necessário(para realização posterior da busca) a concatenação do texto original e texto de indexado, assim como também foi necessário o armazenamento no inicio do arquivo a posição limite no texto(*ManipulationFile::CreateIndexFile*).

Exemplo de arquivo .idx gerado:

```
68
As causas exatas da encefalopatia hepática ainda são desconhecidas.
38 52 37 51 43 2 16 54 19 9 33 49 0 42 18 32 48 44 25 66 14 7 12 29 4 41 3 22 63 58 17 47 65 55 62 23 20 35
56 10 24 61 34 31 40 64 45 26 21 46 60 53 59 27 36 28 50 67 1 15 8 6 57 13 30 39 5 11
```

Codificação

Huffman foi o algoritmo utilizado para codificação e descodificação. Na primeira versão da aplicação foi implementado uma forma simples, no entanto não eficiente do algoritmo LZ78. Tal versão realizava a codificação em um tempo maior que a versão atual, além de gerar um arquivo incoerente ao realizar a descodificação(abordado com mais detalhes na seção seguinte.). Como material de auxilio para construção foi utilizada as aulas de estrutura de dados de [2].

Para realização da codificação é lido como arquivo de entrada o arquivo de index(exemplo acima). A partir desse arquivo é gerada uma tabela com os caracteres, o numero de ocorrências de cada caracteres e sua representação, além de "converter" as strings do texto em uma cadeia de bits. A tabela é gerada ao ser verificado o numero de ocorrências de cada caracter no texto.

O método *HuffmanAlgorithm::BuildHuffmanTree* cria uma árvore que verifica a frequência de cada caracter original. Para cada caracter cria um nó com a representação do caracter e da frequência que ele aparece. Através da ordenação gerada na árvore será feita a criação criação do *codeword*, representação binaria do texto, de forma a que cada código seja único.

A escrita da construção da tabela foi realizada no arquivo original de entrada e para auxiliar na geração dos códigos e do arquivo final foi necessário a criação de um arquivo temporário que é removido ao final da codificação, restando como resultado do processo, o arquivo de índex inicial de forma comprimida. Ou seja é gerado um arquivo único com a tabela e os códigos(*HuffmanAlgorithm::Encode*).

Foi pensado na possibilidade de utilização de um arquivo auxiliar que poderia ser lido no momento da decodificação. Nesse caso após a codificação teríamos um arquivo com os códigos gerados e um arquivo temporário com a tabela. Mesmo essa abordagem gerando um arquivo de codificação menor, ela foi descartada por poder gerar uma complexidade, vista como desnecessária no momento de realização do comando de busca. No cenário pensado, para realização da busca, deveria além do arquivo .idx comprido, ser passado como parâmetro também, a adição do arquivo com a tabela ou em outro caso assumir que o arquivo de tabela sempre estará presente no diretório onde o programa for executado. Afim de dispersar essas dependências e alteração no comando, foi escolhida a abordagem de arquivo único.

Exemplo do arquivo gerado na codificação:

```
31
        0.285714        11
4       0.065637        1010
3       0.065637        0111
1       0.065637        0110
2       0.065637        1001
5       0.065637        1000
```

6	0.054054	0100
a	0.046332	0010
0	0.027027	00111
s	0.027027	00110
8	0.023166	00001
e	0.023166	00000
9	0.023166	00011
7	0.023166	00010
c	0.019305	101110
d	0.015444	101100
i	0.015444	010101
n	0.011583	1011111
o	0.011583	1011110
t	0.011583	1011011
	0.007722	1011010
h	0.007722	0101101
p	0.007722	0101100
x	0.003861	01010001
	0.003861	01010000
.	0.003861	01010011
A	0.003861	01010010
f	0.003861	01011101
l	0.003861	01011100
u	0.003861	01011111
	0.003861	01011110

```
0100010010110100101001000110111011100010010111110011000100011011000000101000100101011011001 0
001101110110000101100000101111110111000000010111010010010111001011110010110000101011011010101
0010110101101000000101100010111101011011010101101110001011001001010110111111011000010110011 0
0101000010111101110110000000000110101110101111010111110101101000001011100101011011000010001100
1010011101101001110001011100000111111010100111100111011001001110001001110110000111100011110111
0111111010000011100111111010011011011000001110111100111101000010111010011111100110001101001010
1101101010110001011011010011110010001111101011101000111110111111001100111010001101110000100110
1100001011101001001101000111111000011111010000111111001011111100100111110111100011100010101101
1000111111001101011100000001111011110101101110110110111000111101001001111010101011100101001110 0
1011011101010001110000000111100001101110000000101110010001011011101001110010000111101000011110
1001000110110110110100011000011101001110001000110110011111011100111110111000011110001101100 11
011
```

Descodificação

Assim como na realização da codificação a descodificação se faz por meio da leitura da tabela gerada na codificação e frequência de cada caracter. Para construção do arquivo de descompressão foi utilizado um arquivo auxiliar, pois a "reconstrução" dos caracteres se dá de forma dinâmica*(HuffmanAlgorithm::Decode)*. O arquivo de entrada é substituído no final pelo arquivo temporário que recebe a mesma nomenclatura do arquivo de entrada. Tal arquivo servirá como entrada para a execução da busca.

Busca

O mecanismos de busca utilizado para verificação de ocorrências do padrão foi por meio da busca binária[3]. O arquivo de índex, que no momento da busca deverá estar descomprimido, conterá a mesma estrutura gerada através indexação.

O método *ManipulationFile::ReadIndexFile* é responsável por realizar a leitura do do arquivo de índex e armazenar na estrutura *IndexFileProperty,* que servirá como parâmetro para o método de busca (*Search::BinarySearch*). O texto é armazenado em um vetor de strings contendo cada índice uma linha do texto. Já o índex é armazenado em um porteiro de inteiro.

A busca é estruturada lendo cada padrão e realizando a busca em todo o texto. Caso o texto seja uma estruturada em várias linhas, a aplicação deve ler cada linha e realizar a busca utilizando o ponteiro de indexes.

Através do método *SearchResult::ShowTextLinesOccurrences* exibe na tela as linhas com as ocorrências dos padrões e caso o uso da opção *-c* ou *—count,* é exibido o número de ocorrências(*SearchResult::ShowOccurrenceNumbersPatterns*).

Uma observação importante é que os arquivos gerados no decorrer da execução não foram salvos em formato binário, pois nos testes iniciais realizados a redução no tamanho do arquivo não era larga. Outra questão levantada era a necessidade de visualização do arquivo de index ou codificação gerado no terminal.

3. Teste de performance

Para realização de testes foram utilizados arquivos de texto em inglês[4], em tamanhos variáveis. Com base em um arquivo de 50mb[5], foram extraídos diversos trechos variados de 1 - 5 mb. Para tal foi criada uma ferramenta auxiliar[6]. Os arquivos utilizados para teste não continham quebra de linha('\n'), dessa forma todo o conteúdo era armazena do memória com escrita e leitura. Utilizando o mesmo arquivo inicial(arquivo de 50MB) foram gerados dois grupos de arquivos com 5 arquivos de 1 - 4MB. Até os 4 primeiros MB foi criado um grupo de arquivos e a partir de então foi criado outro grupo de arquivos. os arquivos foram criados de forma incremental, dessa forma o arquivo de 1Mb está contido no seus arquivos seguintes.

Como padrões de busca foi utilizada a abordagem de uma string simples contendo apenas uma palavra, para ser procurada no texto de index, como também foram gerados arquivos de padrão com as mesma proporções dos arquivos de index. Para geração dos arquivos de padrões, foi utilizado o mesmo texto base utilizado na geração dos arquivos de texto.

O arquivo *benchmark.hpp,,* presente dentro do projeto principal(src/) foi desenvolvido para que os métodos de indexação, codificação, descodificação e busca fossem executados individualmente e tendo seus dados de métricas mareados em um arquivo auxiliar. Abaixo é exibido uma tabela com os tamanhos de cada arquivo de acordo com seu estado(original, indexado e codificado).

Tamanho do arquivo de texto	*Tamanho do arquivo de index*	*Tamanho do arquivo codificação*
grupo de arquivo I		
1000000KB	7888891KB	32240584KB
2000000KB	16888891KB	67847927KB
3000000KB	25888891KB	103442194KB
4000000KB	34888891KB	139946100KB
grupo de arquivo II		
1000000KB	7888891KB	32131987KB
2000000KB	16888891KB	67730048KB
3000000KB	25888891KB	103310047KB
4000000KB	34888891KB	139748726KB

Tabela 1 - Comprimento do arquivo original ao longo da execução dos comandos.

A seguir serão exibidos os gráficos com relação tamanho do arquivo vs tempo de execução, para cada comando.

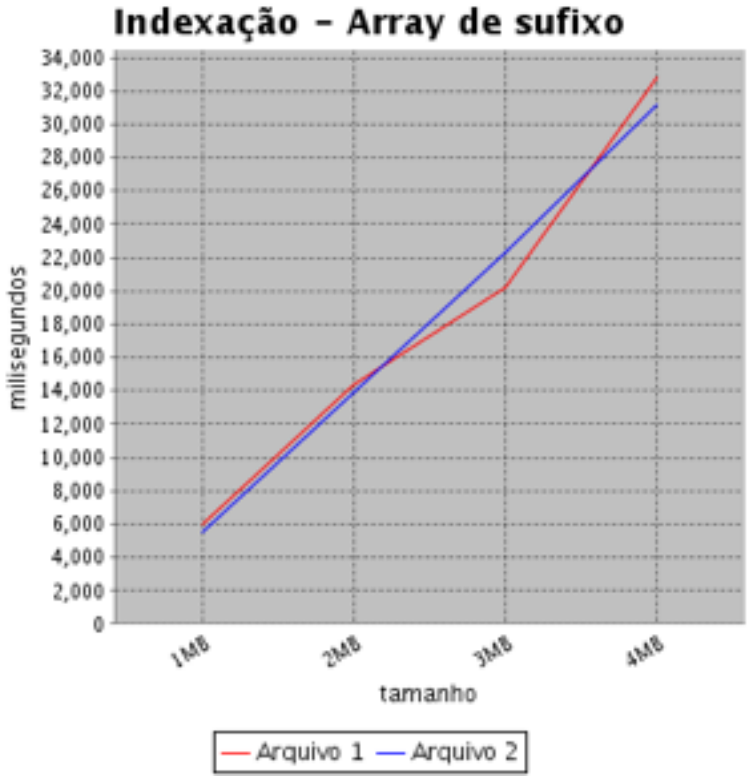

Figura 1 - representação da execução da indexação dos arquivos de teste.

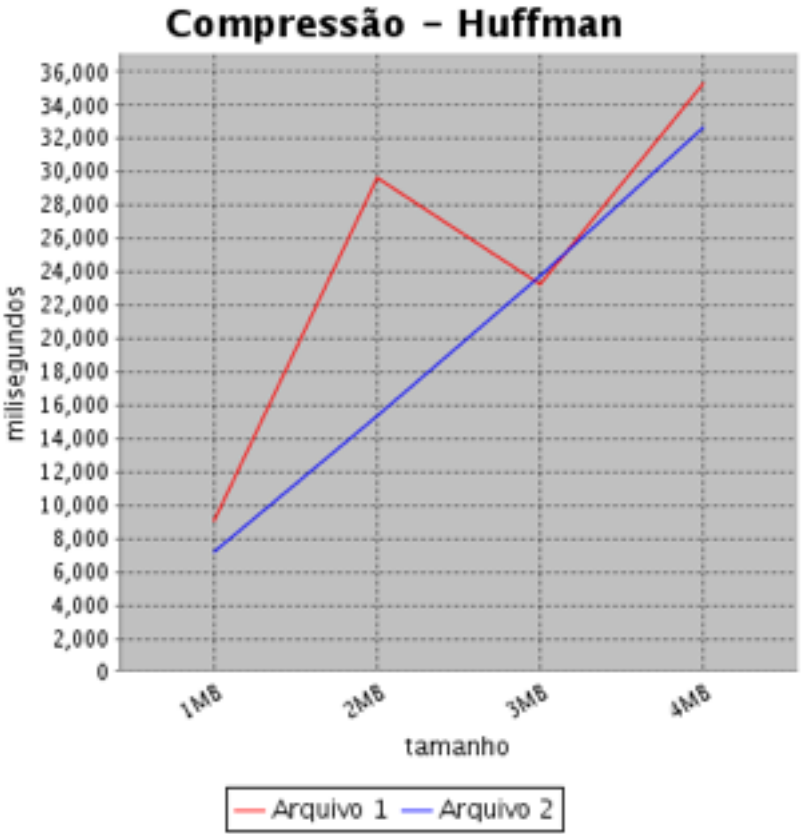

Figura 2 - representação da execução do algoritmo de compressão.

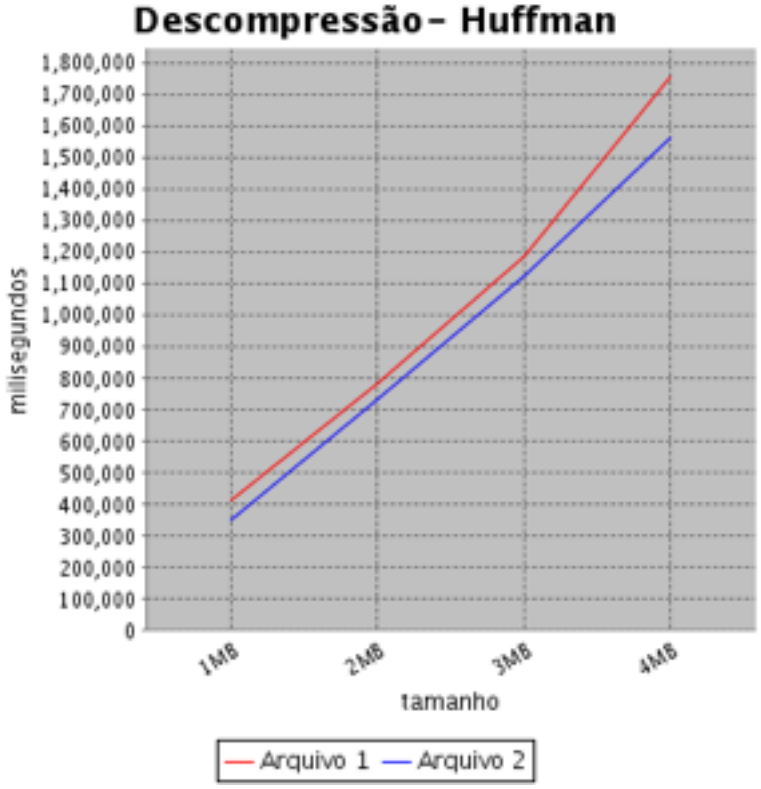

Figura 3 - representação da execução do algoritmo de descompressão.

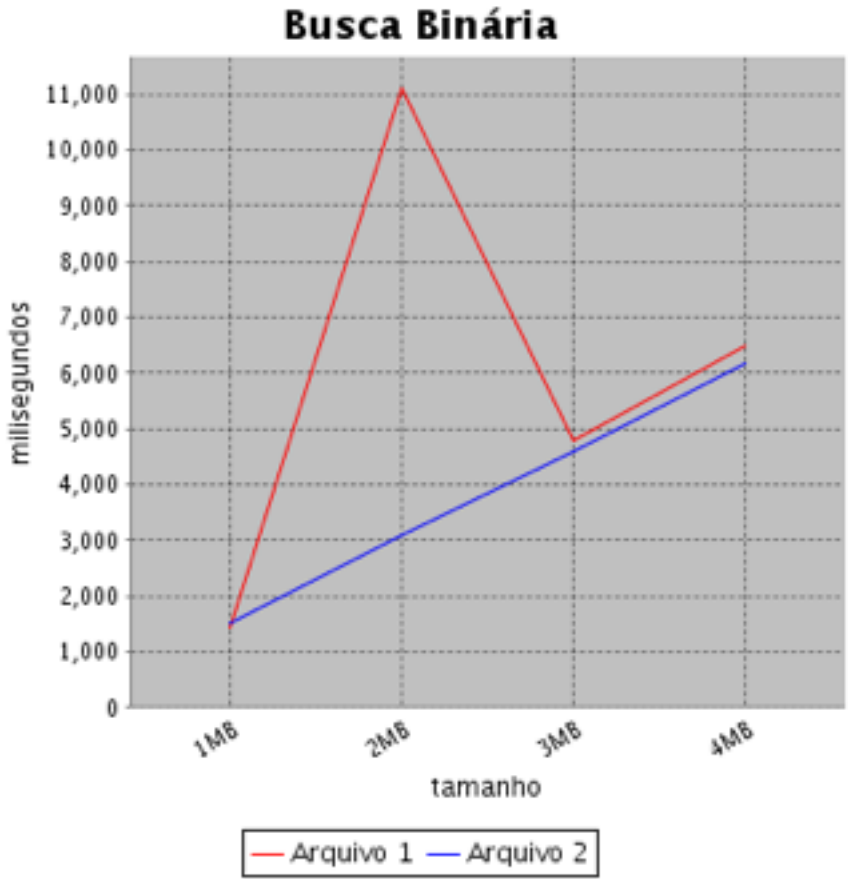

Figura 4 - Execução da busca binária utilizando a relação arquivo 1(de 1-4MB) com padrão 1(de 1-4MB) e arquivo 2(de 1-4MB) com padrão 2(de 1-4MB)

Tamanho do arquivo de texto	***Ocorrências***
grupo de arquivo I	
1000000KB	100441
2000000KB	228185
3000000KB	290382
4000000KB	367747
grupo de arquivo II	
1000000KB	69460
2000000KB	130680
3000000KB	204061
4000000KB	281442

Tabela 2 - representação das ocorrências em cada arquivo. Busca utilizando um padrão com mesmo valor do arquivo de texto.

Algo importante que deve ser levado em consideração é que os arquivos de texto utilizados tinham um alfabeto de 239 caracteres.

As execuções de indexação, compressão e descompressão foram realizadas de forma sequenciais. Com relação a configuração da máquina de realização dos testes anteriores:

- iMac 2008
- Processador: 2.4 GHZ Intel Core 2 Duo
- Memória: 4GB - (1905M livres)
- HD 250GB (1230GB livres)

Referências

[1] Suffix array. Disponível em:< http://visualgo.net/suffixarray.html >.

[2] Algoritmo de Huffman para compressão de dados. Disponível em: < http://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html#Huffman-trie >.

[3] Árvore binária de busca. Disponível em: < http://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-bst.html >.

[4] The Text Collection. Disponível em: < http://pizzachili.dcc.uchile.cl/texts.html >

[5] English texts. Disponível em:< http://pizzachili.dcc.uchile.cl/texts/nlang/english.50MB.gz >

[6] Extract text. Disponível em:< https://github.com/code-like-a-girl/extract-text >