Universidade Federal de Pernambuco - UFPE
Centro de Informática - CIn
Bacharelado em Ciência da Computação

Processamento de Cadeias de Caracteres

Ferramenta IPMT

Recife, Maio de 2022

Universidade Federal de Pernambuco -
UFPE Centro de Informática - CIn

Bacharelado em Ciência da Computação

Disciplina: Processamento de Cadeias de Caracteres - IF767
Professor: Paulo Fonseca
Período Letivo: 2021.2
Alunos: Amanda Nunes Silvestre Costa e Rodrigo Farias Rodrigues Lemos

Sumário

1. Identificação

A equipe é formada por Amanda Nunes Silvestre Costa (ansc) e Rodrigo Farias Rodrigues Lemos (rfrl).

A maior parte do código foi feita utilizando os conceitos de pair programming, fazendo com que todos tenham uma boa noção e contribuição de boa parte do código, segue a lista de responsabilidades que utilizamos na divisão.

Os códigos utilizados foram baseados na aula do professor.

1.1 Amanda Costa

- Script para testes automatizados e criação de tabelas.
- Preparação de testes.
- Compressão (Zip)
- Descompressão (Unzip)

1.2 Rodrigo Lemos

- Arquivo main com a CLI.
- Arquivo makefile com sistema de compilação.
- Indexação
- Método de busca exata

2. Implementação

2.1 Indexação (Criação do Suffix Array)

Detalharemos como foi feito o modelo de indexação e busca dentro do nosso projeto. Incluindo os algoritmos de criação do suffix array.

2.1.1 Criação do vetor de posicionamento.

Durante essa fase é criado um array bidimensional que representa a posição lexicográfica da subcadeia específica sendo calculada. Progredindo o tamanho dessa subcadeia em potências de dois.

Cada par (x, y) desse vetor de posicionamento vai representar a ordem lexicográfica da primeira e da segunda metade da cadeia em análise, esses dois valores juntos são suficientes para indicar a ordem lexicográfica da cadeia como um todo já que é utilizando os dois que realizamos a operação de ordenação.

Na primeira iteração o valor do array será representado pela posição lexicográfica do carácter unitário dentro do texto, as demais seguiram a seguinte lógica:

$i = Posição\ na\ iteração\ (2^i\ representará\ o\ tamanho\ da\ subcadeia\ atual)$
$j = Posição\ no\ texto$
$n = Tamanho\ do\ texto$
$pos[i][j] = \{pos[i-1][j],\ j + 2^i < n\ ?\ pos[i-1][j + 2^i] :\ -1\}$

O algoritmo tem complexidade final de $O\ (n\ log^2 n)$ já que ao todo são realizadas $logn$ iterações, cada uma delas com uma ordenação no vetor contendo os n estados criados.

2.1.2 Criação final do Suffix Array.

Agora que temos o vetor de posicionamento cada valor *i* no estado final representará a ordem lexicográfica da subcadeia começando da posição i, já que as iterações só são finalizadas quando o tamanho da subcadeia atinge o cumprimento da própria cadeia completa.

A complexidade total do algoritmo é $O(n)$ onde *n* é a quantidade de caracteres do texto base. Já que o resultado final do algoritmo de posicionamento já está ordenado, então podemos colocar os valores em suas posições corretas em apenas uma passagem pelo array.

Cada valor final no vetor de sufixo vai representar o índice da primeira letra que inicia o sufixo e estarão ordenados de forma crescente pela ordem lexicográfica.

2.1.3 Criação do arquivo de index.

Cada linha do arquivo original será transformada em 3 linhas no arquivo de index.

A primeira terá um natural *n* que representa a quantidade de caracteres presentes naquela linha, a segunda será uma cópia exata do conteúdo da linha e já a terceira terá *n* inteiros simbolizando o array de sufixo para o dado texto.

2.2 Algoritmo de busca exata

Foram implementados nesta etapa o algoritmo de busca exata baseado no array de sufixo.

2.2.1 Sarr Search

O algoritmo do Sarr Search aproveita que temos a ordenação lexicográfica dos sufixos do texto original para operar utilizando buscas binárias.

Procuramos no array o primeiro valor que tem ordem lexicográfica maior ou igual ao padrão a ser procurado e depois o mesmo, porém apenas considerando os maiores. Ambos os processos têm complexidade de $O\ (log\ n\ *\ (custo\ checar\ cadeias))$ sendo *n* novamente o tamanho do texto original. A diferença entre esses dois valores vai nos representar a quantidade de ocorrências desse padrão no texto, podemos ir mais a fundo e também saber em quais posições do texto exatamente estão localizados esses matches.

2.3 Zip e unzip

Para os comandos de zip e unzip utilizamos o algoritmo de Huffman para compressão e descompressão de textos.

2.3.1 Construção do algoritmo de Huffman (Compactando o texto)

O algoritmo de Huffman se baseia no uso de um vetor de frequências para cada um dos caracteres do texto. A partir dessas frequências, nosso objetivo será criar uma árvore que corresponderá ao código de prefixo, codificando cada um desses caracteres utilizando o caminho percorrido pela árvore de uma forma que aqueles que possuem as maiores frequências ficariam mais próximos da raiz e por consequência teriam um menor valor de codificação.

Para criarmos a árvore, utilizamos uma *priority_queue*, de uma forma que aqueles com a menor frequência acabam ficando no topo dessa prioridade, para a compressão em si.

Para a criação do arquivo binário usaremos sempre blocos de 8 bits escrevendo cada um deles diretamente na composição binária do arquivo, esse processo seguirá duas partes.

Informações sobre a árvore:

Primeiro salvamos dados referente a árvore de Huffman, contendo as informações necessárias para reconstruí-la. salvamos primeiramente o número de caracteres únicos a serem lidos (Chamaremos de *n*), seguido de então de n blocos que seguem o

seguinte formato: 1 bloco contendo o identificador inteiro do carácter, então 1 bloco contendo o número de caracteres no caminho desse caráter na árvore de Huffman (Chamaremos de *m*) e para finalizar $m/8$ blocos cada um contendo um pedaço do código único do respectivo carácter.

Informações sobre o texto:

Para finalizar todo o restante do arquivo terá valores binários referentes a codificação do texto original utilizando o método de Hoffman, fazendo a compactação desse texto.

2.3.2 Descompressão

Tratamos o arquivo de entrada primeiro angariando as informações deixadas em evidência no início a fim de reconstruir a árvore de Huffman de nó em nó. Com esta etapa concluída, passamos pelo arquivo completo, lendo os códigos dos caracteres e consultando através da árvore recriada. Os caracteres resultantes são adicionados no arquivo de saída e formam o texto original.

2.4 Detalhes de implementação relevantes

- Comandos principais:
  - index: Gera o arquivo de texto .idx utilizando como base o arquivo original
  - Search: Faz uma busca exata por um padrão utilizando um arquivo .idx.
  - zip: Faz a compactação de um arquivo de texto em um arquivo .myzip
  - unzip: Descompacta um arquivo .myzip para novamente ter seu formato original.
- Opções de linha de comando:
  - -c —count-only: imprime apenas a quantidade total de ocorrências do(s) padrões contidas no(s) arquivo(s) de texto.
  - -p —pattern-file: arquivo contendo o padrão de busca desejado, aceita múltiplos padrões de uma vez, sendo um por linha do arquivo.
  - -h –help: retorna um guia com os possíveis comandos.
- Alfabeto de entrada
  - *getopt* utilizado para leitura dos argumentos do programa
  - O programa foi feito para funcionar utilizando como base textos com os padrões de caracteres ASCII.

3. Testes e Resultados

Os testes foram realizados através de um script python e bash, utilizando a ferramenta GNU time para obtenção do tempo de execução. Os resultados encontrados foram armazenados em uma tabela do Google Sheets a fim de comparar o desempenho da execução de cada comando quando enfrentando um determinado dataset.

Os arquivos utilizados são de diferentes tamanhos e estão disponíveis no Pizza&Chili, já os padrões, foram obtidos de maneira aleatória nos arquivos. Os testes foram realizados em um MacBook Pro modelo 2019 com 8 GB de memória e um processador Intel Core i5 1,4 GHz.

3.1 Comando de Indexação

Comparação de tempo de execução do algoritmo de criação do index utilizando diferentes arquivos em diferentes tamanhos. Percebemos durante os testes que pouco importava o tipo do arquivo, os tempos de execução ficaram muito similares quando utilizando arquivos de mesmo tamanho.

Algo importante de se comentar é que o nosso arquivo indexado estava sempre tendo um tamanho aproximadamente quatro vezes maior que o arquivo original, fica como melhoria futura preparar uma forma mais eficiente de armazenamento para essas informações.

Teste de processamento (Analisando o tempo em segundos)

Comando Index	50MB	100MB	200MB	FULL
Sources	44	85	176	173
English	39	80	156	1683
XML	46	90	177	250

Teste de tamanho (Analisando o tamanho do arquivo .idx)

Tamanho Index	50MB	100MB	200MB	FULL
Sources	392,7 MB	392,7 MB	784,9 MB	789,1 MB
English	206.1 MB	402,3 MB	804,7 MB	8,48 GB
XML	201.1MB	391,8 MB	786 MB	1,11 GB

3.2 Comando de Busca Exata

Para critério de comparação, durante o método de busca foi utilizada a opção “count-only”, retornando apenas a quantidade total de ocorrências.

Para os testes também foi utilizado um padrão aleatório, porém curto o suficiente para que tivesse aparições constantes durante os textos.

Assim como o algoritmo de indexação também foi detectada pouca variação dependendo do tipo do texto e todos seguiram uma crescente linear à medida que vamos aumentando em tamanho.

Teste de processamento (Analisando o tempo em segundos)

Comando search	50MB	100MB	200MB	FULL
Sources	13	25	48	49
English	12	24	47	519
XML	13	24	49	69

3.3 Comandos Zip e Unzip

Para os testes de compressão e descompressão de texto, foram utilizados três tipos de texto diferentes, o foco aqui foi, além de observar como o tamanho inicial do texto influencia no final, entender como o nosso algoritmo se comporta diante a alfabetos diferentes.

As características dos textos utilizados são:

- DNA:
  - Composto por sequências de DNA, majoritariamente formado pelos caracteres ‘A’, ‘C’, ‘G’ e ‘T’.
  - Tamanhos utilizados: 50MB, 100MB, 200MB e 400MB.
- English
  - Composto por textos na língua inglesa.
  - Tamanhos utilizados: 50MB, 100MB, 200MB e 2,21GB.
- Sources:
  - Composto por concatenações de corpos de código nas linguagens C e Java.
  - Tamanhos utilizados: 50MB, 100MB e 200MB.

3.3.1 Zip

Para o teste de compressão usando como critérios de comparação o tamanho do arquivo comprimido e no tempo de execução.

Dentro de cada tipo de texto, observamos comportamento linear no crescimento do texto. Notamos também que quão mais simples um alfabeto, menor em comparação vai ser o tamanho do texto comprimido que o utiliza, tal qual o tempo de execução.

Teste de tamanho e tempo de processamento do texto DNA

Texto	Tamanho Original	Tamanho Comprimido	Tempo de Execução
DNA	52,4 MB	14,5 MB	3,0 s
	104,8 MB	28,9 MB	6,3 s
	209,7 MB	57,7 MB	13,0 s
	403,9 MB	111,62 MB	23,9 s

Teste de tamanho e tempo de processamento do texto English

Texto	Tamanho Original	Tamanho Comprimido	Tempo de Execução
English	52,4 MB	29,9 MB	5,4 s
	104,8 MB	60,19 MB	11,0 s
	209,7 MB	119,5 MB	22,5 s
	2,21 GB	1,26 GB	4 min 4 s

Teste de tamanho e tempo de processamento do texto Sources

Texto	Tamanho Original	Tamanho Comprimido	Tempo de Execução
Sources	52,4 MB	36,4 MB	6,0 s
	104,8 MB	72,9 MB	12,0 s
	210,8 MB	144,6 MB	23,6 s

3.4 UnZip

Para o teste de descompressão, após a confirmação de que os arquivos descomprimidos estavam corretos em tamanho e conteúdo, usamos como critério de comparação o tempo de execução.

O comportamento deste algoritmo foi semelhante ao Zip em linearidade de tempo dentro do mesmo alfabeto, mas não mostrou grandes variações entre tipos de textos diferentes.

Teste de corretude e tempo de processamento do texto DNA

Texto	Tamanho	Tamanho Descomprimido	Tempo de Execução
DNA	14,5 MB	52,4 MB	3,0 s
	28,9 MB	104,8 MB	6,3 s
	57,7 MB	209,7 MB	13,0 s
	111,62 MB	403,9 MB	23,9 s

Teste de corretude e tempo de processamento do texto English

Texto	Tamanho	Tamanho Descomprimido	Tempo de Execução

English	14,5 MB	52,4 MB	2,5 s
	28,9 MB	104,8 MB	5,1 s
	57,7 MB	209,7 MB	10,2 s
	111,62 MB	2,21 GB	1 min 48 s

Teste de corretude e tempo de processamento do texto Sources

Texto	Tamanho	Tamanho Descomprimido	Tempo de Execução
Sources	36,4 MB	52,4 MB	2,8 s
	72,9 MB	104,8 MB	5,6 s
	144,6 MB	210,8 MB	11,3 s

4. Conclusões

Em conclusão, nós percebemos que todos os algoritmos que nós implementamos acabam sendo pouco variáveis em relação ao conteúdo do texto em si, trazendo mais estabilidade para esses métodos e apenas variando de forma linear no tamanho dos arquivos.

Nós também notamos que quando comparado com o projeto anterior que se voltava apenas para a busca de padrões tivemos uma melhora no tempo de processamento, porém somando com o tempo de execução da indexação vimos uma piora considerável, isso já era esperado, porém achamos que poderíamos otimizar mais o nosso algoritmo de criação do arquivo *.idx* em momentos futuros.