Universidade Federal de Pernambuco – UFPE

Centro de Informática – CIn

Recife – 2021.2

Processamento de

Cadeias de Caracteres

Relatório IPMT

Escrito por:

Eidson Sá (ejas),

Vinícius Dantas (vdj)

Entrega do relatório: 15/05/2022

Sumário

Identificação

A equipe é formada por Eidson Jacques Almeida de Sá e Vinícius Dantas Januário. Vinícius fez a parte de compressão, indexação, parte do frontend e os scripts e execução dos testes dos algoritmos. Eidson fez a parte de descompressão, busca, bem como parte do frontend, processamento dos dados de teste para geração de gráficos e tabelas, bem como a documentação.

Implementação

Descrição da ferramenta

A ipmt é uma ferramenta de uso por linha de comando (CLI) que permite fazer busca utilizando a técnica de indexação, bem como compressão e descompressão de arquivos. Suporta apenas arquivos no formato ascii, mas pode facilmente ser adaptada para suportar arquivos em UTF-8 (unicode).

Para indexação de textos e busca de padrões, implementamos o suffix array, suportando a criação de índices de arquivos de até 25MB. Para compressão e descompressão implementamos o algoritmo Huffman. Por padrão, cada modo de uso da ferramenta usa os algoritmos descritos.

A aplicação possui 3 opções de uso:

- c –count para imprimir apenas a quantidade de padrões no texto
- p –pattern para indicar o uso de um arquivo com padrões
- h –help para imprimir o help da aplicação

Detalhes de implementação relevantes

Ao contrário da primeira entrega, utilizamos mais a classe std::string e std::fstream para representar os textos, ler e escrever os arquivos. Criamos duas classes com métodos e variáveis estáticas, cada uma para lidar com um algoritmo específico.

Assim como na primeira entrega, buscamos implementar os algoritmos vistos na disciplina, buscando fazer melhorias na medida do possível e se atentando aos detalhes de implementação ao se trabalhar com C++. Também utilizamos a classe BenchMarkTimer para nos ajudar na medição de performance de um determinado escopo da aplicação.

Na construção do suffix array conseguimos utilizar o radix sort, reduzindo a complexidade da versão da demonstração de $O(n \log^2 n)$ para $O(n \log n)$. Porém, não conseguimos reduzir a complexidade de espaço na construção da ordem do sufixo de cara interação (os P's da demonstração) de $O(n \log n)$ para $O(n)$, impactando severamente no tamanho do arquivo que pode ser indexado, bem como no próprio desempenho da indexação, na necessidade de alocar o devido espaço em memória.

No arquivo de índice, armazenamos o tamanho do texto, bem como os vetores de sufixo, Llcp, Rlcp e as frequências dos caracteres no texto. Com esses dados o texto pode ser facilmente reconstruído no modo de busca.

O modo de busca com a opção “-c” funciona dentro da complexidade esperada, porém sem essa opção o programa acaba sendo bem mais lento pela necessidade de ordenar as ocorrências dos padrões no texto para imprimir as linhas em ordem.

Na parte compressão com o algoritmo de Huffman, as informações que são guardadas no .myz são: frequências das letras no texto (128 * 4 bytes), quantidade de bits a ser lida no último byte (1 byte), o tamanho original do arquivo (1 * 8 bytes) e por fim a própria string binária codificada. Durante a construção da árvore binária usamos funções da própria biblioteca de C++ que constrói a heap on place dentro de um vetor, com destaque a std::make_heap, que segundo a documentação da standard library constrói a heap em tempo linear. Para armazenar os códigos durante a compressão utilizamos uma implementação de hashtable da standard library, a std::unordered_map. Durante o processo de compressão o arquivo é lido de uma vez como um vetor de bytes e a codificação é feita nesse mesmo vetor, com o uso de memória se limitando ao tamanho do texto, alfabeto e tamanho da árvore gerada.

Na parte de descompressão a árvore é reconstruída a partir das frequências dos caracteres. Cada bit do código é processado, quando um caractere é descodificado ele é transferido para uma fila (std::queue) auxiliar, e sempre que possível esses caracteres são escritos no buffer de saída que é o próprio texto de entrada. O uso de memória se limita ao tamanho do texto comprimido, tamanho do texto original, bem como tamanho do alfabeto e da árvore gerada.

A aplicação não possui nenhum bug conhecido, a corretude foi averiguada fazendo testes manuais com a ferramenta diff e a com a comparação da saídas de outras ferramentas como a grep.

Testes e Resultados

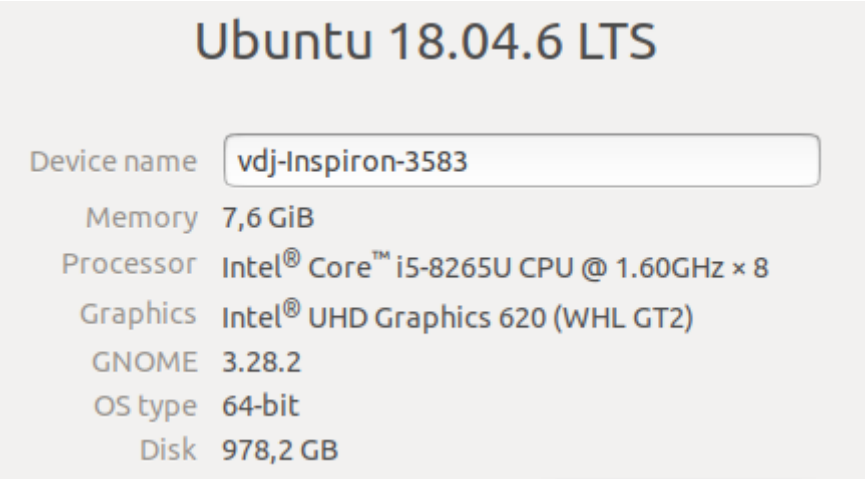
Para os testes de desempenho da aplicação foram utilizados 3 datasets, disponíveis em http://pizzachili.dcc.uchile.cl/texts.html. Foram eles:

- english.1024MB: Texto em língua inglesa de 1gb.
- sources.200MB: Código fonte na linguagem C de 200mb.
- dna.200MB: Sequências de DNA, com alfabeto limitado, com 200mb.

Para os dois primeiros datasets foi necessário fazer um pré processamento do texto, devido a estarem no formato ISO-8859-1. Utilizamos a ferramenta *iconv* do linux para converter os arquivos em ASCII.

Como nossa implementação de indexação se limita à arquivos de até 25MB, foram utilizados recortes de aproximadamente 19MB do primeiro (english.1024MB) e terceiro (dna.200MB) dataset como input.

Todos os testes foram realizados em uma máquina com disco rígido na seguinte configuração:



Indexação e busca

Para indexação foi feito um teste e para busca 5 testes, sendo 2 deles com busca de um só padrão e 3 com busca de múltiplos padrões.

No primeiro simplesmente indexamos os dois recortes dos datasets english e dna, obtendo o seguinte tempo de execução:

Tempo de execução indexação

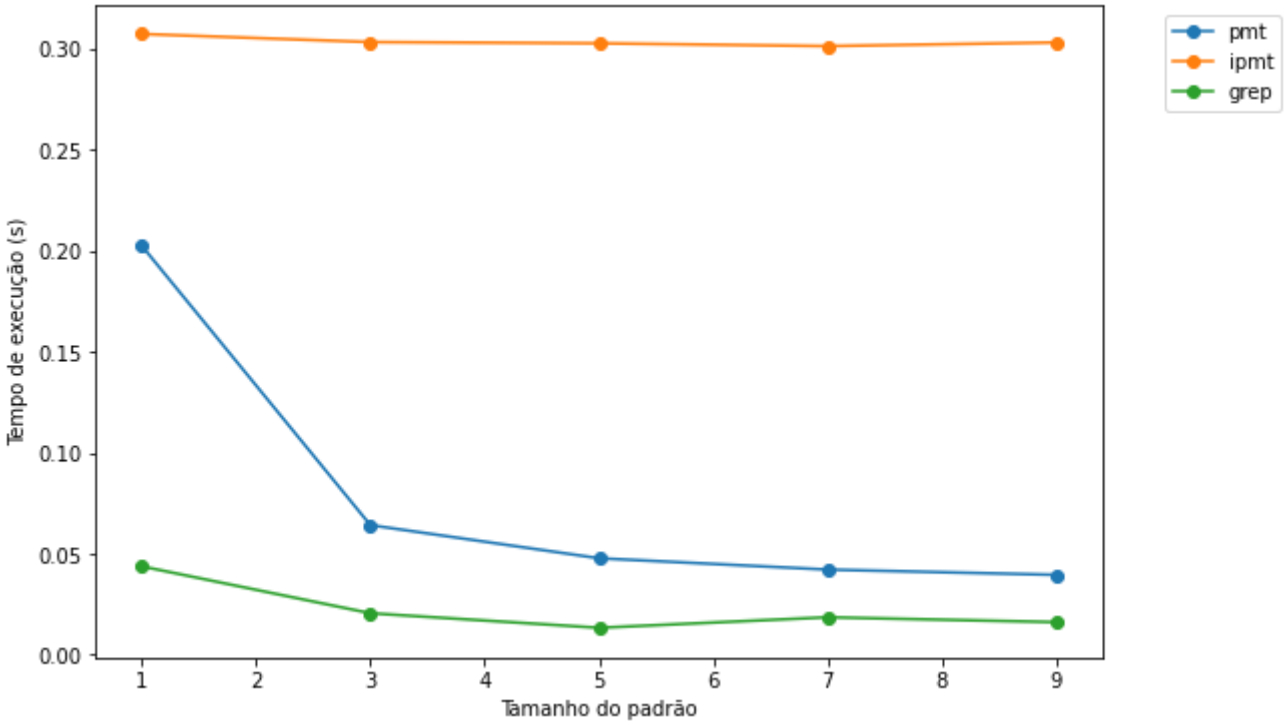
	english	dna
Tempo de execução(s)	137.9	144.9

Apesar de o processo de indexação ser o mais custoso, ficou evidente que guardar todas as ordens dos sufixos na construção do vetor de sufixos foi o grande gargalo desse processo. Com a ajuda da classe auxiliar BenchmarkTimer notamos que cerca de 120 segundos desse tempo é gasto na construção do vetor de sufixos dos dois casos.

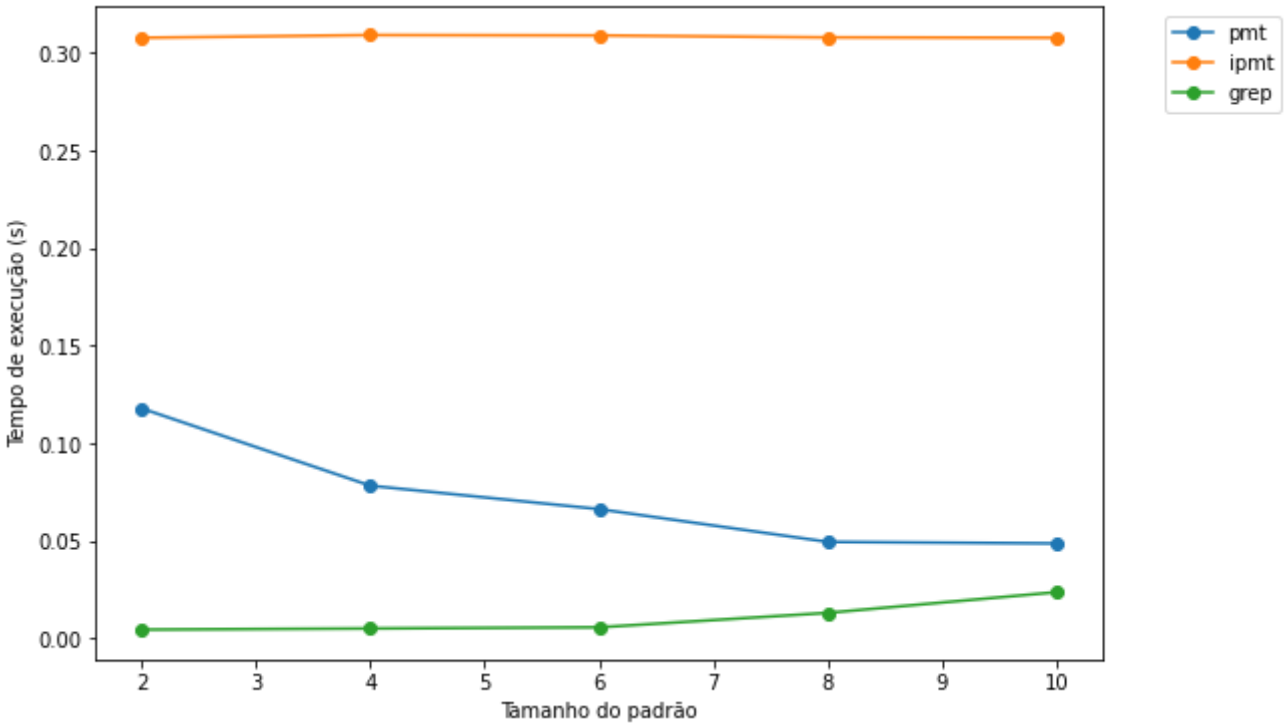
Para o segundo e terceiro testes usamos os arquivos previamente indexados, buscamos palavras de tamanho 1 a 10 dentro de cada dataset, para cada tamanho buscamos 5 palavras e consideramos o tempo de execução como a média aritmética do tempo de execução das 5 palavras. Para comparação utilizamos a ferramenta grep e a pmt, implementada na primeira entrega. Todos os testes foram feitos com a opção -c e em seu modo de busca padrão.

Segue os gráficos com os resultados dos testes:

Palavras em inglês, recorte de english.1024MB



Sequência de dna, recorte de dna.200MB



Podemos ver que o tempo de execução do suffix array apesar de maior, se mantém constante independente do tamanho do padrão, enquanto que as ferramentas grep e pmt flutuam como esperado.

Nos próximos 3 testes, testamos múltiplos padrões de uma só vez, com o uso da opção -p para a pmt e ipmt e -f no grep. No primeiro teste utilizamos todas as palavras que separamos do primeiro teste de busca para o recorte do dataset english.1024MB, totalizando 25 palavras, e de maneira análoga realizamos o segundo teste para para o recorte do dataset dna.200MB. No terceiro

se realizou um teste de estresse com 3000 palavras encontradas neste link, no recorte do dataset english.1024MB. A seguir as tabelas com os resultados:

25 palavras, recorte de english.1024MB, 5365052 ocorrências.

	pmt	ipmt	grep
Tempo de execução(s)	0.203	0.301	0.019

25 sequências de dna, recorte de dna.200MB, 7223692 ocorrências.

	pmt	ipmt	grep
Tempo de execução(s)	0.244	0.309	0.005

3000 palavras, recorte de english.1024MB, 6464185 ocorrências.

	pmt	ipmt	grep
Tempo de execução(s)	0.338	0.307	0.040

Observando os resultados desses testes, notamos que a ferramenta ipmt mesmo com o acréscimo de palavras a serem buscadas se manteve na faixa dos 0.3s de tempo de execução, enquanto que com o pmt, que no caso usa o algoritmo Aho Corasick como padrão, seu tempo de execução aumenta. No caso de 3000 palavras chega a ser maior que o do Suffix Array do ipmt. Os resultados do grep foram tão rápidos quanto pesquisando um só padrão, sendo difícil de se tomar algo conclusivo.

Depois de feito esses testes de busca, averiguamos com o uso da classe BenchmarkTimer o quanto de tempo de execução estava sendo gasto propriamente com as operações de busca com as palavras. Vimos que no último teste com 3000 palavras esse tempo não chegou a passar de 3.4ms (0.0034s). Portanto, a maior parte do tempo no suffix array é gasto com carregamento de índice e reconstrução do texto, que são operações custosas.

Compressão e descompressão

Na parte de compressão e descompressão realizamos 2 testes, basicamente executamos as funcionalidades para cada um dos datasets que utilizamos, a fim de medir seu tempo de execução, bem como aferir a taxa de compressão que o algoritmo obteve. Para comparação utilizamos a ferramenta gzip em seu modo de execução padrão, mantendo o arquivo de origem com a opção -k. A taxa de compressão foi calculada como sendo a divisão do tamanho original pelo tamanho do arquivo comprimido. Segue as tabelas com os resultados:

Tempo de execução compressão

Dataset/Ferramenta	ipmt	gzip
english.1024MB	35.76s	64.33s
dna.200MB	4.84s	20.97s
sources.200MB	6.91s	5.81s

Tempo de execução descompressão

Dataset/Ferramenta	ipmt	gzip
english.1024MB	21.35s	6.24s
dna.200MB	1.71s	1.06s
sources.200MB	4.02s	0.95s

Taxa de compressão

Dataset/Ferramenta	ipmt	gzip
english.1024MB	1.754	2.633
dna.200MB	3.631	3.541
sources.200MB	1.457	4.434

Durante a compressão, o ipmt teve melhor desempenho nos dois primeiros datasets, enquanto que o gzip foi melhor foi melhor no último. Na descompressão o gzip foi mais rápido em todos os casos. No quesito compressão, o algoritmo de huffman só obteve um melhor resultado no dataset de sequências de dna, por uma pequena margem.

Tirando o fato que em relação ao tempo de descompressão nosso algoritmo deixou um pouco a desejar, o resultado é de certa forma esperado. Lendo um pouco a respeito do algoritmo do gzip, vemos que ele se baseia no algoritmo Deflate, que primeiro aplica a eliminação de strings repetidas no texto com o algoritmo LZ77 e depois reduz a representação de símbolos com o algoritmo de Huffman. Com isso o gzip consegue tirar proveito do caso de uso de ambos os algoritmos e conseguir no geral uma boa taxa de compressão. Isso fica bem evidente no terceiro teste (sources.200MB), um texto com código fonte em C, onde o gzip consegue tirar mais proveito das repetições que são mais comuns de ocorrer nesse tipo de texto, enquanto que no nosso caso a compressão não é tão boa, já que existem muitos símbolos no alfabeto e que naturalmente gera uma árvore grande. Já no teste onde o alfabeto do texto era reduzido (dna.200MB), o nosso algoritmo conseguiu ter uma melhor taxa de compressão que a do gzip, mostrando que ter um alfabeto reduzido ajuda a compressão de Huffman ter melhores desempenhos.

Conclusões

Sobre o que foi visto de suffix array, apesar de não termos conseguido implementar o algoritmo totalmente otimizado na parte de indexação, foi que apesar de não conseguir um desempenho tão bom para busca de padrões como é feita no grep e no pmt, o tempo de busca efetivo é ínfimo se comparado às das duas ferramentas. Se torna ideal para casos de uso onde se tem um programa com um maior tempo de vida e que ao longo do tempo é necessário fazer buscas dentro de um texto, sendo o maior overhead na parte de construção e carregamento do índice do texto.

Já no algoritmo de Huffman, notamos que é um algoritmo que comprime muito bem textos com alfabeto reduzido, porém que não consegue melhores taxas de compressão para textos com alfabeto grande. Os tempos de execução também seguem esse padrão, são melhores com alfabetos reduzidos, se comparado ao gzip, porém particularmente no tempo de descompressão nossa implementação talvez deixe um pouco a desejar, seja por demérito nosso ou mérito do gzip.