UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA - CIN

IPMT - Indexed Pattern Matching Tool

Processamento de Cadeias e Caracteres - 2015.2
Prof. Paulo Gustavo Soares

Equipe:
Guilherme Palma Peixoto - gpp
Rafael Acevedo de Aguiar - raa7

Recife, 2015

Sumário

1. Identificação

1.1 Equipe

Guilherme Palma Peixoto (gpp)
Rafael Acevedo de Aguiar (raa7)

1.2 Descrição da contribuição dos membros

Guilherme Peixoto:

- Implementação do(s) algoritmo(s) de indexação e de *string matching*;
- Execução dos testes relacionados.

Rafael Acevedo

- Implementação do(s) algoritmo(s) de compressão e descompressão;
- Execução dos testes relacionados.

2. Implementação

2.1 Algoritmos implementados

Foram implementados primariamente dois algoritmos na totalidade do projeto. Para a indexação, foi utilizado *array* de sufixos com construção em $O(n \log n)$, permitindo assim busca em $O(m + occ)$ (t.q. *m* é o tamanho do padrão e *occ* o número de ocorrências) após indexação. Para compressão e descompressão, foi utilizado o *LZW*.

2.2 Situações nas quais cada algoritmo é empregado

Sempre que um texto for indexado pelo usuário, será utilizado *array* de sufixos para gerar a indexação do texto. Além disso, o texto será comprimido por uso do LZW e os índices juntamente com o texto comprimido serão concatenados para gerar o arquivo ".idx".

2.3 Detalhes de implementação relevantes

Foi utilizado C++11 para realização do projeto e interpretador de Python para a execução de testes. Durante a leitura das entradas, fazemos a chamada dos algoritmos quando uma linha do arquivo de texto é lida (evita que um arquivo inteiro seja carregado na memória). Caso haja algum pré-processamento, este é feito antes da leitura do arquivo de texto. Isso foi feito para evitar o carregamento de um arquivo de texto inteiro na memória.

Todo o texto fornecido pelo usuário é concatenado "por '\n'" em uma única *string* antes de realizar a indexação e consequentemente a criação do *array* de sufixos. O projeto foi desenvolvido para trabalhar com caracteres ASCII.

Implementamos as seguintes opções relevantes para a busca exata por um padrão:

Short option	*Long option*	Descrição
-c	--count	Conta o número total de ocorrências que ocorreram um match com o padrão procurado em todo o texto. Análogo ao "grep -o <pattern>".
-p	--pattern	Busca por todos os padrões contidos em um arquivo.
-t	--time	Imprime na tela o tempo de execução[1] da busca,indexação, *encoding* ou *decoding*.

Tabela 1: Opções (e respectivas descrições) suportadas pelo *ipmt* implementado.

[1] O tempo calculado não levará em conta as operações de IO.

3. Testes

A fim de testar a performance do projeto desenvolvido, podemos dividir os testes nas seguintes fases: busca exata e busca aproximada. A menos que explicitamente anotado, todos os testes foram feitos em: OS X 10.11, 2.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3.

3.1 Construção do *array* de sufixo

A fim de realizar a busca, primeiro deve-se indexar o texto. Devido ao tempo necessário para indexar os textos e ao espaço dos arquivos ".idx" gerados, os arquivos testados não excederam o tamanho de 45MB. Podemos observar o tempo necessário para indexar o arquivo em função do tamanho do mesmo, que é caracterizado pela complexidade de tempo de construção do *array* de sufixos com as operações de IO necessárias (leitura do arquivo e escrita da representação do *array* e o texto comprimido em arquivo). O primeiro teste foi realizado num arquivo gerado[2] a partir da concatenação de várias linhas de "texto *lorem ipsum*". Os dois gráficos a seguir representam os testes realizados nesses arquivos:

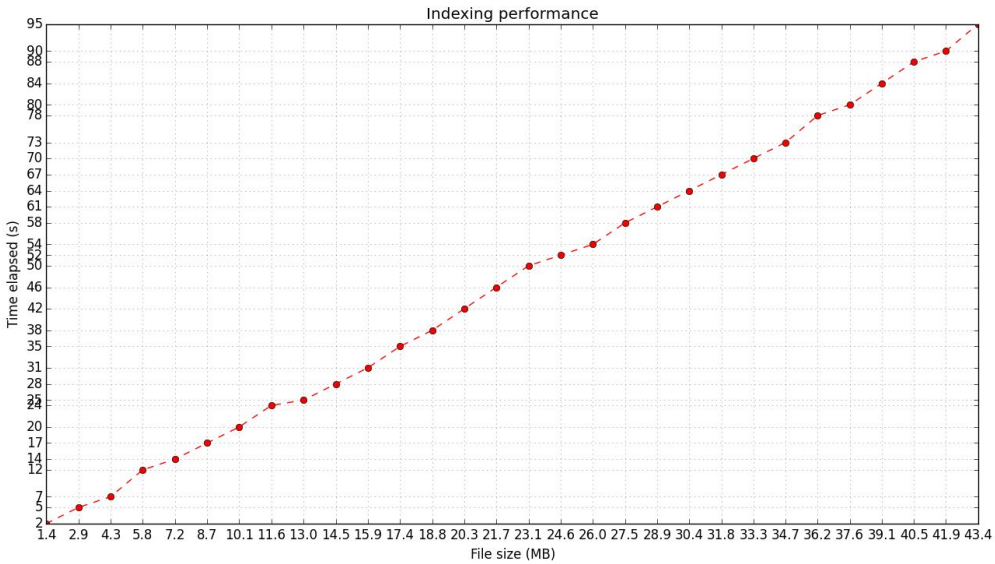

Figura 1: Gráfico do tempo necessário para construir a representação em arquivo do array de sufixo com a compressão do texto em relação ao tamanho do arquivo de entrada.

Podemos observar também o tamanho do arquivo '.idx' gerado na Figura 2:

[2] Foi utilizada a biblioteca open source *Faker* para *Python* a fim de gerar esses arquivos. Disponível em: https://github.com/joke2k/fakerm.

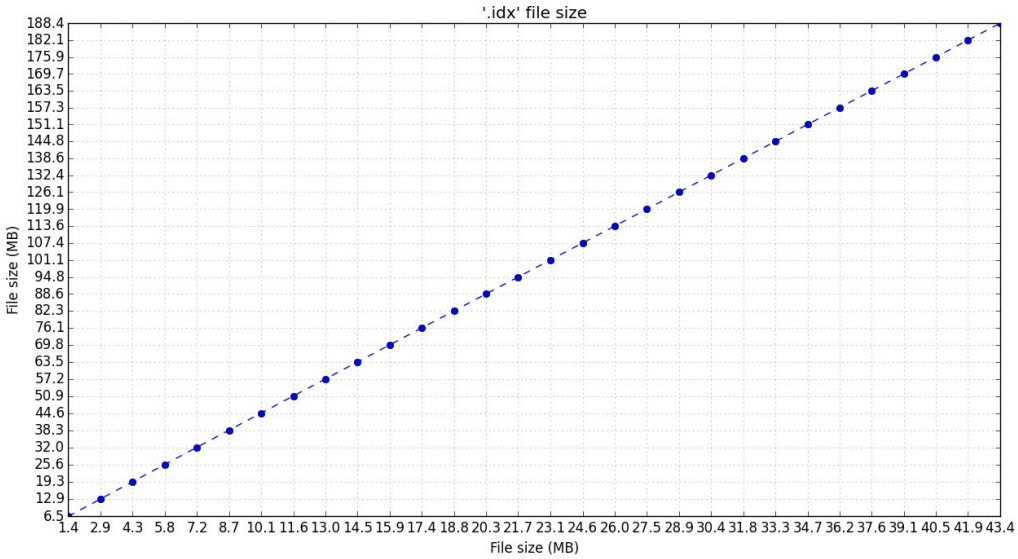

Figura 2: Gráfico do tamanho do arquivo '.idx' gerado em relação ao tamanho do arquivo de entrada.

Podemos observar o comportamento da indexação também em um conjunto de arquivos com *strings* aleatórias[3], onde cada linha do arquivo consiste em 500 dígitos tirados a partir de uma distribuição uniforme, onde cada caractere do alfabeto[4] possui igual probabilidade (10%) de aparecer em uma dada linha. Mostramos o resultados da performance em tempo e em espaço (tamanho do arquivo '.idx' gerado) da indexação a seguir:

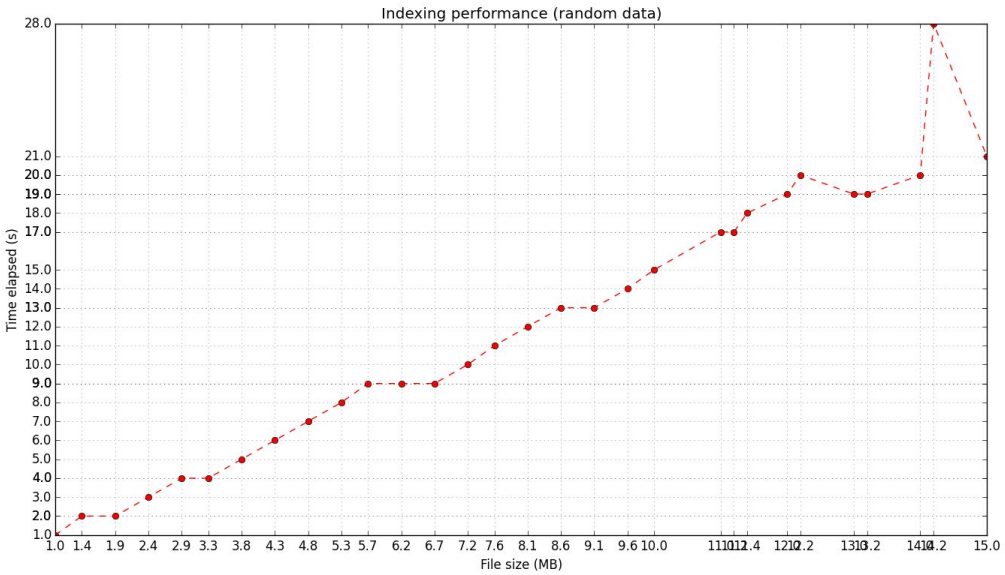

Figura 3: Gráfico do tempo necessário para construir a representação em arquivo do array de sufixo com a compressão do texto em relação ao tamanho do arquivo de entrada gerado a partir de *strings* aleatórias.

[3] Foi utilizado código auxiliar em *Python* para gerar esses arquivos.
[4] O alfabeto utilizado foi [abcdefghij].

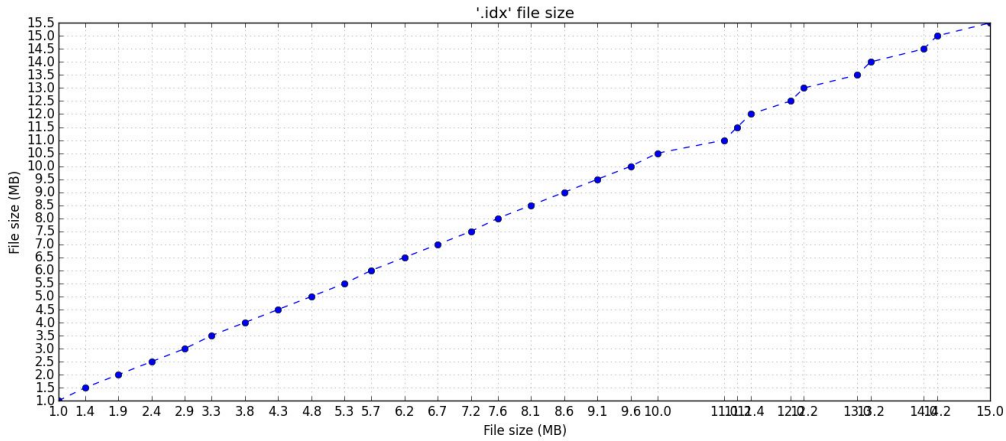

Figura 4: Gráfico do tamanho do arquivo '.idx' gerado em relação ao tamanho do arquivo de entrada gerado a partir de *strings* aleatórias.

3.2 Busca

Após a fase de indexação, é possível realizar a busca por padrões. Testamos a performance do projeto em cima dos arquivos mencionados nos testes acima, comparando com a ferramenta *grep*. Como a ferramenta *grep* retorna as linhas em que ocorreram *match*, para representar o número total de ocorrências utilizamos "grep -o <pattern> <file> | wc -l" para recuperar apenas o número total de ocorrências de um padrão em um dado arquivo. Similarmente, usamos "./ipmt search -c -t <pattern> <idxfile>" para recuperarmos apenas o número de *matches* e obtermos o tempo de execução. Testamos as seguintes configurações de busca:

- Texto natural de *lorem ipsum*, padrão fixo, tamanho do arquivo de entrada variado;
- Texto natural de *lorem ipsum*, tamanho do padrão variado, tamanho do arquivo de entrada fixo;
- Texto formado por *strings* aleatórias, padrão fixo, tamanho do arquivo de entrada variado;
- Texto formado por *strings* aleatórias, tamanho do padrão variado, tamanho do arquivo de entrada fixo.

Reportamos a seguir o desempenho do *ipmt* e comparativamente o *grep* para cada uma das situações descritas acima.

3.2.1 Texto *lorem ipsum*

Devido a ser um texto de linguagem "natural", o número de ocorrências não é uniforme de acordo com os caracteres presentes no alfabeto. Podemos observar, primeiro, o desempenho em relação ao tamanho do arquivo de entrada e ao número de ocorrências do padrão (o padrão escolhido foi "*lorem*"):

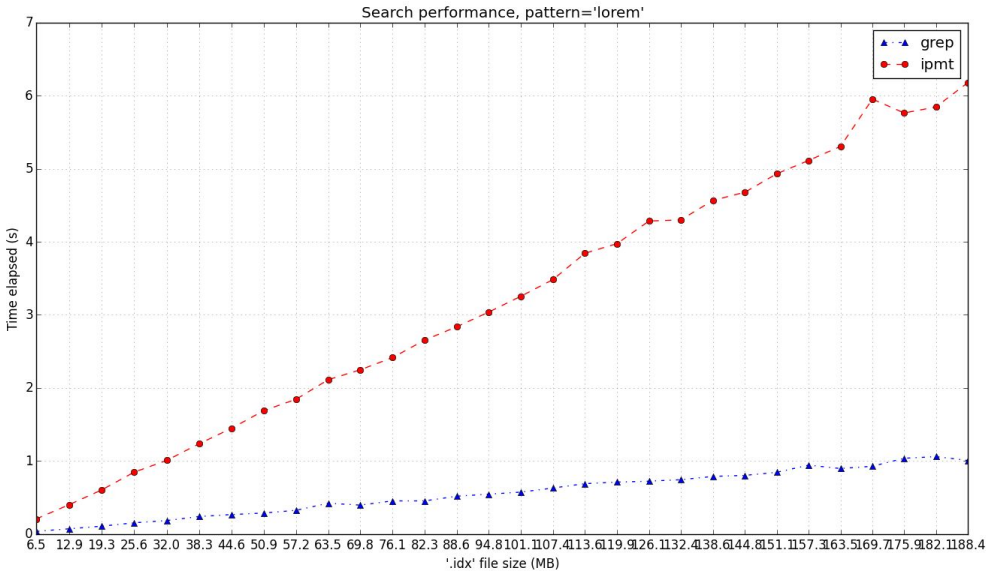

Figura 5: Gráfico da performance do *ipmt* e *grep* para busca de um padrão fixo (*lorem*) em um texto de linguagem natural em relação ao tamanho do arquivo de entrada.

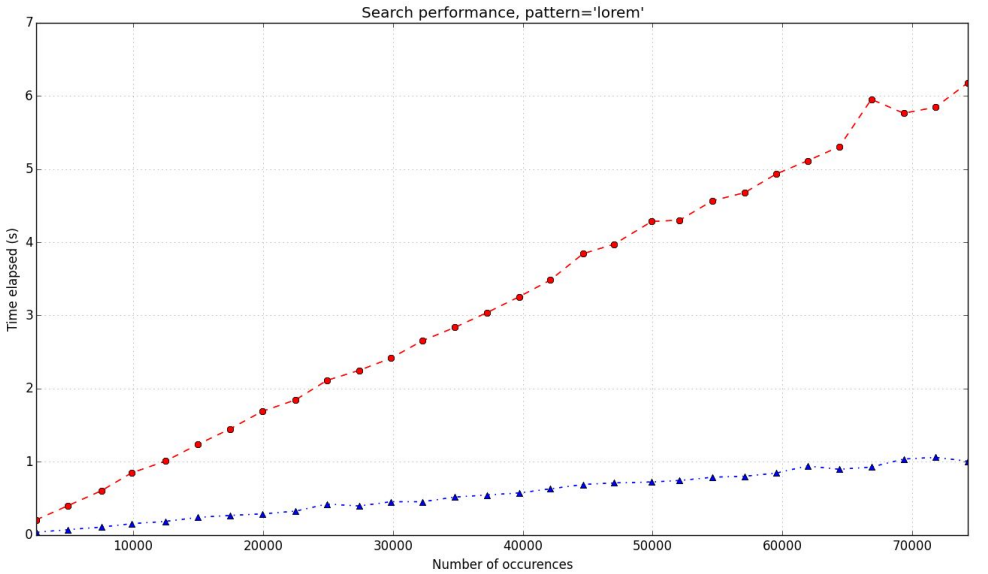

Figura 6: Gráfico da performance do *ipmt* e *grep* para busca de um padrão fixo (*lorem*) em um texto de linguagem natural em relação ao número de ocorrências.

Esse comportamento é esperado, uma vez que o algoritmo dependa do número de ocorrências de um determinado padrão. Para observar o desempenho em relação ao tamanho do padrão, primeiro observamos a relação entre o tamanho do padrão e o número de ocorrências:

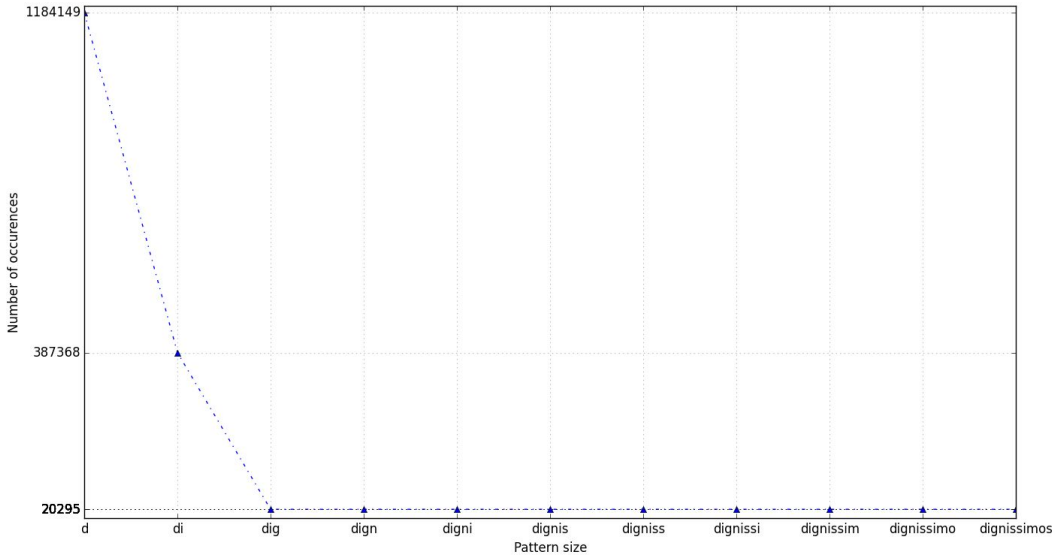

Figura 7: Relação entre um determinado padrão de tamanho [1…11] e o número de ocorrências do mesmo.

Vimos que a partir de "dig", todas as ocorrências seguem de *substrings* da palavra *dignissimos*. Por se tratar de um texto de linguagem natural, poucas palavras são grandes o suficiente para testar o efeito do tamanho do padrão na performance da ferramenta (uma vez que há muitos espaços/caractere ' '). Avaliamos melhor o desempenho do tamanho do padrão no conjunto de arquivos gerados a partir de strings aleatórias.

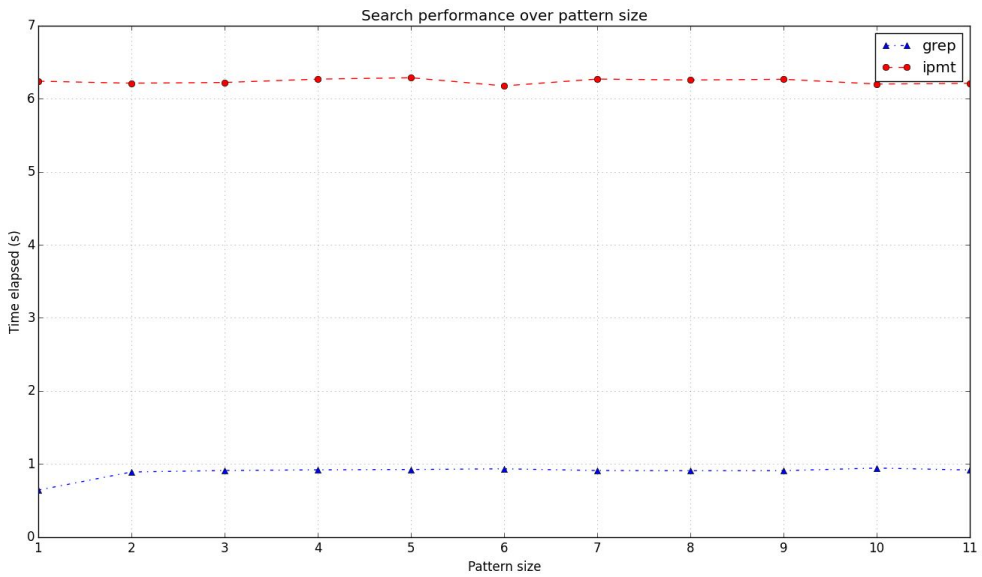

Figura 8: Efeito do tamanho do padrão na performance da busca em texto *lorem ipsum*.

Esse comportamento é esperado, uma vez que a performance do algoritmo depende do tamanho do padrão e do número de ocorrências, vimos que há pouca diferença na performance, visto que a partir do tamanho 3 é o mesmo número de ocorrências para todos os padrões restantes.

3.2.2 Texto aleatório

Nesses arquivos, cada linha contém 500 caracteres do alfabeto [a, b, c, d, e, f, g, h, i, j]. De forma análoga ao conjunto de texto *lorem ipsum*, analisamos a performance do *ipmt* comparativamente ao *grep* de acordo com o tamanho do arquivo de entrada e com o tamanho do padrão.

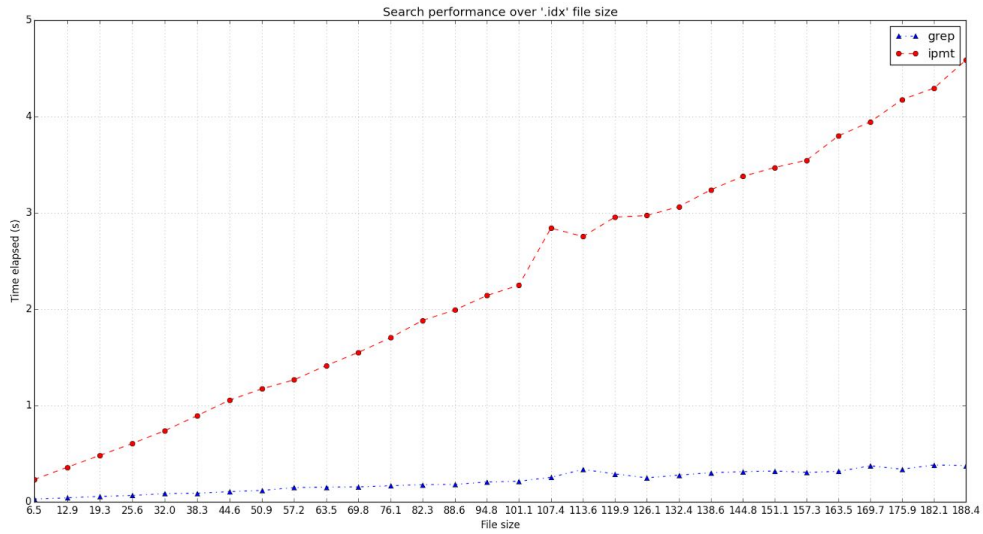

Figura 9: Gráfico da performance do *ipmt* e *grep* para busca de um padrão fixo (*abc*) em um texto de *strings* aleatórias em relação ao tamanho do arquivo de entrada.

Antes de analisar a performance de acordo com o tamanho do padrão, é interessante observar o número de ocorrências de um padrão em função do tamanho. Para calcular o desempenho em função do tamanho do padrão, foi selecionada uma string *s* de 200 caracteres e posteriormente era selecionada a substring s[1..t] como padrão. Reportamos o número de ocorrências:

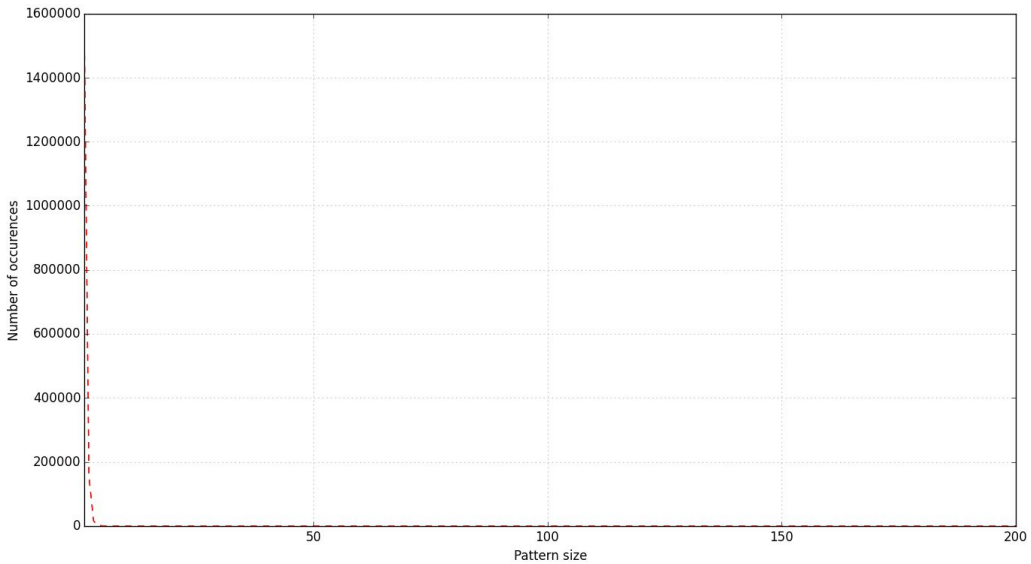

Figura 10: Relação entre um determinado padrão de tamanho [1…200] e o número de ocorrências do mesmo.

Foi observado que após t=7, todos os padrões tinham exatamente uma ocorrência no texto. Isso dá-se ao fato de como cada caractere tem probabilidade de 10% de aparecer, qualquer sequência específica longa ocasionaria uma probabilidade baixa de ocorrência. Analogamente, vimos que o padrão de tamanho 1 ocorria 1552442 vezes. Uma vez que o arquivo utilizado para esse teste continha 31,000 linhas, logo há 15,500,000 caracteres no texto. É fácil perceber que 1,552,442 é aproximadamente 10% de 15,500,000. Uma vez que um padrão de tamanho 1 seja um único caractere, é esperado que ele corresponda a aproximadamente 10% de todos os caracteres do texto. Analisamos então a performance do *grep* e do *ipmt* ao longo do tamanho do padrão:

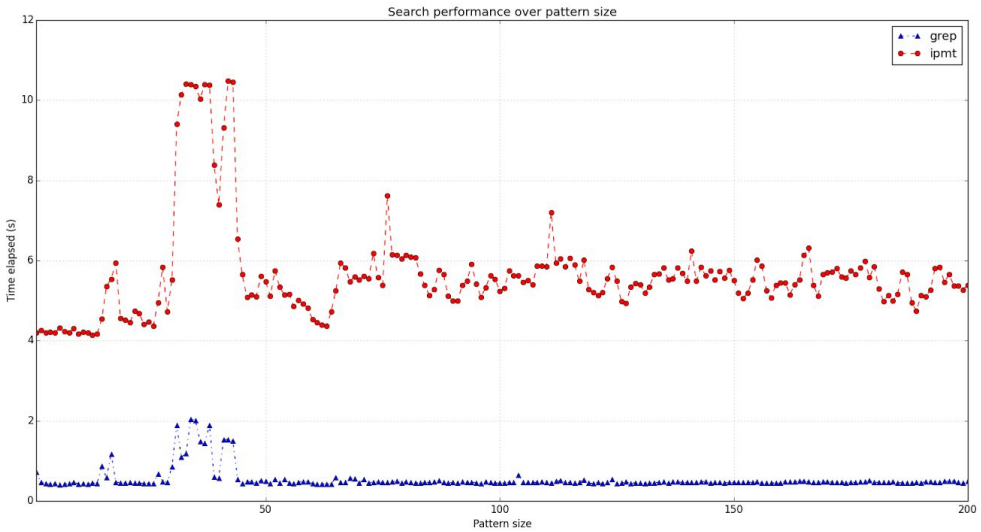

Figura 11: Efeito do tamanho do padrão na performance da busca em texto aleatório.

3.3 Compressão e descompressão

A fim de medir a eficiência do algoritmo de compressão (LZW) implementado, retiramos temporariamente a representação do array de sufixo do arquivo '.idx' gerado, uma vez que essa aumenta consideravelmente a quantidade de informação. Medimos a performance do algoritmo em termos de taxa de compressão e velocidade de compressão. Para avaliarmos melhor a performance do tempo de execução, desconsideramos as operações de IO e avaliamos apenas o tempo de compressão. Realizamos os testes nos mesmos arquivos utilizados nos testes acima.

3.3.1 Texto lorem ipsum

Foi comprimido 30 arquivos de texto de linguagem natural "*lorem ipsum*", com o tamanho variando de 1.5MB a 41MB. Cada arquivo foi comprimido utilizando a nossa implementação no *ipmt* do LZW. Analisamos a performance nesse tipo de arquivo em termos de espaço e tempo a seguir:

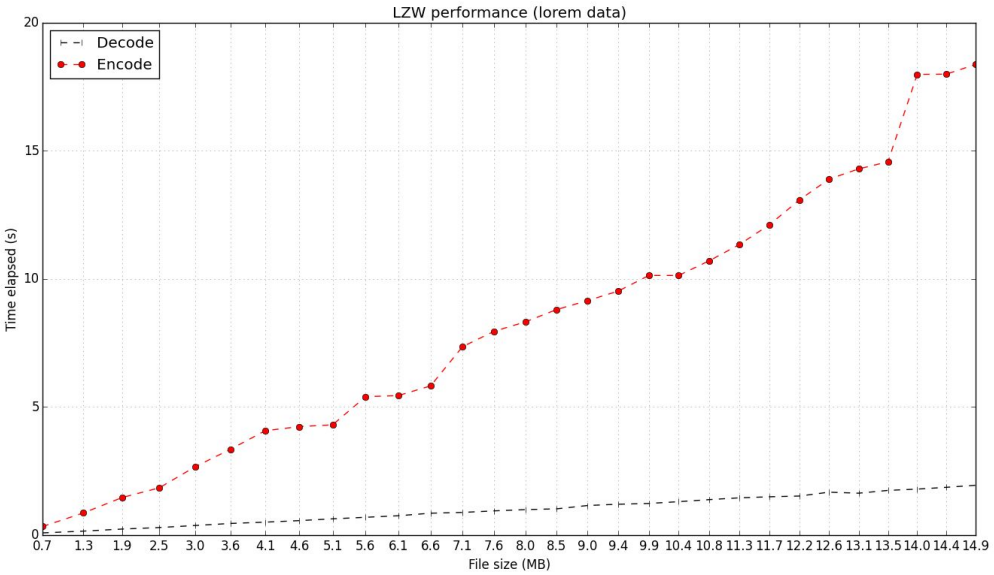

Figura 12: Desempenho (tempo) da nossa implementação do LZW no *ipmt* em arquivo gerado por texto *lorem ipsum*. Em vermelho o tempo necessário para comprimir e em preto o tempo necessário para descomprimir.

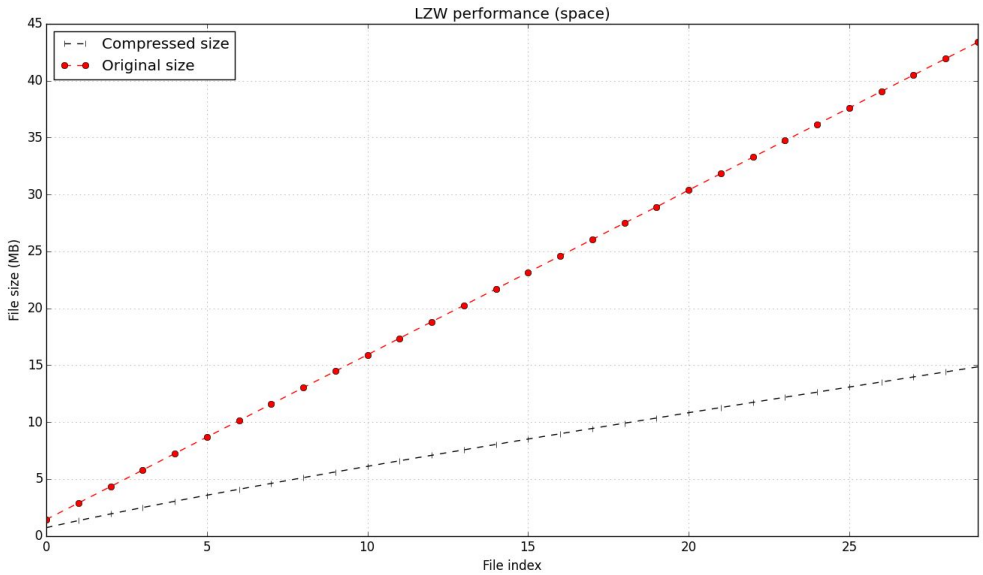

Figura 13: Desempenho (espaço) da nossa implementação do LZW no *ipmt* em arquivo gerado por texto *lorem ipsum*. Em vermelho o tamanho original do arquivo, em preto o tamanho do arquivo resultante.

3.3.2 Texto aleatório

Primeiro, foi comprimido 30 arquivos de texto aleatório, cujo alfabeto era [a … j], com o tamanho variando de 1MB a 15MB. Analisamos em seguida a performance de tempo e espaço do algoritmo nesse caso:

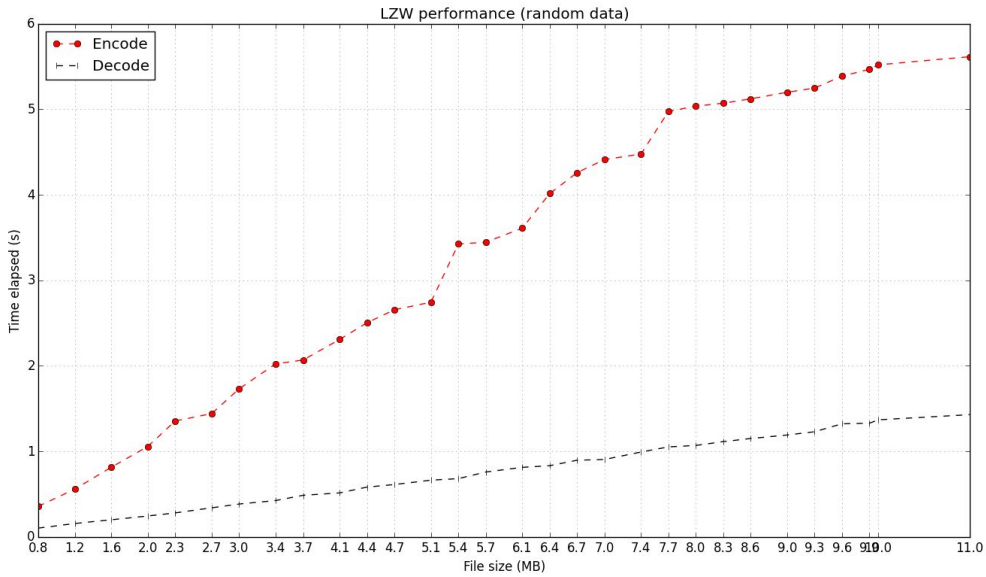

Figura 14: Desempenho (tempo) da nossa implementação do LZW no *ipmt* em arquivo de texto de *string* aleatórias. Em vermelho o tempo necessário para comprimir e em preto o tempo necessário para descomprimir.

É fácil perceber que o algoritmo está obtendo desempenho linear no tamanho do arquivo, o que é esperado.

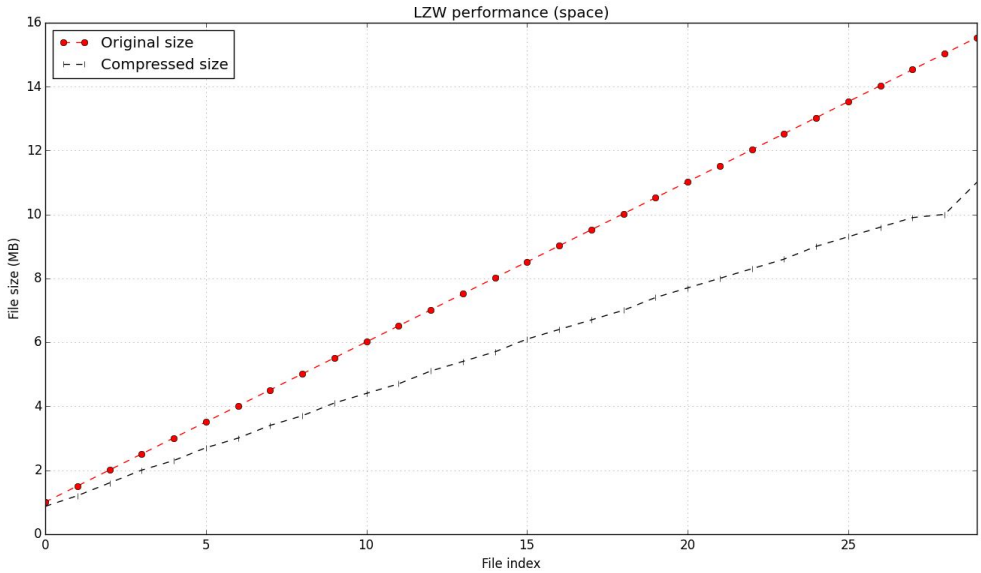

Figura 15: Desempenho (espaço) da nossa implementação do LZW no *ipmt* em arquivo de texto de *strings* aleatórias. Em vermelho o tamanho original do arquivo, em preto o tamanho do arquivo resultante.

Analogamente, a taxa de compressão obtida pelo algoritmo também é linear em espaço. Para o texto de caráter aleatório, vimos que a nossa implementação do algoritmo reduz o tamanho do arquivo original em aproximadamente 30% ao longo desses arquivos. Ao contrário de muitos algoritmos de compressão que não possuem boa performance em dados aleatórios (deixando os arquivos maiores que o original), vimos que o LZW é uma escolha eficiente para comprimir dados aleatórios.

3.4 Testes Gerais

Para finalizar a fase de testes, testamos a busca e a compressão e descompressão em um arquivo de linguagem natural em inglês de 200MB[5][6]. Esse tamanho foi escolhido devido ao tempo necessário para a criação do arquivo '.idx' (~1h10min incluindo todas as operações de IO), além das condições de memórias ('.idx' resultante de tamanho ~0.9GB). Comparamos o desempenho do *ipmt* com o *grep* para busca e com o *gzip* para a fase de compressão e descompressão.

Arquivo	Modo	ipmt	gzip
english.200MB	*encode*	1m51s	18s
	decode	48s	1s

Tabela 2: Comparação do desempenho entre *ipmt* e *gzip* num arquivo de linguagem natural (inglês) de 200MB.

Arquivo	Padrão	Ocorrências[7]	ipmt	grep
english.200MB	*she*	208,723	57.902s	6.249s
	thy	33,052	59.894s	6.170s
	countenance	2,736	55.576s	4.027s

Tabela 3: Comparação do desempenho entre *ipmt* e *grep* num arquivo de linguagem natural (inglês) de 200MB.

[5] O corpus foi obtido em http://pizzachili.dcc.uchile.cl/texts/nlang/. Foi usado "english.200MB" tanto para a busca como para compressão e descompressão. Os tempos medidos nessa tabela desconsideram operações de IO.
[6] O arquivo foi normalizado para remover caracteres indesejados ('\0', *non-ASCII*, etc.) com o comando 'LC_ALL=C sed -i "" "s|[^a-zA-Z0-9.,\!\?]| |g" english.200MB'.
[7] O número de ocorrências foi contado com "grep -o <pattern> <file> | wc -l". Os resultados estão de acordo com o da nossa ferramenta desenvolvida.

Referências

[1] Manber, Udi, and Gene Myers. "Suffix arrays: a new method for on-line string searches." *siam Journal on Computing* 22.5 (1993): 935-948.

[2] Ziv, Jacob, and Abraham Lempel. "A universal algorithm for sequential data compression." *IEEE Transactions on information theory* 23.3 (1977): 337-343.

[3] Welch, Terry A. "A technique for high-performance data compression." Computer 6.17 (1984): 8-19.