Relatório Ferramenta IPMT

1. Identificação

1.1 Equipe:

Davi Yan Macedo Brandão — *dymb*
Edson de Melo Neto — *emn2*

1.2 Descrição das contribuições dos membros da equipe:

- Davi:
    - Implementação da indexação e busca.
    - Implementação dos algoritmos de Suffix Array.
    - Otimização, correção e refatoração do código.
    - Desenvolvimento e execução de testes.
    - Escrita de documentação e relatório.

- Edson:
    - Implementação da compressão e descompressão.
    - Implementação do algoritmo de compressão LZ-77.
    - Otimização, correção e refatoração do código.
    - Escrita de documentação e relatório.

2. Implementação

2.1 Funcionamento Geral da Ferramenta e Opções

O ***ipmt*** é uma ferramenta de indexação, busca, compressão e descompressão em linha de comando, que possui sintaxe:

- *./**ipmt** type*

Onde *type* pode ser:

1) *index*: Indexa um arquivo ".txt" em um arquivo ".idx"
    - Possui a opção de escolha de algoritmo de indexação (padrão: "***DC3***"), e permite ao usuário decidir se o texto deve ou não ser salvo no arquivo ".idx" (padrão: "***False***"). Caso não seja, o texto é reconstruído na busca a partir do array de sufixos e a frequência dos caracteres no texto original.
2) *search*: Busca um padrão, ou um conjunto de padrões, em um arquivo ".txt"
    - Permite que o usuário decida se devem ser mostradas as ocorrências pelo texto, se múltiplos padrões em um arquivo ".txt" deverão ser processados e se o programa deve imprimir o número de ocorrências por padrão.
3) *zip*: Comprime um arquivo ".txt" em um arquivo ".myz"
4) *unzip*: Descomprime um arquivo ".myz"
5) *help*: Imprime a ajuda com informações sobre o comando e sua sintaxe.

2.2 *Algoritmos de indexação*

Decidimos implementar dois algoritmos de indexação: o algoritmo clássico por ordenação, de complexidade O(N logN), que chamaremos de ***"sort"*** e o algoritmo de complexidade linear, idealizado por Kärkkäinen, Sanders e Burkhardt [1], denominado ***"DC3".***

Iremos então detalhar a implementação de ambos.

2.2.1 Sort

O algoritmo "***sort"*** monta o array de sufixos através de *log2*(**n**) iterações, onde **n** é o tamanho do texto, sorteando par a par as posições dos sufixos. Para agilizar a ordenação dos pares, implementamos o *radix sort*, que ordena linearmente elementos de um intervalo conhecido. Apesar dessa otimização, o custo de tempo deste algoritmo é crítico, sendo incapaz de processar em tempo razoável textos com mais de 10^6 caracteres. Além disso, o consumo de memória do algoritmo é muito alto, e por isso alertamos que textos com mais de 10^7 caracteres **NÃO DEVEM** ser processados com esta opção.

2.2.2 DC3

O algoritmo "***DC3*"** tem uma complexidade linear e por isso demonstra uma performance extremamente superior ao "***sort***". As ideias principais por trás do algoritmo são:

1) Cálculo do array de sufixo das posições **i,** tais que **i** mod 3 != 0. É possível realizar esse passo recursivamente, unindo as posições em um texto com 2/3 do tamanho do original.
2) Montagem dos valores nas posições **i,** tais que **i** mod 3 = 0, através das posições montadas no passo 1)
3) Junção dos valores calculados em 1) e 2) no array de sufixos final.

O passo 1 é realizado calculando os vetores $\mathbf{R_1}$ e $\mathbf{R_2}$, onde $\mathbf{R_x}$[i] = (**T**[x:][3*i], **T**[x:][3*i+1], **T**[x:][3*i]). Concatenando $\mathbf{R_1}$ e $\mathbf{R_2}$, teremos um array de triplas cuja tripla na posição i corresponde à **T**[i:] para i mod 3 != 0. Ou seja, se todas as triplas forem únicas, podemos ordená-las e teremos a posição relativa no array de sufixos cada posição não divisível por 3. Se não forem, basta utilizar o algoritmo "***DC3*"** para recursivamente calcular as posições relativas. (Para uma string "acdef", como todos os caracteres são diferentes o array de sufixos pode ser calculado simplesmente ordenando os caracteres. A mesma ideia é utilizada no "***DC3*"**.)

No pior caso, o algoritmo irá realizar em torno de n + 2/3 n + 4/9 n .. = 3 n operações, tornando-o O(3n).

A implementação do *"**DC3"*** na ferramenta é inspirada na implementação dos próprios autores, com alterações condizentes com as necessidades da ferramenta, como o uso de estruturas de dados diferentes e otimização de partes do código.

2.3 *Algoritmo de busca*

Neste projeto, a implementação da busca pelas posições de ocorrência de um padrão num texto é realizada através de uma busca binária simples. Preferimos não implementar o algoritmo de busca com o vetor **lcp**, pois percebemos que éramos capazes de conseguir um desempenho equiparável apenas com heurísticas de busca. Assim, foi possível diminuir o tamanho dos arquivos de índice e evitar a leitura de mais dados. Como veremos nos testes, esses são justamente os gargalos do nosso programa.

Seja um padrão **p** tal que **len**(**p**) = **sz,** a busca binária do ***ipmt*** em um texto **T**, com array de sufixos **SA**, verifica, em cada iteração, se:

1) **T[SA[**m**]], T[SA[**m**] + 1] ... T[SA[**m**] + sz-1] == p[0], p[1] ... p[sz-1]**

Caso **1)** seja verdadeiro, encontramos uma ocorrência do padrão no texto. Para encontrar todas as ocorrências, buscamos a primeira posição da ocorrência mais à esquerda ***'l'*** e da ocorrência mais à direita ***'r'*** no texto com alguns truques de busca binária. Assim, temos que o número de ocorrências deve ser ***'r - l + 1'.***

Com essa ideia em mente, implementamos algumas heurísticas que agilizam a verificação e reduzem consideravelmente o tempo de busca. A mais importante, entretanto, tem a ver com uma propriedade do maior prefixo comum entre o padrão e a posição de busca na iteração:

Sejam l e r os intervalos de busca da busca binária, e $m = (l + r) // 2$, temos que:

2) $lcp(\mathbf{T}[\mathbf{SA}[m]:], \mathbf{p}) = \min(lcp(\mathbf{T}[\mathbf{SA}[l]:], \mathbf{p}),\ lcp(\mathbf{T}[\mathbf{SA}[r]:], \mathbf{p}))$

O que isso indica é que podemos ignorar uma certa quantidade de comparações entre **T**[**SA**[m]:] e p caso já tenhamos feito comparações com **T**[**SA**[l]:] e **T**[**SA**[r]:]. Ajustamos o algoritmo de busca binária para que isso aconteça com uma certa frequência, agilizando o processo de comparação.

2.4 Estruturas de dados utilizadas e observações

- Quanto a indexação:
  - O algoritmo "sort" **NÃO DEVE** ser utilizado em textos com mais de 10^7 caracteres.
- Quanto a busca:
  - Devido a maneira em que os arrays de sufixos são planejados, não é possível de maneira fácil imprimir (e buscar) os padrões linha-por-linha. Portanto, caso a opção "-c" não esteja ativada, a ferramenta imprimirá o texto até a última linha em que um padrão aparece.
- Quanto ao algoritmo *LZ-77*:
  - Limitamos a codificação em 3 bytes, utilizando:
    - 12 bits para o tamanho da janela do buffer de procura, ou seja, 4096 possíveis valores para posição de início do casamento
    - 4 bits para o tamanho da janela de "look ahead", ou seja, podemos encontrar padrões de tamanho até 16
    - 1 byte para o caractere de "mismatch"
  - Para a construção da máquina de estados, foi utilizado um "map<pair<int, char>, int>" para representar os estados.
  - Observamos que o tamanho do buffer de procura (search buffer) e do "look ahead" buffer, interferem no tempo de execução e na qualidade da compressão, já que limita o tamanho da cadeia a ser procurada e a distância em que essa cadeia pode ser buscada, por isso decidimos utilizar os limites apontados anteriormente.

3. Testes e Resultados

A ferramenta foi testada com diferentes arquivos de entrada de tamanhos variados. Para análise, decidimos testar arquivos contendo sequências de DNA, onde índices são mais comumente utilizados pelo alfabeto curto (tamanho 4) com alta repetição, e arquivos com textos em inglês, por se tratar do exato oposto, um alfabeto longo (tamanho 128) e pouca repetição.

Ambos os conjuntos de textos estão disponíveis para download em http://pizzachili.dcc.uchile.cl/texts/dna/ e http://pizzachili.dcc.uchile.cl/texts/nlang/,

respectivamente. Além disso, as ferramentas gzip e grep também foram utilizadas nos testes para comparação do desempenho com o algoritmo de LZ-77.

3.1 Compressão e Descompressão

3.1.1 Quanto ao tempo de compressão

Arquivos contendo sequências de DNA

Tamanho do arquivo de texto	**4.5kB**	**45kB**	**450kB**	**4.5MB**
LZ-77	0.238s	2.867s	25.689s	259.506s
gzip	0.001s	0.001s	0.001s	0.16s

Arquivos contendo textos em inglês

Tamanho do arquivo de texto	**4.5kB**	**45kB**	**450kB**	**4.5MB**
LZ-77	0.45s	3.962s	43.802s	415.733s
gzip	0.001s	0.002s	0.017s	0.097s

Observou-se que o tempo de compressão do algoritmo LZ-77 foi bem maior em relação ao gzip. Isso se deve ao tamanho do "search buffer" e do "look ahead buffer", que foram escolhidos visando a melhor qualidade da compressão, em detrimento do tempo de execução do algoritmo. Além disso, o tempo de execução foi bastante afetado devido à utilização das estruturas *"std::strings"* na implementação do algoritmo, decisão tomada com o objetivo de facilitar a implementação.

Ademais, quando comparamos o tempo de execução do arquivo contendo sequências de DNA em relação ao tempo de execução do arquivo contendo textos em inglês, podemos notar que o primeiro tipo tem um tempo de execução menor, em razão de que os textos em inglês têm uma menor repetição de padrões quando comparados às sequências de DNA.

3.1.2 Quanto à porcentagem de compressão

Arquivos contendo sequências de DNA

Tamanho do arquivo de texto	**4.5kB**	**45kB**	**450kB**	**4.5MB**
LZ-77	2.6 kB	24.8 kB	246.9 kB	2.4 MB
gzip	1.4 kB	13 kB	128.2 kB	1.3 MB

Arquivos contendo textos em inglês

Tamanho do arquivo de texto	4.5kB	45kB	450kB	4.5MB
LZ-77	4 kB	34.1 kB	358.8 kB	3.5 MB
gzip	2.1 kB	17.0 kB	171.4 kB	1.7 MB

Observou-se que o algoritmo *LZ-77* obteve um tamanho comprimido de aproximadamente 50% dos arquivos originais contendo sequências de DNA, enquanto o gzip obteve um tamanho comprimido de aproximadamente 30% do arquivo original.

Já para os arquivos contendo textos em inglês, podemos notar que tanto o *LZ-77* quanto o gzip, tiveram uma queda nos seus desempenhos. Isso se deve ao fato de que esse tipo de arquivo, apresenta uma menor taxa de repetição de padrões, então se torna mais difícil a compressão. Para esses tipos de arquivos, o *LZ-77* obteve em média um tamanho comprimido de 80% do arquivo original, enquanto que o gzip obteve em média um tamanho comprimido de 40% do arquivo original.

3.1.3 Quanto ao tempo de descompressão

Arquivos contendo sequências de DNA

Tamanho do arquivo de texto	4.5kB	45kB	450kB	4.5MB
LZ-77	0.004s	0.026s	0.225s	2.531s
gzip	0.001s	0.001s	0.001s	0.001s

Arquivos contendo textos em inglês

Tamanho do arquivo de texto	4.5kB	45kB	450kB	4.5MB
LZ-77	0.005s	0.038s	0.432s	3.825s
gzip	0.001s	0.001s	0.001s	0.001s

Como esperado, quando comparamos os tempos de descompressão dos arquivos contendo sequências de DNA com os arquivos contendo textos em inglês, podemos perceber que os arquivos do primeiro tipo tiveram um tempo de descompressão um pouco menor quando comparados com os arquivos contendo textos em inglês, pois como visto anteriormente, a compressão dos arquivos contendo textos em inglês resulta em arquivos com tamanhos maiores em relação aos arquivos contendo sequências de DNA.

3.2 Indexação

3.2.1 Quanto ao tempo de indexação

Arquivo contendo sequências de DNA

Tamanho do arquivo de texto	**4.5 kB**	**45 kB**	**450 kB**	**4.5 MB**
Tempo de Exec. Algoritmo "DC3"	0.002s	0.076s	0.069s	3.797s
Tempo de Exec. Algoritmo "Sort"	0.004s	0.091s	2.254s	56.22s
Tempo de Exec. Algoritmo "DC3", Com a flag "-s"	0.002s	0.077s	0.068s	4.185s
Tempo de Exec. Algoritmo "Sort", Com a flag "-s"	0.009s	0.067s	2.137s	60.02s

Como previsto, o algoritmo "***DC3*"** demonstrou uma performance superior ao algoritmo ***"sort"*** em praticamente todos os casos, evidenciando sua efetividade. Podemos observar também que a diferença entre o tempo de ordenação não é tão afetado com o salvamento do texto junto com o array de sufixos, chegando a no máximo 0.5ms de diferença. Obviamente, essa diferença escala linearmente de acordo com a ordem de grandeza das entradas, mas no intervalo em que consideramos razoável a execução da ferramenta, pode-se dizer que essa diferença é desconsiderável.

3.2.2 Quanto ao tamanho dos índices

Tamanho do arquivo de texto	**4.5 kB**	**45 kB**	**450 kB**	**4.5 MB**	**45 MB**
Tamanho do arquivo de índice	18.5 kB	180.5 kB	1.8 MB	18 MB	180 MB
Tamanho do arquivo de índice, com a flag "-s"	22.5 kB	225 kB	2.3 MB	22.5 MB	225 MB
Tamanho do arquivo de texto	4 kB	45 kB	450 kB	4.5 MB	45 MB
Tamanho do arquivo de índice	18.5 kB	180.5 kB	1.8 MB	18 MB	180 MB

Tamanho do arquivo de índice, com a flag "-s"	22.5 kB	225 kB	2.3 MB	22.5 MB	225 MB

O tamanho do arquivo de índice é aproximadamente 4 vezes o tamanho do texto, e aumenta em 1.25x caso o usuário escolha salvar o texto no índice. O resultado já era esperado, pois o tipo de dado utilizado nos índices *(int)* é 4 vezes maior que o dado utilizado nos textos *(char)*.

3.3 Busca

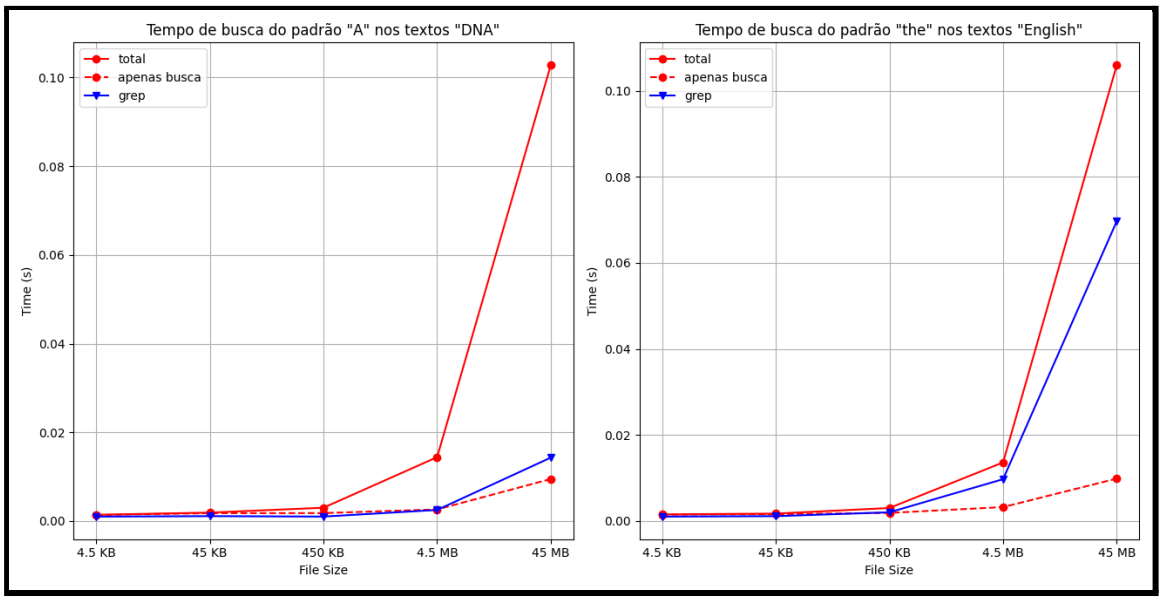
3.3.1 Quanto ao tempo de carregamento dos índices

Tamanho do arquivo de texto	4.5 kB	45 kB	450 kB	4.5 MB	45 MB
Tempo total de busca, Índice padrão	0.001s	0.002s	0.004s	0.042s	0.865s
Tempo de preparação Índice padrão	0.00003s	0.0002s	0.002s	0.037s	0.852s
Tempo total de busca, Índice com texto	0.002s	0.002s	0.003s	0.021s	0.192s
Tempo de preparação Índice com texto.	0.00003s	0.0002s	0.001s	0.016s	0.175s

Neste teste, buscamos evidenciar o maior gargalo da implementação da busca: o carregamento dos índices e a preparação dos textos. É possível notar que, para arquivos de tamanho cuja ordem $>10^6$, o tempo de carregamento do índice ocupa uma larga porção do tempo total de execução da busca, com a busca de fato ocorrendo em pouquíssimo tempo (se comparado ao tempo total). Caso o texto não esteja salvo no índice, é necessário reconstruí-lo a partir do array de sufixos e um vetor com a frequência de cada caractere no texto original. Essa operação também se mostrou extremamente custosa, como é possível perceber na tabela, pela diferença entre o tempo de execução com e sem o texto salvo.

Pelos motivos demostrados nos testes 3.2.1 e 3.3.1, nos testes seguintes as buscas foram realizadas em índices que mantinham o texto salvo, com "tempo total" demonstrando o tempo de execução da ferramenta e "tempo de busca" demonstrando a diferença entre o tempo total e o tempo de carregamento do índice.

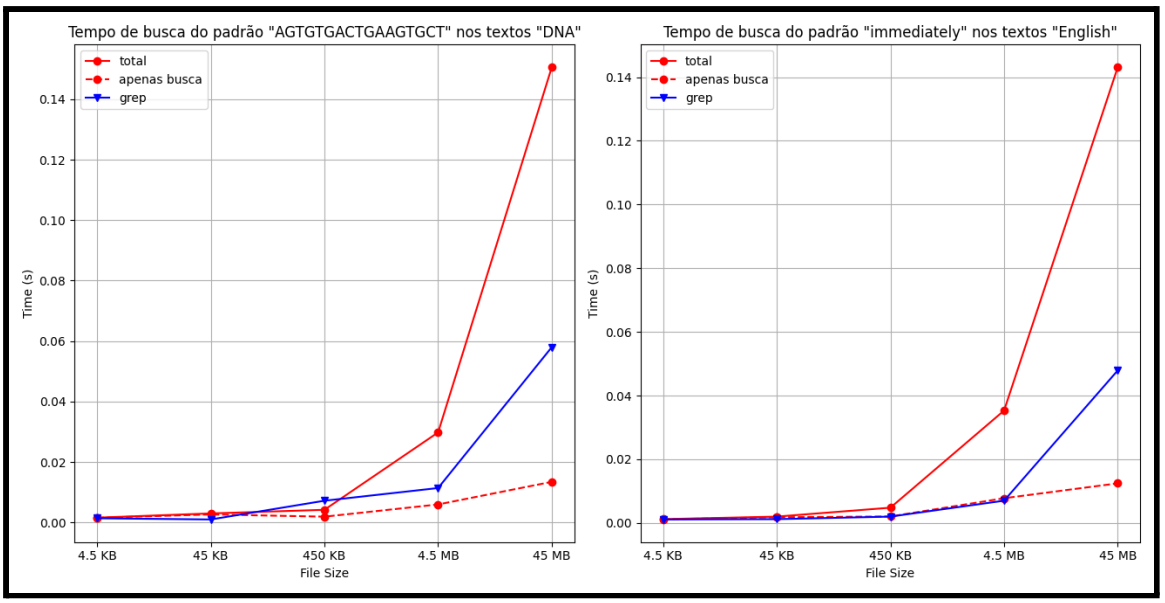
3.3.2 Quanto ao tempo de busca de padrões curtos



Com esse teste, fomos capazes de demonstrar que o tempo de busca no índice corresponde à expectativa de tempo, pois demonstra realizar um número menor ou igual de operações que a ferramenta “grep”, eventualmente tendo performance superior. Entretanto, o tempo de carregamento do índice ainda se mostra um enorme problema para que essa vantagem seja aproveitada.

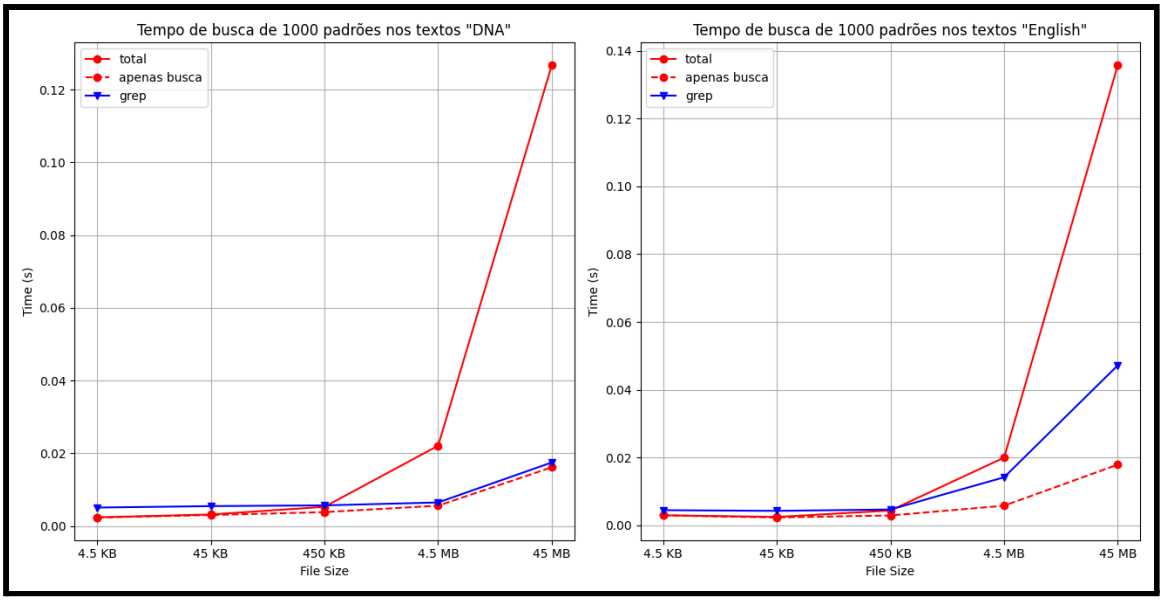
Escolhemos padrões pequenos de alta repetição (“A” e “the”, respectivamente) para garantir que a execução ocorresse corretamente.

3.3.3 Quanto ao tempo de busca de padrões longos



Um resultado proporcional ao que obtivemos anteriormente é visível nesse teste. Mesmo com o tamanho do índice aumentando em ordens de grandeza, o tempo de busca permanece basicamente o mesmo, pois é praticamente linear com o tamanho do padrão.

3.3.4 Quanto ao tempo de busca de vários padrões



Aqui, realizamos a busca de 1000 padrões (no texto DNA, de tamanho e conteúdo aleatório. no texto em inglês, as 1000 palavras mais frequentes do texto). Notamos pouquíssima diferença prática com os resultados adquiridos nos 2 testes anteriores. Como esperado, o tempo de carregamento do índice ainda se mostrou o maior gargalo da execução, e o tempo de busca continuou consideravelmente menor. Porém, pudemos ver uma diferença menor entre o tempo de busca e o tempo total do “grep”. Imaginamos que isso se dê graças ao algoritmo, provavelmente linear no tamanho do texto somado com o tamanho dos padrões, executado pelo “grep”.

4. Conclusões

Neste trabalho, vimos a importância dos algoritmos de indexação, e o ganho bruto em desempenho que eles podem proporcionar na busca por padrões em textos fixos. Pudemos experimentar algoritmos diferentes para indexação e medir seu desempenho. Notamos que é crucial elaborar estratégias que minimizem o tempo de carregamento do índice, assim como o quão efetivo podem ser heurísticas de auxílio na busca binária. Concluímos que uma compressão de alta qualidade é custosa em termos de tempo, nas implementações mais canônicas dos algoritmos.

Acreditamos que a ferramenta atingiu os objetivos propostos e esperados, de acordo com as implementações.

5. Referências

[1] **Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt.** 2006. Linear work suffix array construction. *J. ACM* 53, 6 (November 2006), 918–936. https://doi.org/10.1145/1217856.1217858