UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CIÊNCIAS DA COMPUTAÇÃO

Projeto 2 - Relatório

PEDRO HENRIQUE TÔRRES SANTOS (phts)

ULLAYNE FERNANDES FARIAS DE LIMA (uffl)

Recife, Pernambuco

2018

1. IDENTIFICAÇÃO

O projeto foi desenvolvido por Pedro Henrique Tôrres Santos e Ullayne Fernandes Farias de Lima. A implementação dos algoritmos foi divida tal que cada integrante ficasse com uma das partes do projeto. Portanto, Pedro implementou o algoritmo de indexação, o array de sufixo, e Ullayne implementou o algoritmo de compressão, o lz78. Adicionalmente, Pedro implementou o algoritmo de compressão lz77 com objetivo de fazer uma análise desempenho e comparar ao lz78 em tempo de execução e taxa de compressão. Em outras etapas do projeto, houve contribuição de ambos os integrantes.

2. IMPLEMENTAÇÃO

A ferramenta desenvolvida possui dois módulos: o módulo de indexação, que tem como objetivo deixar a busca no texto mais rápida e o módulo de busca.

O módulo de indexação recebe um arquivo de texto como entrada e este arquivo passará por dois processos, o primeiro vista indexar o texto utilizando o array de sufixo, além disso, os índices são comprimidos antes de serem gravados.

O módulo de busca recebe um padrão como entrada e o arquivo de índices. Para recuperar as ocorrências do padrão na entrada do módulo o arquivo é descomprimido e desserializado, recuperando-se o array de sufixos juntamente com o L-LCP e o R-LCP.

A seguir, será descrito detalhes de implementação dos algoritmos utilizados para desenvolver a ferramenta.

2.1. OTIMIZAÇÕES

Para otimizar as operações em string, foi reimplementado o tipo string_view (presente em C++ 17 mas ausente até C++ 14), que é uma janela sob uma posição de memória. Ela guarda uma weak-reference para a posição de início da memória e seu comprimento em bytes. Isto permite operações como substr em tempo constante e com espaço adicional constante de 2 palavras da arquitetura do processador.

Foram usados streams otimizar a para leitura e escrita de arquivos e passagem de bytes de um módulo para outro da aplicação. Onde se faz necessário escrita ou cópia de espaços largos na memória, utiliza-se leitura e escrita na memória em blocos, agilizando o processo.

Já a serialização foi feita através de um dump de blocos de memória dos vetores que contêm Suffix-Array, L-LCP, e R-LCP e da string que guarda o texto. O que faz com que o processo seja o mais rápido possível, O(n) onde n é a quantidade de bytes a ser feito dump. Esse método impacta na portabilidade do arquivo serializado, que requer que o endianness do processador que serializou o arquivo seja o mesmo que o do que irá deserializa-lo. E outras linguagens de programação podem não se beneficiar da mesma forma que C e C++, podendo ser necessário um parse mais complexo da informação.

Por fim foi imposta uma delimitação para o tamanho máximo indexável pelo nosso programa, redefinindo size_t para unsigned int, o que força o programa a usar inteiros de 32 bits ao invés de 64 bits. Isso se mostrou bem eficiente, cortando pela metade o tempo de execução de certos módulos do código. Ao custo de limitar-se a entrada do programa a menos de 4GB.

2.2. ARRAY DE SUFIXOS

Inicialmente foi utilizado os algoritmos vistos em sala para indexação do texto (criação do array de sufixo e computação do L-LCP e R-LCP). Após alguns testes achou-se necessário otimizar a computação do L-LCP e R-LCP, pois em arquivos maiores estava-se havendo o estouro da stack.

A otimização da computação do L-LCP e R-LCP foi feita através da criação do chamado H-LCP, utilizando o algoritmo de Kasai[1], que é pré-computado. O H-LCP é um array que dado uma posição I, retorna o LCP entre o sufixo na posição I e o sufixo na posição I + 1. Assim podemos calcular o L-LCP de H como o mínimo de H-LCP no intervalo [L, H), para o L relativo a H. E o R-LCP de H é o menor LCP em H-LCP no intervalo [H, R), com o R relativo a H.

[1] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM '01), Amihood Amir and Gad M. Landau (Eds.). Springer-Verlag, London, UK, UK, 181-192.

Em seguida foi utilizado o algoritmo DC3[2] que permite a construção do array de sufixos em tempo linear, que mostrou diferença significativa em tempo de execução mesmo com arquivos de apenas 1MB, quando comparado ao algoritmo base que possui tempo $O(n \log^2(n))$. Este algoritmo foi usado apenas para propósitos de comparação, a implementação se baseia no artigo que o descreve, que apesar de ter sido otimizado, não foi possível compreender a fundo seu funcionamento.

2.3. LZ77

O LZ77 foi implementando da sua forma tradicional, tendo como base o algoritmo mostrado em sala. A janela de busca foi ajustada de tal forma que tenha tamanho 2^9, enquanto o look-ahead tem tamanho 2^7. Os tamanhos foram escolhidos tendo em vista que o look-ahead não necessita um tamanho grande em human-written texts e que uma grande janela de busca impacta significativamente no tempo de execução.

O tamanho do átomo comprimido é de 24 bits (3 bytes), sendo 9 bits para armazenar a posição na janela de busca POS, 7 bits para o cumprimento do look-ahead LEN, e 8 bits para armazenar o char emitido pelo algoritmo na posição POS + LEN. Podendo-se ajustar esse valor em tempo de compilação através do template, tendo como única limitação que o número de bits de WIN e LAS seja múltiplo de 8 e menor que o tamanho de palavra da arquitetura do processador.

2.4. LZ78

O algoritmo de compressão lz78 utiliza um dicionário para armazenar os padrões já vistos pelo algoritmo e para cada padrão encontrado mapeia para uma chave única. No algoritmo implementado pela dupla, utilizou-se uma árvore de prefixos (*trie*) onde, inicialmente, cada nó possui um carácter e um índice, o índice representa a chave do padrão que é armazenada no último caractere do padrão de entrada.

---

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

Para tornar a busca e a inserção de elementos mais rápida, a árvore foi modificada para cada nó é representado por um array de tamanho 256 e um índice, o objetivo de modificar foi que cada caracter fica na posição do seu valor na tabela ASCII. Então, para verificar se um caractere está na relação dos filhos de cada nó, basta olhar se a posição da tabela ASCII dele é diferente de *nullpointer*, caso for, ele não é filho. Esta otimização evita olhar para todos os filhos de um nó no momento da inserção e busca.

A codificação do algoritmo foi desenvolvida utilizando número fixo de bits. Para par inserido no dicionário é adicionado no arquivo de saída do método de compressão: número + char e cada par é utilizado vinte e quatro bits para representar, sendo dezesseis bits para os números e oito para cada char. Por exemplo, a representação do par (0, a) é feita da seguinte maneira:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1

Com a representação feito em bits, gerou-se um problema relacionado a memória porque quando o índice do dicionário atinge 2^{16} é necessário refazer o dicionário. Inicialmente, a trie era inicializada novamente, contudo em testes com um arquivo de 50MB foram gastos 20 GB de memória ram. Por este motivo, foi adicionado a cada nó da árvore um valor que indica em qual versão do dicionário está sendo trabalhada, por exemplo, quando o índice atinge a primeira vez 2^{16} então a versão do dicionário é atualizada para 1 e todos os nós que forem inseridos a partir deste momento é adicionado o valor 1, junto com cada par e se o caractere já estiver no nó quando estiver inserido, atualiza apenas a versão do dicionário. Esta otimização ajudou no tempo de execução da compressão e no consumo de memória.

O tempo gasto pelo algoritmo de codificação é em sua maioria pela inserção e busca na árvore, onde cada um gasta O(n) sendo n o tamanho do padrão já que a escrita é feita byte-a-byte e mesmo retirando esta etapa o tempo de execução reduz pouco.

Para fazer decodificação, não é necessário reconstruir a trie, é utilizado um array estático de tamanho 2^{16} e para cada par, o padrão é inserido na posição do índice e a cada padrão que referencia um antigo, a string inserida no array é a junção das antigas.

O tempo de execução da decodificação é dado pela leitura do arquivo que é feita a cada 3 bytes e já são processados e adicionados no array, por isso, o tempo de decodificação fica O(n), sendo n o tamanho do arquivo de entrada.

3. TESTES E RESULTADOS

A análise de desempenho da ferramenta foi medida através de teste de performance, a dupla buscou analisar o tempo de execução tanto do módulo de busca como do módulo de indexação. Os testes foram realizados em um notebook com o sistema operacional macOS Majove com processador Intel Core i7, memória de 16GB 1600MHz DDR3 e flash storege.

A seguir, será listado os testes de performance feitos. Os testes foram desenvolvidos utilizando o lz77 e lz78.

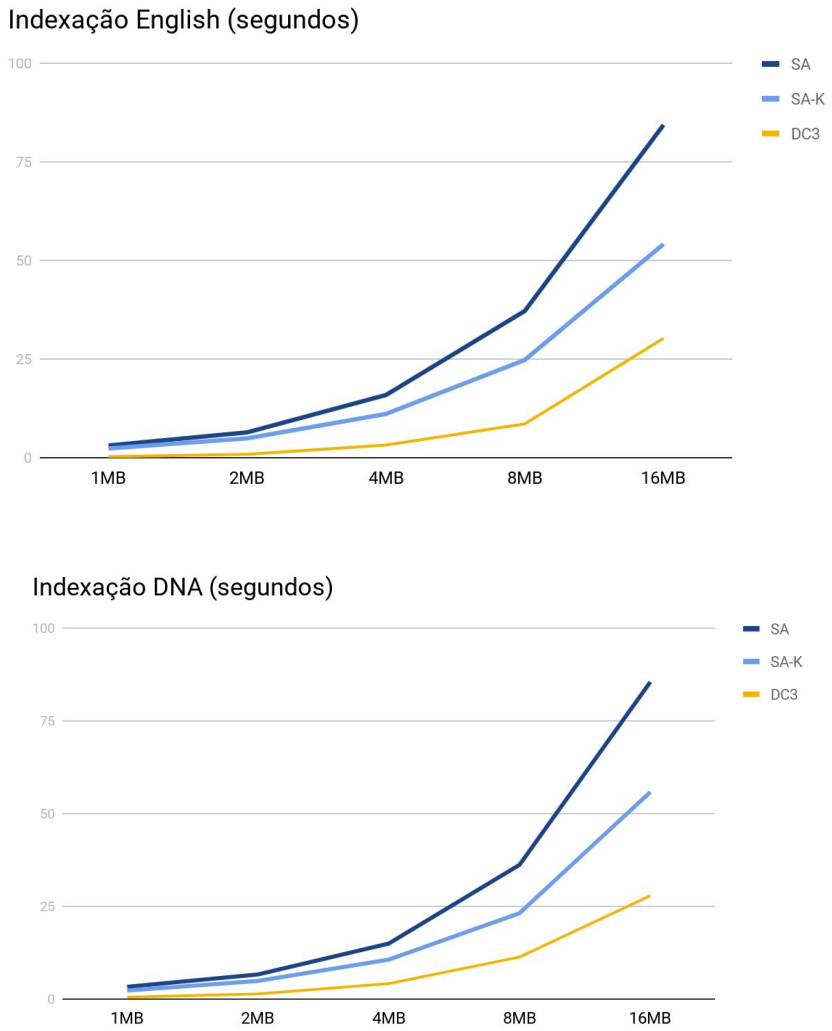
3.1. Tempo de execução vs Tamanho da Entrada.

A seguir, será analisado os desempenhos dos algoritmos: construção do array de sufixo, tempo de compressão e descompressão, além do tempo de execução do módulo de busca. É importante ressaltar que o tempo de execução do módulo de busca está na base 10^{-5} .

Os testes realizados abaixo foram feitos utilizando a base de dados english e DNA, onde foram divididos em tamanhos diferentes: 1MB, 2MB, 4MB, 8MB e 16MB. Para analisar o desempenho do módulo de busca foi utilizada a *string "unworthily"*.

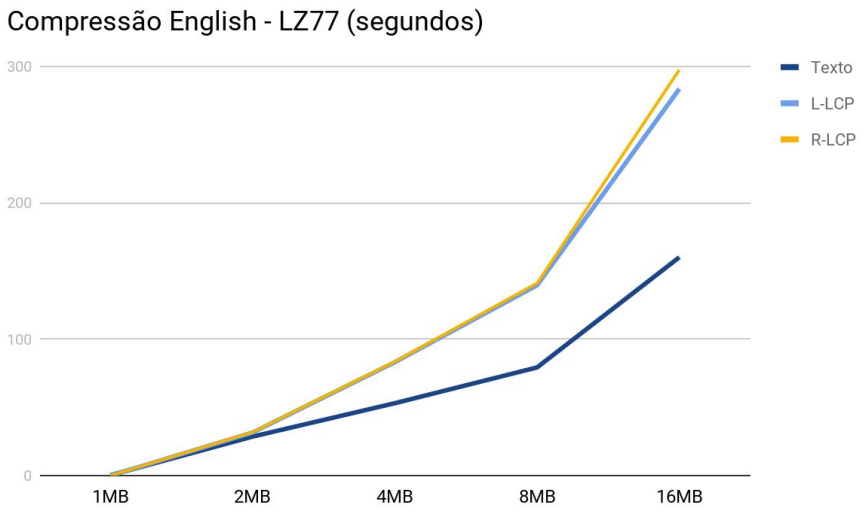
A dupla quando foi desenvolver os testes nas bases citadas acima, observou que o lz78 obteve um problema com a descompressão os arquivos lcp e rlcp, contudo achou-se válido manter os testes realizados nos textos a fim de comparar com o lz77, mas o algoritmo de compressão ofical da ferramenta se manteria o lz77.

3.1.1. Tempo de Execução do Array de Sufixo



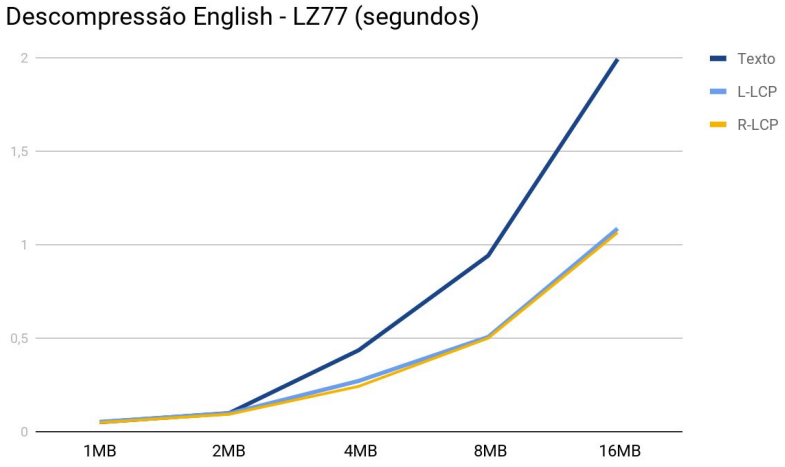
Na figura acima é possível observar que o tamanho do arquivo de fato influencia no tempo de execução dos algoritmos, porém é possível observar a diferença no desempenho dos algoritmos SA e DC3, pois é possível ver claramente que a curva do SA é bem mais acentuada em comparação ao DC3.

3.1.2. Tempo de Execução da Compressão do LZ77

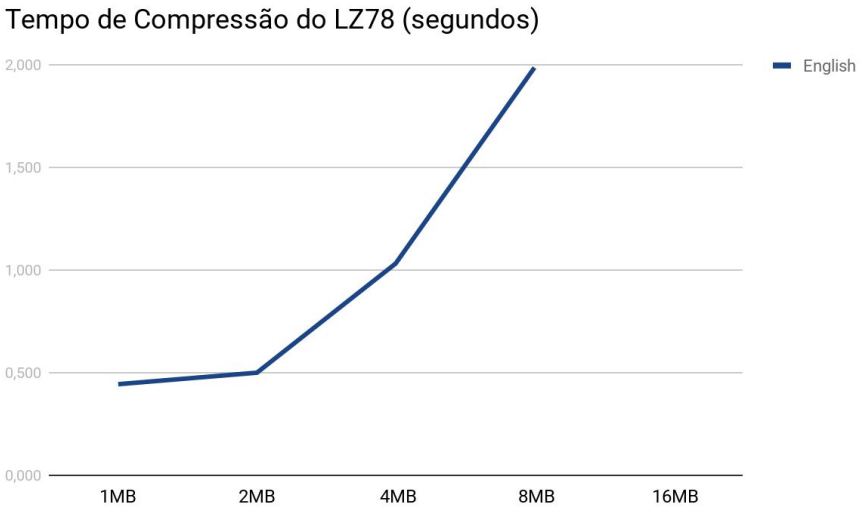


É possível observar que o tempo de compressão do lz77 é grande em relação ao tempo dos demais algoritmos da ferramenta.

3.1.3. Tempo de Descompressão do LZ77

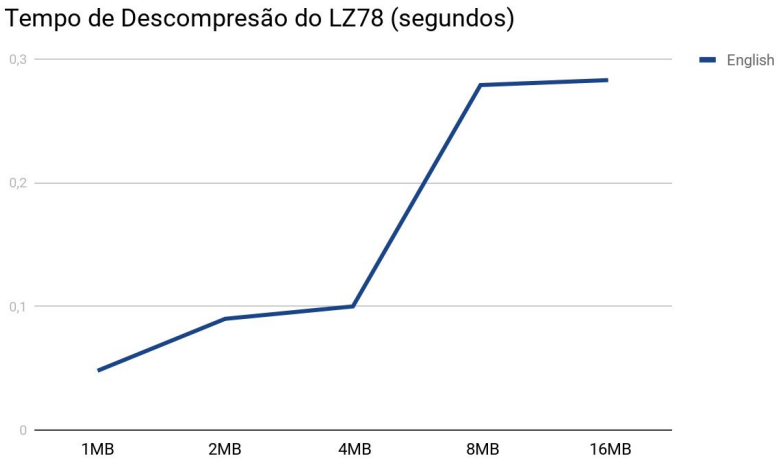


3.1.4. Tempo de Execução da Compressão do LZ78



É possível observar que o tempo de compressão do lz78 é menor em comparação ao tempo do lz77, apesar de ser compreensível pois o algoritmo é mais simples.

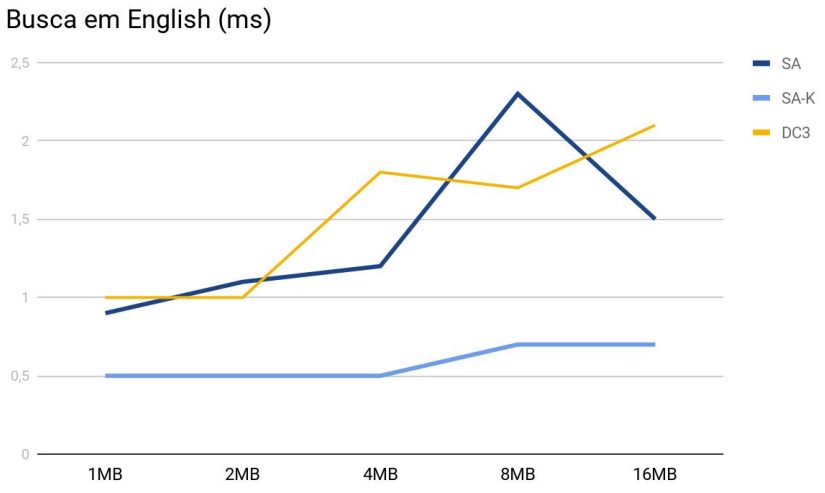
3.1.5. Tempo de Descompressão do LZ78

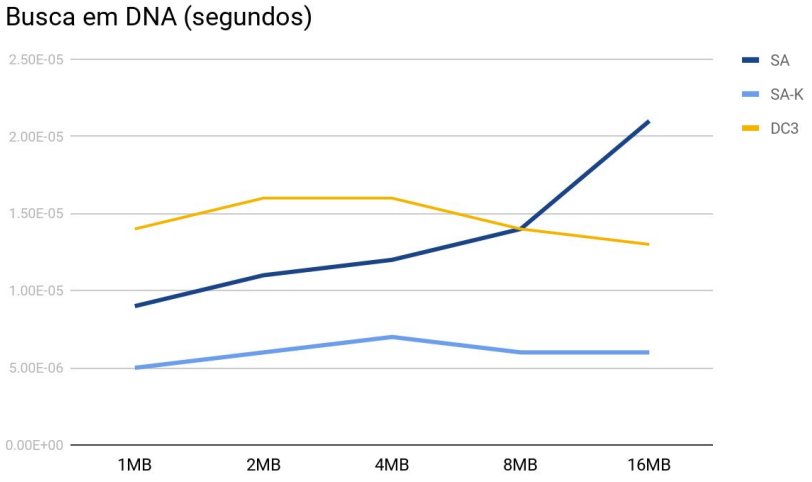


O tempo de descompressão do algoritmo lz78 é muito pequeno e varia pouco mesmo com o tamanho arquivo dobrando, como é possível observar entre os arquivos de 8MB e 16MB. Este

fato é explicável porque o algoritmo necessita apenas passar pelo arquivo comprimido uma vez e consegue decodificar para o texto original.

3.2. Tempo de execução na busca





4. CONCLUSÃO

Com base nos resultados obtidos acima, é possível observar que o tempo da compressão do algoritmo lz77 se torna o tempo mais significativo da ferramenta. Utilizando outro algoritmo de compressão, o array de sufixo é uma boa opção, contudo o tempo de execução ainda é bastante expressivo, como é possível observar no gráfico do tópico 3.1.1. Foi-se possível observar,

também, que o tempo de execução do lz78 é bastante interessante e sua participação tornaria a ferramenta mais rápida, onde apenas o array de sufixo o limite de tempo.