Universidade Federal de Pernambuco
Centro de Informática
Processamento de Cadeias e Caracteres

Relatório do Projeto 2

Aluno:	Arnaldo Rafael Morais Andrade	Data: 09/12/2018
	Marvson Allan Pontes de Assis	

1) Identificação

Nosso grupo inclui **Arnaldo Rafael Morais Andrade** e **Marvson Allan Pontes de Assis**. Arnaldo cuidou da parte do vetor de sufixos e interface, já Marvson, ficou com a compressão/descompressão e os testes.

2) Implementação

2.1) Funcionamento da Ferramenta

O uso é bastante simples, consiste de 4 partes independentes. Primeiramente faz-se um vetor de sufixos, a partir dele e do texto a ser procurado, construímos um arquivo .idx baseado na compressão LZ78. Para o modo de busca, fazemos o caminho inverso até termos os valores correspondidos pelo vetor de sufixo, ou seja, 3 vetores auxiliares e o próprio texto. Então é realizada a busca do padrão ao longo do texto, como visto em sala, foi utilizado do LCP-LR.

2.2) Detalhes Relevantes

a. Estratégias de Leitura

A leitura foi feita linha por linha, não foram feitas otimizações adicionais aqui.

b. Convenções

Usamos namespaces em vez de classes para ficar menos complexo e não precisar ficar instanciando classes. Qualquer estado é mantido pelo ipmt.

Escolhemos usar o namespace completo sempre e evitar o uso de “using namespace = …”, exceto quando isso prejudica a leitura.

c. Indexação

A fim de manter os vetores correspondentes do array de sufixos e o próprio texto, criamos uma string contendo os 4 principais elementos, separados por “$”. Foram concatenados um por um e o elemento final seria o próprio texto. A partir dessa nova

variável, criamos o arquivo .idx.

d. Compressão e descompressão

O algoritmo de compressão e descompressão escolhido foi o LZ78. Duas versões do mesmo foram implementadas para fins de comparação. Ambas são baseadas no código visto em sala. A primeira versão utiliza o *map* padrão, enquanto que uma *trie* foi implementada para a segunda versão. A ideia é comparar a velocidade de cada versão no geral. A *trie* foi implementada utilizando ponteiros na forma *shared_ptr*, que inclui *garbage collector* do tipo que utiliza a contagem de dependências do ponteiro, e quando chega a zero efetua o *destroy*, também é interessante por ser capaz de lidar facilmente com *Nulls* por padrão, de forma que nem precisamos se preocupar com essas coisas.

Uma preocupação era a maneira como percorrer a *trie*. Inicialmente foi feito com uma recursão, mas havia a preocupação de estourar a pilha, então modificamos para usar iteração mesmo. Algumas otimizações quanto a declaração de variáveis fora de laços e o uso de valores auxiliares foram conduzidas. A expectativa é que a compressão e descompressão pela *trie* seja mais rápida.

e. Busca

Uma vez o arquivo .idx descompressado, podemos fazer a engenharia reversa para obter os vetores, úteis para busca e o texto. Basta percorrermos a super string e fazemos o processo de volta a partir do caractere conhecido “$”. Com os itens em mãos, foi realizado a busca no vetor de sufixos, com o auxílio do LCP. Auxílio esse que reduz a forma que é feita busca binária do padrão ao longo do texto. Não foram efetuadas mudanças em relação ao algoritmo visto em sala.

2.3) Limitações

Além de nossa má organização, já que em prática, tivemos um final de semana para fazer o projeto, tivemos diversos bugs até então desconhecidos. Marvson estava numa máquina Windows enquanto Arnaldo, numa Linux. Os resultados mostraram-se diferentes quando rodado em sistemas diferentes e não sabíamos o porquê.

Infelizmente, descobrimos tarde que é por conta de um overflow causado no Linux, por escolhas de variáveis do tipo *char** ao invés de *std::string*. Não havia tempo para troca, então decidimos usar o LZ78 Trie como base, impossibilitante demais testes.

3) Testes e Resultados

3.1) Metodologia

Para comparação, usamos os dados presentes no repositório SMART (https://github.com/smart-tool/smart/tree/master/data) especificamente os textos em inglês e o arquivo que descreve o D.N.A. de uma bactéria E. Coli. As comparações foram feitas manualmente e executadas com um arquivo shell unix.

O tempo foi obtido pela função GNU *date,* que para nossos testes, apresentou uma precisão mais elevada, em relação ao GNU time.

Para uma melhor eficiência das comparações, utilizou-se a flag "-c" em todos os casos de teste. Assim retorna apenas o número de ocorrências do padrão no texto.

Os testes foram feitos em uma máquina com sistema operacional Ubuntu 16.04, Processador Intel® Core™ i7-3770 CPU @ 3.40GHz × 8 , Memória 7,7 GiB, Gráficos Intel® Ivybridge Desktop. Compilador utilizado GCC/G++ 5.4.0.

3.2) Resultados

Com nossas limitações, foi possível apenas fazer um análise dos algoritmos individuais a fim de saber qual seria o "gargalo" entre eles. Fica bastante claro, nas figuras a seguir, que a construção do Vetor de sufixos e seus utilitários, é o que rege todo o funcionamento. Para arquivos de texto muito grandes (i.e > 1mb), seu uso se torna bastante inviável.

Mas ainda sim, com ele em mãos, a busca de um padrão no texto tem um ganho bastante considerável. Fizemos também o teste com a média de padrões de determinado tamanho.

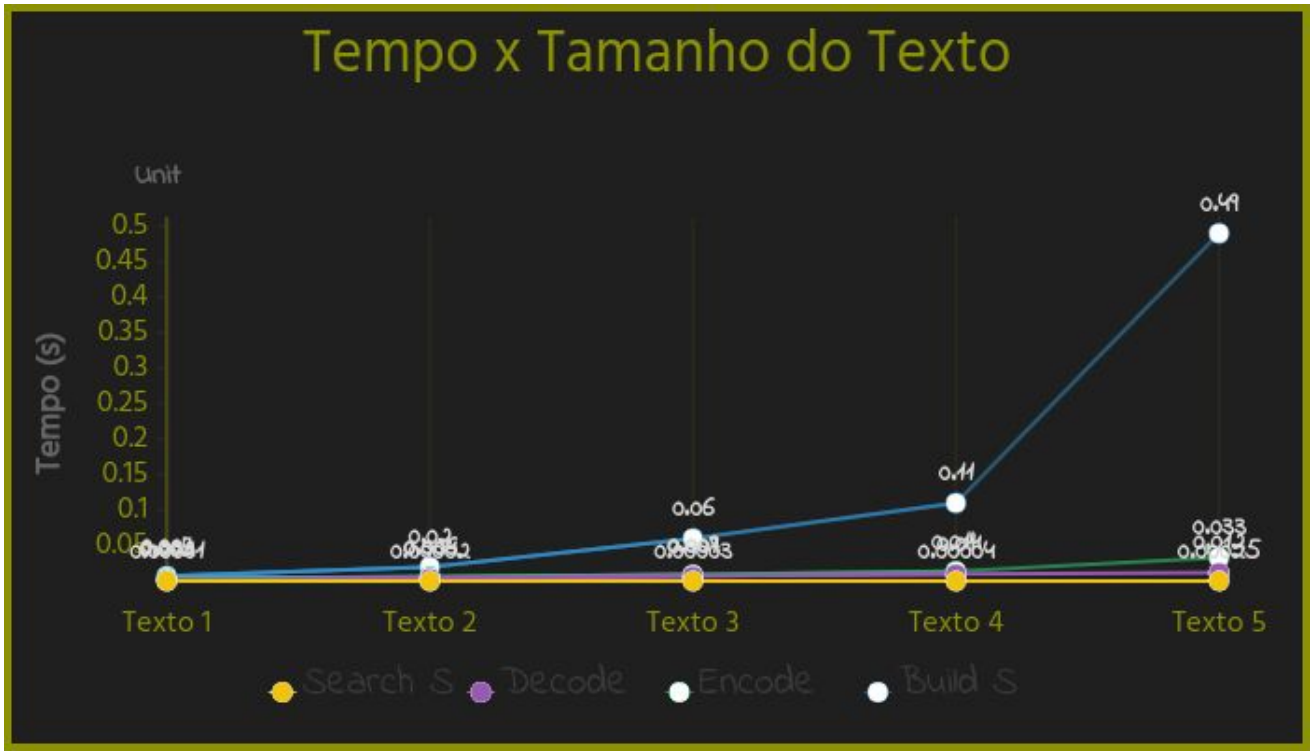

Figura 1 - Análise dos algoritmos

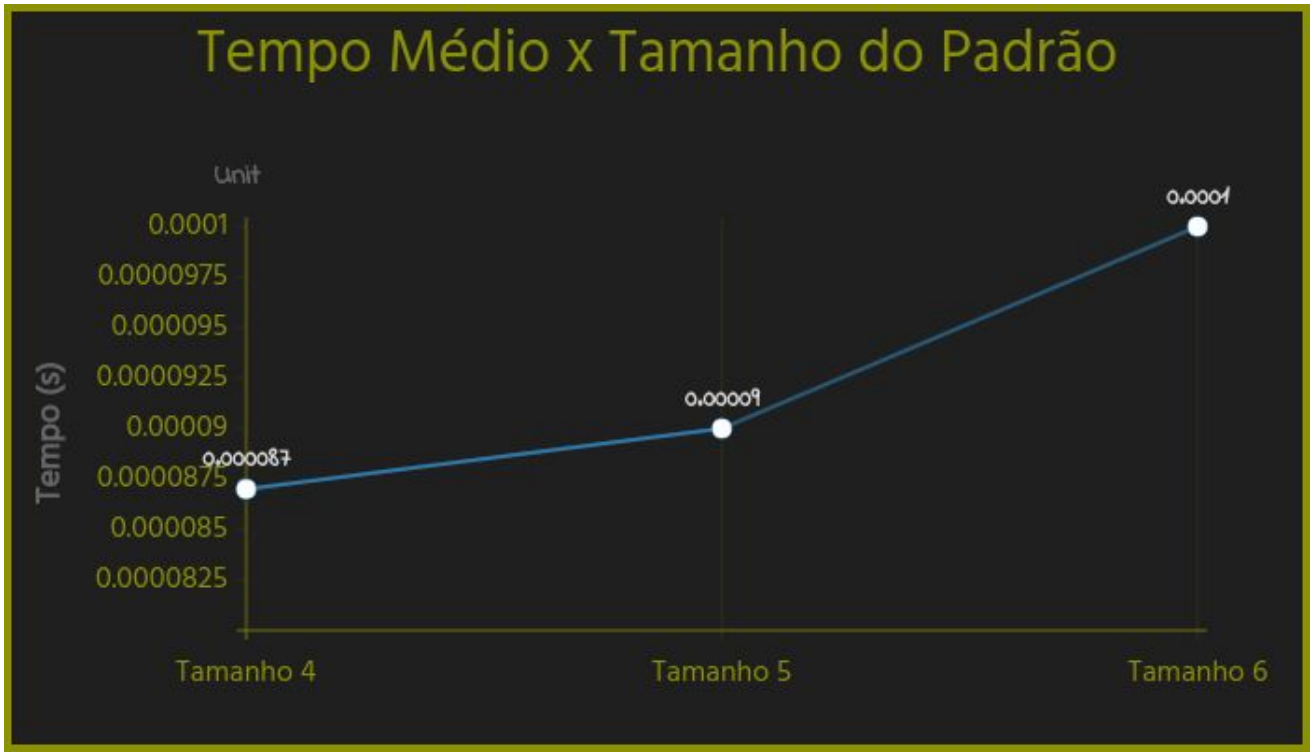

Figura 2 - Análise da busca com variância dos padrões

3.3) **Conclusões**

É visível o quão a construção do vetor de sufixos afeta o programa como o todo. Talvez com uma implementação mais cautelosa e com otimizações, possa vir a ser um tempo menor.

A parte de comprimir/descomprimir mostrou-se bastante ágil em relação ao tamanho do texto. Ainda que não utilizamos uma otimização para o alfabeto, obtivemos um resultado satisfatório.