Universidade Federal de Pernambuco
Centro de Informática

Ferramenta IPMT

Giovanni Antonio Araujo de Barros e Silva - gaabs@cin.ufpe.br
Pedro Henrique Sousa de Moraes - phsm@cin.ufpe.br

IF767 - Processamento de Cadeias de Caracteres

Recife

Dezembro, 2017

Sumário

1. Identificação

1.1 Equipe

Giovanni Antonio Araujo de Barros e Silva - gaabs@cin.ufpe.br
Pedro Henrique Sousa de Moraes - phsm@cin.ufpe.br

1.2 Contribuições

Giovanni

- Implementação dos algoritmos de compressão LZ-77 e LZ-78
- Otimização, correção e refatoração do código
- Desenvolvimento e execução de testes
- Escrita de documentação e relatório

Pedro

- Implementação dos algoritmos de indexação Suffix Tree e Suffix Array
- Otimização, correção e refatoração do código
- Desenvolvimento e execução de testes
- Escrita de documentação e relatório

2. Implementação

A ferramenta IPMT foi implementada usando a linguagem C++ (cross-platform) e possui o conjunto de opções definidas na especificação do projeto. O IPMT possui dois modos de execução: indexação e busca. Foram implementados, para indexação, os algoritmos de Árvores de Sufixos e Arrays de Sufixos. Além disso, foram implementados os algoritmos LZ-77 e LZ-78 para compressão.

No modos de indexação, as opções a seguir estão disponíveis:

- **-e, --compression:** Usada para especificar o algoritmo de compressão a ser usado. Os valores possíveis são "lz77" (Lempel-Ziv-77), "lz78" (Lempel-Ziv-78) e “uncompressed” (sem compressão).
- **-i, --indextype:** Usada para especificar o algoritmo de indexação a ser usado. Os valores possíveis são "suffixtree" (Árvore de Sufixos) e "suffixarray" (Array de Sufixos).

No modo de busca exclusivamente, as opções a seguir também estão disponíveis:

- **-c, --count:** Exibe apenas a quantidade de padrões encontrados em cada arquivo. Como não efetua operações de escrita por linha, a ferramenta pode ficar mais eficiente em casos de muitos matchings no texto.
- **-p, --pattern:** Especifica um arquivo de padrões, cada linha do arquivo é tratado como um padrão. Se essa opção é usada, o primeiro argumento da ferramenta é tratado como um arquivo de texto e não como um padrão, dessa forma um segundo argumento não é obrigatório.

Além dessas, existem opções extras, independente do modo:

- **-h, --help:** Mostra informações sobre o uso e as opções da ferramenta.

- **-v, --verbose:** Mostra informações extras durante a execução da ferramenta.

2.1 Seleção Automática dos Algoritmos na Indexação

Caso nenhum algoritmo seja especificado, a ferramenta seleciona o array de sufixos como algoritmo padrão pois o índice gerado por ele é bem menor que o gerado pela árvore de sufixos.

Além disso, para a compressão realizada durante a indexação, o algoritmo padrão escolhido foi o LZ-78, pois possui e menor tempo de compressão, apesar de não ter a mesma qualidade de compressão do LZ-77.

2.2 Estruturas de dados utilizadas e observações

- Quanto à Árvore de Sufixos:
  - A implementação usa estrutura de dados baseada em nós;
  - As referências dos nós para seus descendentes são armazenadas em um std::map, que é uma implementação de red-black tree, o std::unordered_map (hash table) não foi usado pois não iria oferecer ganho de desempenho assintótico além de consumir mais memória, pois como é usado um alfabeto de tamanho fixo (256), a consulta de um caractere a um nó retorna seu descendente em tempo constante (O(log(256)) = (O(1)));

- Quanto ao Array de Sufixos:
  - Foi implementada uma versão otimizada do algoritmo de criação do array de sufixos que não gera múltiplos arrays parcialmente ordenados, por isso a geração dos LCPs não foi feita usando o algoritmo de Mamber;
  - O algoritmo de ordenação usado foi o std::sort, que é uma implementação de IntroSort (maioria dos compiladores), por isso o tempo de execução é de $O(n.\log^2(n))$
  - A geração dos LCPs foi feita usando o algoritmo Kasai, que usa o array de sufixos pronto para construção dos LCPs o tempo de execução é O(n.log(n)).

- Quanto ao algoritmo LZ-77:
    - Buscou-se limitar cada codificação em 3 bytes, utilizando:
        - 12 bits para o tamanho da janela de buffer, ou seja, 4096 possíveis valores para a posição de início do casamento;
        - 4 bits para o tamanho da janela de lookahead, ou seja, encontrando padrões de tamanho até 16;
        - 1 byte para o caractere de mismatch
    - Para a busca, foram utilizadas listas encadeadas. Existe uma lista para cada possível caracter e as listas armazenam as posições em que os caracteres ocorrem, começando a busca apenas a partir das posições lista do caractere inicial do prefixo. Sendo N o tamanho do buffer e M do lookahead, a complexidade é O(N+M), mas no caso médio, as listas fazem com que a complexidade seja reduzida à ordem de (N+M)/256.
    - Observou-se que as dimensões dos buffers influem diretamente no tempo de execução e qualidade de compressão, uma vez que limita a dimensão da cadeia a ser buscada e condiciona até onde pode ser buscado.

- Quanto ao algoritmo LZ-78:
    - Foi utilizada uma hashmap para armazenar as sequência de caracteres observadas. Especificamente, foi utilizada a estrutura unordered_map da biblioteca padrão de C++.
    - Foi utilizado um esquema de prioridade FIFO quando o limite do dicionário foi alcançado, em vez de recomeçar a partir de um dicionário vazio.
    - Foi adotado o limite de 256 sequências distintas no dicionário (1 byte), de tal forma que cada codificação contém 2 bytes.

- Quanto a leitura das entradas, para tanto fez-se uso da API de streams do C++, uma vez que ela gerencia automaticamente o tamanho dos buffers. Isso facilita a implementação dos algoritmos, mas ocasiona numa pequena perda de performance. Além disso, os arquivos foram completamente carregados na memória.
- Na compressão, foi adicionado um byte que contém a informação do compressor necessária no momento da descompressão.

- A quantidade de memória RAM pode tornar a execução do indexador com árvore de sufixos muito lenta, pois o mesmo usa muita memória, se não houver memória disponível, o sistema usará memória virtual, o que impactará negativamente no tempo de execução do indexador.

3. Testes e Resultados Obtidos

3.1 Ambiente de testes e ferramentas utilizadas

Os testes e comparações da ferramenta e dos algoritmos foram feitos através de scripts em Python. Os scripts executam múltiplas vezes comandos shell e, através da função timeit provida pela linguagem Python, calculam o tempo médio da execução, que é a métrica usada para avaliar o desempenho dos algoritmos. No total, foram executados três testes por arquivo para cada combinação de algoritmos de compressão e indexação. Para a geração dos gráficos, o pacote matplotlib da linguagem Python foi usado.

Os testes foram conduzidos em um computador com sistema operacional Ubuntu 17.10 com processador Intel core i7 2.1GHz e 8GB de memória RAM.

3.2 Descrição dos dados

Para entrada dos testes, foram utilizados arquivos disponíveis no Pizza&Chili Corpus (http://pizzachili.dcc.uchile.cl/). Foram escolhidos dois arquivos de entrada com diferentes tamanhos e categorias de texto de forma a proporcionar uma melhor avaliação do funcionamento da ferramenta em casos mais gerais.

O primeiro arquivo escolhido possui 50MB e possui um vocabulário de apenas 5 letras, tendo sido utilizados apenas os primeiros 10MB do arquivo. Quanto aos padrões, foram escolhidos com os tamanhos 5, 50 e 500, para cada tamanho 50 padrões existentes no texto foram selecionados. O texto e os padrões são compostos de sequências de DNA.

O segundo arquivo escolhido também possui 50MB e possui um vocabulário grande, tendo sido utilizados apenas os primeiros 10MB do

arquivo. Quanto aos padrões, foram escolhidos com os tamanhos 5, 50 e 500, para cada tamanho 50 padrões existentes no texto foram selecionados. O texto em questão é um XML estruturado.

3.3 Compressão e Descompressão

Os algoritmos foram testados com diferentes arquivos de entrada de vários tamanhos. Além disso, a ferramenta gzip também foi incluída nos testes para comparação do desempenho com os algoritmos implementados.

3.3.1 Quanto ao tamanho comprimido

Observou-se que tanto o LZ-77 quanto o LZ-78 obtiveram tamanho comprimido inferior a 50% do tamanho original do arquivo. O resultado também foi próximo ao do gzip, que comprimiu a cerca de 30% do arquivo original.

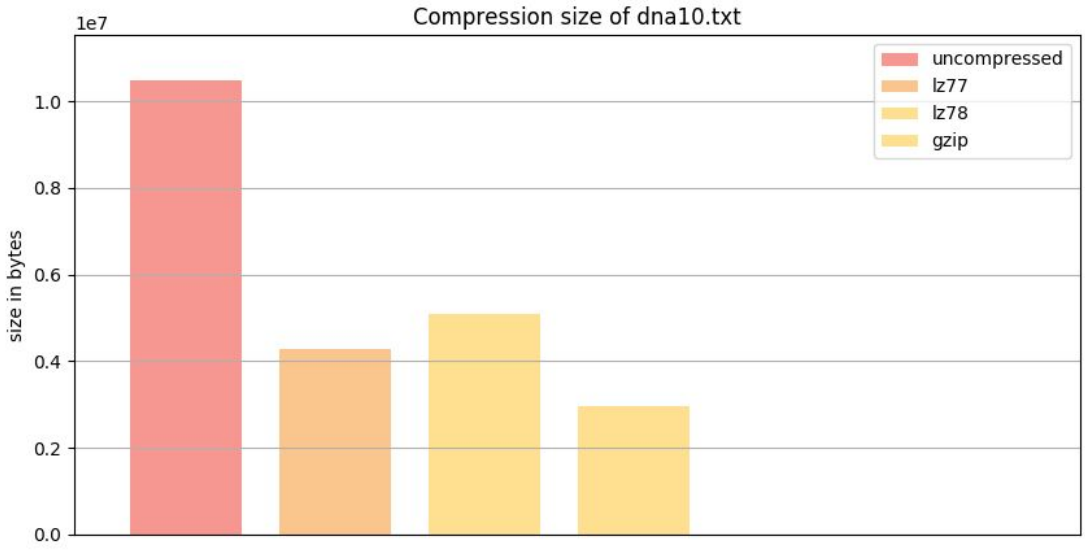

Figura 1: Teste individual dos algoritmos de compressão, avaliando o tamanho do arquivo comprimido. O texto contém sequências de DNA e foi obtido em "http://pizzachili.dcc.uchile.cl/texts/dna/"

3.3.2 Quanto ao tempo de compressão

Observou-se que tanto o LZ-77 teve um desempenho bem inferior ao LZ-78 e ao gzip na questão tempo. O tempo de compressão deve-se ao tamanho da janela, que priorizou melhor compressão ao invés do

tempo. obtiveram tamanho comprimido inferior a 50% do tamanho original do arquivo. O LZ-78, por sua vez, teve desempenho superior ao gzip.

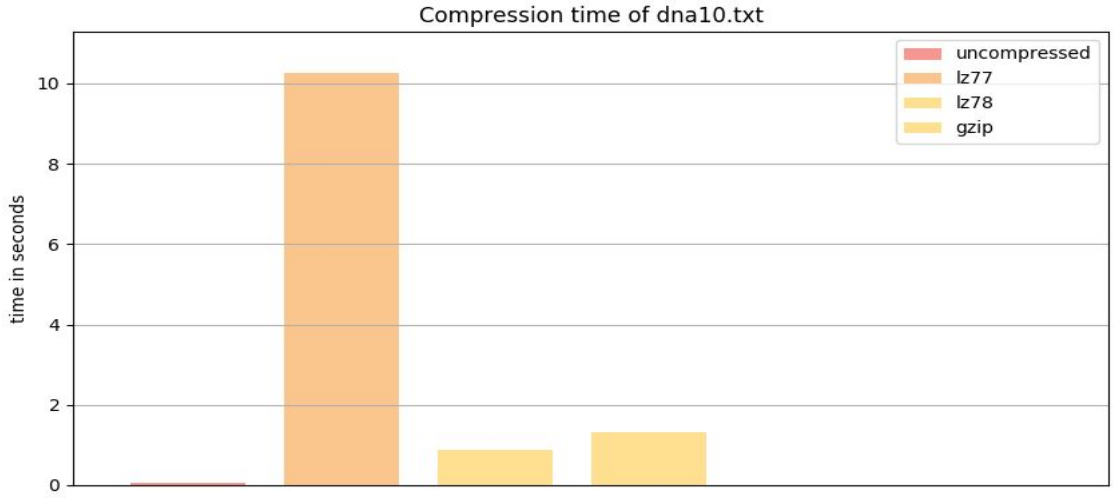

Figura 2: Teste individual dos algoritmos de compressão, avaliando o tempo de compressão dos algoritmos. O texto contém sequências de DNA e foi obtido em "http://pizzachili.dcc.uchile.cl/texts/dna/"

3.3.3 Quanto ao tempo de descompressão

Observou-se que tanto o LZ-77 quanto o LZ-78 obtiveram tempo pequeno de compressão. O LZ-77, inclusive, executa em tempo menor do que o gzip. Ressalta-se o tempo "sem compressão", onde apenas há a leitura do arquivo nos mesmos moldes utilizados no LZ-77 e LZ-78, mostrando que parte significativa do tempo gasto da descompressão nos algoritmos implementados é com a leitura.

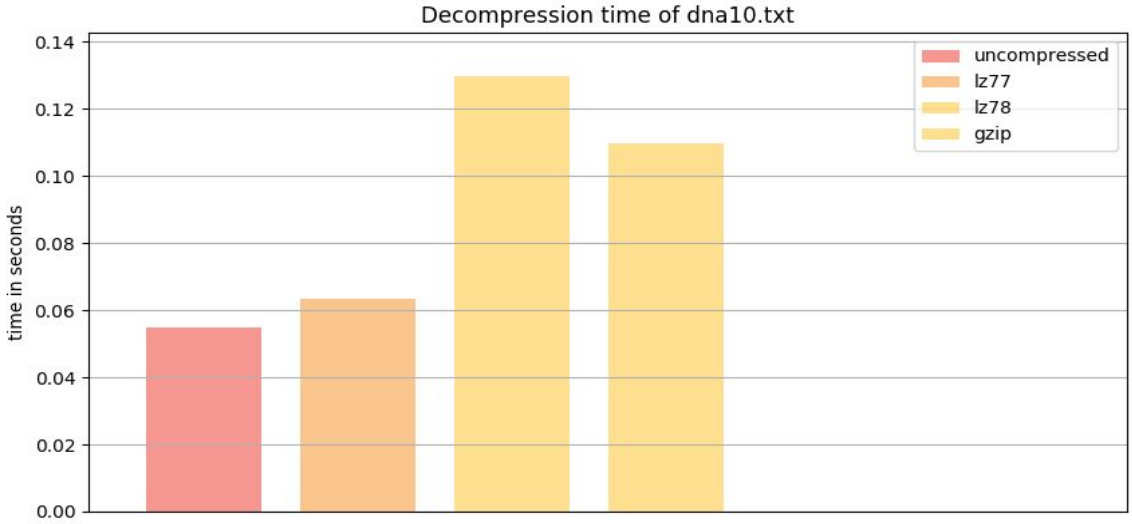

Figura 3: Teste individual dos algoritmos de compressão, avaliando o tempo de descompressão dos algoritmos. O texto contém sequências de DNA e foi obtido em "http://pizzachili.dcc.uchile.cl/texts/dna/"

3.4 Indexação e Busca

3.4.1 Tamanho dos Índices

O tamanho dos índices invertidos gerados pela árvore de sufixos e array de sufixos se mantiveram proporcionais ao tamanho do arquivo. Os tamanhos dos índices gerados com arrays de sufixos foram aproximadamente 10 vezes maiores que o arquivo original, já os índices gerados com árvores de sufixos foram no mínimo 40 vezes maiores, chegando a 70 vezes.

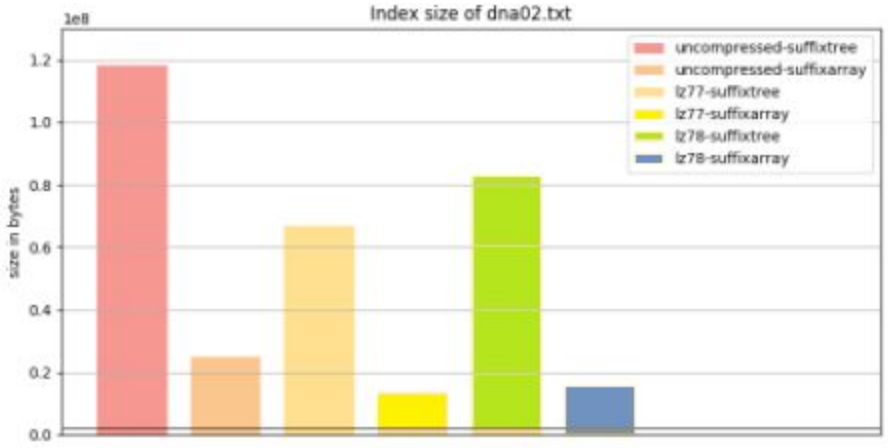

Figura 4: Teste das combinações dos algoritmos de indexação e compressão, avaliando o tamanho do índice serializado, o arquivo possui 2 MB. O texto contém sequências de DNA e foi obtido em "http://pizzachili.dcc.uchile.cl/texts/dna/"

3.4.2 Tempo de Indexação

O tempo de indexação do array de sufixos e na árvore de sufixos foi muito parecido em todos os arquivos, sendo um pouco maior no primeiro, mas essa diferença não mudou muito nos arquivos testados com intervalo de 1 a 10 MB. Mesmo que a complexidade do array de sufixos seja um pouco maior, o tamanho dos arquivos não são suficientes para mostrar essa diferença.

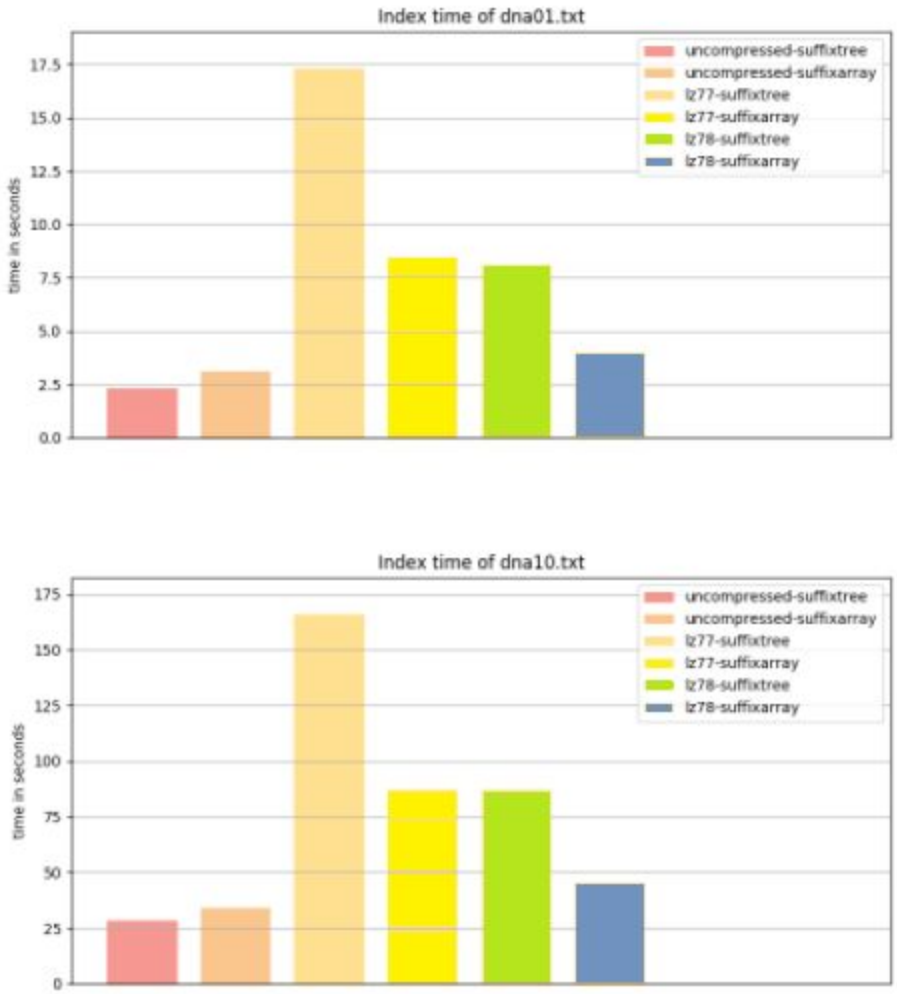

Figura 5: Teste das combinações dos algoritmos de indexação e compressão, avaliando o tempo e execução dos algoritmos com dois arquivos, com 1 e 10 MB respectivamente. O texto contém sequências de DNA e foi obtido em "http://pizzachili.dcc.uchile.cl/texts/dna/"

3.4.3 Tempo de Busca

O tempo de busca levou em consideração o tempo de descompressão (quando existia) e o tempo de reconstrução do índice, em cada para cada tamanho de padrão (5, 50, 500), sendo que existiam ocorrências de todos os padrões ao menos uma vez em nos arquivos, a busca levava em consideração 50 padrões de uma vez.

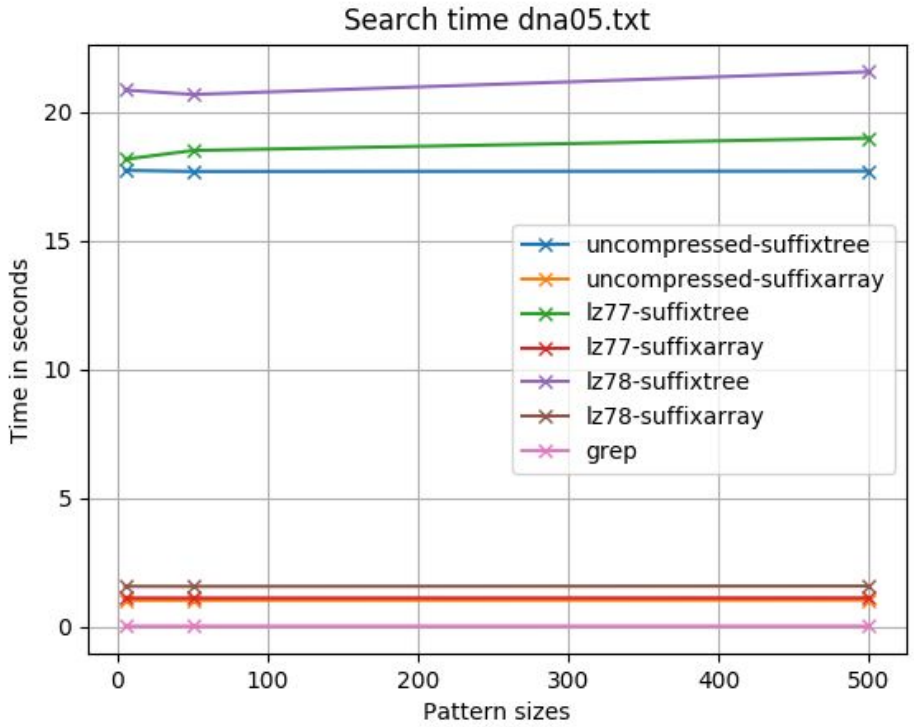

Figura 6: Teste das combinações dos algoritmos de indexação e compressão, avaliando o tempo descompressão e busca por 50 padrões, o arquivo possui 5 MB. O texto contém sequências de DNA e foi obtido em "http://pizzachili.dcc.uchile.cl/texts/dna/"

4. Conclusões

- Quanto à compressão, observou-se que o LZ-77 é superior em taxa de compressão e em velocidade de descompressão, sendo seu uso justificado nos casos em que a prioridade não é a velocidade de compressão;
- O tempo de execução dos algoritmos de indexação é muito alto, mas seu uso é justificado pois, uma vez que terminado, as buscas por padrões são rápidas;
- Um dos problemas dos índices é a quantidade de memória, nos nossos testes, o tamanho de um índice chegou a 70 vezes o tamanho do arquivo indexado, além disso a quantidade de memória RAM usada para manter a estrutura é ainda maior;
- Mesmo que o custo computacional do array de sufixos seja maior que o da árvore de sufixos, o tamanho dos índices gerados foi várias vezes menor, o que valida seu uso em muitos casos;

- O tempo de busca das árvores de sufixos foi pior que os arrays de sufixos, mas isso foi causado pela descompressão dos arquivos de índices muito maiores.