Universidade Federal de Pernambuco - UFPE
CIn - Centro de Informática

Processamento de Cadeias de Caracteres(if767)
Projeto 2

Antonio José Gadelha de Albuquerque Neto(ajgan)
Gabriel Vinícius Melo Gonçalves da Silva(gvmgs)

Recife, 2018.2

1. Identificação

A equipe é formada pelos alunos Antonio Gadelha(ajgan) e Gabriel Melo(gvmgs). Antonio implementou os algoritmos de indexação por array de sufixos e de compressão por LZ77 e fez também a parte da leitura dos comandos. Gabriel implementou os algoritmos de descompressão do LZ77 e de busca, para o arquivo indexado. Ambos contribuíram com a documentação e com os testes.

2. Implementação

A ferramenta foi toda implementada em c++, assim como havia sido sugerido. O funcionamento do programa obedece a entradas por linha de comando no seguinte formato: *"./ipmt index [options] textfile"* ou *"./ipmt search [options] pattern indexfile"*. As opções de comando serão explicadas posteriormente nesse relatório.

2.1 Algoritmos Implementados

- **Indexação - Array de Sufixos**

  O algoritmo de indexação por array de sufixos gera um array com todos os sufixos do texto e ordena por ordem alfabética, resultando num array com os índices ordenados de acordo com os sufixos das suas posições. Os índices são codificados para bits(a quantidade de bits depende do tamanho do texto) e são salvos num arquivo de forma comprimida.

- **Compressão/Descompressão - LZ77**

  O algoritmo de compressão LZ77 mantém toda a informação contida no texto, reescrita de forma que informações repetidas não precisem ser escritas mais de uma vez. Para isso são utilizadas duas janelas, a de dicionário e o lookahead buffer. A janela de dicionário tem tamanho 511(9 bits) e guarda as 511 posições anteriores do texto que o algoritmo acabou de passar e o lookahead buffer tem tamanho 127(7 bits) e olha as 127 posições seguintes do texto para tentar dar match com alguma das substrings presentes na janela de dicionário. Com essas informções o LZ77 guarda triplas no formato <distância, tamanho, caractere> que informa quantos passos o algoritmo deve retornar na janela de dicionário para encontrar um match, qual o

tamanho da substring que deu match e qual o próximo caractere que deve ser escrito depois de contabilizar esse match. Por fim, com todas as triplas do texto, codificamos os valores para bits (9 bits da posição + 7 bits do tamanho + 8 bits do caractere) que nos retorna 24 bits, o equivalente a 3 unsigned chars que são gravados na file comprimida. Quanto mais repetições houverem no texto, melhor pra compressão, pois teremos menos triplas.

Na descompressão, o processo inverso é realizado. Os unsigned chars lidos viram bits, que viram triplas, que formam o texto.

- **Busca - Busca Binária**

  Algoritmo otimizado para realizar buscas em vetores ordenados. A busca é realizada no array de sufixos, como este estava ordenado a única coisa necessária para verificar a ocorrência de um padrão no dado texto era uma busca binária, retornando um valor negativo em caso de inexistência ou a posição inicial do padrão em caso de presença.

2.2 Detalhes de Implementação

2.2.1 Estruturas de Dados

Além do uso amplo das estruturas primitivas, foram utilizadas com certa recorrência a estrutura <vector> para melhor manipulação dos dados.

2.2.2 Estratégia de Leitura das Entradas

A leitura de entradas foi dividida em três partes, na primeira parte verifica-se o arg localizado na segunda posição(a primeira posição é sempre *ipmt*). Esse arg é o que indica qual modo o programa deve obedecer: *index* para o modo de indexação e *search* para o modo de busca. Caso outro termo seja lido, é disparada a flag *help* para o usuário.

Na segunda parte da leitura varre-se o argv e procura por strings que sejam iguais às flags esperadas. As flags podem ser:

- *-c ou --count*:

  Flag para exibir contagem de ocorrências do padrão no texto(por default, não se exibe)
- *-p ou --pattern patternfile*:

Flag para indicar que os padrões devem vir de um arquivo(por default, o padrão é um só e é digitado pelo usuário)

- *-h ou --help*:
  Flag que exibe informações básicas do programa ao usuário.
- *-t ou --text*:
  Flag que comprime o texto, sem indexar os sufixos.
- *-s ou --suffix*:
  Flag que indexa os sufixos, mas não comprime o texto.

Após o programa identificar as flags dentre os argv, varre-se todos os argvs novamente para se identificar os argumentos que não foram identificados no passo anterior, os dois argumentos achados nesse passo serão o padrão e o arquivo de texto, respectivamente. Caso o usuário tenha fornecido a flag *-p*, apenas um argv terá sobrado para esse segundo passo, e ele será o arquivo de texto.

Caso haja alguma inconsistência no comando, como uma flag não esperada ou se estiver faltando alguma informação necessária, a flag *--help* é disparada para o usuário.

2.2.3 Heurística de Combinação de Algoritmos

A equipe decidiu não implementar variações dos algoritmos. Então o programa sempre vai utilizar array de sufixos, LZ77 e busca binária.

2.2.4 Valores padrão dos parâmetros

A equipe setou as janelas para 511 e 127, para manter uma janela grande, mas nem tanto, para não prejudicar o desempenho de tempo. Os valores das janelas também ajudam pois somados dão 16 bits, o que permite que sejam salvos em 2 bytes. Tentamos a opção de utilizar os valores de 1023 e 63 para tentar melhorar a compressão. De fato, conseguimos uma melhora razoável no tamanho dos arquivos, mas o tempo de execução ficou quase 2x mais lento, o que tornaria a abordagem muito ineficiente para arquivos maiores(apesar de dar resultados um pouco melhores).

2.3 Limitações e Bugs Conhecidos

Um ponto que vale a pena ser mencionado é que escolhemos por separar os índices do texto comprimido, para poder manter uma boa ideia do tamanho dos dois arquivos e do tempo necessário para gerá-los. Isso não seria realmente uma limitação da ferramenta, pois poderia ser facilmente contornado, mas já que fizemos diferente do que tem na descrição do projeto, achamos válido colocar isso no relatório.

Por conta de estouros de memória e lentidão no código, foi feita uma mudança no comportamento do array de sufixos. Ao invés de pegar cada posição e levar o sufixo até a última posição do texto, só foi computado 16 posições para frente. Isso é ruim pois se duas posições tiverem um texto igual para as 16 seguintes posições, não dá para garantir que eles serão ordenados corretamente. Mas essa mudança foi necessária, pois qualquer arquivo de tamanho um pouco grande fazia o programa abortar.

Até o momento da elaboração deste relatório o algoritmo de busca ainda não funcionou com eficácia para textos muito grandes, tendo rodado com sucesso apenas em textos pequenos, acreditamos que o problema esteja na etapa de decodificação do arquivo indexado.

3. Testes e Resultados

3.1 Dados e Ferramentas de Comparação

Os testes foram realizados com os textos em inglês de 50 e 200 MB disponível no Pizza&Chili. Também realizamos testes com o texto de Shakespeare de 5.5 MB do projeto Gutemberg.

Comparamos os desempenhos dos nossos algoritmos com a ferramenta grep para casamento exato e gzip para compressão.

3.2 Ambiente de Testes

Os testes foram realizados em um MacBook Air com sistema operacional MacOS Mojave 10.4, memória de 4 GB 1600 MHz DDR3 e processador Intel Core i5 1,4 GHz.

3.3 Experimentos Realizados

Para a compressão foram realizados experimentos para medir tempo e o tamanho resultante da compressão. Nesse modo de

experimentos, foi utilizada a flag que não realiza a indexação(já que ambos os passos são realizados com o comando de *index*).

Para a indexação utilizou-se a flag que impede a compressão, para medir a efetividade real da indexação.

3.4 Resultados Obtidos

3.4.1 Compressão

Arquivo	Tamanho	Tamanho Comprimido	Tempo	Tamanho gzip	Tempo gzip
Shakespeare	5.5MB	4.3MB	36.5seg	2MB	0.54seg
English50MB	52.4MB	43.7MB	5m45seg	19.8MB	4.45seg
English200MB	209.7MB	174,5MB	22m58seg	79.4MB	18.4seg

Para esses resultados o programa foi rodados 3 vezes e então tiramos a média.

Só para exemplificar o que comentamos na seção 2.2.4, rodamos o programa para uma file de 4MB com as janelas em 511 e 127 e obtivemos uma file comprimida de 3,4MB em 29 segundos. Com a configuração em 1023 e 63 obtivemos uma file comprimida de 3,1MB, mas em 51 segundos. Ou seja, entendemos que daria para se obter resultados melhores, mas optamos por manter as janelas fixas em 511 e 127 por conta do tempo gasto, que seria muito alto com files maiores.

O gzip ficou bem à frente da nossa ferramenta em tempo e espaço, mas entendemos que o algoritmo dele é uma versão bem otimizada.

3.4.2 Indexação

Arquivo	Tamanho	Tamanho do .idx	Tempo
Shakespeare	5.5MB	15.7MB	24,7seg
English50MB	52.4MB	170.4MB	4m35seg

English200MB	209,7MB	191,1MB	21m34seg

Para esses resultados o programa foi rodados 3 vezes e então tiramos a média.

Quanto maior o arquivo de entrada, menor será o idx em relação ao original, pois o número de bits necessários para representar as posições tende a crescer de forma mais devagar

3.4.3 Busca

Ocorreram muitos problemas com a decodificação do arquivo indexado para a busca, e o tempo que sobrou não nos permitiu muitos testes. Só deu pra atestar que ele estava retornando o resultado esperado em arquivos de pequeno tamanho, o que nos leva a reforçar a hipótese do bug na decodificação.

3.5 Conclusão

Foi bem interessante trabalhar com compressão e descompressão de arquivos, além de fazer buscas utilizando índices. É bem desafiador sentir na prática as dificuldades de implementação e é muito satisfatório obter resultados que funcionam. Além de tudo, podemos perceber que ainda há espaço para otimizações e que existem abordagens ainda mais efetivas para os problemas em questão. Em geral, a cadeira nos deu uma boa visão da área e os projetos puderam nos desafiar a entender mais.