Trabalho Prático 1
Manipulação de sequências

Vitor Emanuel Ferreira Vital - 2021032072

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

vitorvital@dcc.ufmg.br

1. Introdução

Esta documentação tem como objetivo apresentar a implementação e modelagem da resolução do Trabalho Prático 1 da disciplina de Algoritmos 2, focado na manipulação de sequências. Nesta prática, foram abordados diversos temas relacionados à solução de problemas envolvendo sequências, implementados utilizando a linguagem C++, sendo esses:

- Compressão de arquivos
- Utilização de estruturas de dados estudados na disciplina
- Manipulação de sequências

Ao longo do trabalho, foram explorados tópicos como manipulação de strings, operações com vetores, ordenação de dados, busca e manipulação de elementos em estruturas de dados e compressão de arquivos com base no algoritmo LZ78. Adicionalmente, foram utilizadas diversas técnicas e algoritmos para otimizar o desempenho do código e garantir uma solução eficiente para os problemas propostos.

Por meio deste relatório, espera-se apresentar de forma clara e objetiva os resultados obtidos com a implementação das soluções propostas, bem como as técnicas e estratégias utilizadas para atingir tais resultados. Acredita-se que este trabalho possa contribuir significativamente para o desenvolvimento das habilidades em programação e solução de problemas dos estudantes envolvidos nesta disciplina.

2. Modelagem

O problema foi modelado com base no algoritmo LZ78 para compressão de arquivos texto, utilizando a estrutura de dados Trie. O objetivo do trabalho é reduzir o espaço de armazenamento necessário para o arquivo, considerando a frequência de repetição de caracteres e palavras. Para isso, foi necessário desenvolver uma implementação eficiente da estrutura Trie, capaz de armazenar e buscar sequências de caracteres de forma rápida e eficiente. O algoritmo em questão foi desenvolvido por Abraham Lempel e Jacob Ziv em 1978, baseando-se na construção de um dicionário com os dados encontrados anteriormente no arquivo a ser comprimido. No início o dicionário se encontra vazio. A medida que o arquivo vai sendo lido, caractere por caractere, cada sequência de caracteres não encontrada no dicionário é introduzida no dicionário e ganha um código. As sequências que já se encontram no dicionário são substituídas pelo seu código no dicionário.

Com base no contexto apresentado, é notável que o algoritmo requer um alto número de operações de busca e inserção no dicionário. Com o intuito de otimizar o desempenho do algoritmo, decidiu-se utilizar uma estrutura de dados em árvore Trie, que possibilita a busca rápida e eficiente de sequências de caracteres. A implementação dessa estrutura permitiu que o algoritmo realizasse as operações necessárias de maneira mais rápida e eficiente, resultando em uma solução mais otimizada e eficaz para o problema em questão.

Durante a modelagem do problema de compressão de arquivos utilizando o algoritmo LZ78, foi observado que a adição do código poderia ter um grande impacto na quantidade de dados a serem armazenados. Para mitigar esse efeito, optou-se por utilizar uma representação dinâmica dos bytes de representação dos índices dos prefixos. Dessa forma, ao realizar a compressão, armazena-se também o maior índice que é prefixo de algum caractere, permitindo que todos os outros códigos sejam representados com um número suficiente de bytes para representar o maior código.

Com essa abordagem, é possível armazenar uma quantidade reduzida de bits para representar cada código, sempre que possível. Entretanto, é importante ressaltar que o número de bytes utilizados na codificação é fundamental para a decodificação, razão pela qual a primeira linha do arquivo compactado contém o inteiro que representa o maior índice de prefixo utilizado.

a. Algoritmo de Compressão LZ78

No contexto da compressão de texto pelo algoritmo LZ78, os prefixos identificados são armazenados em um dicionário representado pela estrutura de dados Trie. Cada prefixo é mapeado para um índice no dicionário, com o primeiro índice correspondendo ao caractere vazio (representado como "(vazio), 0"). Durante a compressão, cada caractere é lido e, se houver correspondência com um prefixo no dicionário, o código correspondente é utilizado e a iteração continua até que um novo caractere seja encontrado. Caso não haja correspondência, o código do prefixo é incluído, juntamente com o caractere novo, para uso futuro na descompressão. Essa abordagem permite uma compressão eficiente do texto, reduzindo o espaço de armazenamento necessário, e demonstra a utilidade da estrutura Trie na manipulação de sequências de caracteres.

A inserção na árvore é feita da seguinte forma, começando da raiz ((vazio), 0) :

1. Existe alguma incidência do caractere analisado como filho do nó atual
    a. se sim, prossiga para o passo 2;
    b. se não, insira o caractere e seu respectivo índice, finalizando o algoritmo.
2. Vá para o nó casado, pegue o novo caractere do texto, e refaça as iterações do passo 1.

Durante o processo de inserção no arquivo comprimido pelo algoritmo LZ78, é utilizada a mesma ideia de busca e codificação de prefixos, com a adição da inserção dos códigos correspondentes na estrutura de dados do caractere casado na etapa 1.a. Na etapa 1.b, o código do prefixo e o novo caractere são inseridos no arquivo comprimido, possibilitando a reconstrução do texto original na etapa de descompressão.

b. Algoritmo de Descompressão LZ78

Ao descomprimir um arquivo comprimido pelo algoritmo LZ78, é necessário seguir o método utilizado na compressão, convertendo as informações obtidas do arquivo para as respectivas palavras. Cada palavra é representada pelo seu código de prefixo seguido do caractere final, o que permite sua reconstrução no texto original.

Para otimizar as buscas na estrutura de dados Trie, foi implementado um vetor que armazena os elementos adicionados na estrutura, com o índice no vetor correspondente à inserção do elemento

no arquivo comprimido. Essa abordagem reduz o tempo de busca na árvore Trie, porém aumenta o custo de armazenamento do algoritmo.

Além disso, cada nó da árvore tem um ponteiro para seu nó pai, para facilitar a reconstrução do prefixo.

Com base nisso, realiza-se a descompressão com base no seguinte algoritmo, iniciando pela raiz da árvore cujo valor é ((vazio), 0) e o ponteiro para o pai é nulo:

1. Procura-se o código da palavra em questão no vetor, obtendo nó respectivo.
2. Armazenamos o novo caractere como filho do nó encontrado
3. Reconstruímos a palavra, iniciando do novo nó até a raíz, prefixando os caracteres encontrados.
4. Imprimimos a palavra encontrada no arquivo de saída

Com o processo de descompressão finalizado, é possível obter novamente o arquivo original antes da compressão, mantendo todas as informações e estrutura do arquivo original. Isso garante que a compressão e descompressão foram realizadas de forma eficiente e sem perda de informações.

3. Implementação

O programa foi implementado na linguagem C++ e executado no sistema operacional Linux Ubuntu 20.04.6 LTS.

4. Análise de Compressão

O algoritmo de compressão LZ78 tem por objetivo manter uma representação do arquivo original para que seja transportado ou armazenado de forma a utilizar menos espaço de memória e, quando houver a necessidade de leitura, basta aplicarmos o processo de descompressão.

Nesse sentido, é importante analisarmos a taxa de compressão obtida com a implementação do algoritmo em questão.

Para isso serão analisados 10 arquivos no formato .txt. Todos eles ocupam um espaço de armazenamento inicial entre 1KB e 2MB, após a compressão esperamos que todos ocupem um espaço significativamente menor que o arquivo original.

Na tabela a seguir podem ser visualizados os resultados obtidos.

Arquivo (TXT)	Tamanho Inicial (bytes)	Tamanho Final (bytes)	Taxa de Compressão
Biblia_Sacra	22.028	22.017	0,05%
Bohr_Theory	366.769	248.117	32,35%
El_Tratado	1.869.044	1.297.825	30,56%
Jaime	56.109	54.093	3,59%
Journal_Marocco	1.112.731	872.410	21,60%
Natalie	383.458	343.065	10,53%
Prayer	72.958	64.105	12,13%

Shan_Shui_Qing	339.000	348.325	-2,75%
The_Captives	1.063.388	833.155	21,65%
Van_den_Noordpool	1.209.070	959.055	20,68%

Com base na tabela acima, é possível observar que a implementação do algoritmo de compressão LZ78 resultou em uma taxa média de compressão de 15,04% para os arquivos analisados. É importante ressaltar que a eficácia da compressão pode variar dependendo do tipo de arquivo e da sua estrutura.

Após uma breve análise na tabela é possível a visualização da ineficácia do algoritmo no arquivo “Shan_Shui_Qing”. O texto em questão trata de um livro escrito em mandarim, cujos caracteres utilizados dificultam a concatenação, pois trata-se de palavras inteiras não sequências de caracteres como o alfabeto romano. Dessa forma, adicionam-se os caracteres do código respectivo do prefixo mais o caractere do texto, fazendo com que o texto ocupe maior espaço de armazenamento.

Um fator importante a ser considerado na análise dos resultados obtidos é o tamanho do arquivo original. Textos menores tendem a apresentar uma taxa de compressão menor, uma vez que há menos sequências de caracteres repetidos para serem aproveitadas pelo algoritmo. Isso ocorre porque há menos texto para análise, o que pode levar a uma menor identificação de padrões repetitivos e, consequentemente, a uma menor redução no tamanho do arquivo. Por outro lado, textos maiores tendem a apresentar uma taxa de compressão maior, já que há mais possibilidades de identificar sequências repetitivas e, assim, reduzir significativamente o tamanho do arquivo final.

Outro fator relevante, quanto a escolha da modelagem da solução, é os bytes de representação dos códigos nos arquivos compactados. Em geral há uma economia de um a dois bytes, isso porque a maior representação para o inteiro no C++, e com as condições da máquina apresentada, é de 4 bytes, além disso, por obter-se os dados do arquivo com char, a menor quantidade de bytes de representação do código é de 2 bytes. Por tal motivo, há formas melhores de manipulação dos bits para compactar ainda mais os arquivos.

Além disso, poderia-se utilizar uma compactação baseada em bits, não mais em bytes, o que levaria a uma melhor economia de memória em diversos casos. Tratamento não utilizado na modelagem em questão.

Por fim, outra análise importante é que a compactação dos dados poderia também ser feita nos caracteres. Nesse cenário, ambas representações (código e caractere) possuiriam representações enxutas, maximizando o ganho de compactação.

5. Análise de complexidade

O programa possui duas funções relevantes para resolução do problema, sendo o custo de tempo ditado por elas. Portanto, analisaremos o custo dela e de suas chamadas sucessivas.

Função LZ78Compress - complexidade de tempo - itera por cada caractere do arquivo inicial O(n), inserindo-o na respectiva altura da árvore, caso não haja ocorrência. A busca pelo caractere tem custo O(m), no qual m são o número de filhos do nó casado até então. Custo de tempo O(nm).

Função LZ78Compress - complexidade de espaço - custo de armazenamento de cada caractere com base no seu prefixo O(n) no pior caso, além do armazenamento no vetor utilizado para impressão dos dados no arquivo final O(n) no pior caso. Custo de espaço igual O(n) .

Função LZ78Decompress - complexidade de tempo - itera por cada caractere do arquivo compactado O(n), busca pelo código do prefixo O(1), baseando-se no vetor, inserção O(1) e reconstrução do prefixo O(log(n)). complexidade de tempo final igual a O(n*log(n)).

Função LZ78Decompress - complexidade de espaço - custo de armazenamento de cada caractere com base no seu prefixo O(n) no pior caso, além do armazenamento no vetor utilizado para encontrar o nó do prefixo O(n). Custo de espaço igual O(n) .

6. Conclusão

Com a conclusão do trabalho prático sobre manipulação de sequências, é possível observar como a aplicação do algoritmo LZ78 pode ser útil em diversos contextos, principalmente em situações em que há a necessidade de armazenar ou transmitir dados em formato de texto. A capacidade de reduzir o espaço necessário para armazenamento ou transmissão de arquivos de texto pode trazer benefícios significativos em termos de otimização de recursos.

Além disso, durante a implementação do algoritmo, foi possível perceber a importância de se utilizar estruturas de dados eficientes na busca e análise de sequências de caracteres repetidos. A Trie tradicional se mostrou uma opção viável e eficiente para esse propósito, permitindo que o algoritmo LZ78 pudesse identificar e codificar as sequências repetidas de forma rápida e precisa.

Vale ressaltar que, embora o algoritmo LZ78 tenha apresentado uma taxa de compressão média de 15,04%, a eficiência do algoritmo pode variar de acordo com o tipo de texto que está sendo comprimido. Textos menores, por exemplo, tendem a apresentar uma taxa de compressão menor, enquanto textos maiores tendem a apresentar uma taxa de compressão maior. Além disso, podem ser realizadas otimizações que ampliaram o ganho na compactação.

Em suma, o trabalho prático sobre manipulação de sequências mostrou como o algoritmo LZ78 e a utilização de estruturas de dados eficientes podem ser úteis na redução do espaço necessário para armazenamento ou transmissão de arquivos de texto. O conhecimento adquirido nesse processo pode ser aplicado em diferentes áreas, desde a transmissão de dados pela internet até o armazenamento de arquivos em sistemas de armazenamento de dados.