EDB2 - Projeto da 1ª unidade

Contexto e objetivo

Como vocês sabem, tudo nos computadores se restringe a 0s e 1s!!! (pelo menos, quem foi meu aluno de ITP deveria saber disso 😉).

Porém, uma sequência de 0s e 1s não significa nada!

...exceto se dermos sentido a essa sequência através de uma **convenção**, ou seja se adotamos um **padrão** que todos devem entender da mesma forma. Assim, usamos padrões para representar basicamente qualquer informação. Usamos para representar números inteiros (representação binária, complemento de 2...), números reais (padrão IEEE para ponto flutuante), cores (RGB, CMYK ...), textos (ASCII, UTF-8...), sons, vídeos etc.

Normalmente, esses padrões possuem um número fixo de bits. Por exemplo, o padrão ASCII usa 8 bits (na verdade, 7) para representar o conjunto de 128 símbolos que nós (seres humanos 😁) chamamos de letras... e que, juntas, criam palavras e frases, que damos significados. Assim, uma frase como "algoritmos e estruturas de dados", que possui uma sequência de 32 símbolos (letras), requer, usando o padrão ASCII, $32 \times 8 = 256$ 0s e 1s em sequência.

Porém, padrões podem ser quebrados! (assim, como na vida 😁)
Podemos nos rebelar e representar os símbolos que usamos da forma que quisermos. Podemos usar nossa própria convenção... nossa própria codificação...

Esse é o intuito da codificação de Huffman. Para representar uma sequência de símbolos, ele foge dos padrões convencionais e define um novo padrão . Porém, ele não é um simples *rebelde sem causa*. Seu propósito é criar um padrão para os símbolos de forma que **a quantidade de bits usados seja a menor possível**.

Usando a estratégia de Huffman para codificar sequências de símbolos, um texto como "algoritmos e estruturas de dados", que seria representado com 256 bits se usasse o padrão ASCII, passaria a ser representado com bem menos bits. Por isso que o método de criação de novas codificações definidas por Huffman é normalmente usado para **compressão de dados**.

Apesar de ter sido proposto inicialmente para criar codificações para representar textos, o algoritmo de Huffman (ou suas variações) tem sido utilizado para reduzir a representação de muitos outros tipos de informação, como áudio (mp3), imagens (jpeg) e vídeos (mpeg).

Se o algoritmo de Huffman consegue gerar novos padrões para contextos tão diferentes... por que não usá-lo para gerar uma codificação otimizada para teus programas (código-fonte)??? 🤔

O objetivo desse projeto é de criar um programa (executável em linha de comando) capaz de comprimir e descomprimir arquivos de texto em geral, mas especializado em arquivos de código-fonte (`.cpp`, `.py`, `.java`, `.js`...). Para isso, será necessário criar um novo padrão, baseando-se no algoritmo de Huffman, e usar esse novo padrão para codificar os códigos de seus programas.

Qual a sua linguagem de programação favorita? A que você já fez mais código? Será que você consegue fazer um programa para comprimir seus códigos de forma mais eficaz que os programas de compressão de uso geral?

Algoritmo de Huffman

Esta seção resume o funcionamento do algoritmo de Huffman para você ter uma rápida ideia dos conceitos e etapas necessárias. Porém, será necessário se aprofundar melhor, e algumas referências foram colocadas no final do documento para ajudar.

A ideia central do algoritmo é de usar a frequência dos símbolos para criar uma codificação. É similar ao Código Morse (para quem conhece 😅), no qual as letras mais comuns têm códigos mais curtos (ex: `.` e `-`), enquanto letras menos frequentes têm códigos mais longos (ex: `--.-`). A diferença é que a codificação não é fixa, mas dinâmica. Ela é gerada em função dos dados que apresentamos para o algoritmo. Assim, a codificação é criada de forma otimizada para qualquer tipo de dado.

O algoritmo pode ser resumido em 4 passos:

1. **Contagem das frequências**: a primeira coisa que precisamos saber é a frequência dos símbolos para podermos identificar quais são mais frequentes e quais são mais raros. Ou seja, precisamos montar uma *tabela de frequência*, que simplesmente mapeia cada símbolo ao número de vezes que ele aparece.
2. **Criação da Árvore de Huffman**: a partir da tabela de frequência, criamos uma árvore binária, onde cada símbolo é representado em um nó-folha. A árvore é criada num processo *bottom-up*, ou seja a partir dos nós-folhas até a raiz, em função das frequências de cada nó. As folhas se juntam criando uma subárvore, cuja frequência é a soma das frequências dos seus filhos. Essas sub-árvores se juntam também criando outras subárvores e o processo continua repetidamente até todas se juntarem na raiz. O algoritmo usa uma fila de prioridade para escolher sempre as duas árvores que possuem as menores frequências.
3. **Geração dos códigos binários**: com a árvore construída, para gerar um código para um símbolo que se encontra na folha, percorremos a árvore da raiz até cada folha. Ao descer para um filho da esquerda, adicionamos o bit `0` ao código que está sendo criado, e ao descer para um filho da direita, adicionamos o bit `1` ao código.
4. **Codificação e decodificação**: para codificar (compactar) o texto original, basta substituir cada símbolo do texto pelo seu novo código de Huffman. Leve em conta que a codificação deve ser agrupada em grupos de 8 bits (bem... só teremos compressão se agruparmos os bits). Para decodificar (descompactar), precisamos ler a sequência

codificada, bit-a-bit, e ir percorrendo a árvore para identificar qual símbolo se encontra no nó-folha. Sempre que um nó-folha é encontrado, inserimos o símbolo associado a ele na sequência "descomprimida".

Descrição do projeto

O projeto deve ser desenvolvido em duplas (de dois!... não há duplas de três! 😆) e atender os seguintes requisitos:

- O projeto deverá ser desenvolvido em **C++** (pelo menos C++11, mas versões mais recentes são encorajadas).
- A biblioteca padrão do C++ (STL) é permitida e incentivada, especialmente para usar estruturas como `std::map`, `std::vector` e `std::priority_queue`, e recursos como *smart pointers*.
- Porém, **não é permitido usar bibliotecas que implementem árvores ou que encapsulem a lógica de compressão de Huffman**. Essa parte deve ser implementada pela equipe.
- O projeto deve gerar 2 programas, ambos executáveis em linha de comando:
  - **contador de frequência**: esse programa tem como objetivo gerar uma tabela com a frequência dos símbolos mais usados na linguagem de programação escolhida. Esses símbolos podem ser tanto caracteres comuns (`a`, `b`, `1`, `&`, `*`,...) quanto palavras-chaves da linguagem. Por exemplo, em C++, podemos usar `int`, `double`, `while`, `return`, entre outros como símbolos a serem representados por uma sequência única de bits no algoritmo de Huffman. Vale salientar que os caracteres comuns também precisam estar nessa tabela para que eles possam ser codificados.
    Você pode usar arquivos fontes de projetos abertos (github) como fonte de "treinamento" para a contagem da frequência dos símbolos. Quando mais dados for usado na contagem, mais precisa será a probabilidade de surgimento dos símbolos do arquivo que você irá comprimir.
  - **compressor/descompressor**: esse programa vai adotar uma tabela de frequência (criada no programa anterior) para gerar uma codificação base. A codificação dos símbolos pode ser *hard-coded* (ou seja, você vai adotar como padrão para não ser necessário guardar a árvore de Huffman no arquivo comprimido). Essa codificação será usada para comprimir ou descomprimir arquivos de texto em geral (use uma opção em linha de comando para diferenciar a operação). Porém, como a codificação usará uma tabela gerada no programa anterior, ela estará (provavelmente) mais adaptada para comprimir códigos de programas.
- Além dos códigos do projeto, você deve também enviar um documento (pdf ou md) explicando a complexidade da solução implementada. Mais precisamente, você deve apresentar o custo de execução de cada uma das operações do processo de compressão e descompressão. Além disso, a equipe deve executar o seu programa de compressão e compará-lo com outros programas (zip, 7z, gzip ...). O documento deve

então fazer uma análise da taxa de compressão ($1 - tamanho\ comprimido / tamanho\ original$) obtida em diferentes tipos de arquivos (textos e códigos), comparando com essa taxa com a taxa de compressão de outros programas de compressão usando os mesmos arquivos.

Critérios de Avaliação

- **Contador de frequência (1 ponto):** O programa de geração da tabela de frequência funciona corretamente? A tabela é gerada condiz com os arquivos de entrada?
- **Compressão/descompressão (5 pontos):** O programa de geração de código implementa uma estrutura de Árvore de Huffman correta para a tabela de frequência usada? O programa codificam e descodifica corretamente usando a codificação adotada?
- **Qualidade do código (2 pontos):** Código encontra-se bem estruturado, comentado, legível e organizado em funções ou classes?
- **Relatório e análise (2 pontos):** O documento-relatório explicando faz uma análise correta da complexidade da solução (em tempo de execução)? É feita também uma análise da taxa de compressão obtida em diferentes tipos de arquivos?

Referências (para aprofundamento)

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos teoria e prática. Elsevier Editora Ltda., 3a Edição, 2012. Capítulo 16.3.
- R. Sedgewick, K. Wayne. Algorithms. Pearson, 4a Edição, 2014. Capítulo 5.5 - Data compression.