RELATÓRIO DO TRABALHO PRÁTICO 3: COMPACTAÇÃO DE ARQUIVOS DE TEXTO

Diogo Tuler Chaves - 2022043663

1. Introdução

Essa documentação lida com o problema da criação de um sistema de compactação baseado no algoritmo de Huffman fazendo as escolhas de projeto necessárias.

Ao ler o livro “Projeto de Algoritmos” do professor Nivio Ziviani foi descoberto que o Huffman baseado em palavras comprime o texto para aproximadamente 25% do tamanho normal, enquanto o baseado em caracteres apenas para 60%. Vale ressaltar que o baseado em palavras tinha um enorme problema, visto que se o arquivo fosse constituído apenas por palavras formadas por caracteres aleatórios (“ud8qidn9281dn129 ncie9dn1i20” por exemplo) ele não seria compactado, pois só a escrita das palavras e suas chaves já tornaria o arquivo maior que o original, além de que o tempo seria exorbitante.

Tendo em vista que nas especificações do trabalho o objetivo dele é “manter as informações presentes em todas as edições do jornal”, deduzi que todos, ou a maioria, dos arquivos que vão ser passados são, em teoria, textos normais presentes em jornais. Assim sendo, a melhor escolha,na minha visão, foi usar o Huffman por palavra, tendo em vista que iria acatar em uma compactação muito maior. Além disso, optei por não guardar espaços e quebras de linhas para uma maior compactação, colocando entre cada palavra um espaço simples na hora da descompactação. Para conseguir implementar o Huffman por palavras minimizando o gasto de memória e tempo, além de algumas funções hash para fazer buscas também foi utilizado o algoritmo de Moffat e Katajainen visto no livro “Projeto de Algoritmos” que calcula as alturas que cada palavra estaria em uma árvore canônica de huffman sem precisar de fato montar a árvore, através dessas alturas temos os comprimentos de cada código e podemos codificar cada palavra.

Como foi descrito nas especificações do trabalho que era necessário um “sistema de compactação baseado no algoritmo de Huffman” não vi problemas em utilizar o algoritmo de huffman com a implementação de Moffat e Katajainen que utiliza a árvore canônica de Huffman sem de fato ter que monta-la.

Vale ressaltar que esse trabalho foi realizado em C++ e compilado com g++ em um “Lenovo Lenovo IdeaPad S145-15 IWL" com o processador “Intel® Core™ i5-8265 U CPU @ 1.60GHz × 8” com "NV 118 / Mesa Intel® UHD Graphics 620 (WHL GT2)” e com um sistema operacional “Ubuntu 22.04.2 LTS 64-bit”

2. Métodos

2.1. Organização

- **bin:** Contém o arquivo executável “main”, resultado da compilação dos arquivos .o da pasta obj pelo G++.
- **include:** Contém os arquivos “.hpp” com as declarações de funções modularizadas ou de Tipos Abstratos de Dados. Além disso, alguns arquivos acabam por conter implementações, fato que será detalhado abaixo:
    - **tempo.hpp:** É o arquivo onde estão definidas funções que auxiliam na hora de analisar o tempo.

        - **Compare_Clock_Time**: Função auxiliar para "Time_Clock_Combine".
        - **Time_Clock_Combine**: Função que devolve a comparação de tempo em s.
    - **entrada.hpp:** É o arquivo onde está definida a função para análise dos argumentos da linha de comando e as exceções que ela pode enviar.
        - **le_args:** Função para ler os argumentos da linha de comando enviando exceções quando necessário
    - **compactar.hpp:** É o arquivo onde estão definidas funções que fazem a compactação.
        - **escreve_bit:** Função para escrever bit a bit no arquivo de saída.
        - **ocorrencia_com_hash:** Função para receber um arquivo de entrada e realizar a leitura, retornando as palavras que esse arquivo apresenta e sua frequência.
        - **escrever_chave:** Função para escrever do arquivo de saída as palavras que foram usadas e suas frequências.
        - **escrever_compactado:** Função para escrever no arquivo compactado.
        - **compactar:** Função para compactar o arquivo de entrada no de saída.
    - **descompactar.hpp:** É o arquivo onde estão definidas funções que fazem a descompactação.
        - **le_bit:** Função para ler bit a bit do arquivo de entrada.
        - **ler_chave:** Função para ler do arquivo de entrada as palavras que vão ser usadas e suas frequências.
        - **escrever_descompactado:** Função para escrever no arquivo descompactado.
        - **descompactar:** Função para descompactar o arquivo de entrada no de saída
    - **huffman.hpp:** É o arquivo onde estão definidas as funções relacionadas ao do algoritmo de huffman utilizando as técnicas de implementação e otimização de Moffat e Katajainen
        - **calcula_comprimento:** Função que, baseado no algoritmo de huffman, utiliza um processo de três fases proposto por Moffat e Katajainen para calcular as alturas que cada palavra estaria na árvore de Huffman sem de fato montar a árvore, economizando tempo e espaço.
        - **primeira_fase:** Função que transforma o vetor de frequências das palavras em um que contém em sua segunda posição o peso da árvore e no resto os índices dos pais dos nós internos da árvore de huffman.
        - **segunda_fase:** Função que transforma o vetor gerado na primeira fase em um que contém a profundidade dos nós internos.
        - **terceira_fase:** Função que transforma o vetor gerado na segunda fase em um que contém as alturas que as palavras estariam na árvore que é igual ao comprimento dos códigos.
        - **monta_off_base:** Função para mudar os vetores "base" e "offset" que serão vitais na hora de codificar.
        - **codifica:** Função para codificar uma palavra.
        - **monta_dicionario:** Função para montar um dicionário contendo as palavras, o tamanho dos seus códigos e seus códigos em número inteiro.
    - **utilidades.hpp:** Arquivo para armazenar funções e structs que são úteis ao longo do código.
        - **struct palavra_ocorrencia:** Struct para representar uma palavra e a quantidade de vezes que ela aparece.
        - **struct vetor_palavra_ocorrencia (vocabulario):** Struct que representa um vetor do tipo palavra_ocorrencia e seu tamanho.

        - **struct palavra_codigo (dicionario):** Struct para armazenar uma palavra, seu código e o tamanho do seu código.
        - **soma_letras:** Função para calcular a soma das letras de uma palavra.
        - **particao & ordena & quicksort:** Funções para fazer um quicksort com melhoria na escolha de pivô para ordenar as palavras pelas suas frequências.
    - **hash.hpp**: Arquivo com as classes e estruturas usadas ao longo do código para fazer hash e diminuir a complexidade de tempo.
        - **struct tipo_item_leitura**: Struct que representa um item da classe "elemento_hash_leitura".
        - **class elemento_hash_leitura**: Classe que é uma espécie de uma pilha, sendo totalmente adaptada para ser usada para fazer um hash na hora de leitura, a fim de diminuir a complexidade de tempo na hora de achar se certa palavra já foi adicionada ou não e incrementar frequências.
            - **elemento_hash_leitura::elemento_hash_leitura:** Construtor da classe.
            - **elemento_hash_leitura::~elemento_hash_leitura:** Destrutor da classe.
            - **elemento_hash_leitura::vazio:** Função que retorna se está vazio ou não.
            - **elemento_hash_leitura::procura_incrementa:** Função que procura a palavra nos itens armazenados e incrementa a ocorrência se encontrar.
            - **elemento_hash_leitura::insere:** Função para inserir uma nova palavra.
            - **elemento_hash_leitura::remove:** Função para remover o último item adicionado.
            - **elemento_hash_leitura::limpa:** Função para limpar os itens.
        - **struct tipo_item_escrita**: Struct que representa um item da classe "elemento_hash_escrita".
        - **class elemento_hash_escrita**: Classe que é uma espécie de uma pilha, sendo totalmente adaptada para ser usada para fazer um hash na hora de escrita, a fim de diminuir a complexidade de tempo na hora de achar as palavras correspondentes a certo código ou vice-versa.
            - **elemento_hash_escrita::elemento_hash_escrita:** Construtor da classe.
            - **elemento_hash_escrita::~elemento_hash_escrita:** Destrutor da classe.
            - **elemento_hash_escrita::vazio:** Função que retorna se está vazio ou não.
            - **elemento_hash_escrita::procura_cod:** Função que retorna o dicionário que contém o código passado como argumento.
            - **elemento_hash_escrita::procura_palavra:** Função que retorna o dicionário que contém a palavra passada como argumento
            - **elemento_hash_escrita::insere:** Função para inserir uma nova palavra.
            - **elemento_hash_escrita::remove:** Função para remover o último item adicionado.
            - **elemento_hash_escrita::limpa:** Função para limpar os itens.

- **obj**: Contém os arquivos .cpp da pasta src traduzidos para linguagem de máquina pelo compilador GCC e com a extensão .o (object).
- **src**: Contém as implementações dos arquivos.hpp da pasta "include" em arquivos .cpp, além disso contém a função main Nestes arquivos estão programadas toda a lógica e a robustez contra erros do programa.

2.2. Funcionamento

É passado como argumento na entrada a opção ("-c" = compactação e "-d" = descompactação) , um arquivo de entrada e um de saída.

3. Análise de Complexidade

Muitas funções servem apenas para uma maior organização do código e sua complexidade depende totalmente das funções chamadas internamente por elas. Tendo isso em vista, aqui está uma análise de complexidade das funções principais do programa:

- **compactar.hpp**
  - **escreve_bit:** Complexidade de tempo e espaço Θ(1);
  - **ocorrencia_com_hash:** Em relação a complexidade de espaço essa função gera um vetor da classe "elemento_hash_leitura" de tamanho MAX_HASH (definido como 1000) sendo que ao todo esse vetor vai armazenar n palavras e suas frequências, sendo n o número de palavras distintas no arquivo.Dessa forma sua complexidade de espaço é Θ(max(MAX_HASH,n))sendo que max retorna o maior entre os dois. Todavia, a função possui uma complexidade de tempo um pouco mais relativa, já tendo definido n como o número de palavras distintas no arquivo e definindo m como o total de palavras seu pior caso é dado por O(n*m), já seu melhor caso é quando n<=MAX_HASH e pode ser dado como Ω(n).
  - **escrever_chave:** Sendo n o número de palavras distintas possui como complexidade de tempo Θ(n), a sua de espaço é constante.
  - **escrever_compactado:** Em relação a complexidade de espaço essa função gera um vetor da classe "elemento_hash_escrita" de tamanho MAX_HASH (definido como 1000) sendo que ao todo esse vetor vai armazenar n palavras e suas frequências, sendo n o número de palavras distintas no arquivo.Dessa forma sua complexidade de espaço é Θ(max(MAX_HASH,n))sendo que max retorna o maior entre os dois. Todavia, a função possui uma complexidade de tempo um pouco mais relativa, já tendo definido n como o número de palavras distintas no arquivo e definindo m como o total de palavras seu pior caso é dado por O(n*m), já seu melhor caso é quando n<=MAX_HASH e pode ser dado como Ω(n).
  - **compactar:** Em relação a espaço possui como complexidade Θ(n) e em relação a tempo sua complexidade é a maior dentre as funções acima pois usa todas elas.
- **descompactar.hpp**
  - **le_bit:** Complexidade de tempo e espaço Θ(1);
  - **ler_chave:** Sendo n o número de palavras distintas possui como complexidade de tempo Θ(n), a sua de espaço é Θ(n) pois armazena essas chaves.
  - **escrever_descompactado:** Em relação a complexidade de espaço essa função gera um vetor da classe "elemento_hash_escrita" de tamanho MAX_HASH (definido

como 1000) sendo que ao todo esse vetor vai armazenar n palavras e suas frequências, sendo n o número de palavras distintas no arquivo.Dessa forma sua complexidade de espaço é Θ(max(MAX_HASH,n))sendo que max retorna o maior entre os dois. Todavia, a função possui uma complexidade de tempo um pouco mais relativa, já tendo definido n como o número de palavras distintas no arquivo e definindo m como o total de palavras seu pior caso é dado por O(n*m), já seu melhor caso é quando n<=MAX_HASH e pode ser dado como Ω(n).
    - **descompactar:** Em relação a espaço possui como complexidade Θ(n) e em relação a tempo sua complexidade é a maior dentre as funções acima pois usa todas elas.
- **huffman.hpp**
    - **calcula_comprimento:** Não utiliza nenhum espaço adicional e tem como custo de tempo o maior entre as três fases.
    - **primeira_fase & segunda_fase:** Sendo n a quantidade de palavras distintas no texto essa fase realiza Θ(n) operações, sendo essa sua complexidade de tempo.Suas operações são feitas no vetor de entrada e não usam espaço extra, sendo sua complexidade de espaço constante.
    - **terceira_fase:** Sendo n a quantidade de palavras distintas no texto essa fase realiza O(n) operações, sendo essa sua complexidade de tempo.Suas operações são feitas no vetor de entrada e não usam espaço extra, sendo sua complexidade de espaço constante.
    - **monta_off_base:** Apesar de possuir um laço duplo essa fase realiza Θ(n) operações(sua complexidade de tempo é essa),sendo n a quantidade de palavras distintas no texto, pois os dois laços possuem uma relação. Suas operações são feitas no vetor de entrada e não usam espaço extra, sendo sua complexidade de espaço constante.
    - **codifica:** Possui complexidade O(1) em questão de tempo e espaço, pois faz uma operação sem gasto de memória adicional.
    - **monta_dicionario:** Utiliza a função codifica n vezes, sendo n o número de palavras distintas no arquivo, sua complexidade é O(n), em relação a espaço são gastos O(n)
- **utilidades.hpp:** Arquivo para armazenar funções e structs que são úteis ao longo do código.
    - **soma_letras:** Sendo k a quantidade de char na palavra essa função realiza Θ(k) operações, sendo essa sua complexidade de tempo, não utiliza espaço adicional.
    - **particao & ordena & quicksort:** Sendo um quicksort tem complexidade de tempo no melhor caso de Θ(n log n) e como pior caso de $O(n^2)$, todavia foi implementado uma melhoria para evitar cair no pior caso. Suas operações são feitas no vetor de entrada e não usam espaço extra, sendo sua complexidade de espaço constante.
- **hash.hpp**: Arquivo com as classes e estruturas usadas ao longo do código para fazer hash e diminuir a complexidade de tempo.
    - **class elemento_hash_leitura**
        - **elemento_hash_leitura::procura_incrementa:** Supondo que o hash possui n elementos seu pior caso é ter que percorrer todos os elementos, logo sua complexidade de tempo é O(n), não gasta memória adicional, logo sua complexidade de tempo é O(1).
        - **elemento_hash_leitura::insere & remove:** Insere ou remove no inicio não gastantando tempo nem memória adicional O(1).
    - **class elemento_hash_escrita**
        - **elemento_hash_escrita::procura_cod & procura_palavra:** Supondo que o hash possui n elementos seu pior caso é ter que percorrer todos os elementos,

logo sua complexidade de tempo é O(n), não gasta memória adicional, logo sua complexidade de tempo é O(1).

- **elemento_hash_escrita::insere & remove:** Insere ou remove no inicio não gastantando tempo nem memória adicional O(1).

4. Estratégias de Robustez

4.1. Decisões de projeto

Para o funcionamento efetivo do código foram desenvolvidos diversos mecanismos de defesa que visam a programação defensiva, deixando o código mais robusto e impedindo sua falha em diversas situações. Antes de falar sobre esses mecanismos, é importante salientar sobre algumas decisões de projeto que impactaram diretamente a forma com que esses mecanismos foram implementados:

- Foi usada a compactação por palavra tendo em vista que será usada para textos de jornais, dessa forma temos uma compactação maior. Caso tenhamos arquivos constituídos apenas por palavras formadas por caracteres aleatórios (“ud8qidn9281dn129 ncie9dn1i20” por exemplo) eles não serão compactado, pois só a escrita das palavras e suas chaves já tornaria o arquivo maior que o original, além de que o tempo será exorbitante. Além disso, arquivos muito grandes e formados por caracteres aleatórios podem possuir problemas de memória, visto que inúmeras “palavras” terão de ser armazenadas para a compactação.
- Como não foi especificado que o arquivo descompactado tem que ser totalmente igual ao original, espaços e quebras de linhas não foram levados em conta no processo de compactação. Isso acabou por garantir uma maior compactação que continha apenas as palavras e pontuação, sendo essas, de fato, as informações do arquivo. Ao descompactar um espaço é colocado automaticamente depois de cada palavra, logo podemos ter um arquivo descompactado um pouco menor que o original.

4.2. Mecanismos defensivos e de tolerância a falhas

Tendo em vista as decisões de projeto e as possíveis falhas, foram usados bloco try/catch nas partes sensíveis do código e diferentes estruturas de dados que representavam os diferentes erros. Elas são:

- **struct muitos_argumentos=** Struct para enviar um erro no try catch quando temos um argv com mais argumentos que o necessário. Usado na função le_args.
- **struct poucos_argumentos =** Struct para enviar um erro no try catch quando temos um argv sem os argumentos necessários. Usado na função le_args.
- **struct arquivo_invalido=** Struct para enviar um erro no try catch quando temos como entrada uma opção inválida. Usado na função le_args.
- **struct opcao_invalida=** Função para ler os argumentos da linha de comando enviando exceções quando necessário.

Além dessas estratégias foram implementados pequenos ‘if’ dentro de algumas funções para evitar possíveis erros.

5. Análise Experimental

5.1. Análise de Tempo

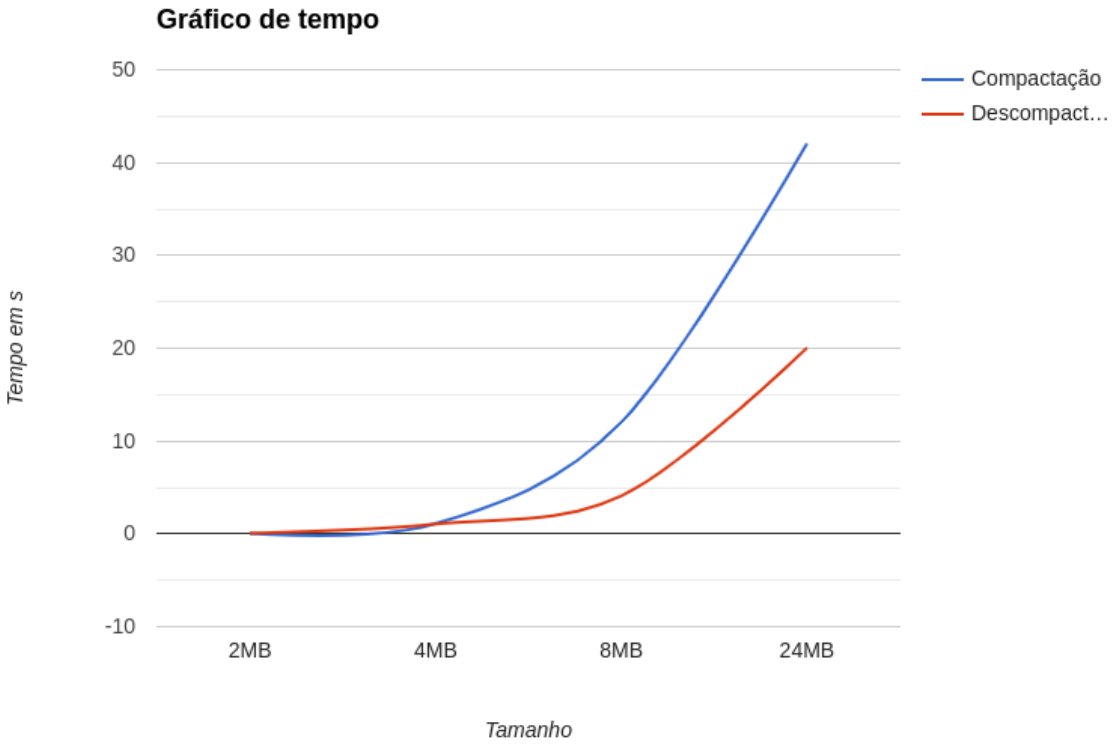
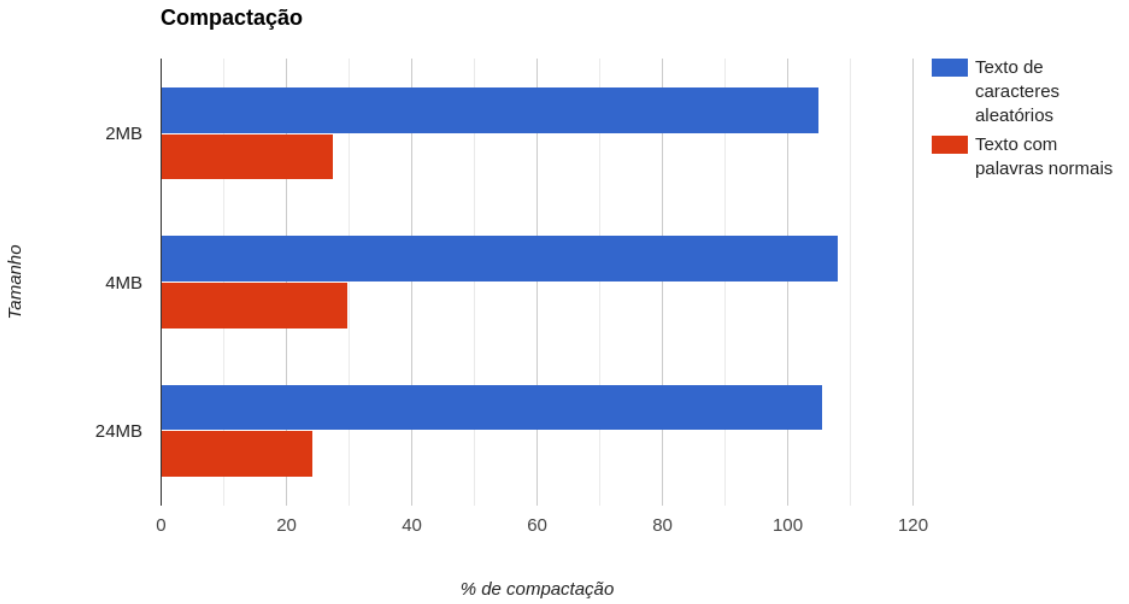


Gráfico do tempo de execução do programa para compactar e descompactar.

Em geral o algoritmo é limitado por O(n*m) sendo que as funções hash tentam ao máximo minimizar esse tempo. O gráfico acima acaba possuindo uma grande disparidade no eixo x, complicando um pouco a visualização. Todavia é visível que a função não é linear nem quadrática, mas sim algo intermediário que,de fato, era o de se esperar de acordo com a análise de complexidade feita. Por fim, é possível ver que o tempo para compactar acaba sendo maior pois, apesar de possuírem complexidades assintóticas parecidas, essa função precisa fazer buscas de palavras pelo dicionário mais vezes. Vale ressaltar que os arquivos utilizados foram de palavras naturais e que foi utilizado uma média, tendo em vista que uma variação dos arquivos podem resultar em mudanças exorbitantes no tempo graças a função hash.

5.2. Análise de Resultado



Como previsto, a compactação possui resultados fantásticos quando estamos trabalhando com textos normais. Todavia, em compensação, quando trabalhamos com textos formados apenas por caracteres aleatórios temos uma baixa diversidade de palavras, assim sendo o código acaba não se saindo bem, não realizando a compreensão.

Além disso, para códigos formados por caracteres totalmente aleatórios, o código pode apresentar falhas de segmentação dependendo da quantidade de palavras diferentes e falhas na compressão e descompressão, tendo em vista que ele foi pensado para comprimir textos de jornais que, de fato, não vão ser esse tipo de arquivo.

6. Conclusões

Em conclusão, no trabalho foi desenvolvido um programa para comprimir e descomprimir arquivos de texto utilizando como base o algoritmo de huffman.

Foi muito interessante trabalhar com tipos diferentes de algoritmos e otimização, fazendo as escolhas de projeto para o que melhor encaixava com o proposto. Meu maior foco foi a maior compactação possível de textos em linguagem natural, assim sendo acabou apresentando falhas para arquivos de caracteres aleatórios, todavia resultados absurdos de compactação para arquivos de linguagem natural. Isso foi interessante para entender o conceito de ‘trade-off’, sendo necessário uma busca árdua para otimização do código.

Além disso, é sempre interessante trabalhar com estruturas de dados e formas de programar novas. Este trabalho foi bastante importante para colocar em prática coisas vistas durante as aulas, como por exemplo o “Quicksort” e o “Hash”.

Por fim, o relatório teve extrema importância na absorção de conhecimento, pois tornou obrigatório o bom planejamento das ideias e uma implementação organizada, limpa e coerente. Tais fatores são vitais no desenvolvimento de qualquer programa e treinar isso é sempre muito agregador.

7. Bibliografia

- **https://stackoverflow.com/questions/1856514/writing-files-in-bit-form-to-a-file-in-c?rq=1 Acesso em: 26 de abril de 2023., Acesso em: 26 de junho de 2023.**
- **https://media.oiipdf.com/pdf/3aa7cada-0c5d-4ce4-b331-e2f8a0f4a305.pdf, Acesso em: 24 de junho de 2023.**
- **https://www.youtube.com/watch?v=5pYYy3Z0m9I, Acesso em: 24 de junho de 2023.**
- **https://www.youtube.com/watch?v=wrOrg1GdS-0, Acesso em: 24 de junho de 202**

8. Instruções para compilação e execução

- Acessar a pasta do TP pelo terminal
- Adicionar o arquivo a ser lido na pasta raiz do projeto, isto é, na pasta que contém o Makefile.
- Para compilar é necessário dar o comando make
- Para rodar é necessário dar o seguinte comando:
    - ./bin/main <op> <arquivo_entrada> <arquivo_saida>
        - ‘op’ pode ser ‘-c’ ou ‘-d’
- O make clean é importante na limpeza dos objetos e o executável gerado na compilação do programa, sendo recomendado sua utilização entre testes.