Trabalho Prático: Ordenação em memória externa

Gabriel Teixeira Carvalho

1. Introdução

Esta documentação lida com o problema de implementar um ordenador em memória secundária. O objetivo principal desta tarefa é criar um programa que, dada uma série de entidades (formadas por uma URL e um número de visitas), consiga dividi-las em fitas, ordenar as fitas decrescentemente segundo o número de visitas (ou ordem alfabética, em caso de empate) e depois intercalar as fitas, ordenando-as na memória secundária segundo os mesmos critérios. Além disso, outro foco importante foi a revisão de conceitos de classes, ordenação, estruturas de dados, desempenho e robustez de código, sendo muito aplicados ao longo da implementação. Para resolver o problema citado, foi utilizada uma classe que gera fitas ordenadas através do algoritmo QuickSort e uma que as intercala através da estrutura de dados Heap.

A seção 2 desta documentação trata mais a fundo sobre como o programa foi implementado, enquanto na seção 3 é apresentada uma análise da complexidade temporal e espacial do trabalho. Já na seção 4, são apresentados aspectos de robustez e proteção contra erros de execução. A seção 5 destrincha o comportamento do programa em termos de tempo e espaço através de diversos experimentos realizados após a implementação do programa. Por fim, a seção 6 conclui e sumariza tudo que será falado nesta documentação, sendo seguida por referências bibliográficas utilizadas na confecção do código e por um manual de compilação e execução do programa.

2. Método

2.1. Ambiente de desenvolvimento

O programa foi implementado na linguagem C++, compilado pelo G++, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL2), com um processador Intel Core i7-6500U (2.50GHz, 4 CPUs) e 8192 MB RAM.

2.2. Organização, Arquivos e Funcionamento

O código está organizado em 4 diretórios na pasta raiz:

- `obj`: Esta pasta contém os arquivos `.cpp` da pasta `src` traduzidos para linguagem de máquina pelo compilador GCC e com a extensão `.o` (object).
- `bin`: Contém o arquivo executável `main`, que consiste nos arquivos .o da pasta `obj` compilados em um só arquivo pelo GCC.
- `include`: Contém os arquivos de declaração (contrato) das classes utilizados no programa, com seus atributos e métodos especificados dentro

de arquivos da extensão `.hpp`, que serão incluídos e implementados nos arquivos de extensão `.cpp`:

- `entidade.hpp`: Define o TAD Entidade que guarda a URL, o número de visitas e eventualmente o arquivo de onde ela veio.
- `quicksort.hpp`: Define a classe QuickSort, com métodos para ordenação de um vetor de entidades e um atributo que guarda o tamanho do vetor a ser ordenado.
- `heap.hpp`: Define a classe `Heap`, formada por um atributo de tamanho máximo, o número de entidades presentes e um vetor de entidades alocado na memória secundária. Possui métodos para adicionar e remover elementos do `Heap` e métodos para construir ou refazer o `Heap`.
- `geradorderodadas.hpp`: Define a classe `GeradorDeRodadas` que possui um atributo que indica o número de entidades que serão ordenadas na memória principal. Essa classe possui métodos que permitem ler entidades de um arquivo de entrada, ordenar as entidades lidas e escrever as entidades em ordem em arquivos de saída separados chamados de fitas.
- `intercalador.hpp`: Define a classe Intercalador, formada por um `Heap` e responsável pela intercalação das fitas geradas pelo `GeradorDeRodadas` e pela impressão das fitas intercaladas e ordenadas em um arquivo de saída. O número de fitas e de entidades a serem manipuladas e o nome do arquivo de saída são guardados em atributos da classe.
- `memlog.h`: Define um TAD que gerencia o registro dos acessos à memória e o registro do desempenho, através de métodos utilizados sempre que há leitura ou escrita em memória.
- `msgassert.h`: Define macros que conferem condições necessárias para o funcionamento correto do programa, abortando a execução ou imprimindo um aviso caso estas condições sejam descumpridas.

- `src`: Contém as implementações das classes declarados nos arquivos `.hpp` da pasta `include` em arquivos `.cpp`; Nestes arquivos estão programadas as regras, a lógica e a robustez contra erros que é o alicerce de todo o programa:
  - `quicksort.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `quicksort.hpp`. Métodos mais importantes:
    - `particao`: Particiona um vetor de entidades a partir de um pivô escolhido. Elementos maiores do que o pivô são colocados à sua esquerda e elementos menores vão para sua direita. O pivô escolhido é a mediana de 3 elementos do vetor para evitar que o QuickSort caia no seu pior caso.
    - `ordena`: Chama o método `particao` e o próprio `ordena` para ordenar o vetor de entidades recursivamente.

- `heap.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `heap.hpp`. Métodos mais importantes:
    - `constroi`: Constrói um `Heap` dado um vetor de entidades.
    - `refaz`: Refaz a estrutura do vetor de entidades para ficar no formato de um `Heap`.
    - `adiciona`: Insere uma entidade no fim do `Heap` e depois reconstrói o `Heap` para ficar no formato correto.
    - `remove`: Remove a entidade que está no topo do Heap (possui o maior número de visitas) e depois refaz o `Heap`.

- `geradorderodadas.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `geradorderodadas.hpp`. Métodos mais importantes:
    - `leEntidades`: Lê de um arquivo de entrada o número de entidades especificadas por um parâmetro passado para o programa e armazena em um vetor de entidades.
    - `ordena`: Ordena o vetor de entidades passado como parâmetro através do método QuickSort.
    - `escreve`: Escreve em um arquivo de saída as entidades de um vetor de entidades. O nome do arquivo de saída é definido pelo programa principal de acordo com a ordem de leitura das entidades do arquivo de entrada e tem o formato rodada-{numeroDaRodada}.txt, sendo numeroDaRodada um parâmetro passado para o método.

- `intercalador.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `intercalador.hpp`. Métodos mais importantes:
    - `intercala`: Recebe um vetor de arquivos (fitas) como parâmetro, lê e adiciona no `Heap` a primeira entidade de cada fita. Após isso, enquanto há elementos no `Heap`, remove a entidade do topo, imprime-o em um arquivo de saída e adiciona no `Heap` o próximo elemento que vem do mesmo arquivo que o elemento removido, intercalando as fitas recebidas. Ao final desse método, as entidades recebidas no arquivo de entrada estarão ordenadas no arquivo de saída.
- `main.cpp`: Este arquivo implementa a função principal do programa, responsável por controlar o fluxo de execução do código. Esse arquivo é responsável pela leitura e atribuição dos parâmetros passados para o programa, instanciação dos objetos das classes implementadas, abertura dos arquivos de entrada e saída e chamada dos métodos das classes para que a ordenação seja feita.

- `memlog.c`: Neste arquivo está presente a implementação do TAD que registra os acessos de memória e desempenho de tempo. Métodos mais importantes:
    - `leMemLog`: Registra em um arquivo uma operação de leitura de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
    - `escritaMemLog`: Registra em um arquivo uma operação de escrita de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
    - `defineFaseMemLog`: Separa as fases de execução do programa, fundamental para a análise experimental.

Além disso, há um arquivo `makefile`, responsável pela compilação modularizada do programa, permitindo compilar somente os módulos necessários, ao invés de recompilar todos os arquivos a cada mudança.

2.3. Detalhes de implementação

As estruturas de dados fundamentais para o trabalho, utilizadas para ordenar as entidades, foram vetores alocados estática e dinamicamente. Os vetores que foram alocados dinamicamente foram devidamente desalocados para evitar vazamentos de memória. Outra estrutura importante é o TAD `memlog_tipo`, que guarda informações necessárias para o registro de performance.

Além disso, o programa recebe 4 parâmetros pela linha de comando:

- Nome do arquivo de entrada
- Nome do arquivo de saída
- Número de entidades armazenadas na memória principal (Tamanho da memória)

O número de fitas é calculado de acordo com o número de entidades no arquivo de entrada e o tamanho da memória. Se o tamanho da memória for maior ou igual ao número de entidades no arquivo, a intercalação se torna desnecessária pois só uma fita é gerada, logo, esse caso é tratado pelo código. Por fim, cabe ressaltar que todas as passagens de vetores como parâmetro para as funções e métodos é feita por referência, passando o endereço ao invés de copiar a matriz em questão. Isto economiza grandes quantidades de memória a cada chamada de função.

3. Análise de Complexidade

Nesta seção será analisada a complexidade tanto de tempo como de espaço das funções e métodos descritos acima. Sendo N = tamanho da memória (valor passado como parâmetro), P = número de fitas, M = número de elementos no `Heap` e assumindo que as operações de leitura e escrita em arquivo sejam O(1) em complexidade de tempo:

Método `QuickSort::particao` – complexidade de tempo:

Esse método percorre o vetor de entidades comparando e trocando elementos que estão antes e depois do pivô, organizando o vetor, até que o índice que vem da esquerda encontre o índice que vem da direita. Portanto, esse método percorre todas as posições do vetor de tamanho N, sendo assim, $\Theta(N)$.

Método `QuickSort::particao` – complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada e o vetor de entidades é passado como parâmetro por referência, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `QuickSort::ordena` - complexidade de tempo:

Esse método chama o método `particao` e também chama o próprio `ordena` recursivamente uma vez em cada partição do vetor de entidades. No melhor caso, no qual as duas partes têm sempre tamanhos iguais, esse método tem equação de recorrência $T(N) = 2(N/2) + N$, que possui solução $\Theta(N\log N)$. No caso médio, é demonstrado que o QuickSort também possui complexidade $\Theta(N\log N)$. No pior caso, quando o pivô escolhido é repetidamente uma das extremidades do vetor, o QuickSort é $\Theta(N^2)$, porém, esse caso é evitado no programa ao definir o pivô como a mediana de uma amostra de três valores do vetor. Portanto, esse método possui complexidade $\Theta(N\log N)$.

Método `QuickSort::ordena` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada e o vetor de entidades é passado como parâmetro por referência, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `Heap::constroi` - complexidade de tempo:

Esse método chama o método `refaz` uma vez para cada nó do `Heap` que não é folha, ou seja, M/2 vezes. Portanto, esse método é $O(M\log M)$.

Método `Heap::constroi` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `Heap::refaz` - complexidade de tempo:

Esse método percorre, no pior caso, todo um galho de uma árvore binária, ou seja, executa $\log M$ operações. Portanto, esse método é $O(\log M)$.

Método `Heap::refaz` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `Heap::adiciona` - complexidade de tempo:

Esse método adiciona uma entidade ao `Heap`, e chama o método constrói uma vez. Portanto, esse método é O(MlogM).

Método `Heap::adiciona` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `Heap::remove` - complexidade de tempo:

Esse método salva e retorna o elemento no topo do `Heap` e chama o método refaz para refazer a estrutura do `Heap`. Portanto, esse método é O(logM).

Método `Heap::remove` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `GeradorDeRodadas::leEntidades` - complexidade de tempo:

Esse método lê N entidades de um arquivo e armazena em um vetor de entidades. Portanto, é Θ(N) em complexidade de tempo.

Método `GeradorDeRodadas::leEntidades` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada e o vetor de entidades é passado como parâmetro por referência, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `GeradorDeRodadas::ordena` - complexidade de tempo:

Esse método executa o algoritmo de ordenação QuickSort, que é Θ(NlogN), como mostrado acima.

Método `GeradorDeRodadas::ordena` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada e o vetor de entidades é passado como parâmetro por referência, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `GeradorDeRodadas::escreve` - complexidade de tempo:

Esse método escreve as N entidades ordenadas do vetor de entidades em um arquivo. Portanto, é Θ(N) em complexidade de tempo.

Método `GeradorDeRodadas::escreve` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada e o vetor de entidades é passado como parâmetro por referência, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `Intercalador::intercala` - complexidade de tempo:

Esse método lê a primeira entidade de cada fita e a adiciona no `Heap` com o método `adiciona`, O(P*MlogM). Depois, enquanto o `Heap` não está vazio, esse método `remove` (O(logM)) uma entidade do topo, imprime no arquivo de saída e `adiciona` (O(MlogM)) mais um elemento no `Heap`. Como em cada iteração desse loop uma entidade é retirada e outra é adicionada, o máximo de elementos no Heap acontece quando o primeiro elemento de cada fita foi adicionado ao Heap. Ou seja, M é menor ou igual a P. Como esta parte do método é executada uma vez para cada elemento lido dos arquivos menos os primeiros elementos de cada fita. Logo, esse loop itera N*P – P vezes, O(N*P). Portanto, esse método é O(P*MlogM + N*P(logM + MlogM)) = O(P*PlogP + N*P(logP + PlogP)) = $O(N*P^2*logP)$.

Método `Intercalador::intercala` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Função `main` - complexidade de tempo:

Essa função controla a execução do programa. Ela recebe parâmetros da linha de comando (Θ (1)), chama os métodos leEntidades (Θ (N)), ordena (Θ (NlogN)), escreve (Θ (N)) P vezes, cria um vetor com P arquivos (Θ (P)) e chama o método `Intercalador::intercala` uma vez ($O(N*P^2*logP)$). Portanto, o programa tem complexidade O((N + NlogN + N)*P + P + $N*P^2*logP$) = O(P*NlogN + $N*P^2*logP$).

Função `main` - complexidade de espaço:

Essa função aloca um vetor de entidades de tamanho N, um vetor de arquivos de tamanho P e, além disso, chama o construtor do Intercalador, que aloca um vetor de entidades de tamanho N*P, portanto, o programa tem complexidade de espaço Θ(N + P + N*P) = Θ(N*P).

Método `memlog::leMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::leMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Método `memlog::escreveMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::escreveMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Método `memlog::defineFaseMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é $\Theta(1)$.

Método `memlog::defineFaseMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é $\Theta(1)$.

Além disso, para o registro dos acessos à memória, a cada operação de leitura ou escrita será realizada uma operação de impressão de complexidade O(1). Essa opção está desativada por padrão no código, mas pode ser ativada facilmente para que as análises sejam feitas. Isso não muda a ordem de complexidade da operação escolhida, porém, aumenta a constante que multiplica a função de complexidade da operação. Portanto, essa opção pode reduzir bastante a performance do código.

4. Estratégias de Robustez

Ao longo do código, a macro `erroAssert é utilizada para tratar erros`. A macro assegura certas condições importantes para o programa, gerando uma mensagem e abortando a execução do programa caso a condição não seja atendida. Os mecanismos de programação defensiva e robustez utilizados foram:

- Asserção de número de argumentos para o programa – aborta o programa se o número de argumentos passados para o programa for diferente do esperado.
- Asserção de nome do arquivo passado por parâmetro – aborta o programa se o nome do arquivo passado por parâmetro não for especificado.
- Asserção de abertura de arquivo – aborta o programa se o arquivo de comandos ou de saída não forem abertos.
- Asserção de parâmetro numérico válido – aborta o programa se algum parâmetro numérico tiver um valor inválido.
- Asserção de leitura correta de arquivo – aborta o programa se alguma linha inválida for lida do arquivo de entrada.
- Asserção de leitura de valor válido de arquivo – aborta o programa se a URL ou o número de visitas lidos do arquivo tenham valores inválidos.

5. Análise Experimental

Essa seção compila uma série de experimentos de diferentes tipos que analisam a performance do programa por meio dos registros do TAD `memlog_tipo`.

5.1. Análise de tempo

Esse experimento tem como objetivo medir o tempo de execução ao performar as operações implementadas no código com diferentes cargas de trabalho

O tempo é obtido ao subtrair o tempo inicial do tempo final, os quais estão presentes no arquivo de registro de desempenho. Foram geradas 10 cargas de trabalho que variam na quantidade de entidades, de 100 até 512000,

organizadas em ordem aleatória. Foi feita a mesma bateria de testes para cada carga: o programa é executado 10 vezes e a média de tempo entre as execuções é calculada. Foram realizados testes com cada número de entidades enquanto o parâmetro tamanho da memória do programa era mantido em um tamanho mediano, relativo ao número de entidades. Além disso, a análise foi feita separadamente para cada fase do programa, geração de rodadas ordenadas e intercalação de rodadas.

Ao analisar os dados obtidos e traçando um gráfico com a ajuda do programa GNUPLOT, percebe-se que o programa tem um comportamento que condiz com a análise de complexidade. A geração de rodadas tem um desempenho que segue aproximadamente a curva NlogN e o intercalador tem desempenho aproximadamente cúbico, como analisado anteriormente.

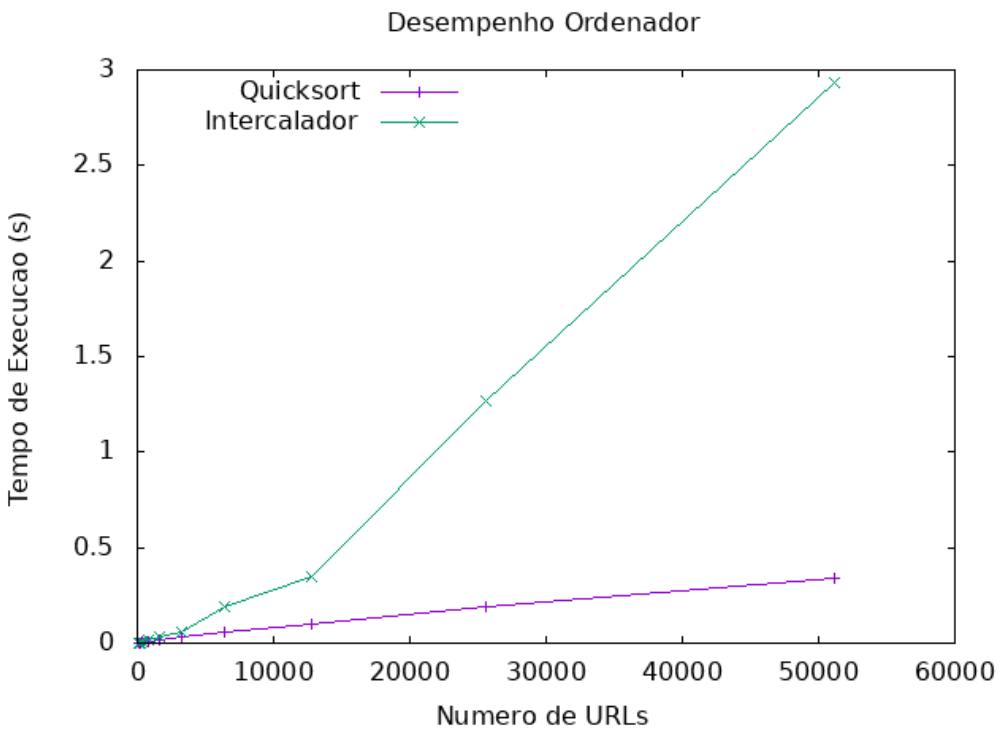

Figura 1: gráfico de desempenho de tempo

Numero de URLs	Quicksort	Intercalador
100	0.002204	0.00131
200	0.003057	0.00235
400	0.004913	0.00850
800	0.008107	0.01443
1600	0.018452	0.03021
3200	0.034517	0.05755
6400	0.060594	0.19293
12800	0.097396	0.35143
25600	0.190513	1.26804
51200	0.343789	2.93474

Figura 2: tabela de desempenho de tempo

Analisando o impacto do parâmetro tamanho da memória em uma entrada de tamanho constante, vemos que, quanto maior o tamanho da memória, menor o tempo de execução. Isso acontece pois com menos fitas serão geradas e, dessa forma, a intercalação será menor. Como a intercalação tem complexidade maior, o tempo de execução acaba sendo menor. Porém, o ato de intercalação é justificado quando a memória principal se torna escassa, tornando-se necessário o processo de ordenação em memória externa.

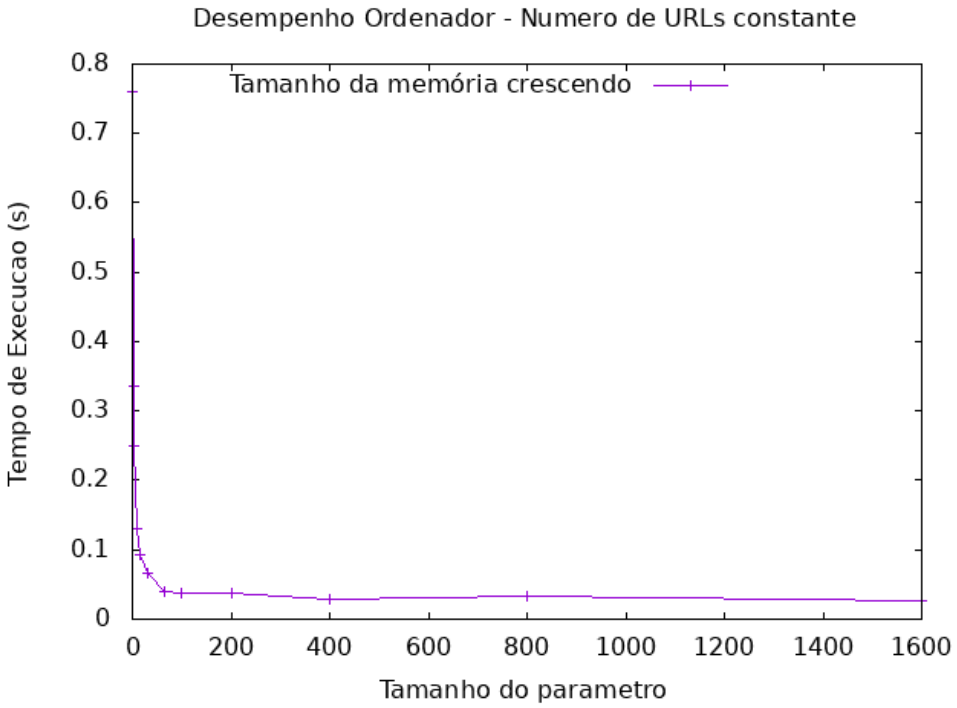

Figura 3: gráfico de desempenho de acordo com o tamanho da memória

Além disso, analisando o perfil de execução do código com o programa GPROF, chega-se a outras conclusões importantes:

Do tempo de execução total do programa, mais de 90 % corresponde à execução do método `intercala` e, dessa porcentagem, mais de 80% corresponde ao método `refaz`.

```
                0.01    0.35       1/1          main [1]
[2]     94.0    0.01    0.35       1         Intercalador::intercala(std::basic_ifstream<char, std::char_traits<char>
                0.01    0.33   25600/25600       Heap::adiciona(Entidade) [3]
                0.00    0.00   25600/25600       Heap::remove() [18]
                0.00    0.00  154112/17950914     escreveMemLog(long, long, int) [6]
                0.00    0.00   76800/3539077     Entidade::~Entidade() [10]
                0.00    0.00  128000/14275240     leMemLog(long, long, int) [8]
                0.00    0.00   25600/3487877     Entidade::Entidade() [9]
                0.00    0.00       1/1           Heap::desaloca() [22]
                0.00    0.00   25601/25601       Heap::getNumeroDeEntidades() [35]
                0.00    0.00   25600/51200       Entidade::Entidade(Entidade const&) [34]
```

Figura 4: perfil de execução do programa

Além disso, o registro de acesso pode aumentar consideravelmente o custo para executar o código, chegando a tomar quase 35% do tempo de execução total.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.36      0.02     0.02  1436985     0.00     0.00  clkDifMemLog(timespec, timespec, timespec*)
 33.36      0.04     0.02   143687     0.00     0.00  Heap::refaz(int, int)
 16.68      0.05     0.01   641713     0.00     0.00  leMemLog(long, long, int)
 16.68      0.06     0.01     1600     0.01     0.04  Heap::constroi()
```

Figura 5: perfil de execução do programa com o registro de acesso ativado

5.2. Localidade de referência

Outra forma de estudar a performance do programa é fazendo um Mapa de Acesso à Memória, o qual permite enxergar claramente como são feitos os acessos à memória do computador durante a execução das operações, tanto no tempo quanto no espaço.

Esse tipo de observação leva em conta os princípios de Localidade de Referência Espacial e Temporal, que dizem que um acesso ao endereço X no tempo T implica que acessos ao endereço X + dX no tempo T + dT se tornam mais prováveis à medida que dX e dT tendem a zero.

Para facilitar essa análise, as funções registradoras (`leMemLog` e `escreveMemLog`) registram uma fase e um ID em todo registro:

- Um ID identifica a qual classe o registro se refere e é atribuído à classe no momento em que são feitos os registros de acesso, pois no arquivo respectivo de cada classe, as funções registradoras estão com o ID correspondente. O ID 0 corresponde ao `QuickSort`, o ID 1 corresponde ao `GeradorDeRodadas`, o ID 2 corresponde ao `Heap` e o ID 3 corresponde ao `Intercalador`.
- Uma fase representa um intervalo de tempo no qual as operações realizadas são semelhantes, o que permite separar o registro em fases com diferentes análises. As fases são definidas através da função `defineFaseMemLog`, dentro de cada operação do código. Fase 0: inicialização das estruturas. Fase 1: Geração de Rodadas. Fase 2: Intercalação de Fitas.

Esse estudo foi feito a partir da geração de gráficos com tempo no eixo X e endereço de memória no eixo Y, após análises dos registros gerados com o registro de acesso ativo ao realizar a ordenação de 16 entidades com um tamanho de memória igual a 4.

5.2.1. QuickSort

Analisando a localidade de referência da Classe `Quicksort`, percebe-se uma concentração de acessos em duas faixas distantes entre si. Além disso, dentro de cada faixa, os acessos estão bastante separados verticalmente.

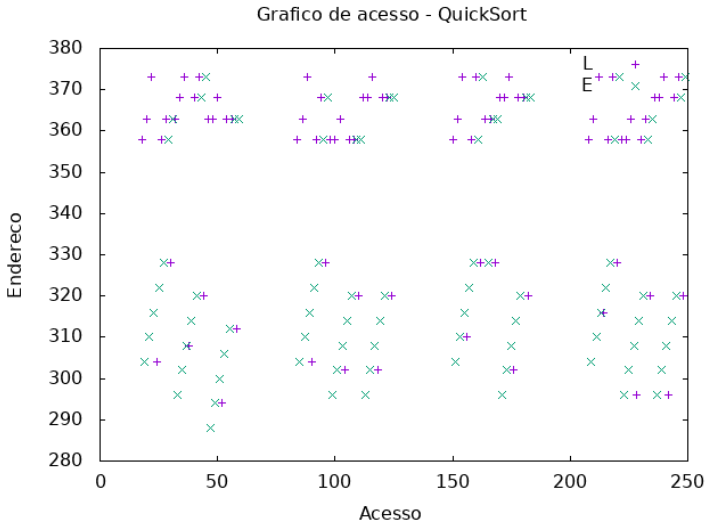

Figura 6: Mapa de acesso à memória da classe QuickSort

5.2.2. Gerador de Rodadas

Analisando a localidade de referência da Classe `GeradorDeRodadas`, percebe-se a mesma dispersão presente na classe `GeradorDeRodadas`, com faixas distantes entre si. Porém, nesse caso, os acessos da faixa inferior estão mais concentrados do que na superior.

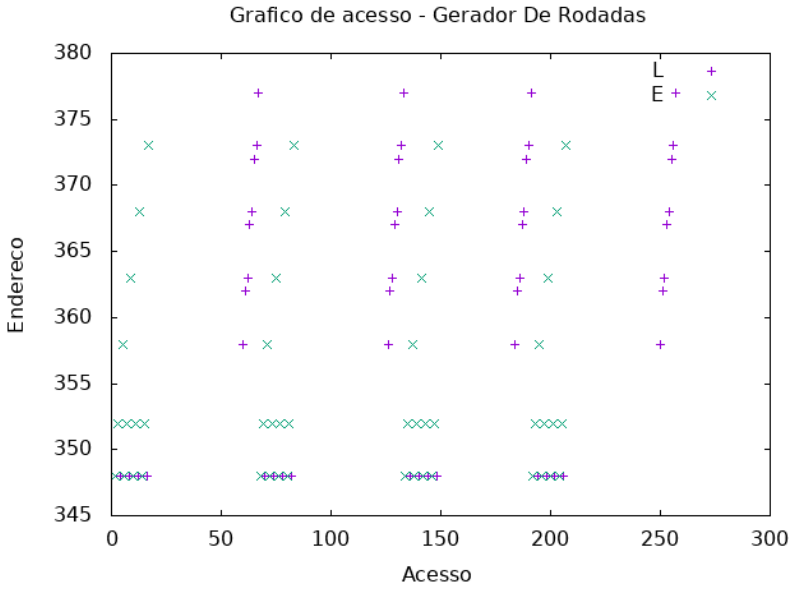

Figura 7: Mapa de acesso à memória da classe Gerador de Rodadas

5.2.3. Heap

No caso do `Heap`, percebe-se acessos concentrados em 4 endereços, indicando uma boa localidade de referência. Esse número de endereços depende do número de fitas pois, como já dito antes, a cada iteração um elemento é adicionado e outro é retirado do Heap, mantendo o número máximo de elementos do Heap constante.

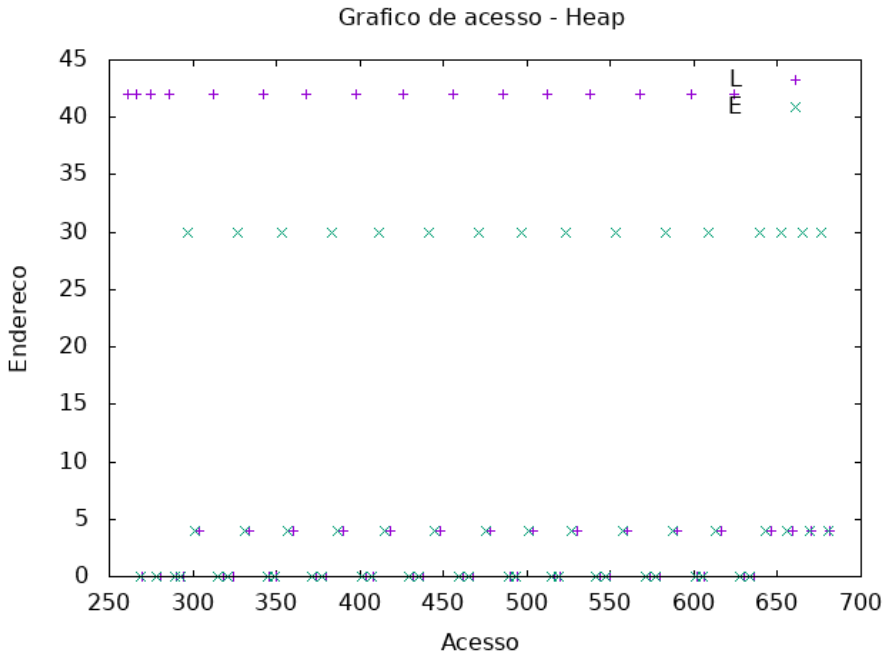

Figura 8: Mapa de acesso à memória da classe Heap

5.2.4. Intercalador

No caso do Intercalador, os acessos estão concentrados em 4 endereços, correspondentes ao número de fitas, indicando uma boa localidade de referência.

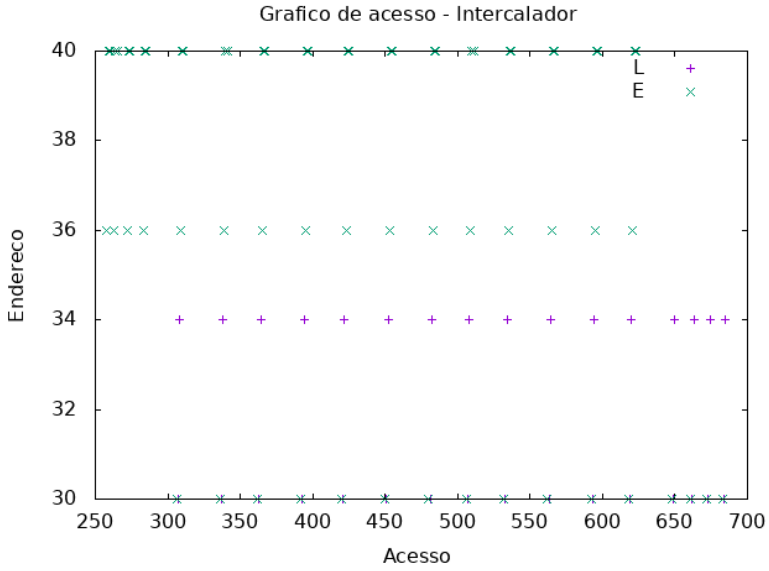

Figura 9: Mapa de acesso à memória da classe Intercalador

De posse desses gráficos, podemos ver que a localidade de referência das 4 classes tem seus pontos positivos e negativos, com acessos concentrados em faixas, mas em faixas separadas entre si, exceto no QuickSort que tem seus acessos relativamente separados em ambas as faixas.

Portanto, para aperfeiçoar o desempenho do programa, seria válido analisar formas de melhorar a localidade de referência dessas classes, evitando movimentações desnecessárias através da memória, mas que não tornassem o

código complexo demais. Um exemplo de melhoria possível seria a separação, se possível e de forma razoável, dos métodos que acessam posições afastadas na memória em classes ou etapas diferentes, evitando acessos alternados a localidades distantes.

5.3. Distância de pilha

Outra maneira de analisar a performance do programa e que corrobora com o que já foi analisado anteriormente é a distância de pilha (DP). Cada vez que um endereço é acessado, ele é colocado no topo de uma pilha inicialmente vazia. Quando o endereço é acessado novamente, sua posição na pilha é a distância de pilha.
Fazendo uma soma ponderada de todas as distâncias de pilha com suas frequências de um conjunto de procedimentos, chega-se à distância de pilha desse conjunto e quanto maior ele seja, pior é a localidade de referência da operação. Isso se dá, pois, um mesmo endereço demora mais a ser acessado novamente se sua distância de pilha é alta, ferindo o princípio da localidade de referência. Analisando esses aspectos, é possível tirar algumas conclusões a respeito das diferentes operações e classes.

Em todos os gráficos, percebe-se aquilo que já foi constatado nos mapas de acesso: A classe `QuickSort` não acessa a memória tão bem, pois possui uma distância de pilha considerável e com distâncias espalhadas. A classe `GeradorDeRodadas` possui comportamento semelhante, mas com distâncias mais concentradas, seguindo melhor o princípio da localidade de referência. As classes Heap e Intercalador, possuem distâncias de pilha menores e seguem também o princípio da localidade de referência, devido aos acessos recorrentes aos mesmos endereços.

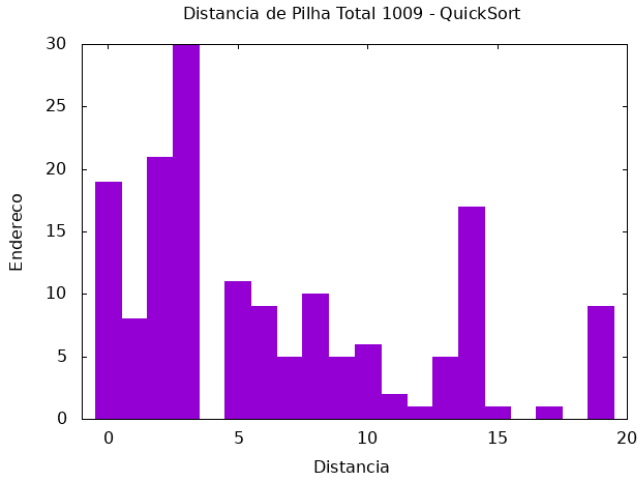

Figura 7: Histograma de distância de pilha da classe QuickSort

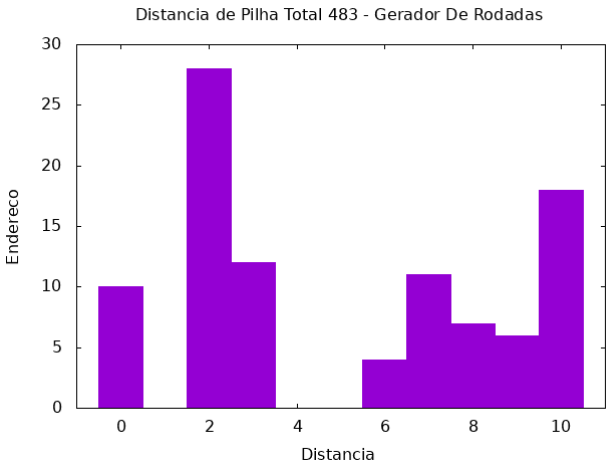

Figura 8: Histograma de distância de pilha da classe Gerador De Rodadas

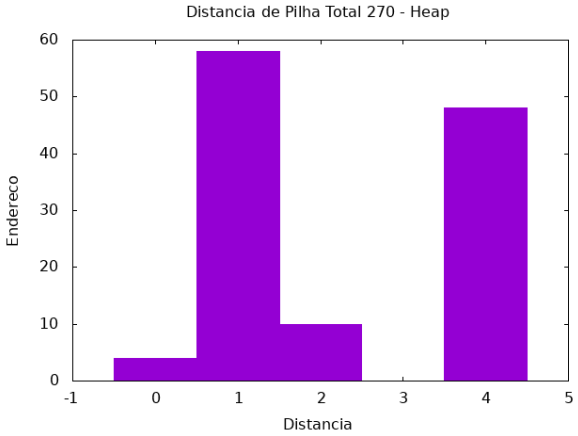

Figura 9: Histograma de distância de pilha da classe Heap

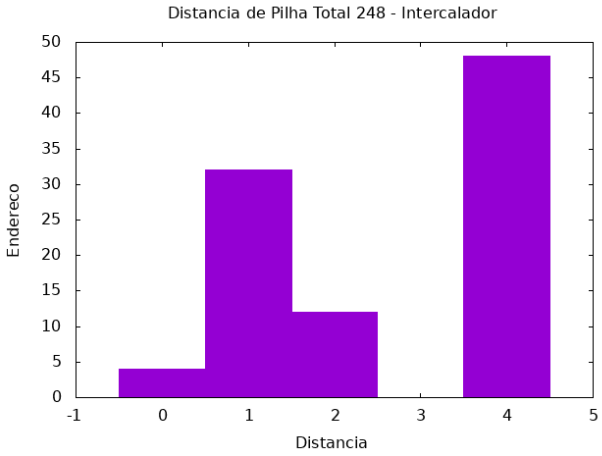

Figura 10: Histograma de distância de pilha da classe Intercalador

6. Conclusão

Este trabalho lidou com a tarefa de implementar um ordenador em memória secundária, na qual a abordagem utilizada para a resolução foi a implementação de classes com métodos para realização das operações em vetores alocados estática e dinamicamente. Com a solução adotada, pode-se resolver um problema comum de uma forma inovadora e robusta, preparada para diferentes cargas de trabalho em termos de memória. Por meio da resolução desse trabalho, foi possível aplicar e revisar conceitos relacionados a estruturas de dados, ordenação, alocação de memória, análise de complexidade de tempo e espaço, programação defensiva e performance de programas.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo, entender a especificação, implementar a lógica de ordenação por ordem alfabética, definição das operações e responsabilidades relevantes a cada classe, análise experimental do desempenho do programa, organização do projeto, definição das frentes de trabalho a serem seguidas e a criação de uma documentação completa.

7. Bibliografia

Chaimowicz, L. e Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Passing an array by reference Stack Overflow, 2011. Disponível em: < https://stackoverflow.com/questions/5724171/passing-an-array-by-reference>. Acesso em: 13 de jan. de 2022.

Instruções para compilação e Execução

1 – Extraia o arquivo `.zip` na pasta desejada.

2 – Execute o comando "`cd TP`" no terminal

3 – Execute o comando "`make all`" no terminal para compilar os módulos do programa

4 – Execute o programa `main` da pasta `bin` passando um arquivo de texto com os comandos pela linha de comando. Um arquivo de teste está presente na pasta tests.