Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Trabalho Prático 1
Ordenador Universal

João Pedro Moreira Smolinski

2024023996
jp.smolinski05@gmail.com

Conteúdo

1 Introdução

Neste documento analisaremos e detalharemos a forma de resolução do Trabalho Prático 1 da disciplina de Estruturas de Dados (DCC205-2025/1).

Deseja-se desenvolver um Ordenador Universal, um TAD (Tipo Abstrato de Dados) capaz de selecionar automaticamente o melhor algoritmo de ordenação para um vetor com base em suas características de entrada. Para isso, vamos usar dois gatilhos para alternar entre algoritmos. Se o vetor estiver quase ordenado ou for pequeno, optaremos pelo InsertionSort; caso contrário, o QuickSort com mediana de 3 será usado.

O desafio é determinar empiricamente, mas também de forma otimizada, os limiares ótimos para os dois gatilhos. Ou seja, quando que o vetor passa a ser pequeno o suficiente ou ordenado o suficiente para que o custo do InsertionSort seja menor. As duas variáveis que definirão isso serão chamadas "limiarQuebras" e "minTamParticao", vetores com menos elementos desordenados que limiarQuebras e vetores com tamanho menor que minTamParticao serão ordenados pelo InsertionSort.

Ademais, a função de custo para a análise empírica segue $f(cmp, move, calls) = a * cmp + b * move + c * calls$. Onde a, b e c são constantes passadas como parâmetros pela entrada e cmp, move e calls representam o número de comparações, movimentações e chamadas de um algoritmo, respectivamente.

Essa documentação será dividida em:

- Detalhes referentes a implementação.
- Instruções para execução do programa.
- Análise de Complexidade das funções implementadas.
- Estratégias de Robustez utilizadas.
- Análise Experimental do código desenvolvido.

2 Implementação

Para a implementação, optou-se pelo uso da linguagem C. A não necessidade de objetos e suas limitações, a simplicidade no código e uma distância menor do hardware (permitindo melhor controle do desempenho) foram fatores que colaboraram com essa escolha.

Além disso, vale notar que o projeto foi feito usando o VS Code como IDE. Seu compilador é o GNU Compiler Collection (GCC) versão gcc 6.3.0. O código foi testado em um ambiente com:

- **Processador -** Intel(R) Core(TM) i3-10100F CPU @ 3.60GHz.
- **Memória -** 16.0 GB.
- **Sistema Operacional -** Windows 10 22H2 e Ubuntu 24.04.1

2.1 Fluxo de Execução

O programa inicia selecionando a entrada; se foi passado algum parâmetro de arquivo na execução, este será a entrada; caso contrário, usará o terminal.

A entrada passa inicialmente 6 parâmetros: Seed (que será usada no cálculo do limiarQuebras), limiar de custo, a, b, c (apresentados na função citada acima) e o tamanho do vetor. Após isso, serão lidas as próximas entradas correspondentes ao tamanho do vetor que ocuparão cada posição. Com esses valores, chama-se a função que calcula o tamanho mais otimizado de partição. Com esse valor calculado, chama-se a função que calcula o número máximo de quebras mais otimizado. Ambas as funções serão aprofundadas a seguir.

Após a chamada de ambas as funções, o ordenador universal poderia ser chamado com o menor custo possível dentro do limiar passado como parâmetro, como desejado.

2.2 Estruturas de Dados

Para a execução do projeto, foram pensados alguns TADs que facilitassem a implementação. Foram usados vetores do tipo double que armazenam os custos de cada teste realizado. Um vetor de cópia do vetor a ser ordenado, que serve para fazer os testes sem perder o vetor original. Além disso, foram criadas as seguintes Structs:

2.2.1 ArrayParameters

Essa estrutura recebe os valores usados para o cálculo de custo e de otimização. As constantes a, b e c anteriormente citadas, bem como o limiar de custo esperado e a seed. O penúltimo é usado também na otimização e será explicado à frente. Já o último valor é usado para o cálculo do limiarQuebras.

2.2.2 Custo

Essa estrutura é passada em cada teste de ordenação para armazenar a quantidade de movimentações, chamadas e comparações que estão sendo feitas. Tem funções simples de mudança de seus valores e também de cálculo do custo (em conjunto com ArrayParameters)

2.3 Funções

As principais e mais complexas funções utilizadas no projeto são as que envolvem o cálculo otimizado para o minTamParticao e limiarQuebras.

2.3.1 Partição

Para o cálculo de tamanho de partição, nosso objetivo é testar diversos valores dentre uma faixa e analisar seus custos. Os valores a serem testados são espaçados pelo cálculo de um passo, de forma que: $passo = (max\_tamanho\_faixa - min\_tamanho\_faixa)/5$. Como exemplo, o valor inicial mínimo é 2 e o máximo é o tamanho do vetor. O passo acaba gerando diversos testes, após o fim destes, detectamos o teste que apresentou menor custo (seguindo a função citada acima) e calculamos uma nova faixa correspondente a esse valor.

A faixa é sempre feita de forma que tenha dois passos de distância. Por padrão, é um passo a mais e um a menos que o valor que apresentou menor custo. Apesar disso, se este for um extremo da faixa, a nova é realocada para caber em suas extremidades.

Após o cálculo da nova faixa, a diferença de custo de seus extremos é usada para verificar se o custo já convergiu. Se for uma diferença abaixo do parâmetro limiarCusto passado na entrada, o menor valor já retorna como tamanho de partição otimizado. O mesmo vale caso menos de 5 testes tenham sido feitos na faixa anterior.

Caso não haja convergência, o laço repete com a nova faixa até que se encontre um valor convergente.

2.3.2 Quebras

Para o cálculo do número de quebras, nosso objetivo é entender como os ordenadores vão divergir quanto mais ou menos desordenado estiver o vetor. Para isso testaremos vários níveis de desordenação. Esses níveis, ou números de quebra, vão seguir uma faixa semelhante a de tamanho da partição, porém iniciando com 1 e $tamanho/2$ como mínimo e máximo. Os testes serão feitos ordenando o vetor e desordenando $t - vezes$ para o teste de valor t. Essa desordenação é feita de forma pseudoaleatória selecionando posições no vetor ordenado para trocar com a função drand48() da biblioteca padrão da linguagem C.

Após o embaralhamento, cálculamos o custo que o QuickSort e o InsertionSort tomariam para ordenar o vetor. Ao fim dos testes, selecionamos como o teste ótimo aquele cuja diferença entre os dois algoritmos foi menor. Em seguida calculamos uma nova faixa de forma semelhante ao cálculo do tamanho da partição.

Quando finalizada a nova faixa, estimamos a diferença entre o custo do InsertionSort relativo ao limite superior e inferior da faixa. Se essa diferença for menor do que o parâmetro limiarCusto passado na entrada, ou se foram feitos menos do que 5 testes na faixa, o teste ótimo retorna como número máximo de quebras otimizado.

Caso ainda não tenha convergido, o algoritmo de testes repete para a nova faixa até que seja estabelecida uma convergência.

2.3.3 Dicionário de Funções

- **defineBreakLimit -** Gerencia os testes realizados relacionados ao número de quebras, as faixas e seus custos. Também define quando encerrar a otimização.
- **definePartitionSize -** Gerencia os testes realizados relacionados ao tamanho de partição, as faixas e seus custos. Também define quando encerrar a otimização.
- **calculaNovaFaixa -** Calcula os novos limites inferiores e superiores da nova faixa de testes.
- **getValueByStep -** Retorna o valor de teste relacionado com seu número de teste.
- **menorCusto -** Retorna a posição com menor custo de um vetor de custos
- **menorCustoDiff -** Retorna a posição de menor diferença entre dois vetores de custos.
- **absolute -** Valor absoluto de um double.
- **swap -** Troca o valor de duas variáveis incrementando o custo.
- **quickSort3Ins -** Algoritmo recursivo de QuickSort com mediana de 3 que chama o InsertionSort se tiver menos que um tamanho passado como parâmetro.
- **partition3 -** Algoritmo de particionamento de um vetor com base em um pivô calculado com a mediana dos valores extremos e central.
- **insertionSort -** Algoritmo de InsertionSort para ordenação do vetor.
- **median -** Retorna a mediana de 3 números.

- **universalSort -** Ordenador que escolhe entre QuickSort e InsertionSort com base no tamanho do vetor e número de quebras.
- **countBreak -** Conta a quantidade de quebras em um vetor.
- **OrdenadorUniversalPartitionOptimizer -** Gerencia o vetor a ser testado para um tamanho de partição e seu custo.
- **OrdenadorUniversalBreakOptimizer -** Gerencia o vetor a ser testado para um número de quebras (embaralhamento) e seus custos.
- **arrayShuffler -** Embaralha o vetor.
- **arrayCopy -** Copia o vetor em outro.

3 Instruções

O projeto e seus arquivos foram organizados em pastas de forma que o arquivo Makefile consiga executar todas as compilações e builds necessárias para a execução.

Caso esteja usando um ambiente Linux ou semelhante, use o comando *make all* na pasta raiz do projeto. Em seguida, execute *./bin/tp1.out arquivo_de_entrada.txt* para rodar o programa. Substitua arquivo_de_entrada pelo nome do arquivo que deseja usar, ou não passe nenhum outro parâmetro e escreva a entrada no terminal.

Caso esteja usando um ambiente Windows, é necessário modificar o código fonte do programa, já que uma das funções utilizadas não está disponível nativamente no sistema. No arquivo **universalsort.c**, dentro da função OrdenadorUniversalBreakOptimizer, troque as linhas `//srand(seed);` e `srand48(seed);` por `srand(seed);` e `//srand48(seed);` respectivamente (essas linhas aparecem mais de uma vez na função). De forma parecida, siga as instruções presentes na função ArrayShuffler sobre comentar e descomentar linhas. Para a compilação e build, use o comando *make all_windows* na pasta raiz do projeto. Em seguida, execute *./bin/tp1.exe arquivo_de_entrada.txt* para rodar o programa. Substitua arquivo_de_entrada pelo nome do arquivo que deseja usar, ou não passe nenhum outro parâmetro e escreva a entrada no terminal.

São esperadas saídas diferentes para o programa caso o código fonte seja alterado. Isso se deve a natureza diferente das funções de geração de números pseudoaleatórios em cada caso.

4 Análise de Complexidade

Para fins de sintetização, serão citadas nessa seção somente funções cuja complexidade de tempo ou espaço não sejam constantes ($O(1)$). Dessa forma o valor mínimo que teremos é $O(log(n))$ e caso a função não seja citada abaixo, assuma que seu custo é $O(1)$.

4.1 Tempo

- **defineBreakLimit -** Usando a lógica apresentada acima, serão calculadas $O(log(n))$ faixas de valores a serem testador. Cada faixa tem uma quantidade fixa de testes $O(1)$ e cada teste tem custo $O(n^2)$ em seu pior caso (que provavelmente vai ser calculado já que são vários testes feitos). Dessa forma a complexidade de tempo da função é $O(log(n)) * O(1) * O(n^2) = O(n^2 log(n))$.
- **definePartitionSize -** Com o mesmo princípio lógico que a função defineBreakLimit: São $O(log(n))$ faixas com $O(1)$ testes que custam $O(n^2)$ cada. Totalizando $O(n^2 log(n))$.
- **menorCusto -** Caminha pelo vetor a procura do menor valor $O(n)$.
- **menorCustoDiff -** Caminha pelos dois vetores procurando a menor diferença entre eles $O(n)$.
- **quickSort3Ins -** Usando a mediana de 3 valores, o algoritmo de QuickSort tem complexidade temporal $O(nlog(n))$.
- **partition3 -** Em complexidade temporal, o pior caso percorre o vetor todo $O(n)$.
- **insertionSort -** Tem como pior caso complexidade de tempo $O(n^2)$.
- **universalSort -** Chama countBreak e quickSort3Ins ou insertionSort, sendo que desses o pior caso é do insertionSort com custo $O(n^2)$.
- **countBreak -** Passa pelo vetor inteiro uma vez para contar as quebras, $O(n)$.
- **OrdenadorUniversalPartitionOptimizer -** Custo $O(n^2)$ no pior caso, isso se deve ao fato de ter de lidar com casos como um vetor completamente desordenado ser ordenado por um InsertionSort.Também chama um arrayCopy, mas por ser $O(n)$ também é $O(n^2)$.
- **OrdenadorUniversalBreakOptimizer -** Chama arrayCopy, $O(n)$; quickSort3Ins duas vezes, $O(nlog(n))$; arrayShuffler duas vzes, $O(b)$ e insertionSort, $O(n^2)$. Totalizando $O(n + nlog(n) + b + n^2)$, como b é sempre menor do que n (o número de quebras não ultrapassa o tamanho do vetor), tem custo de tempo final $O(n^2)$.
- **arrayShuffler -** Com o parâmetro de quebras iqual a b, a função tem custo $O(b)$.
- **arrayCopy -** Custo de tempo $O(n)$ pois passa por cada posição do vetor uma vez para copiá-la.

4.2 Espaço

- **defineBreakLimit -** Cria dois vetores de custo de tamanho constante, mas chama OrdenadorUniversalBreakOptimizer que tem custo de espaço $O(n)$.
- **definePartitionSize -** Cria um vetor de custo de tamanho constante e chama OrdenadorUniversalPartitionOptimizer, que tem custo de espaço $O(n)$.
- **quickSort3Ins -** Usando a mediana de 3 valores, o algoritmo de QuickSort tem complexidade de espaço $O(n)$.
- **universalSort -** Tem como pior caso chamar quickSort3Ins em seu pior caso, totalizando complexidade espacial $O(n)$.
- **countBreak -** Passa pelo vetor inteiro uma vez para contar as quebras, $O(n)$.
- **OrdenadorUniversalPartitionOptimizer -** Custo $O(n)$ no pior caso, isso se deve ao fato de ter que chamar quickSort3Ins que tem pior caso $O(n)$.
- **OrdenadorUniversalBreakOptimizer -** Cria um vetor de tamanho n, $O(n)$ e chama quickSort3Ins duas vezes. Totalizando $O(n + 2n) = O(n)$.

4.3 Total

Dessa forma, a complexidade de tempo e espaço total de execução do programa tem a seguinte análise. Considerando funções em série e em paralelo, a main vai chamar primeiramente definePartitionSize e depois defineBreakLimit. Além disso, também vai criar e preencher um vetor de tamanho n e algumas variáveis fixas. Sendo assim, sua complexidade é de tempo $O(n^2log(n)+n^2log(n)+n) = O(n^2log(n))$ e de espaço $O(n+n+n+1) = O(n)$.

5 Estratégia de Robustez

Os pontos em que mais podem ocorrer erros no código são durante a leitura do arquivo de entrada bem como na alocação e liberação de memória dinâmica utilizada. Tendo isso em vista foram implementados testes que lidam com essas questões.

- **Arquivo menor do que o esperado -** Caso o arquivo chegue no fim e ainda há itens para ler, retorna um erro de entrada e encerra o programa.
- **Faixa pequena demais -** Caso a faixa resulte em um passo igual a 0, incrementa o passo deixando seu mínimo 1 para evitar loops sem fim durante a execução.
- **Erro na alocação de memória -** Caso a alocação de memória retorne um ponteiro nulo, libera todas as memórias alocadas até o momento e retorna um erro, encerrando o programa.

Além desses pontos, toda a memória alocada em algum momento é devidamente liberada quando não é mais utilizada.

6 Análise Experimental

A análise experimental realizada tem como base testar como o TAD se comporta em diferentes cenários. Esses cenários são variações de tamanho do vetor, tamanho da chave a ser comparada, tamanho do registro a ser movimentado e configuração do vetor. As entradas de teste foram geradas automaticamente por um gerador de vetores criado na linguagem Python. Todos os dados são armazenados em três arquivos CSV e seus valores possíveis para cada variação são:

- **Tamanho do Vetor -** Varia de 10 a 7000 em valores fixos.
- **Tamanho da Chave -** Varia de 4 a 64 caracteres em potências de 2.
- **Tamanho do Registro -** Varia de 4 a 64 caracteres em potências de 2.
- **Configuração do Vetor -** Ordenado, inversamente ordenado e aleatório

Para os testes, também é feita uma verificação de forma a garantir que cada teste é único e não se repetirá. Após isso, foi usado um outro código em Python que executasse todas as entradas e armazenasse seus dados.

6.1 Regressão de Coeficientes

Feitos os testes, o primeiro objetivo era calcular uma regressão que estimasse os valores dos coeficientes de comparações, movimentações e chamadas de funções em relação ao tempo. Esses coeficientes são representados por a, b e c no gráfico abaixo (Figura 1) e têm valores 0.000568 0.011345 e 0.000114 respectivamente.

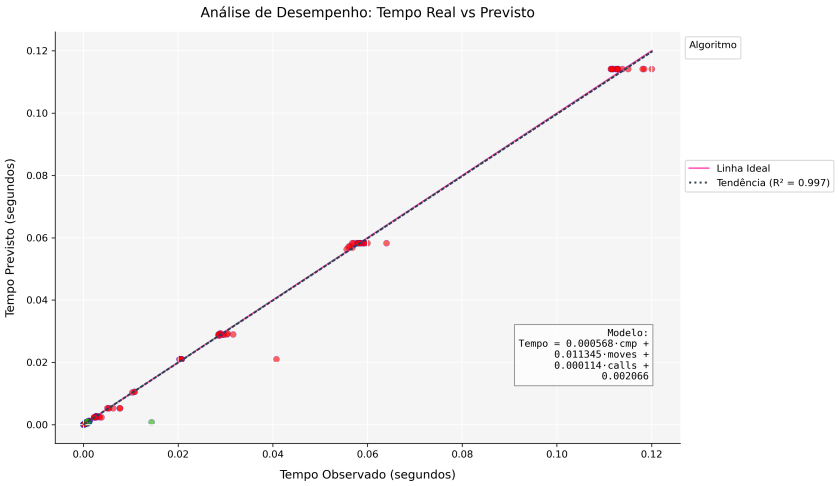

Figura 1: Gráfico de previsão de análise de desempenho.

6.2 Comparação entre algoritmos e custos de etapas

Com os coeficientes bem estimados (vide $R^2 = 0.997$), podemos recalcular os testes com esses parâmetros para o custo. Após a segunda leva de testes, vamos avaliar e comparar o custo temporal de nosso TAD com o custo de outros algoritmos de ordenação (InsertionSort e QuickSort). Separaremos os testes pelas configurações de vetor previamente comentadas, o resultado pode ser conferido na Figura 2.

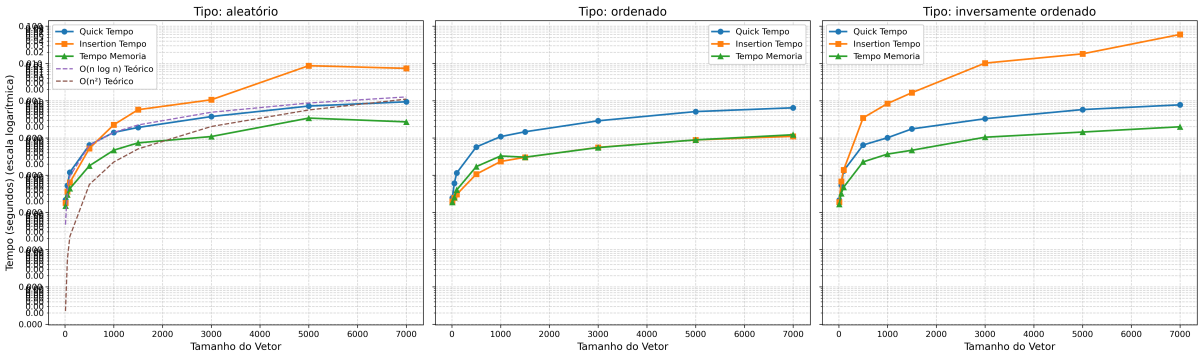

Figura 2: Gráficos comparativos de algoritmo por configuração de vetor.

Como pode ser notado no gráfico, o caso geral (aleatório) e o caso inversamente ordenado apontam o êxito do TAD em otimizar ainda mais a ordenação do vetor. No caso ordenado, ele se assemelha muito ao InsertionSort, isso acontece porque este é chamado quase instantaneamente pelo ordenador universal.

Vamos também analisar o custo por partes do otimizador de nosso ordenador universal (Figura 3).

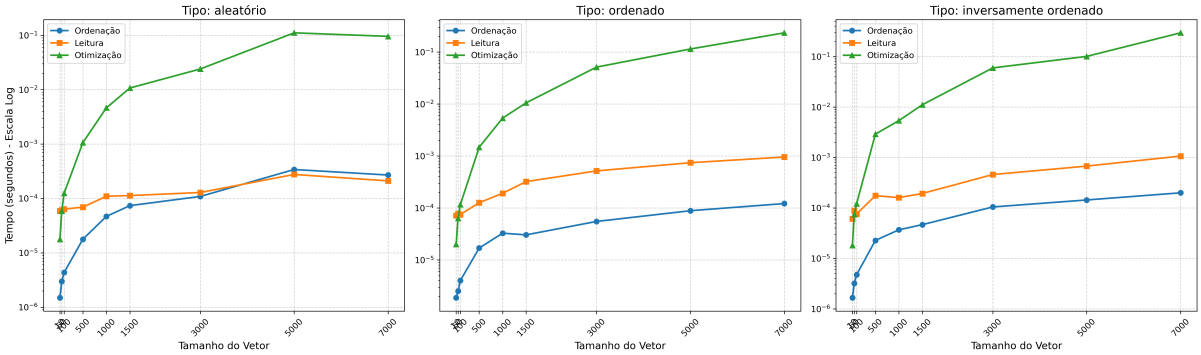

Figura 3: Gráfico de custo temporal de cada etapa do programa

Com base no gráfico, é possível notar o quão caro (em tempo computacional) é o processo de otimização do TAD, evidenciando sua complexidade $O(n^2 log(n))$ apontada anteriormente.

6.3 Correlações entre quebra e partição

A próxima etapa da análise experimental é calcular o impacto das dimensões do vetor e de suas configurações no número ligado ao limiarQuebras e minTamParticao. Para isso, foram feitos os gráficos presentes nas Figuras 4 e 5. É possível reparar a média do Limiar de Quebras e Partição para cada tamanho de vetor e configuração testada. Nesses casos, ambos os parâmetros parecem crescentes em relação ao tamanho do vetor. Quanto à configuração, o limiar de partição tende a aumentar quanto mais organizado estiver o vetor, enquanto o limiar de quebras diminui na mesma direção.

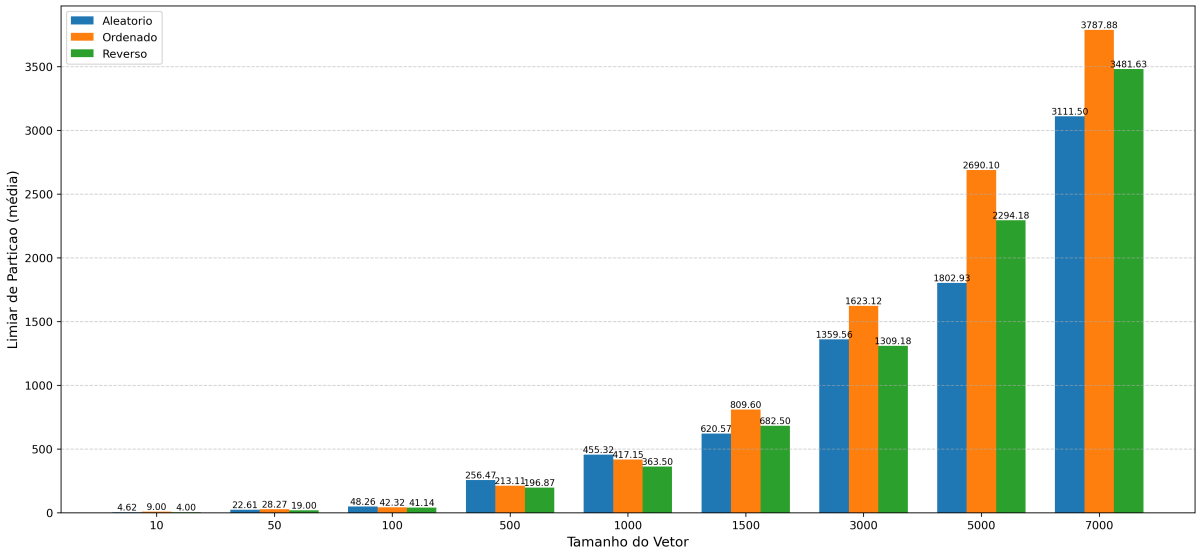

Figura 4: Média do Limiar de Partição para cada tamanho e configuração

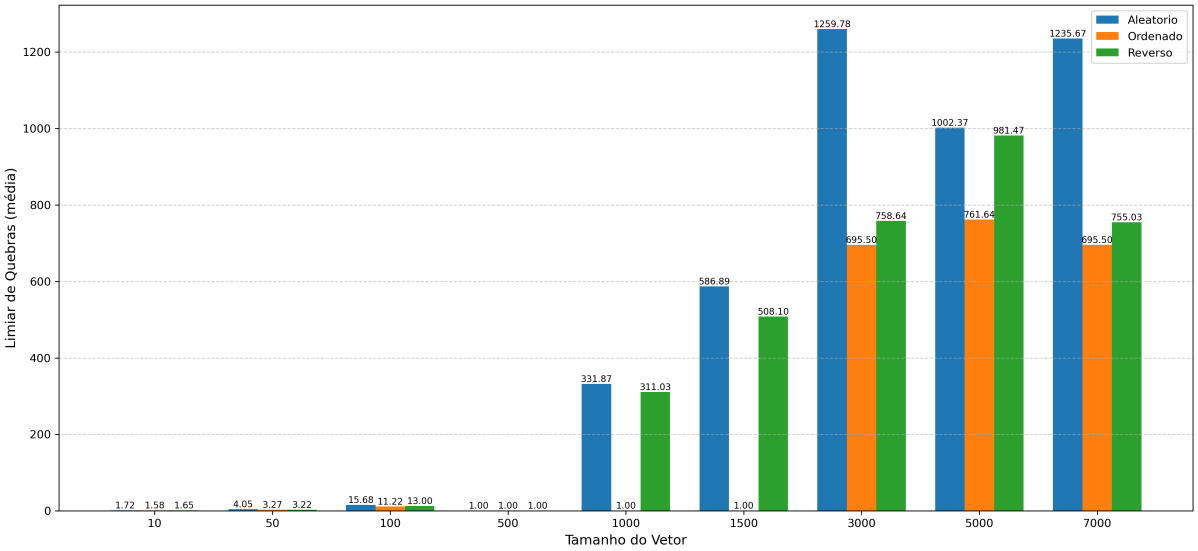

Figura 5: Média do Limiar de Quebras para cada tamanho e configuração

6.4 Impactos do tamanho de Chave e Registro

Por fim, temos de analisar como o tamanho da Chave e do Registro impacta no tempo de execução do programa. É esperado que, quanto maior o registro e a chave, maior será o tempo, já que impactaria bastante nas movimentações e comparações, respectivamente. Isso pode ser conferido no mapa de calor da Figura 6. Uma observação crucial é reparar como as chaves e registros são vetores de tamanho fixo entre 4 e 64. Isso impõe uma complexidade a mais na execução, onde se c é o tamanho da chave e r é o tamanho do registro, teremos complexidade $O(max(c, r))$.

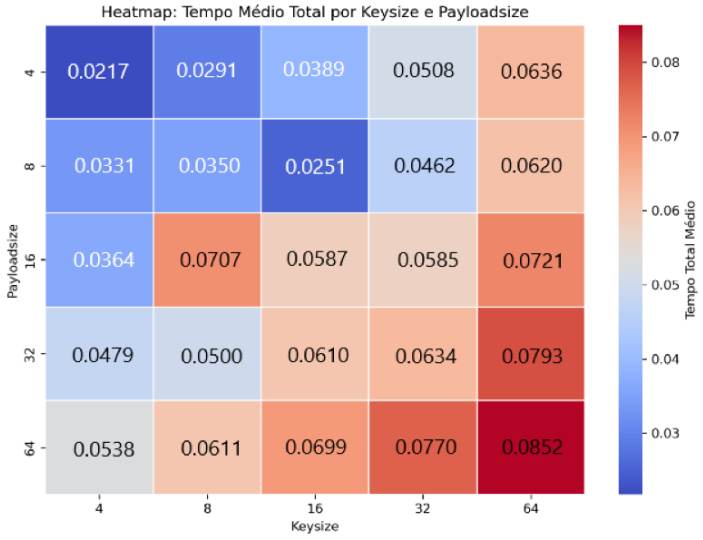

Figura 6: Média do Limiar de Quebras para cada tamanho e configuração

7 Conclusão

Destarte, o problema propunha desenvolver um Ordenador Universal que soubesse quando alternar de algoritmo para obter melhor desempenho. Para resolvê-lo, era necessário calcular empiricamente os limites de número de quebras e de tamanho de partição que serviriam de gatilho para trocar de método. Implementar uma solução que ficasse de acordo com o esperado foi a maior parte do desafio, pois pedia do implementador um grande domínio sobre o assunto de estrutura de dados, otimização e testes de algoritmos e algoritmos de ordenação. Grande parte desse conhecimento foi lecionado em sala de aula na disciplina de Estrutura de Dados e disciplinas

passadas, o que frisa a importância de conhecer e dominar essas técnicas, bem como tomar decisões de projeto relevantes.

Vale comentar que esse trabalho e suas análises posteriores instigam o aluno a pesquisar e aprender mais sobre otimizações e previsões de estados, tópico muito ligado à ciência de dados, área que ganhou destaque e ficou em alta nos últimos tempos.

Referências

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática3a. Edição, Elsevier, 2012

Especificação do Trabalho Prático 1 - Ordenador Universal DCC/ICEx/UFMG.

Anisio Lacerda, Wagner Meira Jr., Washington Cunha, Lucas N Ferreira. Estruturas de Dados, Slides da disciplina DCC205 - Estruturas de Dados, Departamento de Ciência da Computação, ICEx/UFMG.