# Relatório do Trabalho I
# Algoritmos de Ordenação Externa

Grupo: David Lima, Israel Pedreira, Márcio dos Santos
Professor: Dr. George Lima
Curso: Estrutura de Dados e Algoritmos II

**Link do repositório com o código fonte (Github)**

**Ambiente de execução online do projeto (Replit)**

## Algoritmos Avaliados

### Intercalação Balanceada de $p$-Caminhos

O algoritmo de ordenação externa por intercalação balanceada de $p$-Caminhos (ou $p$-Ways) utiliza o conceito de intercalação, combinando $p$ sequências previamente ordenadas em uma sequência maior a cada etapa. Esse processo é repetido até que reste uma única sequência ordenada contendo todos os registros. Inicialmente, as sequências nos arquivos de entrada são combinadas usando intercalação, resultando em uma sequência $p$ vezes maior que a original. Essas sequências são distribuídas uniformemente em $p$ arquivos de saída. Completado o processamento dos valores nos arquivos originais, uma fase do algoritmo é concluída. Na próxima passagem, os arquivos de saída da etapa anterior tornam-se os novos arquivos de entrada, e os arquivos de entrada originais passam a ser os arquivos de saída. A nova etapa intercala os arquivos gerados na primeira fase, formando arquivos $p$ vezes maiores. Esse processo continua até que reste apenas uma sequencia final, que é o arquivo ordenado completo.

As sequências iniciais ordenadas para o algoritmo podem ser obtidas por métodos de ordenação tradicionais, como *Quick Sort*, ou estratégias como *Round Robin*. No nosso caso, foram geradas usando o método de seleção natural com *Heap Sort*.

Como se trata de um método de ordenação externa, a principal particularidade é a determinação do maior valor de $p$ para a ordenação, calculado com base na quantidade de arquivos que podem ser mantidos abertos simultaneamente ($k$). Esse valor é dado por $p = \lceil \frac{k}{2} \rceil$. As sequências geradas devem ser distribuídas de forma balanceada entre os $p$ arquivos para garantir um comportamento ótimo do algoritmo e evitar sobrecarga em determinados arquivos.

Considerando métodos que geram sequências iniciais com tamanho máximo igual à capacidade da memória principal ($m$) e $n$ registros, o algoritmo requer

$$\lceil \log_p \left( \frac{n}{m} \right) \rceil$$

iterações em cada registro para reduzir as $\frac{n}{m}$ sequências iniciais a uma única sequência ordenada [5]. O número de passos necessários é dado pelo logaritmo na base $p$ de $\frac{n}{m}$, já que a cada etapa de intercalação o número de sequências é reduzido por um fator de $p$. No entanto, como usamos o método de seleção natural, não podemos garantir a quantidade exata de sequências geradas pelo *heap* apenas com base em $m$ e $n$, pois isso depende da distribuição dos registros. Portanto, se o *heap* gerar uma quantidade $r$ de sequências, o algoritmo precisará realizar

$$\lceil \log_p (r) \rceil$$

iterações sobre cada registro.

### Ordenação por Heap

O processo de obtenção das sequências iniciais ordenadas com um heap mínimo começa com a leitura e armazenamento de até $m$ registros em uma fila de prioridade, que é estruturada como um heap. Essa estrutura assegura que o menor valor esteja sempre disponível para remoção, sendo este o valor na raiz da árvore. Quando o menor valor (raiz) é removido do heap, ele é inserido em uma sequência ordenada. Em seguida, o próximo registro disponível é adicionado à estrutura.

Se o valor inserido for menor que o último valor removido, ele é "marcado". Essa marcação garante que a fila de prioridade mantenha a ordem correta, mesmo quando o valor inserido é menor que o último removido. No heap, valores "marcados"são tratados como se fossem maiores do que os registros não marcados, independentemente de seu valor real. Quando todos os valores na estrutura estão "marcados", o heap "quebra", removendo todas as marcações e iniciando uma nova sequência ordenada. Esse processo continua até que todos os registros tenham sido distribuídos em uma das sequências iniciais, que serão então passadas para os algoritmos de ordenação externa.

As operações de inserção e remoção em uma fila de prioridade têm complexidade $O(\log m)$, onde $m$ representa o número máximo de registros que podem estar simultaneamente na estrutura. No contexto dos experimentos, esse valor é determinado pelo tamanho da memória principal disponível.

## Ordenação por Intercalação Polifásica

O algoritmo de ordenação polifásica é um método de ordenação externa, projetado para ser eficiente em cenários onde o número de arquivos disponíveis para intercalação é limitado. O processo de ordenação polifásica começa com a geração das runs. Uma vez que os runs são gerados, eles precisam ser distribuídos entre os arquivos disponíveis para a intercalação. Supondo que o sistema tenha $k$ arquivos disponíveis, os runs são distribuídos entre $k-1$ arquivos, enquanto o arquivo restante é reservado como o arquivo de saída. A distribuição dos runs entre os arquivos deve ser feita de maneira equilibrada, para que cada arquivo contenha aproximadamente o mesmo número de runs, embora na prática isso nem sempre seja possível devido à natureza dos dados. A fase de intercalação polifásica começa a partir da comparação dos primeiros elementos dos $k-1$ arquivos de entrada.

O menor elemento é selecionado e escrito no arquivo de saída (vazio), e o próximo elemento do mesmo run é lido para continuar a comparação. Isso ocorre até que um dos arquivos fique vazio, então ele será designado como arquivo de saída. Esse processo se repete até que todos os runs sejam combinados em um único run maior. O algoritmo utiliza todos os arquivos de forma cíclica, redistribuindo os dados em cada fase, o que reduz o número de fases necessárias. Uma característica importante do método de intercalação polifásica é o uso de "sequências falsas"quando o número de runs iniciais não é um múltiplo de $k-1$. Essas sequências falsas são essencialmente sequências vazias, que são utilizadas para preencher os arquivos de entrada e manter o processo de intercalação consistente. O processo de intercalação continua por várias fases, cada uma reduzindo o número de runs ao combinar múltiplos runs menores em runs maiores. [4]

O algoritmo termina quando todos os runs foram combinados em um único run final, que estará completamente ordenado no arquivo de saída. Esse run final é o resultado da ordenação dos dados.

## Ordenação em Cascata

O método de ordenação em cascata foi proposto por Betz et al. em 1959 [1], precedendo o método de ordenação baseado em intercalação polifásica. O algoritmo se baseia na intercalação de $k-p$ arquivos em apenas um arquivo de saída, sendo $k$ a quantidade de máxima de arquivos abertos e $p = 1, 2, 3, \ldots, k-1$. A intercalação do algoritmo presume que todas as sequências estão ordenadas e que existe pelo menos um arquivo vazio para ser usado como output. Esta é aplicada até que outro arquivo se torne vazio, assim, o algoritmo o considera como o novo arquivo de saída e prossegue para outra intercalação, que não inclui o arquivo de saída passado. Note que a cada intercalação desconsideramos um arquivo, isso indica que nosso $p$, indicado na fórmula anterior está crescendo. Quando $p = k-1$, isto é, só resta um arquivo para intercalar, passamos para a próxima fase do algoritmo, ou seja, recomeçamos a intercalação considerando $p = 1$. Em algumas variações do algoritmo, o conteúdo do último arquivo é copiado para o arquivo de output, sem alterações significantes no tempo de execução [3]. Este processo se repete até que reste apenas uma sequência de $n$ registros completamente ordenada em um dos arquivos.

Note que precisamos garantir que ao final de cada intercalação teremos apenas um arquivo vazio durante todo o processo. Para que isso valha, antes da execução do algoritmo realizamos uma simulação de seu comportamento sobre a quantidade de sequências que teremos em cada arquivo, em cada fase. A simulação começa com o final esperado; apenas uma sequência de tamanho $n$ em um dos arquivos, e enquanto a quantidade de sequências seja menor que o número de registros, traçamos o comportamento inverso do algoritmo. Ao final deste algoritmo, teremos $k$ valores, um para cada arquivo, que nos dirá a quantidade de sequências ordenadas que devemos ter em cada arquivo para que o algoritmo funcione perfeitamente. Observe que se $n$ é diferente desta quantidade, não poderemos preencher todos os arquivos com a quantidade esperada de sequências. Neste caso, criamos *dummy runs*, que seriam sequências falsas, criadas para o algoritmo operar de forma esperada. A intercalação de uma dummy run com uma sequência que contém registros retorna uma sequência com todos os registros com um registro “dummy”. Observe também, que para $n$ registros não ordenados, teremos $n$ sequências iniciais, logo, $r = n$.

# Método de Avaliação

## Algoritmos

Os algoritmos foram escritos na linguagem Python sem dependências externas. A única biblioteca utilizada durante o desenvolvimento foi a `random`, padrão da própria linguagem, para a geração de números pseudo-aleatórios. Ademais, todos os testes foram centralizados na classe `Evaluator`, para que a execução de cada algoritmo seja padronizada ao máximo, assegurando uma comparação justa. Os testes foram realizados de forma que o seguinte seja garantido para todo algoritmo: (i) Utilização de sequências com registros aleatórios, (ii) avaliação com base em sua taxa de processamento ($\alpha$), (iii) utilização do mesmo intervalo de $r$ e (iv) as sequências geradas já são ordenadas e são passadas diretamente para o algoritmo, ignorando a etapa da ordenação natural (Ordenação por Heap). Os valores de $r \in R$ foram calculados baseando-se no conjunto $R$ apresentado no roteiro do trabalho:

$$R = \{i \times j | i \times j \leq 5000; i = 1, 2, \ldots, 10; j = 10, 20, \ldots, 1000\}$$

Considerando que $R$ é um conjunto, desconsideramos os valores repetidos de $i \times j$ (ex. $1 \times 20$ e $2 \times 10$) e avaliamos os algoritmos seguindo sua ordem crescente. Este mesmo teste foi realizado cinco vezes para cada algoritmo, um para cada valor de $k$, onde $k \in \{4, 6, 8, 10, 12\}$. Ademais, o teste para cada $k$ foi repetido 10 vezes, sendo cada valor de $\alpha$ uma média destes.

## Heap

Os experimentos relacionados ao Heap foram realizados através da mesma classe, `Evaluator`. Nestes experimentos, a estrutura foi avaliada com base no $\beta$ calculado utilizando seu output. Este é o caso pois, no roteiro, é requisitada que o $\beta$ calculado para avaliação do Heap tenha seu segundo parâmetro, $j$, zerado. O $j$ indica a fase do algoritmo, zerá-lo indica que as sequências foram apenas escritas na memória secundária; O algoritmo de ordenação ainda não moveu nenhum registro. Sendo assim, o $\beta$ calculado depende apenas do resultado do Heap.

Além disso, os experimentos constaram com a variação do tamanho da memória principal ($m$) dada pelo intervalo indicado no roteiro do trabalho:

$$m \in \{3, 15, 30, 45, 60\}$$

Onde, para cada valor de $m$ o algoritmo era responsável de achar sequências ordenadas em 10.000 registros aleatórios. Estes registros eram gerados no começo da avaliação e se mantinham fixos para cada valor de $m$.

Abaixo temos um trecho do código-fonte (Listing 1) referente a avaliação do Heap, da classe `Evaluator`.

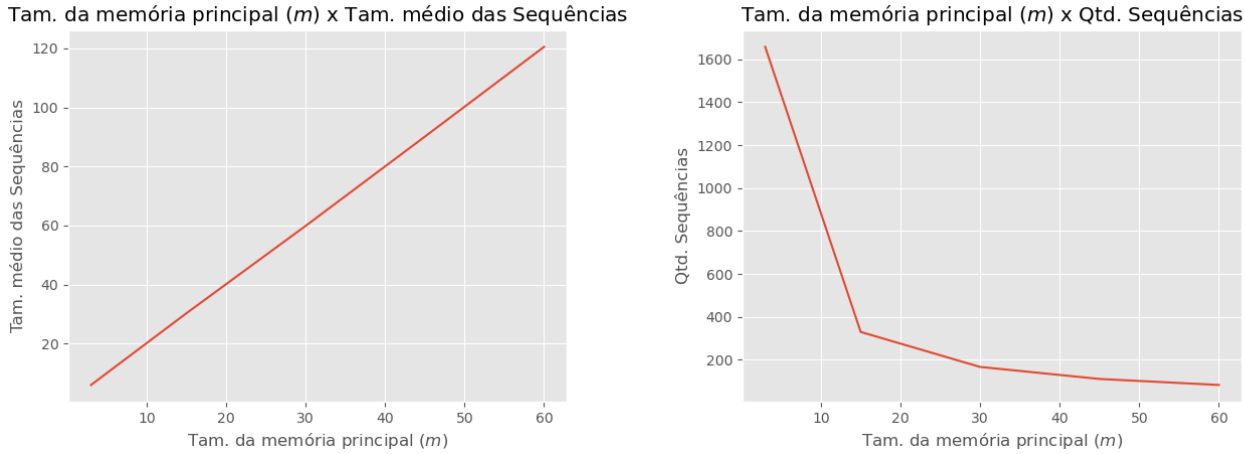
```
betas = []
if fixed_seq:
    regs = Evaluator._generate_random_sequence(N_OF_REGS)

for m in tqdm(m_values):
    if not fixed_seq:
        regs = Evaluator._generate_random_sequence(N_OF_REGS)
    sorted_seqs = Heap(
        main_memory_size=m,
        registers=regs
    ).sort()

    betas.append(
        beta(m, len(sorted_seqs), N_OF_REGS, depth=0)
    )
```

Listing 1: Trecho do código fonte responsável pela avaliação do Heap. Note que durante os experimento foi utilizado `fixed_seq=True`, `N_OF_REGS=10000` e `m_values=[3,15,30,45,60]`.

# Resultados Alcançados

## Heap

No gráfico 1, ($ii$) ilustra a relação inversa entre o tamanho da memória principal ($m$) e o número de sequências geradas durante o processo de criação do *Heap*. Em contraste, ($i$) destaca uma relação linear entre o tamanho médio das sequências geradas e o valor de $m$. À medida que o tamanho da memória principal aumenta, o número de sequências diminui e o tamanho médio das sequências aumenta. Isso ocorre porque tamanhos maiores de $m$ permitem que o *Heap* processe blocos maiores de registros antes de "quebrar".




(i) (ii)

Figura 1: Gráficos referentes à métrica $\beta$. Em (i) temos a relação entre o tamanho médio das sequências geradas pelo Heap e o tamanho da memória principal ($m$) para 10.000 registros, e em (ii) temos a relação entre a quantidade de sequências geradas pelo *Heap* e o tamanho da memória principal ($m$) para 10.000 registros.

A redução no número de sequências beneficia os algoritmos de ordenação externa, pois a diminuição na quantidade de sequências reduz o número de blocos que precisam ser intercalados, aumentando a eficiência do algoritmo. No entanto, o gráfico mostra que, para valores cada vez maiores de $m$, a melhoria na redução do número de sequências torna-se progressivamente menor.

Isso indica que, embora um aumento no valor de $m$ possa melhorar a eficiência do processo, a redução no número de sequências não é tão significativa quando a memória atinge um certo ponto. Em outras palavras, a cada incremento adicional no tamanho da memória, a melhoria na eficiência se torna cada vez menos perceptível.

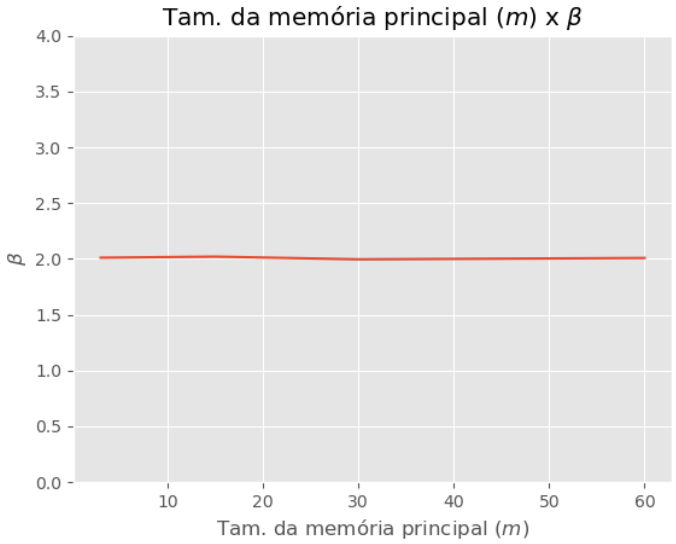



Figura 2: Relação entre o valor de $\beta$ e o tamanho da memória principal ($m$) para 10.000 registros

O gráfico 2 demonstra uma relação quase constante (com valor próximo a 2) entre os valores de $\beta$ para $\beta(m, 0)$ e o tamanho da memória principal $m$. O valor de $\beta$ é dado pela equação 1, onde $r_0$ representa o número de sequências iniciais geradas, e $|S^m_{i,0}|$ denota o tamanho da $i$-ésima sequência inicial.

$$\beta(m, 0) = \frac{1}{m\ r_0} \sum_{i=1}^{r_0} |S^m_{i,0}| \tag{1}$$

Este valor de $\beta$ pode ser interpretado como a média do tamanho das sequências geradas pelo *Heap* dividido por $m$. Em outras palavras, $\beta$ representa a relação entre o comprimento médio das sequências e o tamanho da memória

principal. Um valor constante de $\beta$ indica que, para quantidades de registros consideravelmente maiores (10000 nos testes realizados), à medida que o tamanho da memória principal aumenta, a média do tamanho das sequências geradas pelo *Heap* é proporcional a $m$. Isso sugere que o *Heap* está gerando sequências de tamanho médio que permanecem aproximadamente constantes em relação ao aumento da capacidade da memória, mantendo uma relação linear com $m$.

Em resumo, os gráficos mostram que, para grande quantidades de registros, com o aumento do tamanho da memória principal ($m$), o número de sequências geradas pelo *Heap* diminui, enquanto o tamanho médio das sequências aumenta. Isso indica que tamanhos maiores de $m$ permitem que o *Heap* processe blocos maiores de registros antes de "quebrar". Apesar disso, a melhoria na eficiência, refletida pela redução do número de sequências, torna-se menos significativa à medida que $m$ aumenta.

Além disso, o valor constante de $\beta$ sugere que o comprimento médio das sequências geradas pelo *Heap* é proporcional ao tamanho da memória, mantendo uma relação linear. Isso confirma que, embora um aumento em $m$ melhore a eficiência do processo, a redução no número de sequências e os ganhos adicionais tornam-se progressivamente menores.

## Intercalação Balanceada de $p$-Caminhos

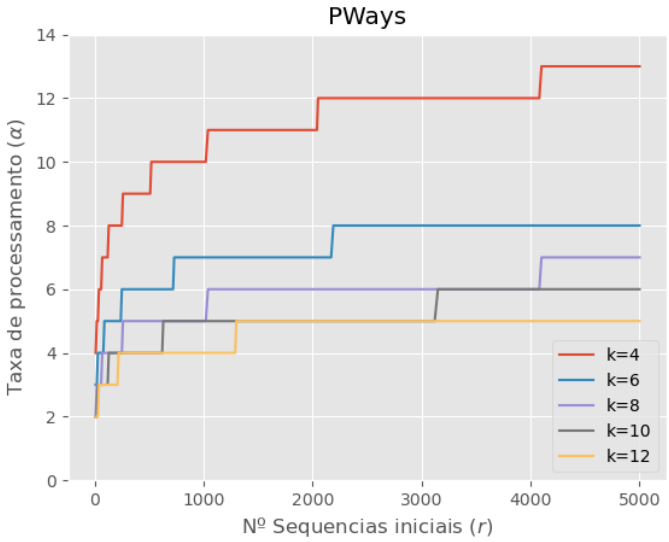



Figura 3: Relação entre a taxa de processamento ($\alpha$) e a quantidade de sequências iniciais ($r$) para o algoritmo de ordenação balanceada com P caminhos. Cada cor representa um valor diferente de $k$, segundo legenda.

Na figura 3, são apresentados os resultados obtidos para o algoritmo de intercalação balanceada de $p$-Caminhos. O gráfico revela um comportamento que se assemelha a uma função em degraus. Esse padrão é explicado pela aplicação da função teto em

$$\lceil \log_p (r) \rceil$$

A função teto arredonda para cima o valor do logaritmo, resultando em uma curva que exibe saltos discretos à medida que $r$ aumenta, refletindo as alterações no número de iterações necessárias conforme o número de sequências iniciais muda.
Considerando que $p = \lceil \frac{k}{2} \rceil$, é possível observar que, para $p = 2$, a taxa de processamento cresce mais rapidamente em comparação com os outros valores de $p$, chegando ao valor já esperado de

$$\lceil \log_2 (5000) \rceil = 13$$

Isso ocorre porque, com $p$ menor, o número de iterações necessárias aumenta rapidamente com o crescimento de $r$. Em contraste, conforme o valor de $p$ aumenta, o número de iterações por registro diminui, embora a redução se torne menos expressiva. A diferença entre bases menores (como $p = 2$ e $p = 3$) tem um impacto mais proporcionalmente mais significativo no crescimento da função em comparação com a diferença entre bases maiores (como $p = 3$ e $p = 4$). Portanto, embora valores maiores de $p$ melhorem a eficiência do algoritmo, a melhoria torna-se progressivamente menos pronunciada conforme $p$ aumenta.

## Ordenação por Intercalação Polifásica

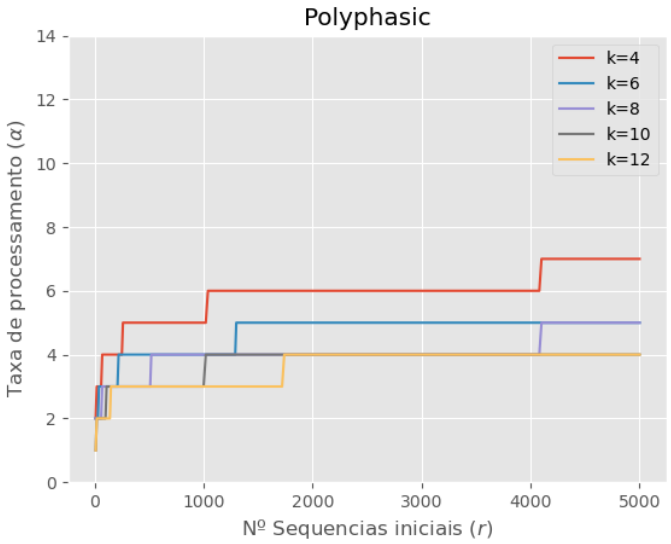



Figura 4: Relação entre a taxa de processamento ($\alpha$) e a quantidade de sequências iniciais ($r$) para o algoritmo de ordenação polifásico. Cada cor representa um valor diferente de $k$, segundo legenda.

Na figura 4 temos a relação entre a taxa de processamento e o número de sequencias inciais para cada valor de k. Podemos perceber que o algoritmo de intercalação polifásica trabalha de maneira eficiente para reduzir progressivamente o número de runs ao longo das fases, e a complexidade do processo de merge é detalhada fase por fase. Cada uma delas requer um número de comparações que diminui ao longo do tempo, mas que inicialmente é elevado, devido ao grande número de runs e ao tamanho dos mesmos. [2]

A análise do gráfico sugere que, embora o número total de comparações ao longo das fases seja elevado, o algoritmo é otimizado para garantir que o número de fases e, portanto, o tempo total de execução, seja minimizado. Isso é particularmente eficaz em cenários onde a I/O é cara e deve ser minimizada.

## Ordenação em Cascata

A figura 5 expõe o resultado de $\alpha$ obtidos após a execução de diferentes testes sobre a ordenação em cascata.

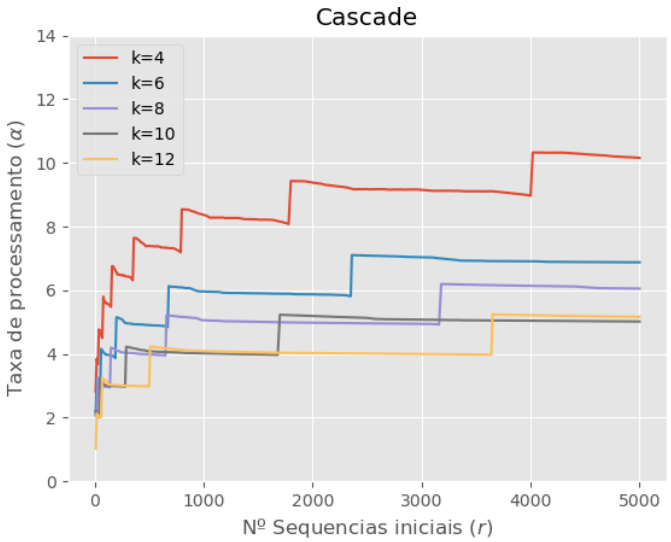



Figura 5: Relação entre a taxa de processamento ($\alpha$) e a quantidade de sequências iniciais ($r$) para o algoritmo de ordenação em cascata. Cada cor representa um valor diferente de $k$, segundo legenda.

Observe que o algoritmo apresenta diferentes "pulos"no gráfico. Estes aumentos significativos repentinos da taxa de processamento ($\alpha$) referem-se à "mudança de nível" na simulação das sequências iniciais, isto é, a quantidade de registros $n$ a se ordenar se tornou maior que a quantidade de sequências iniciais ideias do nível anterior, logo, o algoritmo teve que dar mais um passo na simulação inicial, aumentando a quantidade ideal de sequencias iniciais drasticamente. Nossa hipótese é confirmada ao observar que a frequência destes "picos" diminuem conforme o aumento de $k$; Isso mostra que o algoritmo precisa de cada vez mais registros para passar para o próximo nível da simulação.

Perceba que $\alpha$ também vai diminuindo sutilmente após estes picos. Isso se deve pelo fato de que o cascata vai necessitando de cada vez menos movimentações de registros para ordenar a sequência, pois temos menos dummy runs e mais registros podem ser intercalados simultaneamente.

Além disso, é possível perceber que para $k = 4$ o algoritmo precisa realizar muitas operações de entrada e saída, e consequentemente, apresenta um $\alpha$ muito grande, ao se comparar com os outros valores de $k$. Porém, a medida que aumentamos a quantidade máxima de arquivos abertos ($k$), os valores de $\alpha$ também vão diminuindo.

Este comportamento se assemelha ao visto em [3] (página 290) porém, nele, o autor utiliza o número de passes do algoritmo pelos registros durante a intercalação no eixo das ordenadas, e encontra que, com o aumento de $k$, a transição para novos níveis da simulação tornam-se mais suaves, em comparação às transições abruptas que encontramos em nossos testes com $\alpha$.

# Conclusão

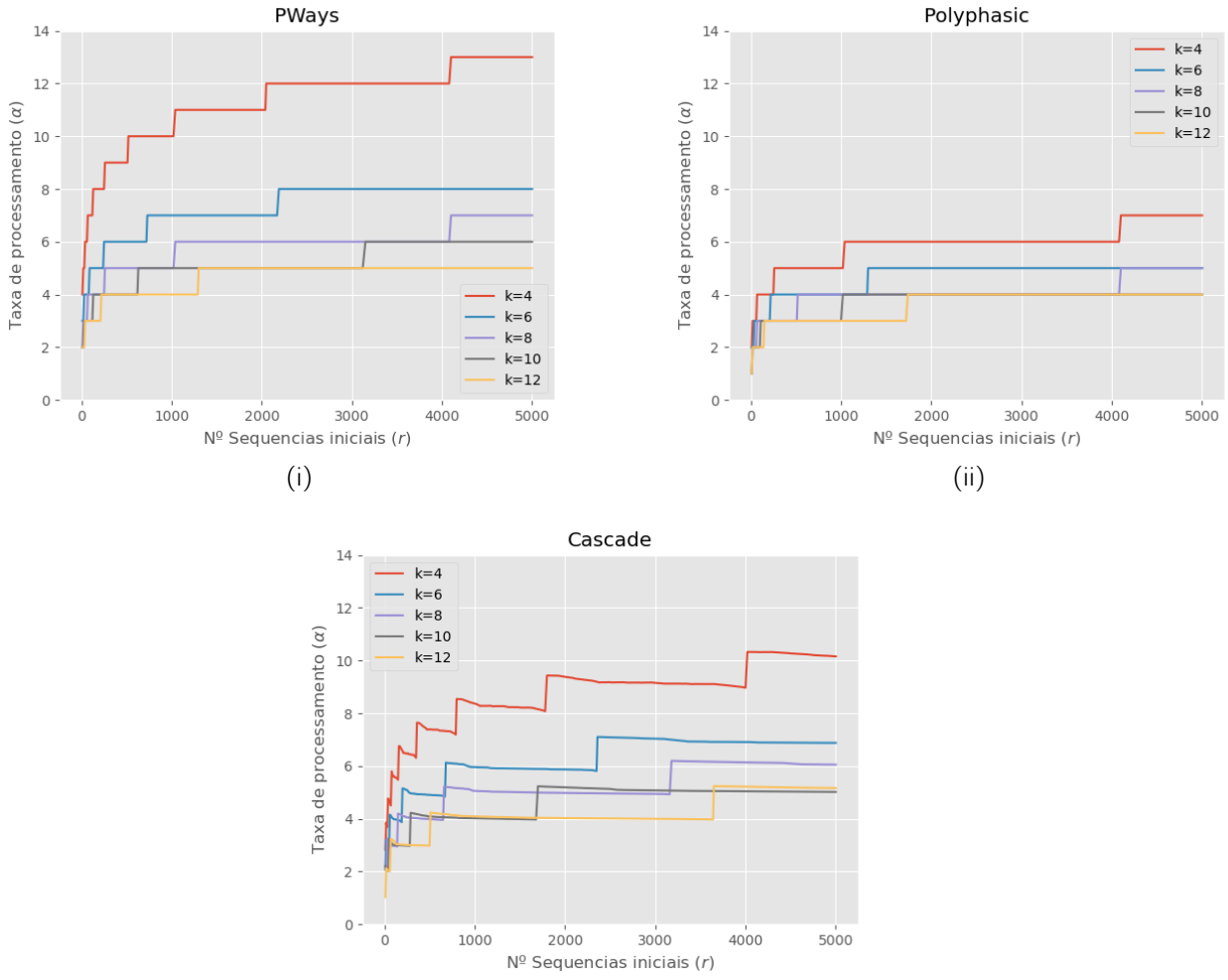



Figura 6: Gráficos de taxa de processamento ($\alpha$) $\times$ Número de sequências iniciais ($r$) de todos os algoritmos apresentado, a fim de comparação. (i) Intercalação Balanceada de $p$-Caminhos, (ii) Ordenação por Intercalação Polifásica e (iii) Ordenação em Cascata.

Observando a figura 6 é possível reparar que, para valores pequenos de $k$, o $p$-Ways apresenta o pior comporta-

mento dentre os algoritmos avaliados. Com o aumento da quantidade máxima de arquivos abertos ($k$), ele aparenta se aproximar assintoticamente dos demais métodos. Ademais, o algoritmo de ordenação polifásica apresenta uma boa performance em relação aos demais em valores mais baixos de $k$, onde a variância deste parâmetro não parece alterar sua taxa de processamento ($\alpha$) como os outros algoritmos.

A figura também deixa mais clara a consistência do polifásico na apresentação do menor $\alpha$ com diferentes parâmetros. Isso ocorre pois o P-Ways, pela sua estrutura, não faz uso otimizado dos arquivos abertos, limitando-se a utilização de $p = \lceil \frac{k}{2} \rceil$ para realização das intercalações, Além disso, o algoritmo pode realizar cópias desnecessárias de sequências, especialmente quando há desbalanceamento entre o número de sequências distribuídas nos $p$ arquivos participantes da intercalação. Isto também é verdade ao comparar o polifásico com o método de ordenação em cascata; devido a grande quantidade de intercalações "extra"que este segundo método realiza, assim, registros são movidos com mais frequência, resultando numa taxa de processamento consistentemente maior que a do polifásico. No entanto, segundo a literatura, para $k \geq 6$, a ordenação em cascata tende a ser assintoticamente melhor do que a polifásica [3].

Dessa forma, levando em consideração a taxa de processamento que o $p$-Ways e o algoritmo de ordenação em cascata apresentaram, concluímos a partir dos experimentos expostos que o polifásico aparenta ser o mais indicado em cenários onde operações de leitura e escrita são lentas de serem realizadas.

# Referências


[1] B. K. Betz e William C. Carter. *New merge sorting techniques*. 1959. URL: `https://api.semanticscholar.org/CorpusID:358711`.

[2] Gordon College. *External Sorting*. `https://www.math-cs.gordon.edu/courses/cs321/lectures/external_sorting.html`. Accessed: 2024-08-23. 1999.

[3] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume 3*. Pearson Education, 1997. ISBN: 9780321635747. URL: `https://books.google.com.br/books?id=x9AsAwAAQBAJ`.

[4] Robert Sedgewick. *Algorithms in C++*. 1st. Addison-Wesley, 1992.

[5] Alan Tharp. *File Organization and Processing*. John Wiley & Sons, Inc, 1988.