Relatório Técnico - Comparação de Algoritmos de Ordenação

https://github.com/mateusdfaria/comparacao-algoritmos-ordenacao

Renan Iomes, Eduardo Klug, Mateus Mautone, Mateus de Faria e Leonardo Rocha

1. Código-Fonte Documentado e Organizado

O código-fonte foi desenvolvido em Python e estruturado em dois arquivos principais:

sorting.py

Contém a implementação do padrão Strategy, com classes para diferentes algoritmos de ordenação:

- Bubble Sort
- Bubble Sort Otimizado
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Tim Sort

Exemplo de Documentação na Classe TimSort:

```
class TimSort(SortingStrategy):
    def sort(self, data):
        """Implementa o algoritmo Tim Sort, híbrido de Merge Sort e Insertion Sort.
        Divide a lista em blocos (run) de tamanho 32, ordena com Insertion Sort
        e combina com Merge Sort. Otimizado para dados parcialmente ordenados.
        """
        def insertion_sort(arr, left, right):
            for i in range(left + 1, right + 1):
                key = arr[i]
                j = i - 1
                while j >= left and arr[j] > key:
                    arr[j + 1] = arr[j]
                    j -= 1
                arr[j + 1] = key
```

```
    def merge(arr, left, mid, right):
        left_arr = arr[left:mid + 1]
        right_arr = arr[mid + 1:right + 1]
        i = j = 0
        k = left
        while i < len(left_arr) and j < len(right_arr):
            if left_arr[i] <= right_arr[j]:
                arr[k] = left_arr[i]
                i += 1
            else:
                arr[k] = right_arr[j]
                j += 1
            k += 1

    def tim_sort(arr):
        n = len(arr)
        RUN = 32
        for i in range(0, n, RUN):
            insertion_sort(arr, i, min((i + RUN - 1), (n - 1)))
        size = RUN
        while size < n:
            for left in range(0, n, size * 2):
                mid = left + size - 1
                right = min((left + size * 2 - 1), (n - 1))
                if mid < right:
                    merge(arr, left, mid, right)
            size *= 2

    tim_sort(data)
    return data
```

main.py

Este arquivo coordena a execução dos algoritmos, gera dados aleatórios e expõe métricas via `prometheus_client`.

Exemplo de Documentação:

```
def main():
    """Inicia o servidor HTTP para métricas e executa testes de ordenação.
    Testa os algoritmos com tamanhos 1000 e 10000 e registra os resultados.
    """
    start_http_server(8000)
```

```
    sizes = [1000, 10000]
    algorithms = [
        ("Bubble Sort", BubbleSort()),
        ("Tim Sort", TimSort()),
        ("Quick Sort", QuickSort()),
    ]

    try:
        while True:
            for size in sizes:
                data = generate_data(size)
                for name, algorithm in algorithms:
                    _, elapsed_time, comparisons, swaps = measure_sorting_time(algorithm, data)
                    print(f"{name}: {elapsed_time:.2f} ms | Comparacoes: {comparisons} | Trocas:
{swaps}")
            time.sleep(30)
    except KeyboardInterrupt:
        print("\nParando o servidor de métricas.")
```

2. Explicação do Uso do Padrão Strategy

O padrão **Strategy** permite encapsular diferentes algoritmos e torná-los intercambiáveis. No projeto:

- `SortingStrategy` define a interface comum.
- `SortingContext` delega a ordenação para a estratégia escolhida.

```
class SortingContext:
    def __init__(self, strategy):
        self.strategy = strategy

    def execute_sort(self, data):
        sorted_data = self.strategy.sort(data.copy())
        return sorted_data
```

3. Descrição do Processo de Geração dos Dados

- **Geração:** Dados são gerados aleatoriamente com:

```
def generate_data(size):
    return [random.randint(0, 100000) for _ in range(size)]
```

- **Execução:** Algoritmos ordenam uma cópia dos dados, e métricas são coletadas via `prometheus_client`.

4. Métricas e Gráficos Comparativos de Desempenho

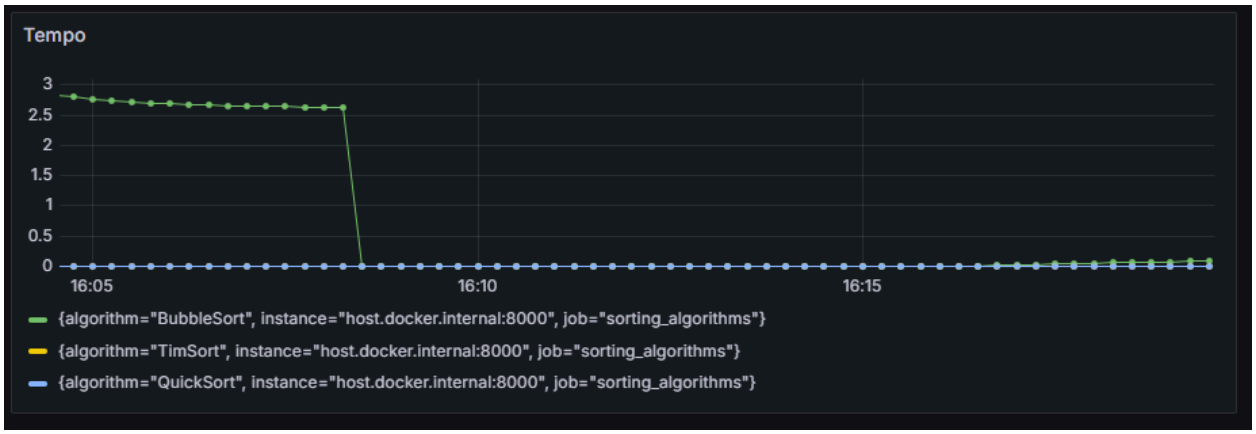
- **Métricas Coletadas:**
  - **Tempo de Execução:** Medido via `Histogram`.
  - **Comparacões:** Contadas via `Counter`.
  - **Trocas:** Contadas via `Counter`.

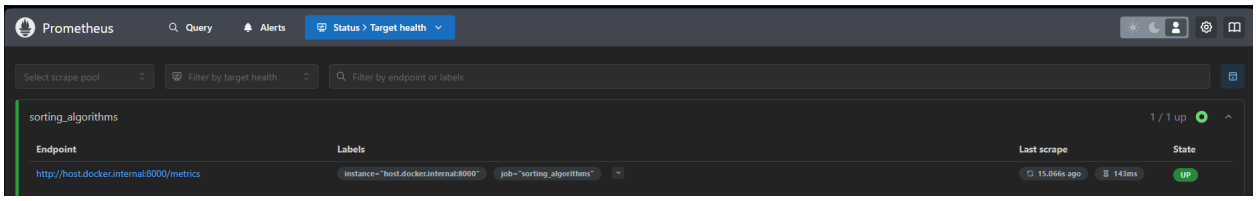
5. Ferramentas Utilizadas para Logs e Análise

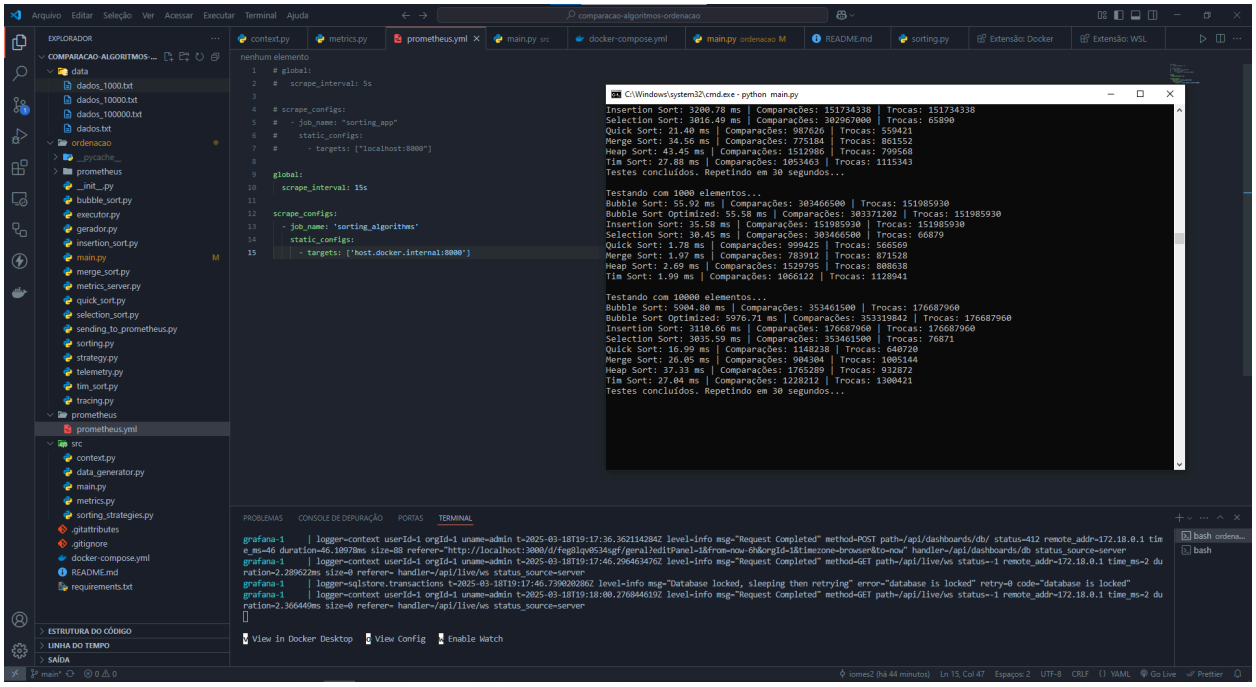
- **Prometheus:** Coleta e armazena métricas.
- **Grafana:** Exibe os dados do Prometheus.

Depois de montar toda a estrutura de arquivos, iniciamos o arquivo main.py com o cmd do computador para os testes, ao mesmo tempo podemos ligar o Grafana no terminal, para entramos no site gerado localmente tendo a sua rota, conectamos a estrutura do grafana com a aplicação Prometheus que está configurada nos códigos python. Vamos para a tela de Dashboards e começamos a criar as queries em suas respectivas tabelas, que são Tempo, Comparação e Troca, configuramos cada uma delas com 3 modelos de algoritmos, Bubble sort, Tim Sort e Quick Sort, seus dados de testes vão sendo gerados pelo cmd do arquivo main.py que ainda está rodando em looping de testes. Configuramos o tempo de leitura dos gráficos para a cada 15 minutos, salvamos o arquivo dos dashboards em formato .json no repositório e também um readme explicativo.


Trocas
2500000000
2000000000
1500000000
1000000000
500000000
0
16:05
16:10
16:15
{__name__="sorting_swaps_total", algorithm="BubbleSort", instance="host.docker.internal:8000", job="sorting_algorithms"}
{__name__="sorting_swaps_total", algorithm="TimSort", instance="host.docker.internal:8000", job="sorting_algorithms"}
{__name__="sorting_swaps_total", algorithm="QuickSort", instance="host.docker.internal:8000", job="sorting_algorithms"}
Comparações
5000000000
4000000000
3000000000
2000000000
1000000000
0
16:05
16:10
16:15
{__name__="sorting_comparisons_total", algorithm="BubbleSort", instance="host.docker.internal:8000", job="sorting_algorithms"}
{__name__="sorting_comparisons_total", algorithm="TimSort", instance="host.docker.internal:8000", job="sorting_algorithms"}
{__name__="sorting_comparisons_total", algorithm="QuickSort", instance="host.docker.internal:8000", job="sorting_algorithms"}


Tempo
3
2.5
2
1.5
1
0.5
0
16:05
16:10
16:15
{algorithm="BubbleSort", instance="host.docker.internal:8000", job="sorting_algorithms"}
{algorithm="TimSort", instance="host.docker.internal:8000", job="sorting_algorithms"}
{algorithm="QuickSort", instance="host.docker.internal:8000", job="sorting_algorithms"}


Prometheus
Query
Alerts
Status > Target health
Select scrape pool
Filter by target health
Filter by endpoint or labels
sorting_algorithms
1 / 1 up
Endpoint
Labels
Last scrape
State
http://host.docker.internal:8000/metrics
instance="host.docker.internal:8000"
job="sorting_algorithms"
15.066s ago
143ms
UP



6. Conclusão: Melhor Algoritmo e Análise

- **Melhor Desempenho:**
  - **Tim Sort** foi o mais eficiente devido à sua abordagem híbrida.
  - **Quick Sort** foi competitivo, mas com piores casos em dados já ordenados.
  - **Bubble Sort** foi o menos eficiente.
- **Vale a Pena Usar "Dividir e Conquistar"?**
  - Sim! Quick Sort e Merge Sort (base do Tim Sort) utilizam essa técnica, reduzindo a complexidade para **O(n log n)**.
  - Para pequenos conjuntos de dados, **Insertion Sort** é mais eficiente devido ao baixo overhead.

Conclusão Final: Tim Sort é a melhor escolha para a maioria dos cenários, especialmente para listas parcialmente ordenadas.