UEPB
Universidade
Estadual da Paraíba

Universidade Estadual da Paraíba
CAMPUS I – Campina Grande

Centro de Ciência e Tecnologia - CCT

Curso de Ciência da Computação

Clara Djeovanna Pereira de Farias

Felipe Eduardo Valdevino da Silva

Zeton Brandon Gonçalves Nascimento

Data: 20/11/2024

SUMÁRIO

1. INTRODUÇÃO
2. DESENVOLVIMENTO
    - 2.1 ALGORITMOS
    - 2.2 CASOS DE TESTE
    - 2.3 AMBIENTE DE EXECUÇÃO
    - 2.4 ANÁLISE COMPARATIVA DOS ALGORITMOS
3. CONCLUSÃO
4. INFORMAÇÕES SOBRE A ATUALIZAÇÃO DO PROJETO

Data	Versão	Descrição	Autores
27/09/2024	0.0	Criação da documentação e projeto	Clara Djeovanna Felipe Eduardo Zeton Brandon
20/11/2024	0.1	Atualização do projeto	Clara Djeovanna Felipe Eduardo Zeton Brandon

INTRODUÇÃO

Este relatório tem como objetivo apresentar os resultados obtidos na análise comparativa de diferentes algoritmos de ordenação aplicados a um dataset contendo informações sobre acidentes de trânsito na República Checa, ocorridos entre 2016 e 2022. Os algoritmos selecionados para este estudo incluem Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Counting Sort, Heapsort e Quick Sort com Mediana 3. Todos esses algoritmos foram previamente abordados em sala de aula, sendo discutidos pelo professor quanto à sua eficiência e aplicabilidade.

O objetivo do projeto foi aplicar transformações, filtragens e ordenações nos dados de acidentes, permitindo uma análise mais aprofundada desses eventos. Serão apresentados os métodos utilizados para a manipulação dos dados, assim como uma discussão sobre o desempenho de cada algoritmo no contexto dos dados de acidentes de trânsito.

DESENVOLVIMENTO

2.1 ALGORITMOS

No projeto, foram utilizados sete algoritmos de ordenação para comparar a eficiência, levando em consideração o tempo de processamento dos dados em diferentes cenários. Os algoritmos aplicados foram:

- **Selection Sort**: Ordena a lista dividindo-a em duas partes: uma parte ordenada e outra não ordenada. Encontra o menor elemento na parte não ordenada e o troca com o primeiro elemento dessa parte. É mais eficaz para listas pequenas.
- **Insertion Sort**: Insere cada elemento da lista não ordenada na posição correta na parte ordenada. Funciona bem com listas quase ordenadas e de pequeno porte.
- **Quick Sort**: Utiliza um elemento pivô para dividir a lista em duas partes e ordená-las recursivamente. É eficaz para grandes volumes de dados, mas pode variar em desempenho dependendo da escolha do pivô.
- **Merge Sort**: Divide a lista em duas metades, ordena cada metade e depois as combina. Oferece um tempo de execução estável, sendo eficiente para grandes datasets.
- **Counting Sort**: Um algoritmo não comparativo que conta o número de ocorrências de valores distintos. Ideal para listas onde os valores têm um intervalo pequeno e conhecido.
- **Heapsort**: Constrói uma estrutura de heap a partir da lista, extraindo os elementos ordenadamente. É eficiente para grandes listas e garante tempo de execução estável.
- **Quick Sort com Mediana 3**: Uma variação do Quick Sort que utiliza a mediana de três elementos como pivô, melhorando o desempenho em listas parcialmente ordenadas.

Cada um desses algoritmos foi aplicado a arquivos CSV contendo informações de acidentes de trânsito, e os tempos de execução foram registrados para análise comparativa.

2.2 CASOS DE TESTE

Para este estudo, os dados foram extraídos do arquivo "road_accidents_czechia_2016_2022.csv". A partir desse arquivo, foram gerados conjuntos de teste representando dois cenários: melhor caso e pior caso, com base na data, tempo e tipo de comunicação do acidente.

Os principais arquivos utilizados como subconjuntos para os testes foram:

- **execution_times_time**: Este arquivo contém os tempos de execução medidos para os diferentes algoritmos de ordenação ao longo do tempo.
- **execution_timesData**: Este arquivo contém dados processados de cada execução dos algoritmos de ordenação, separados por data do acidente, com as métricas necessárias para a análise comparativa.

Os dois tipos de cenários para cada algoritmo seguiram o critério de:

- **Melhor caso**: dados ordenados previamente em ordem crescente.
- **Pior caso**: dados em ordem decrescente.

Os resultados foram organizados em tabelas para facilitar a comparação entre os algoritmos em diferentes situações.

2.3 AMBIENTE DE EXECUÇÃO

O projeto foi desenvolvido na IDE Eclipse, escolhida por sua robustez e suporte ao desenvolvimento em Java. A estrutura do projeto foi organizada em diversas classes, facilitando a manutenção e o reuso do código.

Para a leitura e manipulação de dados, foram criadas classes específicas que realizam a extração e transformação dos dados contidos nos arquivos CSV, permitindo uma análise eficiente. Cada algoritmo de ordenação foi implementado em classes separadas, o que garantiu uma execução organizada e modular.

2.4 ANÁLISE COMPARATIVA DOS ALGORITMOS

A seguir, apresentamos as tabelas de desempenho de cada um dos sete algoritmos de ordenação, considerando diferentes cenários de testes. Os tempos de execução são apresentados em milissegundos, conforme registrado nos arquivos de teste:

EXECUÇÃO PARA O MELHOR CASO POR TEMPO:

Algoritmo	**Melhor Caso (ms)**
Insertion Sort	120287ms
Selection Sort	281559ms
Merge Sort	119ms
Quick Sort	125ms
Quick Sort com Mediana 3	39907ms
Counting Sort	0ms
Heap Sort	120ms

No melhor caso, o algoritmo Counting Sort se destaca por sua eficiência, completando a execução em 0ms. Merge Sort e Heap Sort também apresentam tempos baixos, enquanto Insertion e Selection Sort são significativamente mais lentos.

EXECUÇÃO PARA O PIOR CASO POR TEMPO:

Algoritmo	**Pior Caso (ms)**
Insertion Sort	117657ms
Selection Sort	282294ms
Merge Sort	94ms
Quick Sort	91ms
Quick Sort com Mediana 3	39318ms
Counting Sort	0ms
Heap Sort	104ms

No pior caso, Insertion e Selection Sort mostram desempenhos fracos, levando mais de 117 mil ms e 282 mil ms, respectivamente. Merge Sort e Quick Sort têm desempenhos notáveis, com tempos abaixo de 100ms.

EXECUÇÃO PARA O MELHOR CASO POR DATA:

Algoritmo	**Melhor Caso (ms)**
Insertion Sort	449206
Selection Sort	1176868ms
Merge Sort	216ms
Quick Sort	2401ms
Quick Sort com Mediana 3	2179ms
Counting Sort	45ms
Heap Sort	281ms

Nos organizados pela coluna Date, o Counting Sort novamente se destaca com 45ms. Merge Sort e Heap Sort também apresentam bons resultados, enquanto Insertion e Selection Sort têm tempos muito mais elevados, indicando eficiência muito baixa.

EXECUÇÃO PARA O PIOR CASO POR DATA:

Algoritmo	**Pior Caso (ms)**
Insertion Sort	891628ms
Selection Sort	1211409ms
Merge Sort	255ms
Quick Sort	659ms
Quick Sort com Mediana 3	287ms
Counting Sort	23ms
Heap Sort	283ms

Em cenários de pior caso, Insertion e Selection Sort têm desempenhos extremamente altos, com mais de 890 mil ms e 1,2 milhão ms, respectivamente. Merge Sort e Quick Sort permanecem eficientes, mostrando que esses algoritmos são mais robustos em condições adversas.

CONCLUSÃO

Ao final deste estudo, verificamos que o Counting Sort foi o algoritmo mais eficiente em termos de tempo de execução em todos os cenários. O Merge Sort também se destacou pelo seu desempenho estável, embora consuma mais memória em comparação a outros algoritmos. O Quick Sort e sua variação com Mediana 3 apresentaram variações no tempo de execução, com o Quick Sort tradicional mostrando maior variabilidade no pior caso.

A escolha do algoritmo mais adequado deve levar em consideração a natureza dos dados e as restrições de espaço e tempo de processamento. Este estudo reforça a importância de uma escolha criteriosa de algoritmos de ordenação para otimizar o desempenho em sistemas de análise de dados, especialmente no contexto de grandes volumes de informações como os presentes nos acidentes de trânsito da República Checa.

INFORMAÇÕES SOBRE A ATUALIZAÇÃO DO PROJETO

Execução para o Médio Caso

No cenário intermediário, onde os dados estão parcialmente ordenados ou organizados de maneira aleatória, a análise dos tempos de execução dos algoritmos apresentou os seguintes resultados:

Counting Sort: Demonstrou uma eficiência notável, registrando o menor tempo de execução, com **45ms** no primeiro conjunto de dados e **53ms** no segundo. Este desempenho excepcional reflete a vantagem do algoritmo em trabalhar diretamente com os valores numéricos, explorando sua natureza estruturada e evitando comparações repetitivas.

Merge Sort e **Heap Sort**: Apresentaram tempos de execução mais elevados em comparação ao Counting Sort, com **225ms e 255ms** no Merge Sort e **147ms e 296ms** no Heap Sort, respectivamente, para os dois conjuntos de dados. Embora eficientes, ambos os algoritmos exigem operações adicionais para combinar sublistas ou ajustar estruturas, o que impacta ligeiramente sua velocidade.

Insertion Sort e **Selection Sort**: Registraram tempos significativamente mais altos, com **454596ms e 514956ms** no Insertion Sort e **1008937ms** em ambos os conjuntos para o Selection Sort. Esses algoritmos, devido à sua natureza iterativa e ao número elevado de comparações, são menos adequados para cenários com volumes elevados de dados.

Quick Sort com Mediana de 3 (representado como Quick Sort nos dados): Apresentou uma performance superior ao Insertion e Selection Sort, registrando **3327ms e 367ms** nos dois conjuntos de dados, respectivamente. A escolha estratégica de pivôs minimiza o impacto de partições desequilibradas, conferindo uma melhor eficiência em situações de dados parcialmente ordenados.

EXECUÇÃO ATUALIZADA COM O MÉDIO CASO:

Algoritmo	Casos	Tempo
insertionSort	melhorCaso	325237ms
insertionSort	medioCaso	514956ms
insertionSort	piorCaso	1049842ms
selectionSort	melhorCaso	1290548ms
selectionSort	medioCaso	1008937ms
selectionSort	piorCaso	1937389ms
mergeSort	melhorCaso	122ms
mergeSort	medioCaso	255ms
mergeSort	piorCaso	300ms
quickSort	melhorCaso	3377ms
quickSort	medioCaso	367ms
quickSort	piorCaso	34688ms
countingSort	melhorCaso	45ms
countingSort	medioCaso	53ms
countingSort	piorCaso	47ms
heapSort	melhorCaso	126ms
heapSort	medioCaso	296ms
heapSort	piorCaso	188ms

A principal diferença entre os dois códigos está nas estruturas de dados utilizadas e na modularidade das operações. Vamos analisar em detalhes:

Estruturas de Dados

Primeiro Código:

Utiliza MyArrayList<String[]>, que parece ser uma implementação personalizada de uma estrutura de lista. Isso permite mais flexibilidade na manipulação de registros, como o uso de métodos como shuffle() e reverse() diretamente associados à estrutura.

A modularidade da estrutura personalizada ajuda a encapsular a lógica de manipulação e ordenação dos registros.

Segundo Código:

Utiliza arrays bidimensionais (String[][]). Isso implica que operações como shuffle e reverse precisam ser implementadas manualmente, pois arrays em Java não possuem métodos nativos para essas operações.

Torna a manipulação mais procedural e menos modular, já que métodos como shuffle e reverse são implementados como métodos estáticos adicionais na classe principal.

Modularidade nos Filtros

Primeiro Código:

Os métodos de filtro em AccidentFilter retornam diretamente um MyArrayList<String[]>, que é manipulado e salvo no arquivo em chamadas separadas.

Exemplo:

java

```
MyArrayList<String[]> acidentesComAlcool =
AccidentFilter.filtrarAcidentesPorAlcool(arquivoEntrada);

AccidentFilter.salvarRegistrosEmArquivo(acidentesComAlcool, arquivoSaidaAlcool);
```

Isso permite maior reuso da lista filtrada sem depender imediatamente de gravação em arquivo.

Segundo Código:

Os métodos de filtro em AccidentFilter não retornam os registros diretamente. Eles recebem o arquivo de entrada e saída como parâmetros e salvam os dados processados diretamente no arquivo de saída.

Exemplo:

java

AccidentFilter.filtrarAcidentesPorAlcool(arquivoEntrada, arquivoSaidaAlcool);

Esse estilo de design é mais simples, mas menos flexível para manipulações adicionais antes da gravação no arquivo.

Lista de Algoritmos

Primeiro Código:

A lista de algoritmos inclui: java

{"insertionSort", "selectionSort", "mergeSort", "quickSort", "countingSort", "heapSort"}

Segundo Código:

Adiciona um algoritmo extra:

quickSortMedianaDeTres.java

{"insertionSort", "selectionSort", "mergeSort", "quickSort", "quickSortMedianaDeTres", "countingSort", "heapSort"}

Manuseio dos Casos

Primeiro Código:

O shuffle e reverse são métodos implementados na estrutura MyArrayList, tornando o código mais limpo e intuitivo: java

records1.shuffle();

records1.reverse();

Segundo Código:

Esses métodos precisam ser implementados manualmente como funções auxiliares estáticas na classe principal: java

private static void shuffle(String[][] records) { ... }

private static void reverse(String[][] records) { ... }

Leitura e Processamento

Primeiro Código:

A leitura do CSV retorna um objeto MyArrayList<String[]>.java

MyArrayList<String[]> records1 = AccidentProcessorData.readCsv(csvFilePath);

Segundo Código:

A leitura do CSV retorna um array bidimensional (String[][]).java

String[][] records1 = AccidentProcessorData.readCsv(csvFilePath);

Conclusão

Primeiro Código:

Mais modular e reutilizável devido ao uso de MyArrayList.

Facilita a manipulação de dados com métodos embutidos na estrutura.

Mais robusto para evoluções futuras.

Segundo Código:

Simples e direto, mas menos modular.

Usa arrays nativos, o que reduz a flexibilidade na manipulação de dados.

Inclui um algoritmo adicional (quickSortMedianaDeTres).

Se o objetivo for facilidade de manutenção e escalabilidade, o primeiro código é superior. Se o objetivo for simplicidade ou evitar dependências externas, o segundo código é mais adequado.