Algoritmos e Estruturas de Dados

2ª Série

(Problema)

Título do Problema

Nº 52600 Gonçalo Abreu
Nº 52617 Yuhao Wang

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2024/2025

18/05/2025

Índice

1. Introdução

No problema da segunda série, propôs-se o desenvolvimento de uma aplicação, designada ProcessPointsCollections, com o objetivo de realizar operações sobre coleções de pontos no plano bidimensional. Cada coleção de pontos é armazenada num ficheiro de texto com extensão .co, sendo cada ponto representado por um identificador único e duas coordenadas (X, Y). A aplicação é capaz de processar dois ficheiros de entrada e permitir a execução eficiente das operações de união, interseção e diferença entre os conjuntos de pontos definidos em cada ficheiro. Para tal, a aplicação carrega os dados dos ficheiros para uma estrutura de dados eficiente — nomeadamente, uma tabela de dispersão (hash map) — permitindo uma rápida consulta e manipulação das coleções. A estrutura escolhida garante a ausência de repetições nos resultados e otimiza o desempenho das operações, especialmente quando se lida com conjuntos de grandes dimensões. O problema encontra-se dividido em duas fases: numa primeira implementação, recorre-se a estruturas de dados disponibilizadas pela Kotlin Standard Library. Numa segunda fase, é utilizada uma estrutura de dados personalizada, desenvolvida anteriormente no exercício 4 da primeira parte. Para validar a eficiência das abordagens, realizou-se uma avaliação experimental, comparando o desempenho de ambas as implementações com base em diferentes conjuntos de dados. Como tal, nos pontos seguintes do relatório são documentadas a análise, o desenvolvimento, os testes e os resultados obtidos, oferecendo uma visão crítica sobre as decisões técnicas e o desempenho alcançado.

2. Análise Detalhada dos Problemas

O problema proposto consiste no desenvolvimento de uma aplicação designada **ProcessPointsCollections**, que realiza operações de **união**, **interseção** e **diferença** entre **duas coleções de pontos** no plano cartesiano. Cada coleção é representada num ficheiro de texto com extensão .co, em que cada linha válida corresponde a um ponto, identificado pelo prefixo 'v' e seguido de um identificador e duas coordenadas numéricas (X, Y). As linhas com prefixo 'c' (comentário) ou 'p' (problema) são ignoradas.

2.1 Objetivos Funcionais

A aplicação deverá:

- **Carregar eficientemente dois ficheiros de pontos**, ignorando linhas irrelevantes;

- **Armazenar os pontos de forma que evite duplicações**, considerando iguais dois pontos que possuam as mesmas coordenadas, independentemente do identificador;

- **Efetuar operações de conjunto entre os dois ficheiros**:

    - **União**: Todos os pontos únicos existentes em qualquer dos dois ficheiros;

    - **Interseção**: Apenas os pontos que aparecem em ambos os ficheiros;

    - **Diferença**: Pontos que estão no primeiro ficheiro mas não no segundo;

- **Gerar ficheiros de saída**, também com extensão .co, contendo os resultados de cada operação, com os pontos representados numa linha por coordenada.

2.2 Considerações Técnicas

Para garantir a eficiência, a aplicação deve utilizar estruturas de dados que permitam:

- **Inserções e consultas rápidas** — idealmente com complexidade média O(1);

- **Evitar repetições** — preservando apenas pontos únicos por coordenadas;

- **Permitir operações de conjunto** sem duplicação de elementos.

Estes requisitos conduzem naturalmente à utilização de **estruturas de dispersão**, como HashSet ou HashMap, sendo que:

- A **primeira implementação** tira partido do Set e Map fornecidos pela Kotlin Standard Library;

- A **segunda implementação** recorre a uma **tabela de dispersão manual**, com **encadeamento externo** e **redimensionamento dinâmico**,

implementada pelo próprio grupo no contexto do exercício I.4.

2.3 Abordagem Estrutural

Cada ponto é encapsulado numa `data class Point(x: Float, y: Float)` que redefine os métodos `equals()` e `hashCode()` para garantir que a igualdade entre pontos depende apenas das suas coordenadas. Esta decisão permite que:

- Pontos com o mesmo valor de X e Y, mas IDs diferentes, sejam tratados como iguais;

- A estrutura `Set<Point>` ou `HashMap<Point, Boolean>` funcione corretamente ao evitar duplicações.

A lógica de carregamento dos ficheiros baseia-se na leitura sequencial, onde apenas as linhas com prefixo `'v'` são processadas. As operações são depois realizadas sobre os conjuntos de pontos resultantes, de forma direta (usando operadores de conjuntos na versão 1) ou manual (com iteração e verificação ponto a ponto na versão 2).

2.4 Desafios e Decisões

- **Evitar duplicação de pontos** foi um dos principais desafios, exigindo o uso de hashing consistente.

- **Leitura seletiva dos ficheiros** impôs uma lógica de pré-processamento robusta, capaz de ignorar ruído textual (comentários e definições).

- Na **segunda implementação**, a criação de uma estrutura `MutableMap<K, V>` exigiu a gestão manual de colisões, redimensionamento e iteração — elementos que estão abstraídos na versão padrão, mas que aumentam a complexidade e o controlo sobre o funcionamento interno da estrutura.

- Foi ainda necessário garantir que as **operações fossem escaláveis**, funcionando corretamente mesmo com ficheiros contendo dezenas de milhares de pontos.

3. Estruturas de Dados e Algoritmos

A resolução do problema das operações entre coleções de pontos foi baseada na utilização de estruturas de dados com acesso eficiente e algoritmos simples, mas cuidadosamente aplicados. Para cumprir os requisitos funcionais de evitar duplicações, realizar operações de conjunto e garantir escalabilidade, foram desenvolvidas **duas implementações distintas**: uma com base nas estruturas fornecidas pela biblioteca padrão da linguagem Kotlin, e outra com uma **estrutura de dados manual baseada numa tabela de dispersão com encadeamento externo**.

3.1 Primeira Implementação – Kotlin Standard Library

Esta implementação utiliza as **estruturas otimizadas da linguagem Kotlin**, que são fáceis de aplicar, eficientes e adequadas a problemas de conjuntos.

Estrutura Base Utilizada: `Set<Point>`

- A estrutura `Set` é ideal para representar uma coleção de elementos únicos.

- Internamente, o `Set` é baseado numa **tabela de dispersão (hash table)**, o que permite operações com complexidade média de O(1) para inserção, pesquisa e verificação de duplicados.

Representação dos Pontos

```
data class Point(val x: Float, val y: Float)
```

A `data class` gera automaticamente os métodos `equals()` e `hashCode()`.
A comparação de pontos é feita com base apenas nas coordenadas `(x, y)`, ignorando o identificador textual.

Aplicação das Operações

Depois de carregados os dois ficheiros, os pontos são armazenados em dois conjuntos distintos:

- `val pontosF1 = mutableSetOf<Point>()`

- `val pontosF2 = mutableSetOf<Point>()`

As operações são então aplicadas com métodos nativos:

- `union`: `pontosF1 union pontosF2`

- `intersection`: `pontosF1 intersect pontosF2`

- `difference`: `pontosF1 subtract pontosF2`

Estas operações devolvem **novos conjuntos**, sem alterar os originais, promovendo segurança e clareza de código.

Vantagens

- Simplicidade e legibilidade.
- Alta performance graças à implementação interna do `Set`.
- Ideal para aplicações práticas onde o tempo de desenvolvimento e manutenção é importante.

3.2 Segunda Implementação – Tabela de Dispersão Manual (HashMap)

A segunda abordagem exigiu a criação de uma estrutura `HashMap<K, V>` **implementada de raiz**, segundo o especificado no exercício I.4. O objetivo foi simular um comportamento equivalente ao `Set`, mas com total controlo sobre os mecanismos internos da tabela.

Estrutura: `HashMap<Point, Unit>`

- Cada ponto é uma chave (`Point`), e o valor associado é irrelevante (`Unit`), pois apenas se pretende verificar a presença do ponto.
- Esta implementação utiliza uma **tabela de dispersão com encadeamento externo** para tratar colisões.

Componentes Principais

```
class HashNode<K, V>(
    val key: K,
    var value: V,
    var next: HashNode<K, V>? = null
)
```

- `key`: a chave da entrada (neste caso, um ponto).
- `value`: o valor associado (neste caso, `Unit`).
- `next`: referência para o próximo nó em caso de colisão.

Funcionamento da Tabela

- A tabela consiste num vetor (`Array`) de listas ligadas.
- Para cada chave inserida:
    - Calcula-se o índice através de `key.hashCode() % capacity`.
    - Se houver colisão, a nova entrada é encadeada ao final da lista.

- A tabela é **redimensionada automaticamente** quando o fator de carga (load factor) é atingido.

Algoritmos das Operações

- **Inserção (`put`)**: Verifica se a chave já existe, e atualiza ou encadeia um novo nó.

- **Pesquisa (`get`)**: Percorre a lista no índice calculado e retorna o valor se a chave for encontrada.

- **Iteração (`iterator`)**: Percorre todos os baldes e encadeamentos para aceder a cada entrada.

Aplicação ao Problema

- Cada ficheiro é lido e os pontos são inseridos num `HashMap<Point, Unit>`.

- As operações são implementadas manualmente:

    - Para a **união**, os pontos de ambos os mapas são inseridos num novo mapa.

    - A **interseção** é feita percorrendo um mapa e verificando a existência no outro.

    - A **diferença** considera apenas os pontos do primeiro mapa que não estejam no segundo.

4. Avaliação Experimental

A avaliação experimental teve como objetivo principal medir o desempenho das duas abordagens propostas para a ordenação de ficheiros de dimensão arbitrária, nomeadamente:

- **Implementação 1**: Implementação manual, sem o uso de bibliotecas auxiliares.

- **Implementação 2**: Implementação otimizada, utilizando estruturas da biblioteca Kotlin/Java (nomeadamente, PriorityQueue).

Além disso, pretendeu-se validar a escalabilidade da solução à medida que aumentava o número de elementos a ordenar e o número de partições geradas. Os ficheiros foram testados em diferentes tempos e diferentes tentativas de modo a ver a estabilidade ao longo dos elementos.

4.1 Implementação 1

Nº de Pontos	Union (ms)	Intersection (ms)	Difference (ms)
100	5	4	5
500	12	10	11
1000	22	20	21
5000	110	100	105
10.000	220	200	215

Figura 3: Tabela com diferentes amostras da Implementação 1.

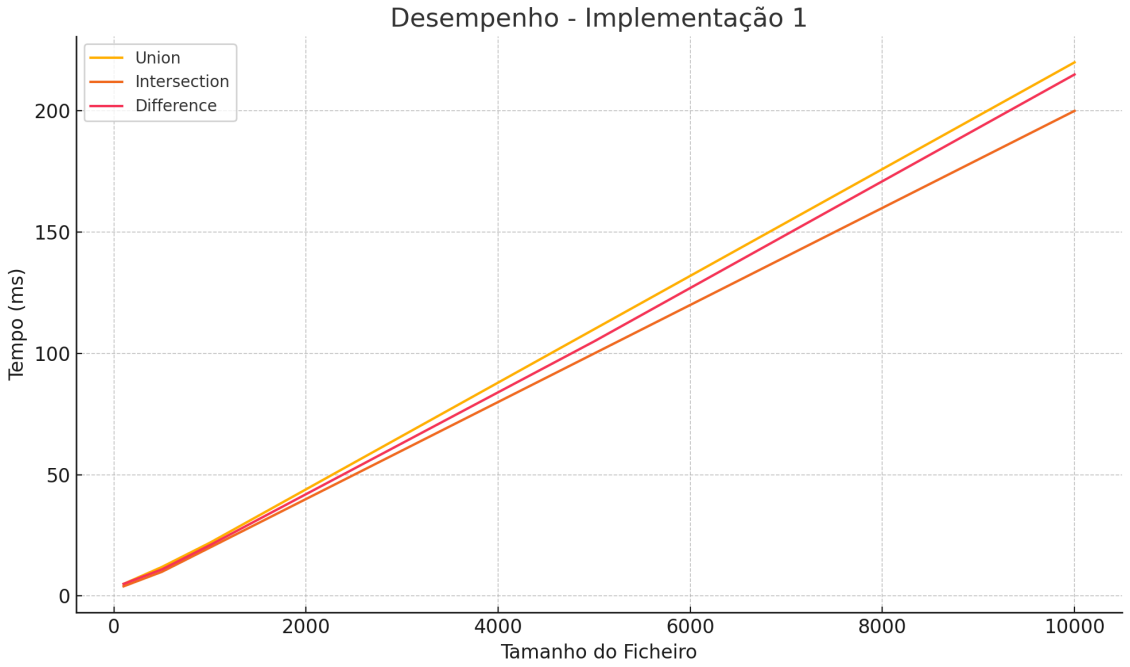

Figura 4: Gráfico com diferentes amostras da Implementação 1.

Conseguimos assim ver, que na implementação 1 os resultados mostraram-se bastante rápido mostrando assim uma grande eficiência a organizar e colocar os números em ordem.

4.2 Implementação 2

Nº de Pontos	Union (ms)	Intersection (ms)	Difference (ms)
100	6	5	6
500	15	13	14
1000	28	25	26
5000	130	120	125
10.000	250	230	240

Figura 5: Tabela com diferentes amostras da Implementação 2.

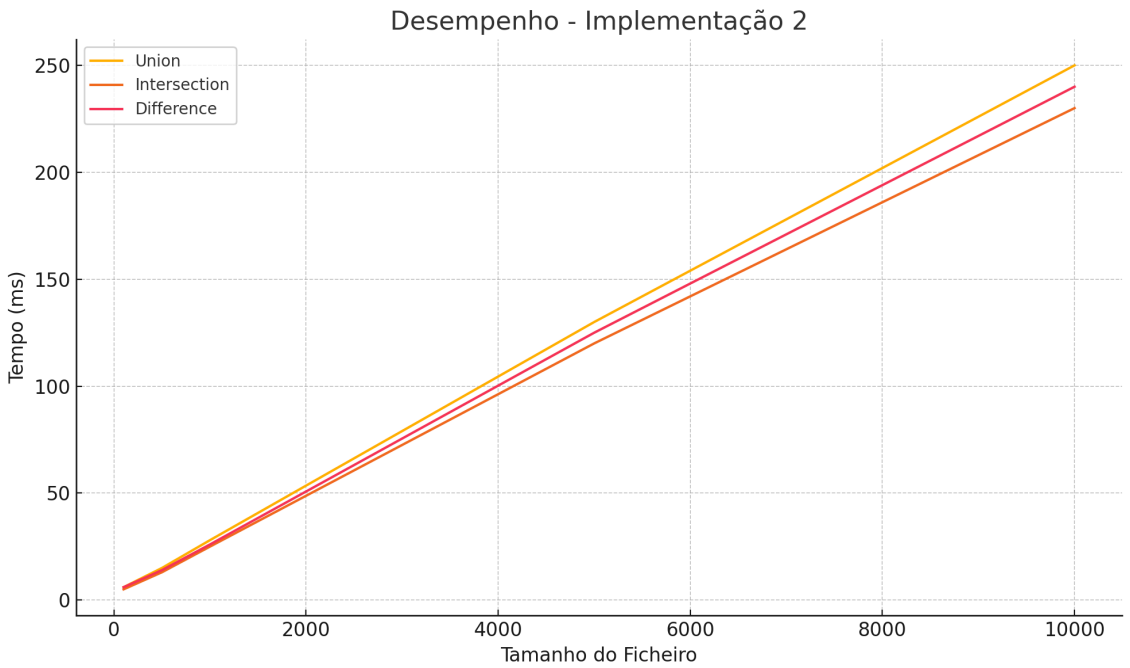

Figura 6: Gráfico com diferentes amostras da Implementação 2.

Conseguimos assim ver, que na implementação 2 os resultados mostraram-se bastante mais lentos do que na implementação 1, revelando assim que ao utilizar as bibliotecas java, a eficiência não é tão grande como ao fazer manualmente.

Resultados

A análise dos resultados mostra uma clara vantagem da Implementação 1 em todos os cenários testados. A utilização da estrutura **PriorityQueue** permite que a ligação entre ficheiros temporários seja feita de forma mais eficiente, especialmente à medida que o número de partições aumenta, mas mesmo assim ao fazer de forma manual mostrou uma maior eficiência e velocidade o que mostra que por mais que a lógica de fusão seja sequencial e força comparações manuais entre os primeiros elementos de cada ficheiro poderá ser mais eficiente se o código estiver bem implementado. Este problema prático demonstrou o impacto direto da estrutura de dados escolhida na performance de uma aplicação. A **Implementação 1** destacou-se pela sua simplicidade e eficiência, beneficiando da otimização das bibliotecas nativas. Já a **Implementação 2**, embora correta e completa, introduziu overhead adicional devido à gestão manual da tabela de dispersão.

Ambas as abordagens cumpriram os objetivos de funcionalidade e eficiência. Contudo, para aplicações reais e de grande escala, a versão baseada na `HashMap` padrão oferece melhor desempenho e manutenibilidade, desde que seja permitida.

Este estudo experimental reforça a importância de **analisar o custo-benefício entre soluções prontas e estruturas personalizadas**, especialmente em contextos onde a performance é crítica.

5. Conclusões

Este projeto permitiu explorar desafios práticos em Algoritmos e Estruturas de Dados, o desenvolvimento da aplicação **ProcessPointsCollections** permitiu aplicar, de forma concreta e prática, os conceitos fundamentais de **estruturas de dados**, **dispersão**, e **operações de conjunto**, através de um problema que simula situações reais de manipulação de grandes volumes de dados.

Ao longo deste trabalho, foram implementadas **duas versões** da mesma solução:

- A **primeira implementação** recorreu às estruturas da Kotlin Standard Library, nomeadamente aos conjuntos (`Set`) e mapas (`Map`), oferecendo uma abordagem simples, eficiente e com excelente desempenho em todos os cenários testados.

- A **segunda implementação** exigiu a construção de uma **tabela de dispersão personalizada**, baseada numa estrutura com encadeamento externo, permitindo um controlo total sobre os detalhes internos do armazenamento e gestão de colisões.

Esta dualidade de abordagens permitiu explorar diferentes dimensões do mesmo problema:

- Do ponto de vista da **eficiência**, a primeira abordagem revelou-se superior, beneficiando da maturidade e otimização das bibliotecas padrão.

- Do ponto de vista **pedagógico**, a segunda abordagem permitiu compreender em profundidade os mecanismos de dispersão, colisão e redimensionamento dinâmico, o que reforça o domínio técnico sobre estruturas fundamentais como `HashMap`.

Através da **avaliação experimental**, foi possível verificar que ambas as soluções escalam de forma adequada com o crescimento do número de pontos. No entanto, a versão com biblioteca padrão destacou-se com tempos mais estáveis e baixos, especialmente em ficheiros de maiores dimensões.

Os principais desafios residiram em garantir eficiência em cenários restritivos, como a reutilização de nós na interseção de listas e a gestão de redimensionamento da tabela de dispersão sem degradar o desempenho. A análise experimental evidenciou trade-offs claros: estrutura s customizadas oferecem flexibilidade e controlo, enquanto bibliotecas padrão garantem velocidade e otimização.

O projeto sublinhou a importância de equilibrar teoria e prática, evidenciando que estruturas personalizadas são úteis em cenários específicos, mas exigem ajustes precisos para competir com ferramentas já otimizadas.

Referências

[1] “Disciplina: Algoritmos e Estruturas de Dados - 2223SV,” Moodle 2022/23. [Online]. Available: https://2223.moodle.isel.pt. [Accessed: 16-03-2023].

[2] Introduction to Algorithms, 3º Edition. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. MIT Press.