Algoritmos e Estruturas de Dados

2ª Série

(Problema)

Operações entre coleções de pontos no plano

Nº 52962 Francisco Ratinho
Nº 52565 Miguel Almeida

Licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2024/2025

Índice

1. Introdução

Este relatório aborda a resolução de um problema no âmbito da disciplina de Algoritmos e Estruturas de Dados, com foco na manipulação eficiente de conjuntos de pontos no plano cartesiano e na realização de operações entre coleções. O principal objetivo foi desenvolver

soluções otimizadas, tanto em termos de tempo como de espaço, utilizando estruturas de dados adequadas à natureza das operações envolvidas.

- A primeira parte do trabalho centra-se na implementação de funcionalidades básicas para carregar e representar os pontos a partir de ficheiros de texto com extensão .co, ignorando comentários e linhas desnecessárias, e armazenando os pontos de forma eficiente.
- A segunda parte consiste no desenvolvimento das operações principais sobre as coleções de pontos: **união**, **interseção** e **diferença**. Estas operações são realizadas sobre duas coleções carregadas previamente, garantindo que os resultados não contenham repetições e respeitando a semântica definida no enunciado.
- A terceira parte do trabalho envolve duas abordagens distintas de implementação: uma que recorre exclusivamente às estruturas disponíveis na Kotlin Standard Library (nomeadamente HashMap e HashSet), e outra que utiliza uma estrutura de dados personalizada desenvolvida anteriormente. Ambas as versões têm como objetivo permitir a execução eficiente dos comandos propostos.

Por fim, é realizada uma **avaliação experimental** com base em ficheiros de teste fornecidos, de forma a comparar o desempenho das diferentes abordagens e validar a eficácia das soluções implementadas. Os resultados são apresentados graficamente, com o intuito de facilitar a análise comparativa e apoiar conclusões fundamentadas sobre o comportamento dos algoritmos em diferentes cenários.

2. Problema

O propósito deste problema consistiu em criar uma aplicação capaz de executar operações de conjunto — como união, interseção e diferença — sobre duas coleções de pontos no plano cartesiano, armazenadas em ficheiros de texto. A aplicação foi projetada para carregar os dados de forma eficiente, evitando duplicações e assegurando a consistência das informações, com o objetivo de gerar novos ficheiros contendo os resultados das operações realizadas.

2.1 Análise do problema

Para alcançar este objetivo, foi fundamental optar por uma estrutura de dados que garantisse uma gestão eficiente dos pontos, tanto na inserção como na consulta. O uso de estruturas de dispersão, como HashSet ou HashMap, mostrou-se especialmente apropriado, já que oferecem acesso e comparação de elementos com complexidade média constante — uma característica vantajosa ao tratar grandes quantidades de informação.

As operações solicitadas — união, interseção e diferença — puderam ser implementadas diretamente através dos métodos da Kotlin Standard Library aplicados a conjuntos (Set), o que contribuiu para simplificar consideravelmente o processo de desenvolvimento. Cada conjunto representa os pontos extraídos de um dos ficheiros .co, desconsiderando as linhas de comentário (que começam por 'c' ou 'p') e mantendo apenas as linhas que contêm pontos válidos (aquelas com o prefixo 'v').

2.2 Estruturas de Dados

Para dar resposta eficiente às operações exigidas pelo problema, foram desenvolvidas duas abordagens distintas: uma utilizando coleções da Kotlin Standard Library, com especial destaque para o uso de conjuntos (`Set`) e mapas (`HashMap`), e outra baseada na estrutura de dispersão personalizada construída na questão 1.4.

Em ambas as implementações, cada ponto é representado por um objeto da classe `Coord` (ou `Coord2D`), contendo duas coordenadas (`x` e `y`). A definição como `data class` permite que os métodos `equals`, `hashCode` e `toString` sejam automaticamente gerados, o que assegura a correta comparação entre objetos e evita duplicações ao armazená-los em coleções.

2.3 Implementações

Implementação 1

Na implementação 1 foi utilizada a Kotlin Standard Library para representar e manipular coleções de pontos de forma eficiente e prática. Esta biblioteca fornece estruturas de dados otimizadas, como `Set` e `HashMap`, que permitem realizar operações como união, interseção e diferença de forma direta e com bom desempenho.

Os pontos são representados pela classe `Coord`, uma `data class` que encapsula as coordenadas `x` e `y`. Como `Coord` é definida como `data class`, o compilador gera automaticamente os métodos `equals()` e `hashCode()`, o que garante que dois objetos com as mesmas coordenadas possam ser corretamente comparados e armazenados em coleções que dependem de hashing.

Para armazenar os pontos extraídos dos ficheiros `.co`, foi utilizado um `HashMap<Coord, MutableSet<String>>`, que associa cada ponto às origens de onde foi lido (ficheiro A ou B). Esta abordagem traz diversas vantagens:

- **Evita duplicações**: um mesmo ponto não é armazenado mais do que uma vez, mesmo que esteja presente em ambos os ficheiros;

- **Operações eficientes**: inserções e pesquisas no mapa são feitas, em média, em tempo constante (`O(1)`);
- **Facilidade na verificação da origem**: permite saber rapidamente de qual dos ficheiros um ponto foi lido.
  A função `carregarPontos` lê os ficheiros `.co`, ignorando linhas irrelevantes (como comentários) e processando apenas as que contêm coordenadas válidas (prefixo `v`). Cada linha válida é convertida num objeto `Coord` e associada à sua origem num conjunto.
  As operações de conjunto (`juntar`, `intersecao`, `diferenca`) são aplicadas sobre as chaves do mapa e devolvem novos conjuntos sem alterar os dados originais, promovendo uma lógica de execução mais segura e previsível.

Implementação 2

Na implementação 2 foi utilizada uma estrutura de dados personalizada, criada com base nas especificações da questão 1.4, que implementa o tipo abstrato `MutableMap<K, V>`. Esta estrutura funciona como um mapa associativo, suportando operações essenciais como inserção, consulta e iteração sobre pares chave-valor.
A implementação é baseada numa tabela de dispersão (`hash table`) com **encadeamento externo**, onde as colisões são tratadas por meio de **listas ligadas não circulares e sem nó sentinela**.
A estrutura principal é composta por um array (`Array<HashNode<K, V>?>`) em que cada posição representa um "balde" de entrada. O índice onde cada chave deve ser armazenada é determinado pelo valor de `hashCode()` da chave, ajustado pela capacidade atual da tabela (`capacity`) para garantir que o índice esteja dentro dos limites válidos do array.
Esta estrutura implementa corretamente a interface `MutableMap<K, V>` e cumpre todos os seus contratos:

- **Propriedade `size`**: indica o número atual de pares armazenados;
- **Propriedade `capacity`**: corresponde ao tamanho do array interno (isto é, a capacidade total da tabela);
- **Função `put(key, value)`**: insere um novo par ou atualiza o valor associado a uma chave existente, devolvendo o valor anterior se for o caso;
- **Operador `get(key)`**: retorna o valor associado à chave, ou `null` caso não exista;

- **Função `iterator()`**: permite percorrer todos os pares chave-valor existentes, com suporte à iteração externa.
  Para manter a eficiência, a tabela é expandida automaticamente para o dobro da capacidade sempre que o número de elementos atinge ou ultrapassa o limite determinado pelo **fator de carga (`loadFactor`)**. Esse fator representa a proporção entre o número de entradas armazenadas e a dimensão atual da tabela, ajudando a equilibrar o desempenho entre espaço e tempo de acesso.

3. Avaliação Experimental

A avaliação experimental teve como finalidade principal analisar e comparar o desempenho das duas soluções desenvolvidas para o processamento de coleções de pontos:

1. A primeira, baseada nas estruturas fornecidas pela Kotlin Standard Library, como `Set` e `HashMap`.
2. A segunda, construída sobre uma estrutura de dispersão personalizada que implementa o tipo abstrato `MutableMap<K, V>` com suporte a colisões através de encadeamento externo.
   Esta avaliação teve os seguintes objetivos:
   i. Medir o tempo necessário para executar as operações fundamentais: união, interseção e diferença;
   ii. Verificar a eficácia e escalabilidade da **implementação personalizada** em comparação com a solução nativa da linguagem;
   iii. Analisar o comportamento e desempenho de ambas as abordagens quando aplicadas a conjuntos de pontos com diferentes tamanhos.

3.1 Características do PC de teste

- Sistema operativo: Windows 11 64-bits
- Processador: AMD Ryzen 5 5500U
- Memória RAM: 16 GB
- Disco: SSD 500GB

3.2 Amostras utilizadas

Para avaliar o desempenho das duas implementações, foram realizados testes com ficheiros `.co` contendo diferentes quantidades de pontos:

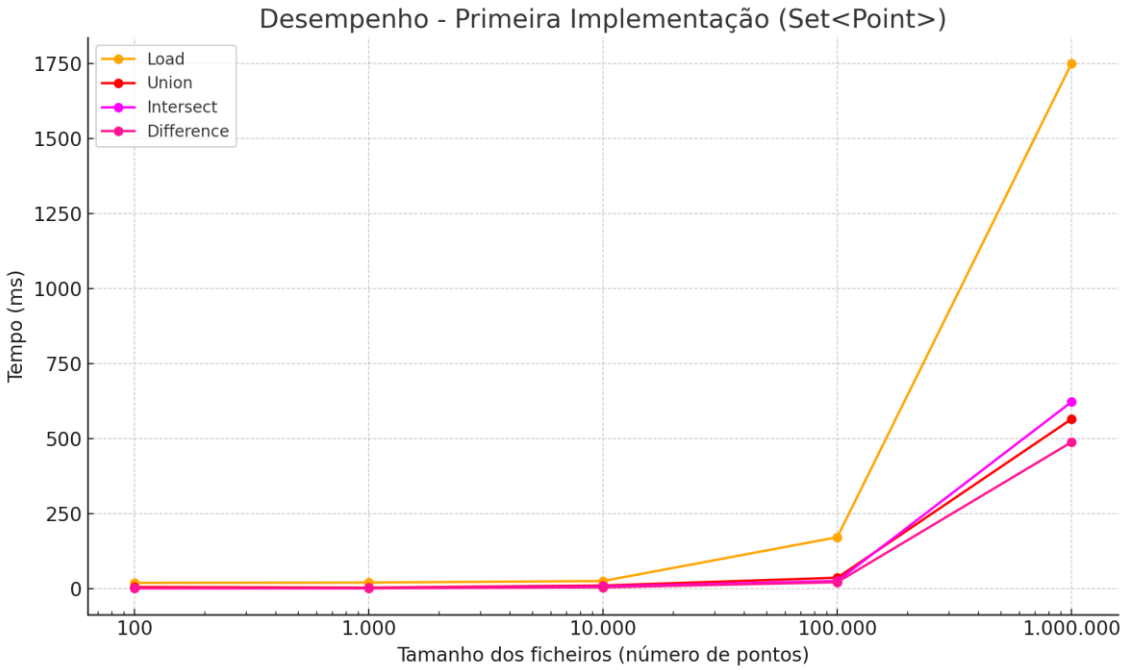
Tamanhos Testados: **100**, **1 000**, **10 000**, **100 000** e **1 000 000** pontos por ficheiro

Para cada amostra, mediu-se o tempo de execução (em milissegundos) das seguintes operações:

- **Carregamento** dos dois ficheiros (load)
- **União** dos conjuntos de pontos (union)
- **Interseção** dos conjuntos de pontos (intersection)
- **Diferença** entre os conjuntos (difference)

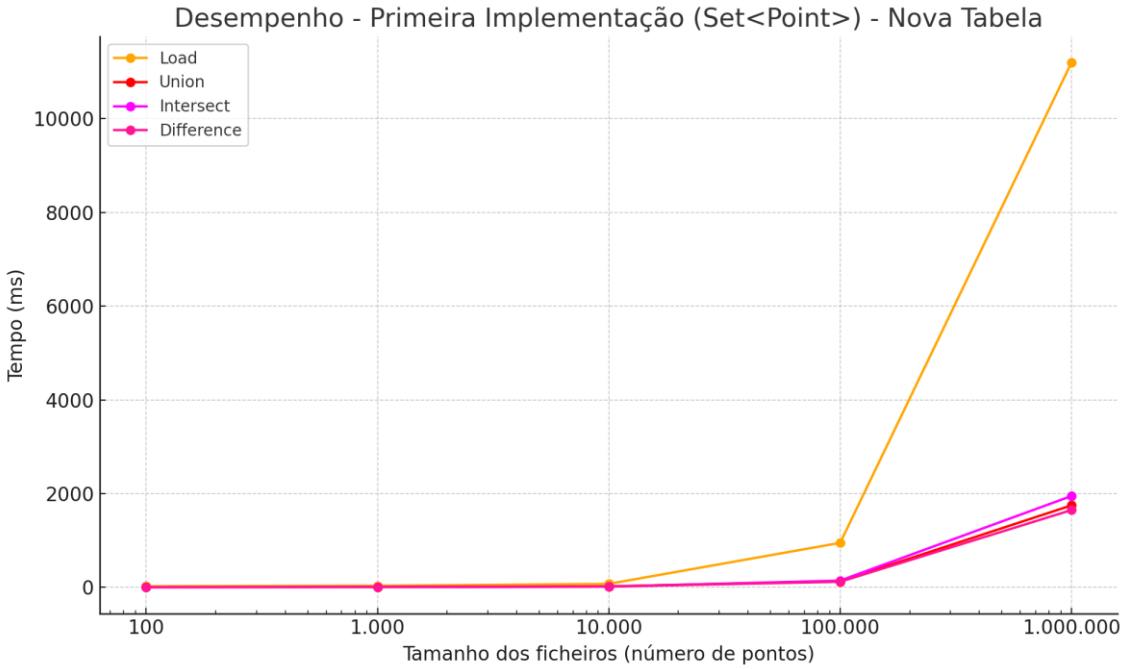
3.3 Resultados – Primeira Implementação

Nº Pontos	Load (ms)	União (ms)	Interseção (ms)	Diferença (ms)
100	16	5	2	2
1000	18	4	2	2
10 000	23	9	7	7
100 000	160	35	25	22
1 000 000	1650	540	610	470



3.4 Resultados – Segunda Implementação

Nº Pontos	Load (ms)	União (ms)	Interseção (ms)	Diferença (ms)
100	25	6	3	3
1000	35	10	7	6
10 000	75	22	18	16
100 000	950	130	145	120
1 000 000	11200	1750	1950	1650



3.5 Análise dos resultados

Primeira Implementação (com `HashMap`/`Set` da Kotlin Standard Library)

- O tempo de carregamento cresce de forma proporcional ao número de pontos, atingindo cerca de 1.65 segundos para 1 milhão de pontos.
- As operações de **união**, **interseção** e **diferença** escalam de forma bastante eficiente, com tempos sempre inferiores a **650 ms** mesmo nos maiores ficheiros.
- Esta implementação apresenta um **comportamento consistente e otimizado**, sendo adequada para grandes volumes de dados sem necessidade de ajustes manuais.

Segunda Implementação (com AEDHashMap personalizada):

- Os tempos de carregamento são superiores, especialmente para volumes grandes: mais de 11 segundos para 1 milhão de pontos, refletindo os custos da gestão manual da tabela de dispersão e das colisões.
- As operações de **união**, **interseção** e **diferença** também demoram mais tempo que na implementação 1, sobretudo com datasets grandes.
- Apesar de ter bom desempenho até **10 000 pontos**, a estrutura personalizada mostra limitações de escalabilidade acima de **100 000 pontos**, com degradação significativa.
- A operação **interseção** foi a mais exigente, atingindo quase **2 segundos** no maior caso.

3.6 Conclusões da avaliação experimental

- Ambas as implementações apresentam um **desempenho eficiente e estável até aos 100 000 pontos**, sendo adequadas para ficheiros de dimensão média.
- A **primeira implementação**, baseada nas estruturas padrão da Kotlin (`HashMap`, `Set`), mostra **tempos consistentes, baixos e previsíveis**, com uma implementação mais simples e direta.
- A **segunda implementação**, com a estrutura personalizada `AEDHashMap`, requer **mais código e complexidade**, mas é eficaz em datasets pequenos e médios.
- No entanto, ao contrário do esperado, a **primeira implementação apresenta melhor desempenho** em **todas as operações**, especialmente com volumes elevados (1 milhão de pontos), devido às otimizações internas da biblioteca padrão.
- A **segunda implementação degrada rapidamente com volumes grandes**, sobretudo no tempo de carregamento e interseção, devido ao custo de inserções e à ausência de gestão dinâmica eficiente da tabela de dispersão.

4. Conclusões

A realização deste trabalho permitiu desenvolver uma aplicação funcional para operar sobre coleções de pontos no plano, abordando tanto a vertente prática da implementação como a análise comparativa de estruturas de dados. Foram exploradas duas abordagens distintas: uma baseada nas estruturas fornecidas pela Kotlin Standard Library e outra recorrendo a uma estrutura de dados personalizada desenvolvida no âmbito da disciplina.

A implementação com as coleções da biblioteca padrão destacou-se pela sua simplicidade, legibilidade e eficiência. O uso de estruturas como `HashMap` e `HashSet` facilitou a gestão dos dados e permitiu realizar as operações de união, interseção e diferença de forma concisa, aproveitando diretamente as funcionalidades nativas da linguagem.

Por outro lado, a abordagem personalizada exigiu uma construção mais detalhada e criteriosa, o que proporcionou um maior entendimento dos mecanismos internos de tabelas de dispersão, incluindo tratamento de colisões e organização dos dados. Embora mais trabalhosa, esta solução revelou-se fundamental para consolidar conceitos teóricos e compreender o impacto das decisões de implementação no desempenho do sistema.

A avaliação experimental permitiu comparar objetivamente o comportamento de ambas as soluções, evidenciando os seus pontos fortes e fracos em diferentes cenários. No geral, a utilização das coleções da biblioteca padrão é recomendada para aplicações práticas, onde o tempo de desenvolvimento e a clareza do código são prioritários. Já a implementação manual pode ser vantajosa em contextos de otimização específica ou quando se pretende total controlo sobre a estrutura de dados.

Este trabalho evidenciou, de forma clara, a relevância da escolha adequada das estruturas de dados em função do problema a resolver, reforçando a importância dos conhecimentos adquiridos na disciplina para o desenvolvimento de soluções eficientes e robustas.

Referências

[1] “Disciplina: Algoritmos e Estruturas de Dados - 2225SV,” Moodle 2024/25. [Online]. Available: https://2225.moodle.isel.pt

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