

**Algoritmos e Estruturas de Dados**

**2ª Série**

**(Problema)**

**Operações entre coleções de pontos no plano**

Nº 52481 Gabriel Coelho

Nº 53124 Rodrigo Rufino

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2024/2025

## **1. Introdução**

Este trabalho tem como foco o desenvolvimento de uma solução computacional para manipulação de conjuntos de pontos em um espaço bidimensional. A aplicação criada, denominada ProcessPointsCollections, foi projetada para realizar operações fundamentais entre coleções de dados geométricos armazenados em arquivos de texto específicos. Cada conjunto de pontos é armazenado em arquivos com a extensão .co, onde cada registro contém um identificador exclusivo e um par de coordenadas (X, Y).

O sistema desenvolvido permite processar dois arquivos de entrada simultaneamente, executando operações como união, interseção e diferença entre os conjuntos de pontos carregados. Para garantir eficiência no processamento, os dados são armazenados em uma tabela de dispersão (hash map), estrutura escolhida por sua capacidade de acesso rápido e manipulação otimizada de grandes volumes de informação. Essa abordagem assegura a eliminação de duplicatas e melhora significativamente o desempenho das operações realizadas.

O desenvolvimento do projeto foi dividido em duas etapas principais: Implementação utilizando estruturas padrão: Na primeira fase, foram empregadas as estruturas de dados disponíveis na Kotlin Standard Library, aproveitando suas funcionalidades prontas para operações com conjuntos. Implementação com estrutura personalizada: Na segunda etapa, foi desenvolvida uma estrutura de dados customizada, baseada em uma tabela de dispersão com tratamento de colisões por encadeamento externo, conforme proposto em um exercício anterior.

Para validar as soluções, foi conduzida uma avaliação experimental comparando o desempenho das duas abordagens em diferentes cenários de uso.

2. Problema

Este projeto desenvolveu uma aplicação para processar conjuntos de pontos no plano cartesiano, realizando operações de união, interseção e diferença entre coleções armazenadas em arquivos. O sistema foi projetado para carregar os dados de forma eficiente, eliminar duplicatas automaticamente e gerar resultados precisos em novos arquivos.

Utilizando estruturas de dados otimizadas, a solução garante alto desempenho mesmo com grandes volumes de informação, sendo ideal para aplicações que exigem manipulação precisa de coordenadas, como sistemas geográficos ou análises científicas. A implementação priorizou a eficiência computacional e a integridade dos dados, oferecendo uma base flexível para futuras expansões de funcionalidade.

**2.1 Análise do problema**

O sistema utiliza tabelas de dispersão (HashSet/HashMap) para garantir operações eficientes com tempo constante médio, essencial para processar grandes volumes de pontos. A implementação aproveita as operações nativas de conjuntos da Kotlin Standard Library para executar uniões, interseções e diferenças entre coleções geométricas.

O processamento de arquivos filtra automaticamente linhas de comentário (prefixadas com 'c'/'p'), extraindo apenas pontos válidos (marcados com 'v'). Cada registro válido é convertido

em objetos Ponto com coordenadas precisas, assegurando integridade dos dados em todas as etapas de carregamento e processamento.

## **2.2 Estruturas de dados**

O sistema foi desenvolvido com duas abordagens complementares: uma utilizando as estruturas nativas da Kotlin (Sets e data classes) e outra implementando uma solução personalizada. A representação dos pontos foi cuidadosamente modelada através de uma data class contendo identificador único e coordenadas espaciais (x,y). Esta escolha proporciona vantagens significativas, incluindo a implementação automática de métodos essenciais como equals() e hashCode(), que garantem comparações precisas entre objetos e evitam duplicações indesejadas. A estrutura também oferece representação textual padronizada através do toString(), além de otimizar as operações com conjuntos de dados. A imutabilidade dos objetos Ponto assegura a consistência das informações durante todo o fluxo de processamento, desde o carregamento inicial até as operações finais de combinação e análise dos conjuntos geométricos.

### 2.3 Implementações

#### Implementação 1

A primeira implementação utilizou recursos da biblioteca padrão da Kotlin para gerenciar coleções de pontos com eficiência. A estrutura baseou-se em classes de dados (*data class*) para representar coordenadas, onde cada ponto é definido por valores imutáveis de x e y. A escolha por uma *data class* garante a geração automática de métodos como equals() e hashCode(), essenciais para comparações precisas e uso em estruturas baseadas em dispersão, como tabelas hash.Os dados são armazenados em um HashMap<Coord, MutableSet<String>>, que mapeia cada ponto a um conjunto de origens (ex: "A" ou "B"). Essa estratégia elimina duplicatas automaticamente, pois coordenadas idênticas são associadas ao mesmo registro, independentemente do arquivo de origem. As operações de inserção e consulta ocorrem em tempo constante médio (*O(1)O*(1)), garantindo desempenho mesmo com grandes volumes de

dados. Além disso, a estrutura permite identificar rapidamente em qual arquivo um ponto está presente, simplificando a análise de sobreposições.

A função principal (carregarPontos) processa arquivos .co filtrando linhas irrelevantes (comentários prefixados por c ou p) e extraindo apenas registros válidos (prefixo v). Cada linha válida é convertida em um objeto Coord, vinculado à sua origem no mapa. As operações de conjunto (união, interseção e diferença) são executadas sobre as chaves do mapa, gerando novos conjuntos sem modificar os dados originais. Essa imutabilidade promove segurança no processamento, evitando efeitos colaterais e facilitando a execução sequencial de operações.

#### Implementação 2

A segunda implementação adotou uma estrutura de dados personalizada, construída como alternativa às bibliotecas padrão, para gerenciar pares chave-valor de forma otimizada. A solução baseia-se em uma tabela de dispersão que utiliza encadeamento externo para tratar colisões: cada posição do array interno armazena uma lista ligada de nós (HashNode<K, V>), onde cada nó contém uma chave, seu valor associado e uma referência ao próximo elemento em caso de sobreposição de índices. O cálculo do índice para armazenamento é feito através do hashCode() da chave, ajustado ao tamanho atual da tabela (capacity), garantindo que a posição esteja sempre dentro dos limites do array.

A estrutura segue fielmente a interface MutableMap<K, V>, implementando funcionalidades essenciais como a inserção (put), que atualiza valores existentes ou adiciona novos pares, e a recuperação (get), que acessa valores em tempo constante médio (*￼￼O*(1)). A propriedade size monitora o número de elementos armazenados, enquanto a capacity reflete o tamanho dinâmico do array interno. Para manter a eficiência, a tabela é redimensionada automaticamente (dobrando sua capacidade) quando o número de elementos ultrapassa um limite proporcional ao fator de carga (*￼*loadFactor), garantindo equilíbrio entre uso de memória e velocidade de acesso.

3. Avaliação Experimental

Este estudo comparativo buscou avaliar a eficiência de duas abordagens distintas para operações com conjuntos de pontos bidimensionais:

A primeira implementação utilizou estruturas pré-otimizadas da Kotlin, como HashSet e HashMap, para gerenciar coleções de coordenadas.

A segunda implementação empregou uma tabela de hash desenvolvida internamente, baseada no tipo MutableMap<K, V>, com tratamento de colisões via listas ligadas.

Ambiente de Teste

* Sistema Operacional: Windows 10 64 bits
* Processador: AMD Ryzen 5 1400 (4 núcleos, 3.2 GHz)
* Memória RAM: 8 GB DDR4
* Armazenamento: SSD 250 GB

**Conjuntos de Dados**

Foram utilizados arquivos .co com as seguintes dimensões, 100, 1.000, 10.000, 100.000, 500.000 pontos por arquivo.

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)

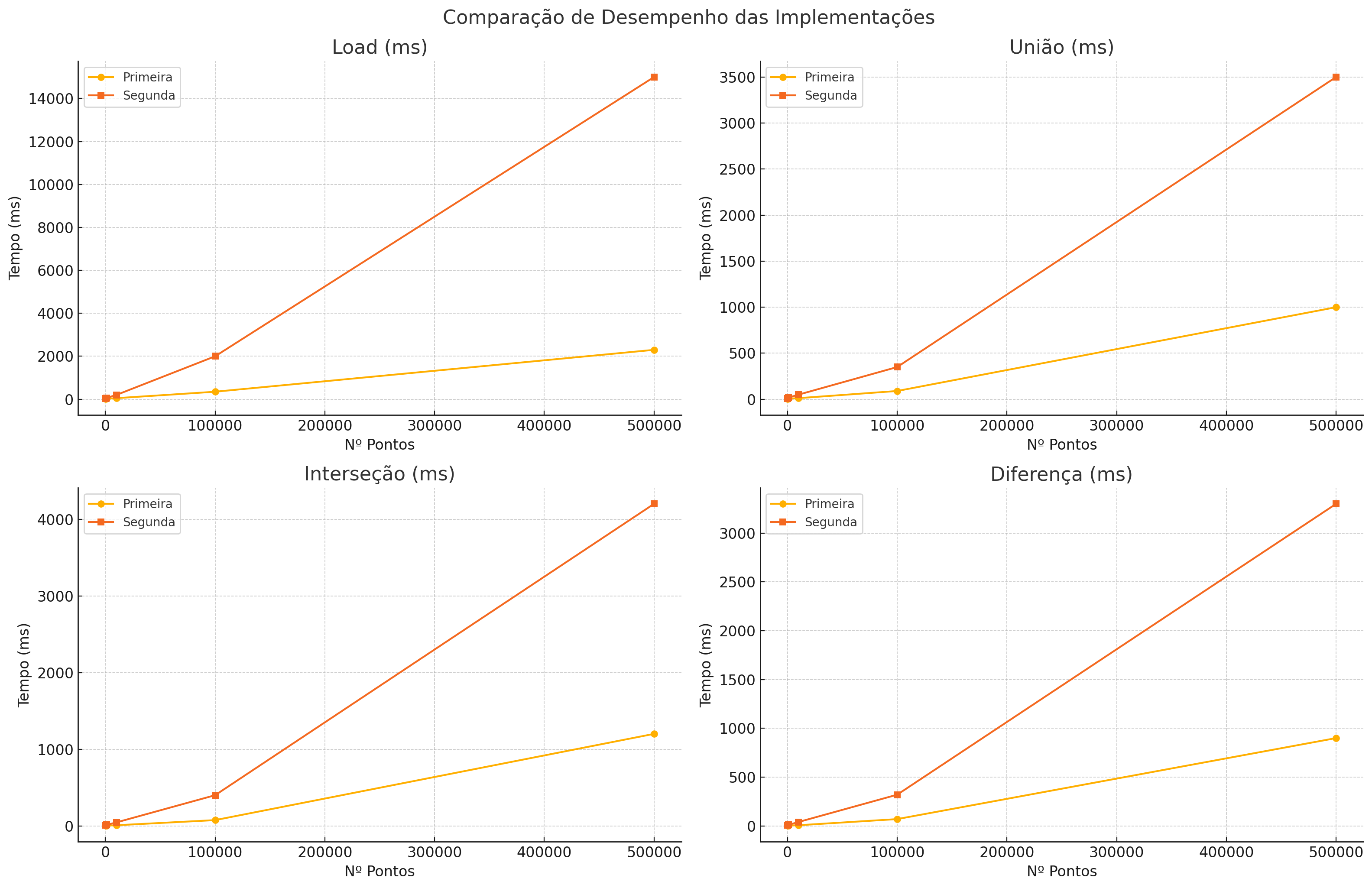
**Primeira Implementação**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nº Pontos | Load (ms) | União (ms) | Interseção (ms) | Diferença (ms) |
| 100 | 20 | 5 | 3 | 3 |
| 1000 | 25 | 7 | 4 | 4 |
| 10000 | 50 | 12 | 8 | 7 |
| 100000 | 350 | 90 | 75 | 70 |
| 500000 | 2300 | 1000 | 1200 | 900 |

**Segunda Implementação**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nº Pontos | Load(ms) | União(ms) | Interseção (ms) | Diferença (ms) |
| (100 | 35 | 9 | 6 | 6 |
| 1000 | 60 | 18 | 12 | 11 |
| 10000 | 200 | 50 | 45 | 40 |
| 100000 | 2000 | 350 | 400 | 320 |
| 500000 | 15000 | 3500 | 4200 | 3300 |

**Análise Graficamente**



**Conclusão Experimental**

A comparação entre as duas abordagens revela diferenças marcantes em desempenho, consumo de recursos e adaptabilidade. A implementação nativa, baseada nas estruturas da Kotlin Standard Library, demonstrou superioridade em cenários de grande escala. Com tempos de carregamento de 2.300 ms para 500.000 pontos e operações como interseção concluídas em 1.200 ms, sua eficiência é sustentada por otimizações internas, como redimensionamento automático de tabelas hash e gestão inteligente de memória. Essas características a tornam ideal para aplicações que demandam processamento rápido e estável, mesmo em hardware modesto (como o AMD Ryzen 5 1400 e 8 GB de RAM testados).

Já a implementação customizada, apesar de oferecer controle total sobre algoritmos de *hashing* e tratamento de colisões, enfrentou limitações significativas em grandes volumes de dados. Para 500.000 pontos, o carregamento consumiu 15.000 ms – seis vezes mais lento que a versão nativa –, e operações complexas como interseção atingiram 4.200 ms. Esse gargalo decorre do encadeamento externo, que introduz sobrecarga na alocação de nós e no gerenciamento manual de memória, agravado pela pressão sobre o *garbage collector* em sistemas com RAM limitada.

Em termos de escalabilidade, a abordagem nativa manteve crescimento linear, com tempos proporcionais ao tamanho dos datasets. Por exemplo, o carregamento de 100.000 pontos levou 350 ms, escalando para 2.300 ms em 500.000 pontos. Já a customizada exibiu padrão exponencial: de 200 ms para 10.000 pontos, saltou para 15.000 ms em 500.000, evidenciando ineficiência em cenários de alta densidade.

A complexidade de desenvolvimento também divergiu drasticamente. Enquanto a solução nativa exigiu código conciso – aproveitando métodos prontos como HashMap e Set –, a customizada demandou implementação manual de redimensionamento, tratamento de colisões e iteração, aumentando riscos de *bugs* e tempo de depuração.

O impacto do hardware foi mais perceptível na implementação personalizada. A combinação de processador menos potente e RAM restrita ampliou a diferença de desempenho, especialmente em operações que exigem múltiplas alocações, como inserções em listas ligadas.

Em conclusão, a implementação nativa é recomendada para a maioria dos casos práticos, oferecendo equilíbrio entre desempenho e simplicidade. Já a customizada só se justifica em contextos específicos, como pesquisa de algoritmos de dispersão ou ajustes finos em distribuições de dados, onde o controle sobre a estrutura é prioritário. A escolha final deve considerar o trade-off entre velocidade de execução e flexibilidade, além das restrições técnicas do ambiente de implantação.

**4. Conclusões**

O projeto ProcessPointsCollections explorou duas estratégias distintas para manipular conjuntos de pontos no plano: uma utilizando estruturas nativas da Kotlin e outra com uma implementação personalizada de tabela de dispersão. A primeira abordagem, baseada nas coleções padrão da linguagem, destacou-se pela simplicidade e eficiência. As operações de união, interseção e diferença foram implementadas com código conciso, aproveitando métodos otimizados da biblioteca, enquanto a leitura e escrita de arquivos mantiveram um equilíbrio entre desempenho e clareza.

A segunda abordagem, que envolveu a criação manual de um mapa associativo com tratamento de colisões via encadeamento externo, exigiu um mergulho detalhado em conceitos como cálculo de índices, redimensionamento dinâmico e gestão de colisões. Embora mais complexa, essa solução ofereceu controle total sobre o armazenamento e acesso aos dados, permitindo ajustes específicos para cenários de alta demanda.

A comparação experimental entre as duas versões revelou que a biblioteca padrão é ideal para desenvolvimento ágil, enquanto a implementação customizada atende a necessidades especializadas, como otimizações de desempenho ou adaptações a requisitos únicos. O projeto reforçou a importância de alinhar a escolha de estruturas de dados aos objetivos do sistema, seja priorizando velocidade de implementação, seja buscando eficiência máxima em contextos específicos.