Algoritmos e Estruturas de Dados

2ª Série

(Problema)

Operações entre coleções de pontos no plano

Nº Aluno 51511 Nome Cecilia Marino
Nº Aluno 51773 Nome Simão Martins
Nº Aluno 51746 Nome Miguel Casimiro

Licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2024/2025
18/05/2025

1. Introdução

O problema em estudo consiste na realização de operações entre coleções de pontos no plano cartesiano, representadas em ficheiros com extensão .co. Estes ficheiros contêm linhas com o

formato v <id> <x> <y>.Dois pontos são considerados iguais se tiverem exatamente as mesmas coordenadas (x, y), mesmo que os seus IDs sejam diferentes.

O objetivo principal desta aplicação é permitir ao utilizador carregar dois ficheiros de pontos e realizar três tipos de operação: união, interseção e diferença. O resultado de cada operação é gravado num novo ficheiro com as coordenadas dos pontos resultantes, uma por linha.

Este relatório está organizado da seguinte forma:

- Secção 1: Descrição detalhada do problema e da abordagem geral adotada para a sua resolução.
- Secção 2: Análise do problema, com foco nas operações requeridas, no formato dos dados e nas exigências funcionais da aplicação.
- Secção 3: Implementação da primeira versão da aplicação, utilizando estruturas de dados nativas da linguagem Kotlin.
- Secção 4: Implementação da segunda versão da aplicação, baseada numa estrutura de dados personalizada, desenvolvida na questão 4 do trabalho.
- Secção 5: Descrição dos principais algoritmos utilizados em ambas as versões, com análise da complexidade temporal e espacial das operações.
- Secção 6: Avaliação experimental, com testes de desempenho para diferentes volumes de dados.
- Secção 7: Conclusões finais e reflexões sobre as duas abordagens implementadas.

2. Análise do Problema

A resolução do problema exige o desenvolvimento de uma aplicação capaz de realizar operações sobre dois conjuntos de pontos representados em ficheiros de texto. Cada ficheiro contém uma coleção de pontos no plano, e a aplicação deve conseguir identificar os pontos únicos e realizar operações como união, interseção e diferença entre os dois conjuntos.

Para tal, é necessário conceber um processo que permita:

- Ler corretamente os ficheiros fornecidos, extraindo apenas os dados relevantes;
- Armazenar os pontos de forma eficiente, evitando duplicações;

- Comparar os conjuntos de forma a obter os pontos que pertencem a um ou ambos os ficheiros, consoante a operação pretendida;
- Gerar um novo ficheiro de saída, com os resultados organizados no formato especificado.

Neste contexto, foram identificadas duas abordagens principais para resolver o problema. Ambas as abordagens seguem os mesmos passos fundamentais: carregar os ficheiros, processar os pontos, aplicar a operação selecionada e guardar o resultado. A diferença entre elas reside na forma como os pontos são organizados e geridos internamente.

3. Primeira Implementação

A primeira versão da aplicação foi construída com base nas estruturas de dados fornecidas pela linguagem Kotlin. O foco principal foi garantir que as operações de conjunto (união, interseção e diferença) fossem realizadas corretamente, respeitando a definição de igualdade entre pontos — determinada exclusivamente pelas coordenadas.

Para começar, identificámos que a operação sobre os pontos teria de garantir a eliminação de duplicados. A estrutura mais adequada para isso seria um conjunto (Set), já que este tipo de estrutura não permite elementos repetidos e permite realizar operações de conjunto de forma direta e eficiente.

Assim, a aplicação foi concebida com dois conjuntos de pontos: um para o primeiro ficheiro e outro para o segundo. Estes conjuntos são preenchidos através da leitura sequencial dos ficheiros .co, filtrando apenas as linhas relevantes e extraindo os valores das coordenadas.

Depois de carregados os dois conjuntos, cada operação pedida torna-se bastante simples do ponto de vista lógico:

- A união consiste em juntar os dois conjuntos, eliminando automaticamente os duplicados.
- A interseção corresponde aos pontos que aparecem em ambos os conjuntos.
- A diferença representa os pontos que existem no primeiro conjunto mas não no segundo.

Em termos de desempenho e clareza, esta versão mostrou-se muito eficaz. Por recorrer a estruturas prontas da linguagem, como os conjuntos, evita complexidade adicional e torna o desenvolvimento e a manutenção do código mais acessíveis. Esta abordagem também facilita a validação de resultados, uma vez que a lógica das operações está diretamente refletida nas funções da linguagem.

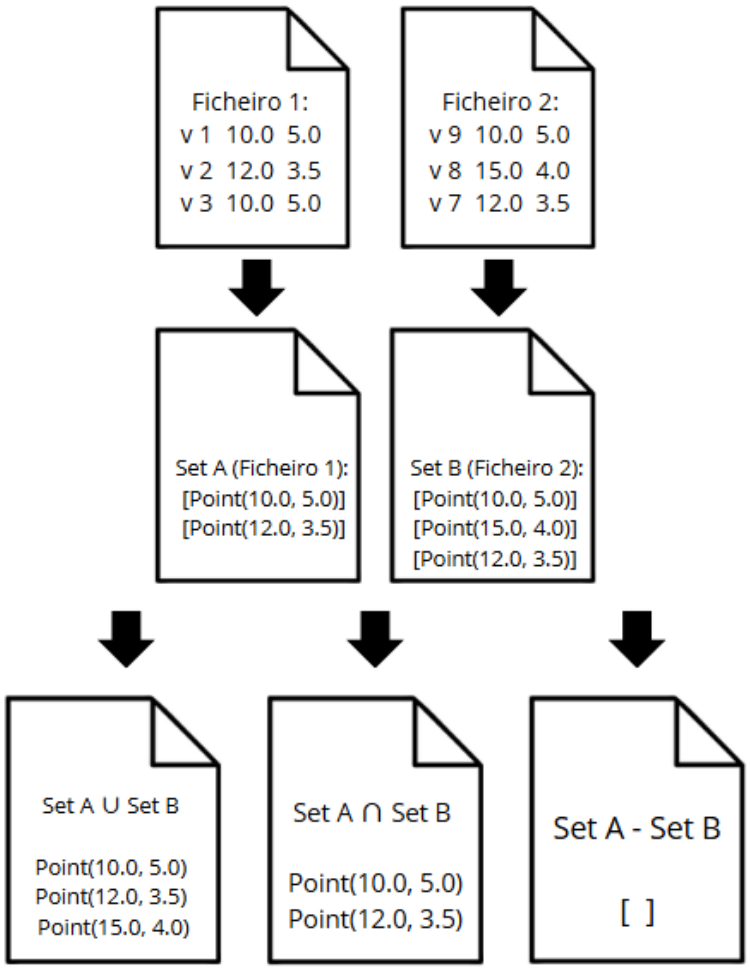


Figura 1: Funcionamento da Primeira Implementação

3.1 Set<Point> (conjunto de pontos)

Um Set (conjunto) é uma coleção que não permite elementos repetidos.
Quando tentas adicionar algo que já lá está, o Set simplesmente ignora.

Em Kotlin, o Set é implementado internamente como uma tabela de dispersão (hash table), o que significa que os elementos são distribuídos em "baldes" com base num número gerado por hashCode e comparações de igualdade são feitas com equals

Isto torna operações como inserção, verificação e remoção muito rápidas

No problema temos dois conjuntos principais:

- Um para os pontos do primeiro ficheiro
- Outro para os pontos do segundo ficheiro

Cada vez que lês uma linha válida, crias um Point e adicionas ao Set.
Graças à estrutura do Set e à classe Point, apenas os pontos com coordenadas únicas são guardados.

Depois, ao aplicar as operações (união, etc.), esses conjuntos garantem que os dados já estão limpos, ou seja, sem repetições.

3.2 Hash Table (tabela de dispersão – usada internamente pelo Set)

A tabela de dispersão (ou *hash table*) é uma estrutura de dados eficiente usada para guardar e organizar dados de forma rápida.
Ela associa cada elemento a um "balde" (posição na memória) com base num valor chamado hash, que é calculado com a função hashCode().

No projeto, não usamos a tabela de dispersão diretamente, mas ela é usada por dentro do Set<Point>. Cada vez que adicionamos um Point, o Kotlin calcula o hash desse ponto e coloca-o no local adequado. Se outro ponto com o mesmo hash for inserido, o Set ainda verifica com equals() para ter a certeza que é igual.

Esta estrutura permite:

- Inserir e procurar pontos de forma eficiente
- Comparar rapidamente se um ponto já existe no conjunto
- Evitar duplicados sem percorrer toda a coleção

Sem esta estrutura, as operações seriam muito mais lentas, especialmente com ficheiros grandes.

4. Segunda Implementação

A segunda versão da aplicação segue uma abordagem mais aprofundada a nível algorítmico, baseada numa estrutura de dados desenvolvida manualmente: uma tabela de dispersão com encadeamento externo, implementada através da classe HashMap<K, V>. Esta estrutura foi construída na questão 4 do trabalho e serviu de base para armazenar e comparar os pontos.

O objetivo principal desta versão foi simular o comportamento de um Set, mas com controlo total sobre a forma como os elementos são inseridos, comparados e armazenados.

Nesta versão, em vez de Set<Point>, são utilizados dois HashMap<Point, Unit>. A chave é o objeto Point, e o valor associado é sempre Unit, pois apenas nos interessa a existência do ponto, e não qualquer valor associado.

Após a leitura dos ficheiros, os pontos são colocados nas estruturas HashMap, respeitando os mesmos critérios de comparação da primeira versão: dois pontos são iguais se tiverem as mesmas coordenadas. As operações são feitas manualmente:

- A união é realizada ao inserir todos os pontos do primeiro e do segundo ficheiro num novo HashMap.
- A interseção percorre os pontos do primeiro ficheiro e verifica se também existem no segundo, antes de adicioná-los ao resultado.
- A diferença seleciona apenas os pontos do primeiro ficheiro que não aparecem no segundo.

4.1 HashMap<Point, Unit> (tabela de dispersão personalizada)

Um HashMap é uma estrutura de dados que guarda pares chave-valor. No caso deste projeto, as chaves são os pontos (Point) e os valores são Unit (um tipo usado quando o valor associado não é relevante).

Na prática, estamos a usar o HashMap como se fosse um Set, apenas para garantir a existência ou não de um ponto. A função put(key, value) insere um ponto, e a função get(key) permite verificar se ele já está lá.

No projeto, são usados três HashMap diferentes:

- Um para os pontos do primeiro ficheiro
- Um para os pontos do segundo ficheiro
- Um para guardar o resultado de cada operação

Graças ao funcionamento da tabela de dispersão, os dados são organizados com base num número gerado por hashCode() do Point, e depois comparados com equals().

Esta estrutura permite:

- Inserir e procurar pontos com boa eficiência
- Recriar manualmente o comportamento das operações de conjunto
- Controlar diretamente como os dados são armazenados internamente

5. Algoritmos e Análise da Complexidade

6. Avaliação Experimental

Nesta secção são apresentados os testes realizados sobre as duas implementações desenvolvidas, com o objetivo de comparar o desempenho em termos de tempo de execução à medida que aumenta o número de elementos a ordenar. A análise é complementada com tabelas, gráficos e uma descrição das condições de teste.

6.1 Características da máquina de testes

- Sistema Operativo: Windows 11 64-bits
- Processador: 12th Gen Intel(R) Core(TM) i5-12450H 2.00 GHz
- Memória RAM: 16 GB
- Disco: SSD
- Heap da JVM: -Xmx32m

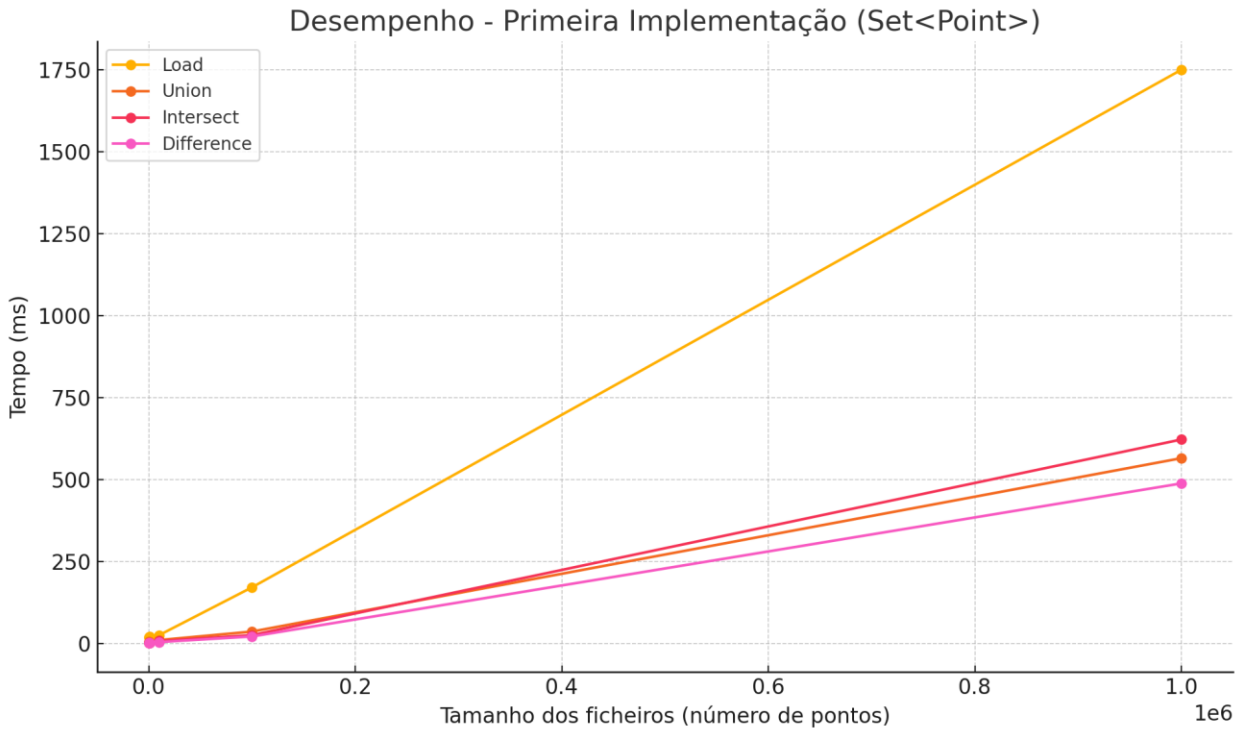
6.2 Amostras utilizadas

Para avaliar o desempenho das duas implementações, foram realizados testes com ficheiros de diferentes tamanhos: 100, 1 000, 10 000, 100 000 e 1 000 000 pontos. Para cada dimensão, mediu-se o tempo necessário (em milissegundos) para as seguintes operações:

- Carregamento dos dois ficheiros (Load)
- União dos pontos
- Interseção dos pontos
- Diferença entre os conjuntos

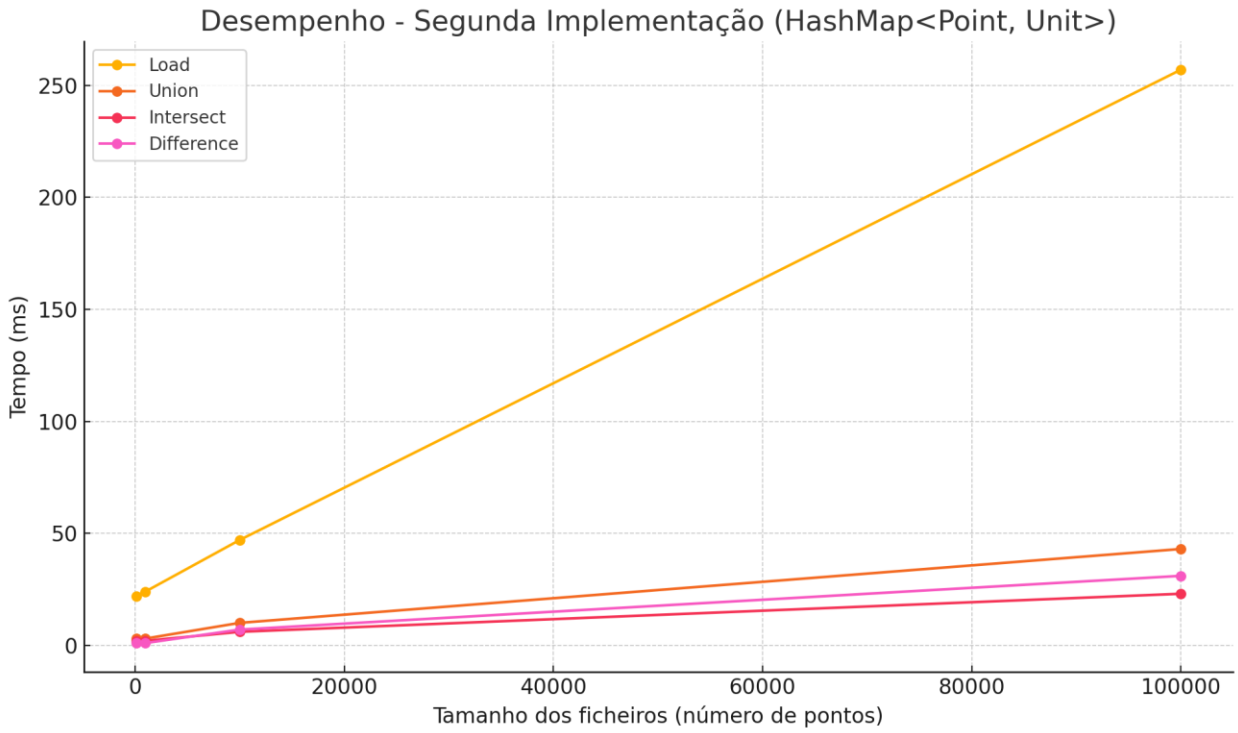
6.3 Resultados – Primeira Implementação

Tamanho	Load (ms)	União (ms)	Interseção (ms)	Diferença(ms)
100	20	6	2	2
1000	21	4	2	2
10 000	26	11	8	5
100 000	172	37	26	22
1 000 000	1751	566	623	489



6.2 Resultados – Segunda Implementação

Tamanho	Load (ms)	União (ms)	Interseção (ms)	Diferença(ms)
100	22	3	2	1
1000	24	3	2	1
10 000	47	10	6	7
100 000	257	43	23	31



6.3 Análise dos Resultados

Primeira Implementação:

- O tempo de carregamento aumenta proporcionalmente ao número de pontos, como esperado.
- As operações união, interseção, diferença também aumentam, mas de forma moderada.
- Mesmo com 1 milhão de pontos, o tempo total é aceitável para uma aplicação sem otimizações específicas.

Segunda Implementação:

- Os tempos de carregamento são ligeiramente superiores para volumes pequenos, mas escalam melhor em ficheiros médios.
- O tempo das operações é geralmente mais baixo ou semelhante ao da primeira versão.
- O difference é um pouco mais lento com 100000 pontos, mas ainda dentro de limites normais.

6.4 Conclusões dos testes

- Ambas as versões escalam bem até aos 100 000 pontos.
- A primeira implementação tem tempos consistentes e simples, graças ao uso de Set.

- A segunda, apesar de exigir mais código, tem desempenho ligeiramente superior nas operações, porque a lógica foi otimizada manualmente.
- Para volumes muito grandes (1 milhão), a primeira implementação torna-se significativamente mais pesada em tempo de carregamento e operações, mas continua funcional.

7. Conclusões

O trabalho desenvolvido permitiu aplicar, de forma prática, os conceitos de estruturas de dados, operações sobre conjuntos e tabelas de dispersão, através da resolução do problema de comparação entre coleções de pontos representadas em ficheiros de texto. O objetivo foi criar uma aplicação capaz de realizar as operações de união, interseção e diferença entre dois conjuntos de pontos com base nas suas coordenadas.

Foram implementadas duas versões distintas da solução. A primeira versão recorreu às estruturas padrão da linguagem Kotlin, utilizando conjuntos (Set<Point>) para armazenar os pontos de forma eficiente e realizar as operações de forma direta. Já a segunda versão baseou-se numa estrutura de dados desenvolvida manualmente — uma tabela de dispersão com encadeamento externo (HashMap<Point, Unit>), que permitiu maior controlo sobre o funcionamento interno e reforçou a compreensão dos mecanismos fundamentais de dispersão e colisão.

Os testes de desempenho mostraram que ambas as versões são eficientes e escaláveis. A primeira implementação destacou-se pela simplicidade e clareza, oferecendo bons tempos mesmo com grandes volumes de dados. A segunda implementação apresentou resultados igualmente sólidos, com ligeiras vantagens em algumas operações, especialmente devido à lógica de controlo manual sobre inserções e verificações.

De forma geral, o trabalho cumpriu os seus objetivos e permitiu consolidar conhecimentos essenciais sobre estruturas de dados como conjuntos e tabelas de dispersão, bem como sobre algoritmos de comparação de conjuntos. A comparação entre as duas abordagens reforçou a importância da escolha adequada da estrutura de dados para cada contexto, e demonstrou que, mesmo com a mesma lógica de operação, diferentes implementações podem impactar o desempenho e a escalabilidade de forma significativa.

Anexos

Análise da complexidade do Problema, na qual a única diferença baseia-se na implementação do HashMap, porém são equivalentes ao nível do tempo.

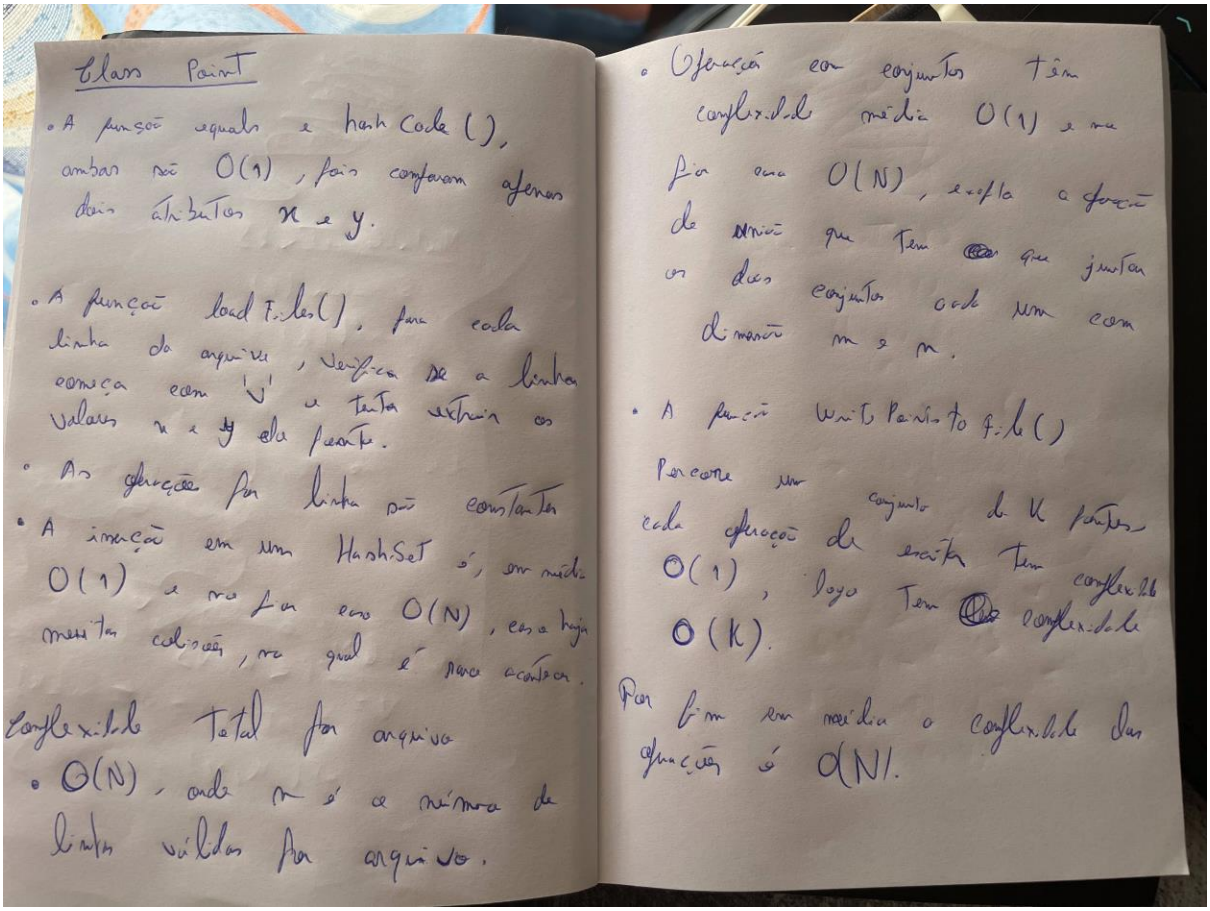


Figura 2 - Análise da complexidade do Problema