Algoritmos e Estruturas de Dados

2ª Série

OPERAÇÕES ENTRE COLEÇÕES DE PONTOS NO PLANO

Trabalho realizado por:

52559 - Isadora Mendes

52848 - Inês Gil

52870 - Martim Garcia

Turma: LEIC22D
Docente: Maria Paula Graça

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

18/04/2025

Conteúdo

Índice **i**

1 Introdução **1**

2 Estrutura de Dados **2**

2.1 Análise Do Problema 2
2.2 Estrutura de Dados Utilizadas na Implementação 1 3
2.3 Estrutura de Dados Utilizadas na Implementação 2 3
2.4 Outras Estruturas Auxiliares 4
2.5 Algoritmos e Análise da Complexidade 4
2.5.1 Implementação 1 4
2.5.2 Implementação 2 5
2.6 Implementação 1 6
2.7 Implementação 2 6
2.8 Comparação de implementações 7

3 Avaliação Experimental **7**

4 Conclusões **9**

5 Referências **10**

6 Anexos **10**

1 Introdução

Pretende-se desenvolver uma aplicação capaz de realizar operações entre conjuntos de pontos no plano, nomeadamente operações de união, interseção e diferença. Estes conjuntos de pontos estão presentes nos ficheiros e produzem ficheiros de saída correspondentes.

O problema principal consistiu na leitura, comparação e manipulação de dois ficheiros contendo pontos em duas dimensões, onde cada ponto é composto por um identificador único e duas coordenadas (x, y).

O relatório foi dividido em diferentes partes:

- Primeira parte: Nesta fase descrevemos as estruturas de dados utilizadas;
- Segunda parte: Nesta fase analisamos os algoritmos desenvolvidos;
- Terceira parte: Nesta fase apresentamos as implementações;
- Quarta parte: Nesta fase realizamos uma avaliação experimental e comparamos os resultados obtidos;
- Quinta parte: Por fim, nesta fase fazemos as discussões todas dos nossos resultados obtidos e quais são as melhores técnicas que poderíamos usar.

2 Estrutura de Dados

Nesta secção do relatório serão especificadas todas as estruturas de dados e todos os algoritmos utilizados no desenvolvimento da nossa série.

2.1 Análise Do Problema

O problema consiste na comparação de dois conjuntos de pontos, representados por ficheiros de texto, onde cada linha contém um ponto identificador e duas coordenadas inteiras (x, y).

Pretende-se identificar:

- Todos os pontos existentes em qualquer um dos ficheiros - união;
- Apenas os pontos que aparecem nos dois ficheiros - interseção;
- Pontos que existem apenas no primeiro ficheiro - diferença.

Os ficheiros podem conter duplicações, ordens diferentes e tamanhos distintos. A aplicação deve processar essas informações de forma eficiente, assegurando que cada ponto é identificado de forma única.

Foi pedido duas implementações para realizar este problema, a primeira implementação consiste em estruturas presentes na Kotlin Standard Library (MutableList()) que considere necessárias, e a segunda consiste em utilizar estrutura de dados implementada na questão 1.4 (HashMap()).

A aplicação funciona através de uma interface de linha de comandos (no IntelliJ), onde o utilizador pode carregar ficheiros e executar comandos que realizam operações de conjunto, exportando os resultados para ficheiros de saída.

2.2 Estrutura de Dados Utilizadas na Implementação 1

Nesta implementação, utilizámos a estrutura de dados *MutableMap()* para armazenar a presença dos pontos de ambos os ficheiros.

A estrutura principal utilizada foi o *map*. Este *map* associa a cada ponto um par de valores booleanos. O primeiro valor indica se o ponto está presente no primeiro ficheiro e o segundo se está presente no segundo.

Esta representação permite realizar as operações perdidas (*union()*, *intersection()*, *difference()*) com base na combinação desses dois valores.

A estrutura *MutableMap()* foi escolhida por ser uma estrutura eficiente, que permite introduções, atualizações e pesquisas em tempo constante na maioria dos casos.

Além disso, o uso de *data class()* permite que os pontos sejam corretamente utilizados como chave do mapa, graças à implementação automática de *equals()* e *hashCode()* baseada nos atributos.

2.3 Estrutura de Dados Utilizadas na Implementação 2

Na segunda implementação, substituímos o *MutableMap()* por uma estrutura *HashMap()* implementada por nós. A nossa *HashMap()* inclui funcionalidades como:

- Inserção (put()) - insere ou substitui um par chave-valor;
- Acesso (get()) - procura uma chave e devolve o valor;
- Remoção (remove()) - elimina uma entrada;
- Expansão automática (duplicação da tabela quando excede o fator carga);
- Iterador sobre todas as entradas (iterator()) - devolve todos os pares existentes.

O *HashMap()* têm um menor desempenho em ficheiros grandes devido à ausência de otimizações internas, mas suficiente para demostrar funcionalidade e validade da estrutura.

2.4 Outras Estruturas Auxiliares

Em ambas as versões, são utilizadas listas (*MutableList()*) apenas para:

- Construir os conjuntos de saída antes de os escrever em ficheiros;
- Temporariamente armazenar os resultados após filtragem do mapa.

2.5 Algoritmos e Análise da Complexidade

Nesta secção descrevemos todos os algoritmos utilizados na implementação do nosso problema e analisámos as suas respetivas complexidades (em termos de tempo e de espaço).

2.5.1 Implementação 1

A aplicação, baseada em *MutableMap()*, segue uma lógica simples.

Algoritmo de carregamento dos ficheiros:

1. Para cada linha de cada ficheiro:
   - Interpretar o ponto (id, x, y);
   - Inserir ou atualizar a entrada correspondente no mapa;
   - Atualizar os valores booleanos (true/false) consoante o ficheiro de origem.

Operações

- union() - filtrar o mapa com condição first||second;
- intersection() - first and second;

- difference - first and !second.

Complexidade

No carregamento de ficheiros a complexidade é de O(n), sendo n o número total de pontos lidos. Nas operações a complexidade é de O(n), pois percorre o mapa uma vez para aplicar o filtro. Já a escrita do ficheiro, a complexidade é mais uma vez O(n), sendo n, o número de pontos no resultado.

2.5.2 Implementação 2

Na segunda implementação, os algoritmos seguem a mesma lógica funcional, mas operam sobre a nossa estrutura *HashMap()*

Algoritmo de carregamento dos ficheiros:

- Abrir os ficheiros;
- Ler cada ponto e calcular o índice hash;
- Inserir ou atualizar os valores booleanos no *HashMap()*.

Operações

- Filtragem dos pontos feita com *entries.filter condição*;
- Utiliza o nosso iterador para varrer todas as entradas na tabela.

Complexidade

No carregamento de ficheiros e leitura a complexidade é de O(1), no melhor caso e O(n) no pior caso com muitas colisões. Nas operações a complexidade é de O(n), como na implementação anterior. Já a escrita do ficheiro, a complexidade é mais uma vez O(n), sendo n, o número de pontos no resultado.

2.6 Implementação 1

Nesta abordagem, foi utilizado o *MutableMap()* para armazenar os pontos. Cada ponto é representado por uma instância da data class *Point*, usada como chave no mapa. O valor associado à chave é um *Pair<Boolean, Boolean>*, que indica se o ponto pertence ao primeiro ficheiro, ao segundo, ou a ambos.

Comandos disponíveis:

- *load ficheiro1.co ficheiro2.co*: lê ambos os ficheiros e constrói o mapa;
- *union ficheiroSaida.co*: escreve todos os pontos que pertencem a qualquer um dos ficheiros;
- *intersection ficheiroSaida.co*: escreve apenas os pontos comuns a ambos os ficheiros;
- *difference ficheiroSaida.co*: escreve os pontos presentes apenas no primeiro ficheiro;
- *exit*: termina a aplicação.

Funcionamento interno:

- Cada linha de cada ficheiro é lida e interpretada;
- Os pontos válidos são inseridos ou atualizados no mapa;
- As operações são implementadas como simples filtros sobre os pares armazenados no mapa.

2.7 Implementação 2

Na segunda abordagem, substituímos a estrutura *MutableMap* pela *HashMap<K, V>*. A interface e o comportamento do problema mantêm-se idênticos, com a principal diferença a nível da estrutura de dados utiliza internamente.

Alterações na estrutura:

- Substituição da data class *Point* por *Ponto*, funcionalmente equivalente;
- Uso da estrutura *HashMap<Ponto, Pair<Boolean, Boolean»*, com lógica de encadeamento para gestão de colisões.

2.8 Comparação de implementações

Critério	**Implementação 1**	**Implementação 2**
Facilidade de uso	Grande	Média
Performance esperada	Alta	Boa (depende da dispersão)
Complexidade do código	Baixa	Elevada
Controlo interno	Limitado	Total

Tabela 1: Comparação entre a 1º Implementação e a 2º Implementação

3 Avaliação Experimental

Nesta secção avaliamos experimentalmente a nossa implementação. Para isso, necessitamos de fazer testes com diferentes valores na quantidade de pontos no plano e registámos os tempos de execução dos algoritmos. O objetivo foi verificar a eficiência de ambas as abordagens, especialmente em operações de leitura, escrita e filtragem de dados.

Estes testes foram realizados numa máquina com processador 13th Gen Intel(R) Core (TM) i7-1355U, 1700 Mhz e 16 GB de memória RAM.

Na Figura 1 e na Figura 2 podemos ver as tabelas das duas implementações e os seus respetivos tempos de execução (em segundos).

Tempos Médios	F1 e F1r	F2 e F2r	F3 e F3r	F4 e F4r	F7x e F8x	F4r e F6r
Load	1.4 s	1.7 s	2.7 s	5.9 s	446.5 ms	32.8 s
Union	64.7 ms	59.3 ms	108.0 ms	206.3 ms	163.1 ms	3.5 s
Intersection	90.2 ms	89.4 ms	190.2 ms	458.8 ms	11.7 ms	150.0 ms
Difference	3.5 ms	2.9 ms	6.6 ms	10.7 ms	122.8 ms	504.6 ms
Média	1.6 s	1.8 s	3.0 s	6.6 s	744.1 ms	36.9 s

Figura 1: Tempos de execução da primeira implementação

Tempos Médios	F1 e F1r	F2 e F2r	F3 e F3r	F4 e F4r	F7x e F8x	F4r e F6r
Load	1.3 s	1.7 s	2.5 s	5.0 s	513.6 ms	35.5 s
Union	111.3 ms	138.0 ms	180.5 ms	356.4 ms	318.9 ms	6.6 s
Intersection	128.2 ms	120.7 ms	192.9 ms	404.7 ms	19.4 ms	333.9 ms
Difference	12.8 ms	13.4 ms	10.1 ms	19.8 ms	114.9 ms	615.8 ms
Média	1.6 s	1.9 s	2.9 s	5.8 s	966.9 ms	43.0 s

Figura 2: Tempos de execução da segunda implementação

Podemos também ver o gráfico destes resultados na Figura 3.

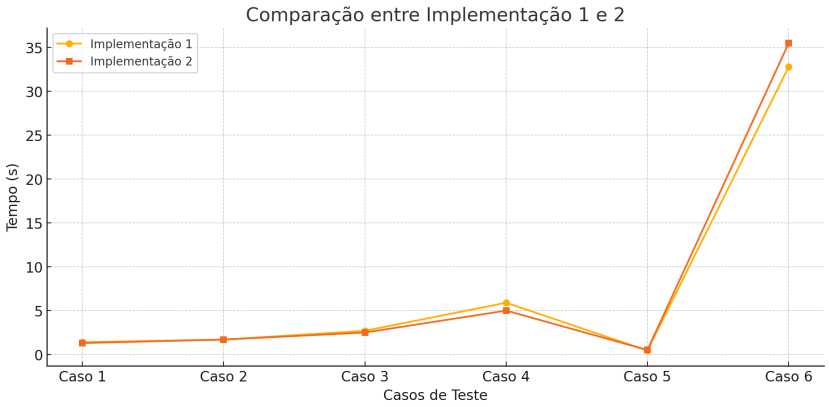

Figura 3: Comparação dos tempos de execução das duas implementações

4 Conclusões

A segunda série permitiu-nos aprofundar significativamente os nossos conhecimentos sobre estruturas de dados fundamentais, explorando não apenas a sua utilização, mas também a sua implementação manual e aplicação prática em problemas reais.

Durante o desenvolvimento do trabalho, implementámos e testámos:

- Estruturas como Max-Heap em array, listas duplamente ligadas circulares, filas circulares com offset e um HashMap genérico com encadeamento externo;
- Algoritmos com foco em eficiência e baixo consumo de memória;
- Uma aplicação interativa robusta, com suporte a comandos de união, interseção e diferença de pontos entre dois ficheiros.

A comparação entre a versão com MutableMap do Kotlin e a versão com HashMap manual revelou que, embora a estrutura nativa ofereça melhor desempenho, a implementação personalizada apresentou-se estável e funcional. Essa nova implementação foi essencial para compreendermos os detalhes internos de uma tabela de dispersão, como a gestão de colisões, expansão da tabela e implementação de iteradores.

Como pontos positivos, destacamos:

- A oportunidade de integrar várias estruturas distintas num único projeto;
- A construção de uma aplicação interativa com interface simples e funcionalidades úteis;
- A validação prática de conceitos teóricos como complexidade algorítmica e gestão eficiente de memória.

5 Referências

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

6 Anexos

Grupo I: Está presente no GitHub.

Grupo II: Ficheiro txt.