Licenciatura em Engenharia Informática e de Computadores

Relatório do Trabalho

2ª Série

Operações Entre Coleções de Pontos no Plano

Trabalho realizado por:

A52767 Oleksandra Zaiets

A52562 Rafaela Pereira

A52691 Whitney Cassiala

Turma: LEIC22D
Docente: Maria Paula Graça

Algoritmos e Estruturas de Dados
2024 / 2025 semestre de verão

Conteúdo

A Introdução

O objetivo desta série é realizar um conjunto de exercícios e resolver um problema prático visando consolidar os conhecimentos adquiridos na unidade curricular de Algoritmos e Estruturas de Dados, através da implementação de soluções eficientes na linguagem Kotlin. Ao longo dos desafios propostos, será necessário aplicar conceitos fundamentais como heaps, listas ligadas, tabelas de dispersão (hash tables) e manipulação de estruturas de dados abstratas.

A primeira parte do enunciado foca-se na implementação de funções específicas com complexidade e desempenho otimizados, sem recorrer às bibliotecas utilitárias fornecidas por java.util ou kotlin.collections, excetuando-se a fase inicial da resolução. A segunda parte explora a aplicação prática desses conceitos na criação de uma aplicação que realiza operações entre coleções de pontos no plano, representadas em ficheiros com formato próprio.

Todas as implementações seguiram princípios de eficiência e reutilização de estruturas de dados, de modo a obter soluções que minimizem o tempo e o espaço computacional. Fizemos também as avaliações experimentais, com base em ficheiros de teste fornecidos, o que permitiu comparar o desempenho das soluções propostas, incentivando uma abordagem crítica e analítica ao desenvolvimento algorítmico.

B Desenvolvimento

B.1 Análise do Problema

O problema proposto consiste no desenvolvimento da aplicação *ProcessPointsCollections*, que tem como objetivo realizar operações sobre coleções de pontos no plano a partir de dois ficheiros de entrada com extensão ".co". Cada ponto é identificado por um identificador único e por duas coordenadas (X, Y). A aplicação deverá ser capaz de processar estes dados e executar um conjunto de operações, garantindo simultaneamente a eficiência no acesso e manipulação das estruturas de dados envolvidas.

As principais funcionalidades da aplicação incluem:

Carregamento de dados:

- Comando: *load document1.co document2.co*
- Deve ler os pontos presentes em ambos os ficheiros e armazená-los numa estrutura de dados eficiente (hash map) que permita acesso e pesquisa rápidos.

Operações sobre conjuntos de pontos:

- *union output.co*: produz um ficheiro com todos os pontos presentes em pelo menos um dos ficheiros de entrada, sem repetições.
- *intersection output.co*: produz um ficheiro com os pontos que ocorrem em ambos os ficheiros, sem repetições.
- *difference output.co*: produz um ficheiro com os pontos que estão em document1.co mas não em document2.co, também sem repetições.

Além das funcionalidades acima, a aplicação deverá respeitar os seguintes requisitos:

Os ficheiros de entrada podem conter três tipos de linhas:

- Linhas de ponto (iniciadas por 'v'), que devem ser processadas.
- Linhas de comentário (iniciadas por 'c') e de problema (iniciadas por 'p'), que devem ser ignoradas.

Os pontos devem ser armazenados de forma garantir:

- Evitar duplicação de entradas com o mesmo identificador.

- Permitir comparação eficiente entre os conjuntos para realizar as operações solicitadas.

A estrutura de dados a utilizar na segunda implementação deverá ser construída manualmente, com base na estrutura definida na secção I.4 (tabela de dispersão com encadeamento externo).

Esta análise estabelece assim a base para o planeamento da implementação, permitindo identificar as estruturas e algoritmos mais adequados à resolução eficiente do problema.

B.2 Estrutura de Dados

A aplicação *ProcessPointsCollections* foi desenvolvida em duas implementações: uma recorrendo às estruturas presentes na Kotlin Standard Library e outra baseada numa estrutura de dados manual, que desenvolvemos anteriormente na primeira parte do enunciado. Ambas as versões utilizam uma estrutura de mapeamento entre coordenadas de pontos e um conjunto de identificadores que indicam a origem desses pontos (ficheiro 1 ou 2), permitindo realizar as operações pedidas de forma eficiente.

B.2.1 Primeira Implementação

Na primeira versão, foi utilizada a classe *HashMap* da biblioteca padrão da linguagem Kotlin, em conjunto com:

Classe *Point*:

- Representa um ponto no plano com duas coordenadas x e y (valores do tipo *Float*).
- Utilizada como chave no *HashMap*.

Estrutura principal de dados: *HashMap<Point, MutableSet<String> >*

- **Chave**: objeto do tipo *Point*.
- **Valor**: conjunto de strings com os valores "f1"ou "f2", indicando a presença do ponto no primeiro e/ou segundo ficheiro.

Esta estrutura permite determinar eficientemente:

- Se um ponto é comum aos dois ficheiros (*intersection*);
- Se pertence apenas ao primeiro (*difference*);
- Ou se pertence a qualquer um dos ficheiros (*union*).

B.2.2 Segunda Implementação

A segunda versão da aplicação substitui as estruturas da biblioteca Kotlin por uma tabela de dispersão com encadeamento externo (hash table), implementada manualmente através da classe *AEDHashMap*.

Classe *AEDHashMap<K, V>*:

- Tabela de dispersão genérica, onde K é a chave (neste caso, do tipo *Point*) e V é o valor associado (conjunto mutável de strings indicando a origem dos pontos), utiliza encadeamento externo para resolução de colisões.
- Implementa a interface *AEDMutableMap<K, V>*, que define as operações *put*, *get* e *iterator*.

Funcionamento da *AEDHashMap*:

- **Inserção (*put*)**: adiciona ou atualiza um elemento na tabela.
- **Leitura (*get*)**: obtém um valor a partir da chave.
- **Expansão (*expand*)**: duplica a capacidade da tabela quando o fator de carga é atingido.

A função *readFiles()* utiliza esta estrutura para ler os pontos de dois ficheiros e construir um *AEDHashMap<Point, MutableSet<String> >*, em tudo semelhante à estrutura da primeira versão, mas criada manualmente.

As funções *union*, *intersection* e *difference* iteram sobre os elementos da *AEDHashMap* para gerar os conjuntos de pontos resultantes.

B.2.3 Outras Estruturas Auxiliares

Classe *IntArrayList*:

- Lista circular de inteiros com tamanho fixo.
- Permite inserção no fim (*append*) e remoção no início (*remove*) com complexidade constante.
- Suporta um valor de offset que é adicionado a todos os elementos quando acedidos, permitindo aplicar um incremento global com a operação *addToAll*.

Função *minimum*:

- Recebe um array representando um max-heap e devolve o menor valor entre as folhas.

- Itera apenas sobre os elementos a partir do índice *heapSize / 2*, onde começam as folhas, otimizando a busca.
- Útil em situações onde se pretende extrair o menor valor de um max-heap, algo não disponível de forma nativa.

Listas duplamente ligadas (*Node<T>*):

- Permitem navegação e manipulação eficiente nos dois sentidos (anterior e seguinte).
- Utilizadas em algoritmos como *splitEvensAndOdds* (separação de pares e ímpares) e *intersection* (criação de nova lista com elementos comuns), onde é necessária remoção e inserção dinâmica de nós.

B.3 Algoritmos e Análise da Complexidade

B.3.1 Leitura e Armazenamento dos Dados

A leitura dos ficheiros consiste em percorrer linha a linha, filtrar as linhas que representam pontos ("v") e armazenar os pontos num mapa (*HashMap* ou *AEDHashMap*) associando cada ponto a um conjunto de origens (ficheiros onde aparece).

Complexidade: Supondo que cada ficheiro tem n linhas, a leitura tem complexidade O(n), pois cada linha é lida uma vez.

A inserção no mapa tem complexidade média O(1), assumindo boa dispersão do hash, logo, o custo total é aproximadamente O(n).

B.3.2 União de Pontos

A operação união consiste em devolver o conjunto de todas as chaves (pontos) presentes no mapa.

Complexidade: Retornar todas as chaves do mapa é uma operação O(k), onde k é o número de pontos únicos armazenados.

B.3.3 Interseção de Pontos

Para calcular a interseção, filtram-se os pontos que pertencem a ambos os ficheiros, ou seja, cujo conjunto de origens contém as duas origens.

Complexidade: A operação percorre o conjunto de pontos e verifica as origens. Isso resulta em O(k), onde k é o número total de pontos.

B.3.4 Diferença de Pontos

A diferença retorna os pontos que aparecem num ficheiro, mas não no outro.

Complexidade: Tal como na interseção, a operação percorre todos os pontos e verifica a presença das origens, resultando em complexidade O(n).

B.3.5 Estrutura de Dados AEDHashMap

A implementação própria de mapa hash utiliza uma tabela de dispersão com tratamento de colisões por encadeamento. O mapa permite operações de inserção, busca e expansão da tabela.

Complexidade:

- Inserção: média O(1), pior caso O(n) (quando ocorre rehashing).
- Busca: média O(1), pior caso O(n).
- Expansão: ocorre quando a carga ultrapassa o fator definido, com custo O(n), amortizado ao longo das inserções.

B.3.6 Outras Estruturas Auxiliares

IntArrayList: lista circular otimizada para operações de inserção e remoção em tempo constante O(1).

Função *minimum*: encontra o valor mínimo entre as folhas de um heap máximo em O(k), onde k é o número de folhas (metade do heap).

Lista Duplamente Ligada (Node): utilizada para operações de inserção e remoção eficientes em listas com complexidade O(1) para inserção/remover, e O(n) para operações que percorrem a lista.

C Avaliação Experimental

Por fim, foi feita a avaliação empírica das duas implementações e foi registrado os resultados para os diferentes tamanhos das partições. A avaliação foi feita no dispositivo com as seguintes características:

Processador (CPU):

- Nome e modelo: Intel Core i7 11th Gen.
- Número de núcleos e threads: 4 núcleos e 8 threads.
- Frequência base e turbo: 2.8 GHz base e 4.7 GHz.

Memória RAM:

- Capacidade total: 12 GB.
- Velocidade: 3200 MT/s (equivalente a 3200 MHz)
- Amrmazenamento SSD: 475 GB SSD.
- Arquitetura do sistema: 64 bits.

		Operações			
File Names	**Implementação**	**Load**	**Union**	**Intersection**	**Difference**
F1.co e F1r.co	1	7.731866400s	385.533300ms	466.885100ms	57.633800ms
F1.co e F1r.co	2	7.078288500s	366.769ms	409.036ms	54.156200ms
F2.co e F2r.co	1	8.988998s	351.199600ms	469.493ms	69.841ms
F2.co e F2r.co	2	9.069158900s	465.017100ms	513.507500ms	59.176900ms
F3.co e F3r.co	1	12.254631700s	435.225ms	763.226600ms	81.394200ms
F3.co e F3r.co	2	12.333921300s	550.500400ms	748.969500ms	79.040700ms
F4.co e F4r.co	1	28.193173s	778.748700ms	1.283286500s	134.433800ms
F4.co e F4r.co	2	29.341375800s	1.070509200s	1.403241500s	157.341600ms
F7x.co e F8x.co	1	2.694516900s	725.331ms	291.104400ms	457.224300ms
F7x.co e F8x.co	2	2.942644s	998.178500ms	337.709200ms	441.077600ms

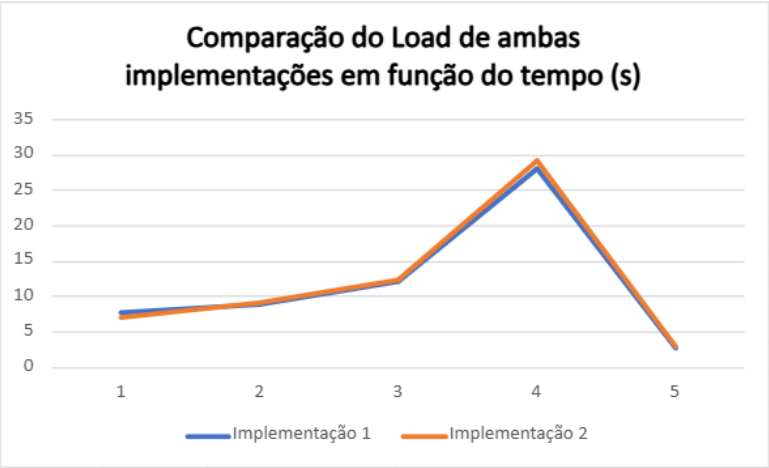

Figura 1

Na Figura 1, apresenta-se um gráfico comparativo do desempenho da função *Load* nas duas implementações desenvolvidas, medido em tempo de execução (segundos). Observa-se que ambas apresentam resultados muito semelhantes em todos os testes realizados. Até ao terceiro ficheiro, o tempo de execução manteve-se inferior a 13 segundos. No caso do quarto ficheiro, registou-se um aumento significativo para cerca de 30 segundos, seguido por uma descida acentuada para menos de 5 segundos no quinto ficheiro.

Esta semelhança nos tempos demonstra que, para a operação de leitura e processamento inicial dos dados, ambas as abordagens oferecem um desempenho equivalente.

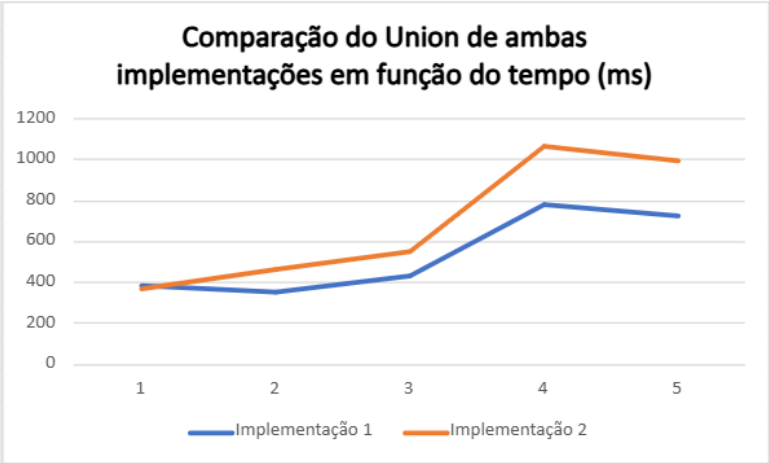

Figura 2

Na Figura 2, apresenta-se a comparação dos tempos de execução da função *union* entre as duas implementações, medidos em milissegundos. Os resultados mostram que a segunda implementação é consistentemente mais lenta do que a primeira, especialmente a partir do segundo ficheiro.

Na primeira implementação:

- O tempo de execução mantém-se próximo dos 400 milissegundos para os ficheiros 1 e 3.
- No segundo ficheiro, observa-se uma ligeira melhoria, com tempo abaixo dos 400 milissegundos.
- Para o quarto ficheiro, há um aumento acentuado, atingindo quase os 800 milissegundos.
- No quinto ficheiro, o tempo desce ligeiramente, aproximando-se dos 700 milissegundos.

Na segunda implementação:

- Verifica-se um aumento gradual do tempo de execução entre os ficheiros 1 e 3, passando dos 400 para quase 600 milissegundos.
- No quarto ficheiro, regista-se um pico superior a 1000 milissegundos.
- No quinto ficheiro, o tempo mantém-se elevado, rondando os 1000 milissegundos.

Estes resultados mostram que a primeira implementação é mais eficiente na operação de união dos conjuntos de pontos, especialmente em cenários com maior volume de dados.

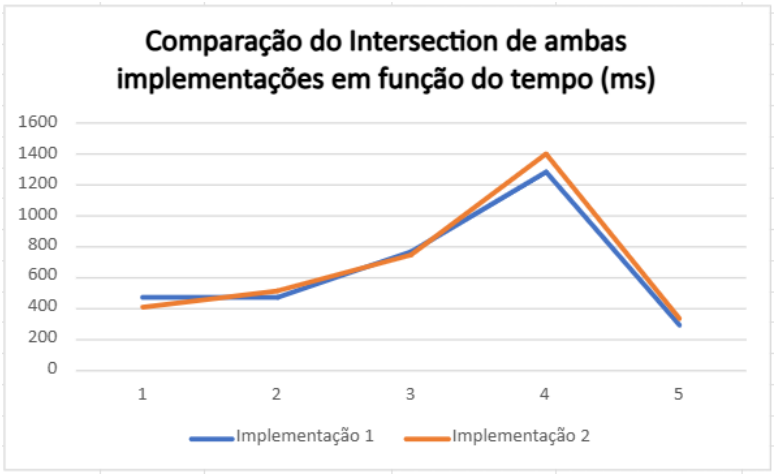

Figura 3

Na Figura 3, é apresentada a comparação dos tempos de execução da função *intersection* para ambas as implementações, em milissegundos. O comportamento do gráfico assemelha-se ao observado na Figura 1, com variações significativas nos tempos de execução consoante o ficheiro processado.

Para os ficheiros 1 e 2, ambas as implementações apresentam tempos de execução em torno dos 500 milissegundos.

No terceiro ficheiro, verifica-se um aumento, com os tempos a aproximarem-se dos 800 milissegundos.

O pico ocorre no quarto ficheiro, onde:

- A primeira implementação atinge cerca de 1200 milissegundos;
- A segunda implementação ultrapassa este valor, chegando aos 1400 milissegundos.

No quinto ficheiro, observa-se uma descida acentuada nos tempos de execução para ambas as implementações, ficando em torno dos 300 milissegundos.

Este comportamento indica que ambas as implementações sofrem degradação de desempenho com o aumento do volume de dados, especialmente no quarto ficheiro, mas recuperam eficiência no último caso analisado.

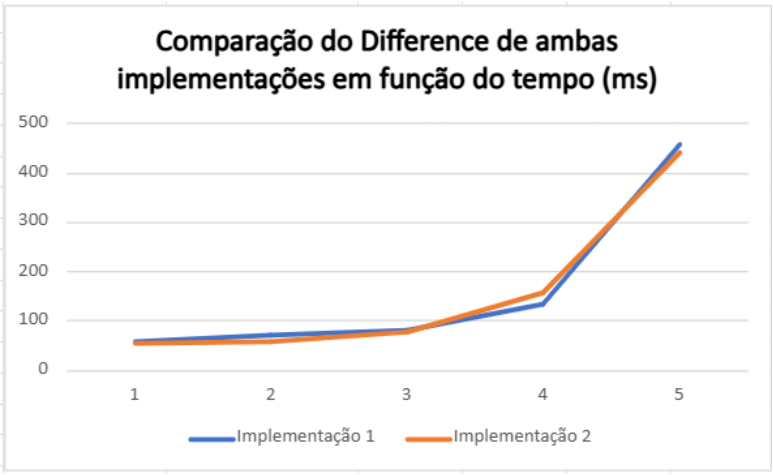

Figura 4

Na Figura 4, observa-se a comparação dos tempos de execução da função difference entre as duas implementações, medidos em milissegundos. Ambas as implementações apresentam um crescimento gradual e semelhante ao longo dos ficheiros analisados:

- Nos três primeiros ficheiros, o tempo de execução mantém-se abaixo dos 100 milissegundos.
- Para o quarto ficheiro, há um aumento significativo, com o tempo a aproximar-se dos 300 milissegundos.
- No quinto ficheiro, verifica-se uma subida mais acentuada, ultrapassando os 400 milissegundos.

Este padrão indica uma escalabilidade semelhante para ambas as implementações, com maior impacto no desempenho para os ficheiros de maior dimensão.

D Conclusões

Neste relatório, foram implementadas e analisadas duas abordagens para manipulação de conjuntos de pontos, com destaque para as funções Load, Union, Intersection e Difference. A comparação de desempenho mostrou que, embora as implementações sejam similares na função Load, diferenças surgem nas operações que envolvem processamento dos dados, onde a escolha das estruturas de dados impactou diretamente o tempo de execução.

A análise da complexidade e o desenvolvimento de uma estrutura própria de hash reforçaram a importância de uma boa seleção algorítmica para garantir eficiência. Este projeto permitiu consolidar conceitos teóricos e sua aplicação prática em programação eficiente.

E 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, 3th Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, MIT Press.