Algoritmos e Estruturas de Dados

2ª Série

(Problema)

Operações entre coleções de pontos no plano

Nº 51616 Afonso Abranja
Nº 51303 Bernardo Lopes
Nº 51786 Sérgio Oliveira

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

18/05/2025

Índice

1. Introdução

No problema da segunda série, propôs-se o desenvolvimento de uma aplicação, designada ProcessPointsCollections, com o objetivo de realizar operações sobre coleções de pontos no plano bidimensional. Cada coleção de pontos é armazenada num ficheiro de texto com extensão .co, sendo cada ponto representado por um identificador único e duas coordenadas (X, Y). A aplicação é capaz de processar dois ficheiros de entrada e permitir a execução eficiente das operações de união, interseção e diferença entre os conjuntos de pontos definidos em cada ficheiro.

Para tal, a aplicação carrega os dados dos ficheiros para uma estrutura de dados eficiente — nomeadamente, uma tabela de dispersão (hash map) — permitindo uma rápida consulta e manipulação das coleções. A estrutura escolhida garante a ausência de repetições nos resultados e otimiza o desempenho das operações, especialmente quando se lida com conjuntos de grandes dimensões.

O problema encontra-se dividido em duas fases: numa primeira implementação, recorre-se a estruturas de dados disponibilizadas pela Kotlin Standard Library. Numa segunda fase, é utilizada uma estrutura de dados personalizada, desenvolvida anteriormente no exercício 4 da primeira parte. Para validar a eficiência das abordagens, realizou-se uma avaliação experimental, comparando o desempenho de ambas as implementações com base em diferentes conjuntos de dados.

Como tal, nos pontos seguintes do relatório são documentadas a análise, o desenvolvimento, os testes e os resultados obtidos, oferecendo uma visão crítica sobre as decisões técnicas e o desempenho alcançado.

2. Problema

O objetivo deste problema era desenvolver uma aplicação capaz de realizar operações de conjunto (união, interseção e diferença) entre duas coleções de pontos no plano cartesiano, representadas em ficheiros de texto. A aplicação permite o carregamento eficiente dos dados, eliminando repetições e garantindo a integridade da informação, com o intuito de gerar novos ficheiros contendo os resultados das operações pedidas.

2.1 Análise do problema

Para atingir este objetivo, foi essencial escolher uma estrutura de dados que permitisse uma gestão eficiente dos pontos, tanto ao nível de inserção como de consulta. A utilização de estruturas de dispersão (como HashSet ou HashMap) revela-se particularmente adequada, uma vez que permitem aceder e comparar elementos em tempo constante médio, o que é especialmente vantajoso quando se lida com grandes volumes de dados.

As operações pedidas — união**,** interseção e diferença — podem ser diretamente implementadas utilizando os métodos disponibilizados pela Kotlin Standard Library sobre conjuntos (Set), o que simplifica significativamente o desenvolvimento da aplicação. Cada conjunto representa os pontos carregados de um dos ficheiros .co, ignorando as linhas de comentário (prefixadas com 'c' ou 'p'), e retendo apenas os pontos válidos (linhas com prefixo 'v').

2.2 Estruturas de dados

Para responder de forma eficiente às operações requeridas pelo problema foi concebida uma estrutura de dados baseada em coleções da Kotlin Standard Library, mais concretamente na utilização de conjuntos (Set) e classes de dados (data class) e outra baseada na estrutura de dados implementada na questão I.4.

Cada ponto é representado por um objeto da classe Ponto, composta por três atributos: um identificador (id), e duas coordenadas (x e y). A estrutura é definida como uma data class, o que permite a geração automática dos métodos equals, hashCode e toString, garantindo que a comparação entre objetos tem em conta os três atributos e que não existam duplicados 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 simples. Esta biblioteca oferece um conjunto abrangente de estruturas de dados otimizadas, que permitem realizar operações de forma direta e com desempenho adequado para a maioria dos casos.

Cada ponto é representado por uma data class, que encapsula os seus três atributos principais: identificador (id) e coordenadas x e y. Ao usar a data class, o Kotlin gera automaticamente os métodos equals() e hashCode(), permitindo que objetos Ponto possam ser comparados corretamente e utilizados em coleções como Map.

Para armazenar os pontos de cada ficheiro .co, foram utilizados conjuntos mutáveis que internamente são implementados como tabelas de dispersão (hash tables). Esta escolha traz várias vantagens:

- Eliminação automática de duplicados: pontos com o mesmo conteúdo não são adicionados mais do que uma vez.
- Operações eficientes: inserções, pesquisas e remoções ocorrem, em média, em tempo constante (O(1)).
- Suporte nativo a operações de conjunto.

Os pontos são carregados a partir de ficheiros .co, ignorando as linhas irrelevantes e extraindo apenas as linhas que contêm coordenadas. Cada linha válida é convertida num objeto Ponto e inserida no conjunto correspondente.

Estas operações devolvem novos conjuntos sem modificar os originais, promovendo código mais seguro e previsível.

Implementação 2

A implementação 2 foi desenvolvida uma estrutura de dados personalizada que implementa o tipo abstrato MutableMap<K, V>, de acordo com as especificações fornecidas no enunciado da questão I.4. Esta estrutura representa um mapa associativo de pares chave-valor, suportando operações básicas como inserção, pesquisa e iteração. A implementação baseia-se numa tabela de

dispersão (hash table) com encadeamento externo, utilizando listas ligadas não circulares e sem nó sentinela para lidar com colisões.

O mapa é composto por um vetor de listas (Array<HashNode<K,V>?>), onde cada índice é calculado a partir do código de dispersão (hashCode()) da chave, ajustado à capacidade da tabela para garantir que o índice esteja dentro dos limites válidos.

A estrutura implementa a interface MutableMap<K, V>, respeitando todos os seus contratos:

- Propriedade size: Mantém a contagem do número atual de pares armazenados.
- Propriedade capacity: Indica a capacidade total da tabela de dispersão (número de posições).
- Função put(key, value): Insere ou atualiza um par chave-valor. Se a chave já existir, atualiza o valor associado e devolve o valor anterior. Caso contrário, insere um novo nó.
- Operador get(key): Devolve o valor associado a uma chave, ou null se esta não existir.
- Função iterator(): Permite iterar sobre todos os pares chave-valor presentes na tabela, através da interface Iterable.

A tabela é expandida para o dobro quando o número de elementos multiplicado pelo *fator de carga* for igual ou superior à capacidade da tabela. O fator de carga é a razão entre o número de elementos presentes na tabela de dispersão e a sua dimensão.

3. Avaliação Experimental

A avaliação experimental teve como principal objetivo comparar o desempenho das duas abordagens implementadas para manipulação de coleções de pontos:

1. A versão baseada nas estruturas de dados da Kotlin Standard Library.
2. A versão baseada numa implementação própria da estrutura MutableMap<K, V>, construída com uma tabela de dispersão com encadeamento externo.

Objetivos da Avaliação

- Medir o tempo de execução das operações principais (união, interseção e diferença).
- Avaliar a eficiência da estrutura de dados personalizada em comparação com a solução nativa.
- Observar o comportamento das estruturas com conjuntos de dados de diferentes dimensões.

Metodologia

Foram utilizados vários ficheiros de teste com diferentes quantidades de pontos, porém para efeitos práticos de exposição dos resultados optou-se por usar os ficheiros F1.co e F2.co para a implementação 1. Mediu-se o tempo necessário para carregar os ficheiros e executar cada uma das três operações suportadas pela aplicação (*union, intersection, difference*), utilizando ambas as implementações. Em baixo encontram-se excertos dos ficheiros F1 (esquerda) e F2 (direita).

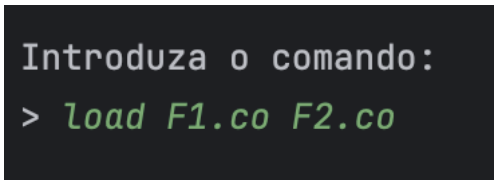
```
v 1 -73530767 41085396
v 2 -73530538 41086098
v 3 -73519366 41048796
v 4 -73519377 41048654
v 5 -73524567 41093796
v 6 -73525490 41093834
v 7 -73531927 41110484
v 8 -73530106 41110611
v 9 -73529341 41125895
v 10 -73529746 41127369
```

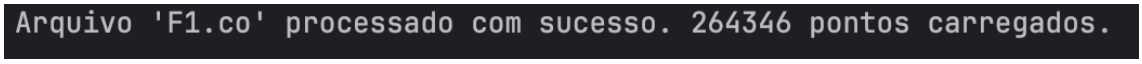
```
v 1 -121745853 37608914
v 2 -121745591 37610691
v 3 -121749163 37614888
v 4 -121747954 37614855
v 5 -121740904 37615568
v 6 -121654486 37620517
v 7 -121696562 37646830
v 8 -121656386 37620082
v 9 -121675244 37619359
v 10 -121684010 37611698
```

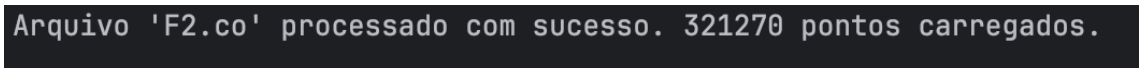
Os testes foram repetidos múltiplas vezes para minimizar o impacto de variações no sistema (ex. processamento paralelo, cache), sendo os valores apresentados correspondentes a uma dessas execuções aquando da estabilidade dos mesmos.

Resultados

Na implementação 1, começou-se por iniciar a aplicação e executar o comando *load F1.co F2.co*. O código está formatado para que pesquise os ficheiros nos diretórios do projeto, visto que foram colocados dentro do projeto os ficheiros a utilizar. Após este comando, é mostrada a mensagem de que a pesquisa foi feita corretamente, ou seja, os ficheiros encontram-se definitivamente no projeto. De seguida, executa-se os comandos *difference, union* e *intersection*, sendo que entre as suas execuções, foi verificado o ficheiro output.co, que ficava com os resultados de cada comando. É de salientar que quando são executados os comandos, são também calculados os tempos de execução. Por último, o comando *exit* terminava a execução do programa. As imagens seguintes representam estes mesmos passos acabados de explicar.





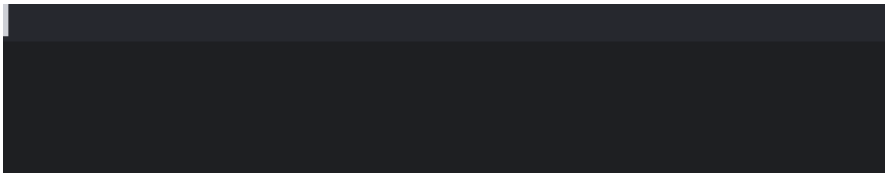


```
-7.3530767E7 4.1085396E7
-7.3530538E7 4.1086098E7
-7.3519366E7 4.1048796E7
-7.3519377E7 4.1048654E7
-7.3524567E7 4.1093796E7
-7.352549E7 4.1093834E7
-7.3531927E7 4.1110484E7
-7.3530106E7 4.1110611E7
-7.3529341E7 4.1125895E7
-7.3529746E7 4.1127369E7
```

```
Introduza o comando:
> union output.co
Union levou 188.58632 ms
```

```
-7.3530767E7 4.1085396E7
-7.3530538E7 4.1086098E7
-7.3519366E7 4.1048796E7
-7.3519377E7 4.1048654E7
-7.3524567E7 4.1093796E7
-7.352549E7 4.1093834E7
-7.3531927E7 4.1110484E7
-7.3530106E7 4.1110611E7
-7.3529341E7 4.1125895E7
-7.3529746E7 4.1127369E7
```

```
Introduza o comando:
> intersection output.co
Intersection levou 145.872902 ms
```



```
Introduza o comando:
> exit

Process finished with exit code 0
```

Na implementação 2, o processo em si foi todo semelhante, sendo que foram executados os mesmos comandos. As imagens abaixo representam a execução completa do programa.

```
Introduza o comando:
> load F1.co F2.co
```

```
Arquivo 'F1.co' processado com sucesso: 264346 pontos carregados.
```

```
Arquivo 'F2.co' processado com sucesso: 321270 pontos carregados.
```

```
Introduza o comando:
> difference output.co
Difference levou 233.878285 ms
Arquivo 'output.co' salvo com sucesso.
```

```
-7.3779924E7 4.1129707E7
-7.3759978E7 4.0996821E7
-7.3785749E7 4.1131464E7
-7.3781799E7 4.1130028E7
-7.3791352E7 4.1140301E7
-7.371805E7 4.1188813E7
-7.3849238E7 4.1071608E7
-7.3721435E7 4.0958913E7
-7.3800316E7 4.1048257E7
-7.3716604E7 4.112416E7
```

```
Introduza o comando:
> union output.co
Union levou 363.992489 ms
Arquivo 'output.co' salvo com sucesso.
```

```
-7.3779924E7 4.1129707E7
-7.3759978E7 4.0996821E7
-7.3785749E7 4.1131464E7
-7.3781799E7 4.1130028E7
-7.3791352E7 4.1140301E7
-7.371805E7 4.1188813E7
-7.3849238E7 4.1071608E7
-7.3721435E7 4.0958913E7
-7.3800316E7 4.1048257E7
-7.3716604E7 4.112416E7
```

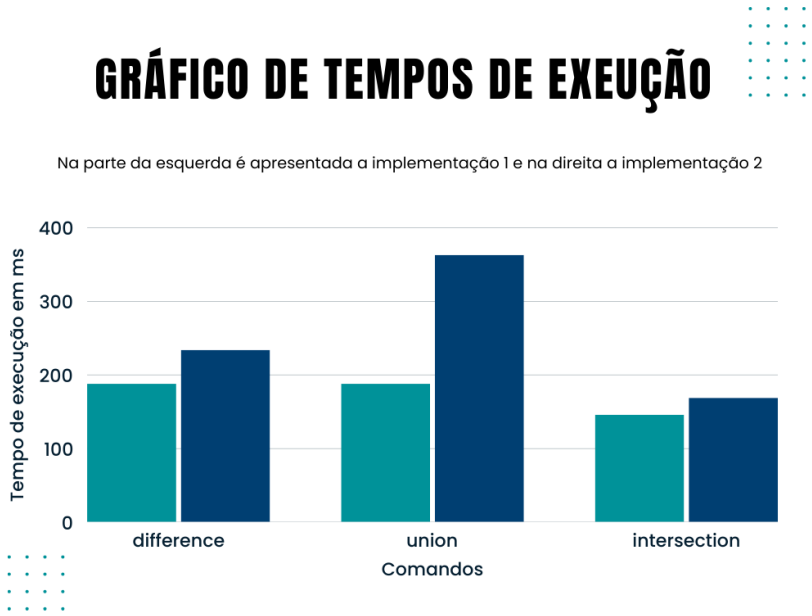
```
Introduza o comando:
> intersection output.co
Intersection levou 169.123991 ms
Arquivo 'output.co' salvo com sucesso.
```

```
Introduza o comando:
> exit

Process finished with exit code 0
```

Verificou-se que, em ambas as implementações, os resultados estavam de acordo com as expetativas. Apenas no comando *intersection*, o ficheiro output.co não tinha pontos, visto que não existiam pontos partilhados entre F1 e F2.

Para uma melhor compreensão da diferença em termos temporais das duas implementações, foi feito um gráfico de barras, que se encontra em baixo.



Como tal, a abordagem baseada na Kotlin Standard Library apresentou um desempenho superior em todos os testes. Isso deve-se à elevada otimização das estruturas internas da linguagem, que utilizam técnicas sofisticadas de gestão de memória e hashing.

Por outro lado, a estrutura de dados personalizada manteve um comportamento funcional e estável, mas apresentou tempos de execução mais elevados, especialmente na execução do comando *union*, e sobretudo na fase de redimensionamento da tabela (quando o fator de carga era atingido). Esta diferença tornou-se mais evidente à medida que o volume de dados aumentava.

Análise

A avaliação experimental demonstrou que:

- A solução com a Kotlin Standard Library é recomendada para aplicações práticas, onde o desempenho e a simplicidade de desenvolvimento são prioridades.
- A implementação personalizada é mais adequada para fins didáticos e para explorar o funcionamento interno das tabelas de dispersão, permitindo uma melhor compreensão dos conceitos envolvidos.

4. Conclusões

O desenvolvimento da aplicação ProcessPointsCollections permitiu implementar e comparar duas abordagens distintas para a manipulação de coleções de pontos no plano: uma baseada nas coleções da Kotlin Standard Library e outra com uma estrutura de dados personalizada do tipo MutableMap<K, V>, conforme solicitado na questão I.4.

A implementação utilizando as coleções da Kotlin Standard Library revelou-se simples, eficiente e de fácil manutenção. Foi possível gerir os pontos no plano de forma direta, tirando partido das funcionalidades integradas da linguagem para realizar operações como união, interseção e diferença com clareza e poucas linhas de código. Esta abordagem também facilitou a leitura e escrita de ficheiros, permitindo uma manipulação eficiente dos dados com um bom equilíbrio entre desempenho e legibilidade.

A segunda abordagem, que envolveu a construção manual de uma tabela de dispersão com encadeamento externo, permitiu compreender em profundidade o funcionamento interno de estruturas de mapas associativos, bem como os mecanismos de dispersão, colisão e redimensionamento dinâmico. Apesar de mais complexa, esta implementação oferece um controlo total sobre a estrutura de dados.

A avaliação experimental, ao comparar o desempenho das duas abordagens, fornece uma base sólida para perceber as vantagens e limitações de cada uma. Em geral, a utilização da biblioteca padrão é recomendada para soluções práticas e rápidas, enquanto a implementação personalizada é ideal para fins de otimização ou em cenários com requisitos específicos.

Esta problema da segunda série demonstrou a importância da escolha da estrutura de dados adequada à natureza do problema.

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