Algoritmos e Estruturas de Dados

2ª Série

Operações entre coleções de pontos no plano

Nº52759 Nome: Stela Rocheteau

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2024/2025

18/05/2025

Índice

1. Introdução

Este trabalho, realizado na disciplina de Algoritmos e Estruturas de Dados, teve como objetivo desenvolver a aplicação ProcessPointsCollections, que consegue carregar dois ficheiros de pontos (com extensão .co) fornecidos pela professora. Durante a leitura, comentários (que iniciam com c) e cabeçalhos (que iniciam com p) devem ser ignorados, de modo a garantir a correta interpretação dos dados. Após o carregamento dos ficheiros, o utilizador tem a possibilidade de solicitar operações de conjuntos: união, interseção e diferença. Os resultados dessas operações devem ser guardados em novos ficheiros (output.co), garantindo que os pontos não se repetem, considerando apenas as coordenadas x e y de cada ponto, já que o identificador de cada linha não é relevante para a distinção.

O que queremos mesmo é criar uma solução que funcione bem e seja eficiente. Para isso, estabelecemos alguns objetivos específicos:

- Implementar as operações de união, interseção e diferença de forma correta e rápida, com uma complexidade linear relativamente ao número de pontos diferentes. Para isso, vamos usar uma tabela de dispersão com encadeamento externo, que é uma estrutura de dados eficiente para este tipo de tarefas.
- Para testar se tudo funciona bem, vamos fazer duas versões da aplicação:
    1. HashMap<Point, List<String> que vem nas bibliotecas do Kotlin/Java;
    2. AEDHashMAP<Point, File> que criei na questão I.4.
- Depois, vamos fazer testes práticos para medir quanto tempo a aplicação leva a executar e quanta memória consome. Vamos usar os ficheiros que se encontram disponíveis juntamente com o enunciado para verificar se a teoria corresponde à prática.

2. Operações entre coleções de Pontos no Plano

Esta secção descreve o segundo desafio apresentado na *Série II*, a implementação da aplicação ProcessPointsCollections. Iniciamos com uma visão geral e com a motivação da tarefa, seguidas de três subsecções:
2.1 Análise do Problema que detalha o enunciado, as operações a suportar e a estratégia adotada para as implementar;
2.2 Estruturas de Dados que apresenta as opções escolhidas para representar de forma eficiente coleções de pontos;
2.3 Algoritmos e Análise da Complexidade que mostra os algoritmos principais, a forma como consultam/atualizam as estruturas de dados e a respetiva complexidade assintótica.

2.1 Análise do problema

Este projeto envolve criar uma aplicação chamada ProcessPointsCollections, que tem como objetivo ler dois ficheiros contendo listas de pontos no formato .co(**load**). A ideia é, a partir dessas informações, gerar três conjuntos diferentes de pontos: um que reúne todos os pontos presentes em ao menos um dos ficheiros, sem repetições(**união**); outro com apenas os pontos que aparecem em ambos os ficheiros ao mesmo tempo(**interseção**); e um terceiro com os pontos que estão no primeiro ficheiro, mas não no segundo(**diferença**).
Cada linha importante nesses ficheiros começa com a letra v to tipo char, seguida de um identificador numérico (id) do tipo, e depois das coordenadas x e y (do tipo Float) do ponto. Linhas que começam com c (comentários) ou p (que descrevem o ficheiro) não trazem dados úteis e podem ser ignoradas.

Para identificar um ponto na aplicação, usei apenas as coordenadas (x, y). Isso significa que pontos com as mesmas coordenadas são considerados iguais, mesmo que tenham identificadores diferentes ou que apareçam várias vezes nos ficheiros.
O que queremos alcançar é fazer uma leitura eficiente, percorrendo cada ficheiro só uma vez, de forma linear, para otimizar o desempenho. Para cada ponto que encontramos, guardamos numa estrutura de dados onde a chave é o ponto (x, y) e o valor indica de qual ficheiro ele veio, na primeira versão é uma lista imutável e na segunda um objeto.
Depois de montar essa estrutura, percorremos tudo de uma vez só para criar os conjuntos que nos interessam (união, interseção e diferença) e escrevemos os ficheiros de saída, sempre no formato simples de x e y, sem prefixos ou identificadores extras.
Resumindo, o que o problema pede é que leiamos os dois conjuntos de coordenadas, registemos de forma eficiente de onde cada ponto veio, e depois percorramos essa estrutura para obter os três conjuntos desejados. Para isso, usamos uma tabela de dispersão (seja a da biblioteca do Kotlin ou uma implementada por nós), que garante que cada ponto seja único, que as operações sejam rápidas e que tudo funcione dentro dos limites de desempenho e formato definidos inicialmente.

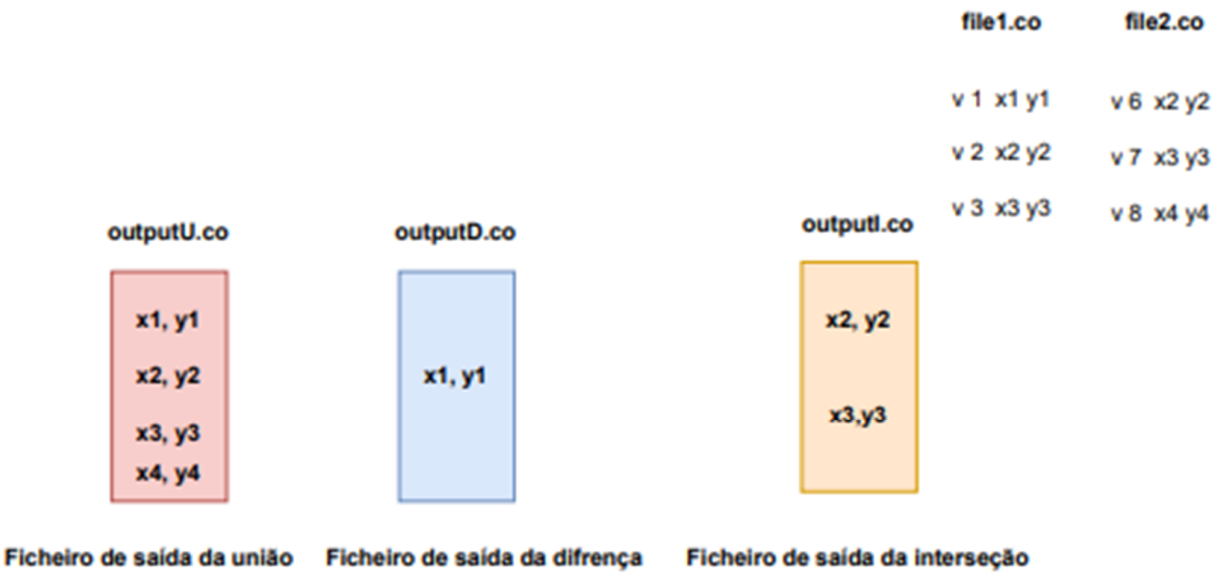

Figura 1: Exemplo dos ficheiros de saída da união, intersecção e diferença gerados a partir do file1.co e file2.co.

2.2 Estruturas de Dados

Para cumprir os requisitos do enunciado, usei duas implementações diferentes, uma mais simples e outra mais elaborada, de modo a mostrar duas formas de resolver o mesmo problema.

Na primeira implementação, usei a estrutura HasMap<Point, List<String>> da biblioteca padrão do Java/Kotlin, que é mais fácil e rápida de implementar. Aqui, cada ponto no plano é a chave de um mapa, e o valor é uma lista imutável com os nomes dos ficheiros onde esse ponto aparece, como "file1" ou "file2". Assim, basta verificar se uma etiqueta está na lista para saber se o ponto está naquele ficheiro. Essa solução é bastante eficiente, porque as operações de procura são rápidas (cerca de O (1)) na média).

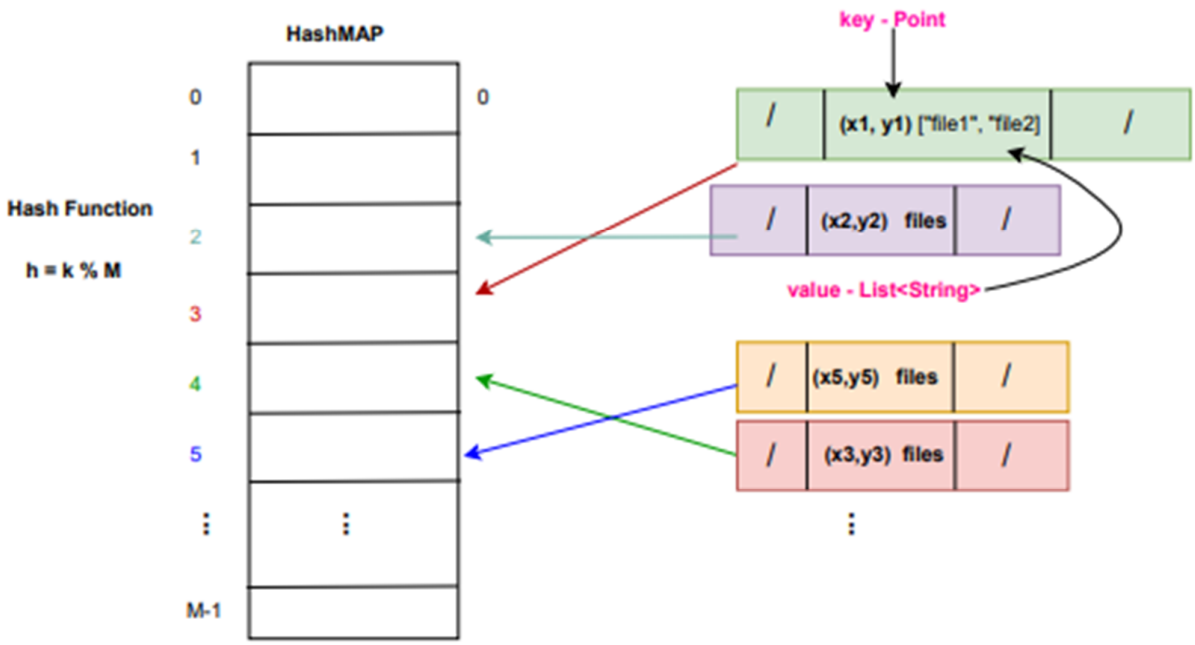

Figura 2: Funcionamento do HashMap da primeira implementação

Na segunda implementação, criei uma estrutura própria, uma tabela de dispersão que nós desenvolvemos na questão 4 da primeira parte do trabalho. Aqui, cada ponto continua a ser a chave, mas o valor é um objeto que indica, com dois bits booleanos, se o ponto está no ficheiro 1, no ficheiro 2, ou em ambos. Essa solução é mais trabalhosa de implementar, mas permite um controlo maior sobre o funcionamento, além de usar menos memória, o que é uma vantagem.
Sobre o tipo Point, que usamos como chave, é uma estrutura imutável, ou seja, depois de criar, não dá para alterar as coordenadas (x e y):
data class Point (val x: Float, val y: Float)
Isso ajuda a garantir que cada ponto é único e consistente. O método hashCode () é criado automaticamente, garantindo que pontos iguais tenham o mesmo código e o método equals () compara apenas as coordenadas x e y, para saber se dois pontos são iguais.
Assim, podemos ter certeza de que cada ponto no plano é tratado de forma única, sem confusões.
Quanto aos valores que guardamos na tabela, na primeira abordagem, usamos uma lista com as etiquetas "file1" e "file2" para indicar em que ficheiro o ponto aparece. Dessa forma, evitamos repetir informações e podemos verificar facilmente se o ponto está em um ficheiro ou no outro.

Na segunda, usamos uma pequena estrutura que indica, com dois booleanos, se o ponto pertence ao ficheiro 1, ao ficheiro 2, ou aos dois:
data class File (val file1: Boolean, val file2: Boolean)
Essa solução ocupa menos espaço e facilita fazer operações como união, interseção ou diferença entre os conjuntos de pontos.

A estrutura que criei, chamada AEDHashMap<Point, File>, funciona de forma semelhante às tabelas de dispersão tradicionais: tem posições onde ficam as listas de colisões. Quando a tabela fica cheia, ela expande-se automaticamente, duplicando o tamanho e redistribuindo os pontos. Para determinar em que posição colocar um ponto, uso o método hashCode () ajustado para um índice válido.
Essa estrutura garante que as operações de inserir ou procurar pontos (get ou put) continuam rápidas, com uma média de O (1), mesmo quando há muitas colisões.

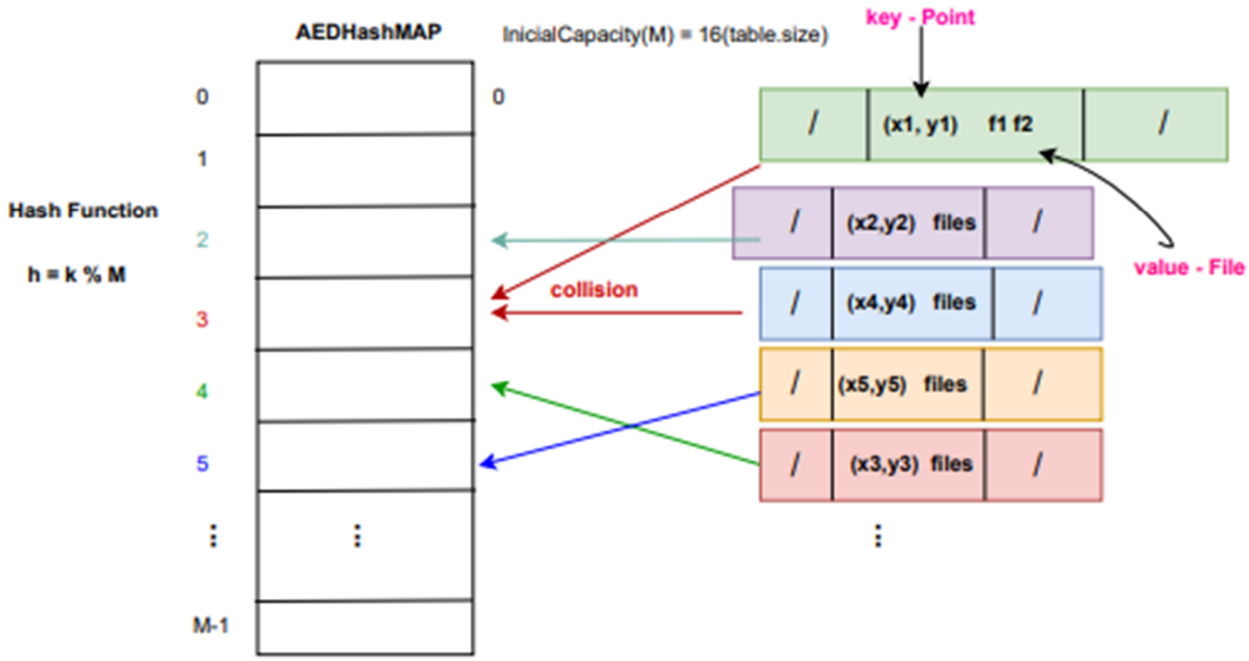

Figura 3: Funcionamento do AEDHashMap da segunda implementação

A estrutura começa com 16 posições (ou seja, um tamanho inicial de M = 16) e um fator de carga de 0,75, o que significa que, quando a tabela estiver 75% cheia ($\alpha = \frac{k}{M} <= 0,75$), neste caso, com 12 elementos (k = 12) ela automaticamente aumenta de tamanho do array (através da função expand). Quando isso acontece, a tabela duplica a sua capacidade, e todos os elementos são redistribuídos com base nos seus novos índices, usando o cálculo k% M.
É importante lembrar que o processo de aumentar a tabela (quando ela fica cheia) tem um custo maior, porque precisa reorganizar tudo, mas como isso acontece só ocasionalmente, o impacto no desempenho geral é mínimo e bem controlado.

Para facilitar a leitura e escrita dos ficheiros, criei funções auxiliares que cuidam de abrir, fechar e ler/escrever ficheiros, de modo a evitar repetir código nas duas implementações.
A primeira implementação é mais fácil de fazer, pois aproveita a biblioteca já pronta. No entanto, ocupa mais espaço por entrada, pois guarda uma lista de etiquetas, a segunda exige mais trabalho, mas é mais eficiente em memória e dá maior controlo sobre o funcionamento, além de permitir ajustes finos na forma como gerimos as colisões.

2.3 Algoritmos e análise da complexidade

O processo principal da aplicação é dividido em quatro etapas: primeiro, a leitura dos ficheiros, e depois três operações que geram os conjuntos de pontos: união, interseção e diferença.
Vamos considerar que o número de pontos distintos obtidos após ler os dois ficheiros é representado por n. A análise do tempo leva em conta uma estrutura de dados eficiente, como uma tabela de dispersão bem dimensionada, que costuma ter um desempenho médio bom.
Na fase de leitura, o programa percorre as linhas de ambos os ficheiros uma única vez, verificando se cada linha representa um ponto (v) e, se sim, cria um objeto com as coordenadas (x, y). Essa operação é rápida, pois cada linha é processada uma única vez. Depois, vai atualizar ou inserir uma entrada na tabela de dispersão, cujo custo médio por operação é constante, ou seja, quase instantâneo, com pequenas variações que se distribuem ao longo de várias inserções. No total, essa fase tem um custo linear, proporcional ao número de pontos, isto é, O(n).
Na memória, o armazenamento é eficiente: cada ponto é guardado com uma chave única (x, y) e um valor que indica de qual ficheiro ele veio. Na primeira implementação, esse valor é uma lista com até duas Strings, e na segunda, apenas dois bits que sinalizam a presença em cada ficheiro.
Depois de carregar os dados, as operações de união, interseção e diferença consistem em percorrer toda a tabela de uma só vez. Para cada ponto, verificamos uma condição simples (por exemplo, se ele veio do ficheiro 1, do 2, ou ambos), e, se a condição for satisfeita, escrevemos o ponto no ficheiro de saída(output.co). Como percorremos cada elemento exatamente uma vez, cada operação tem um custo linear, O(n), além de um tempo constante para verificar a condição e escrever na saída.
Se pensarmos no pior caso possível, onde muitas chaves colidem, o desempenho pode degradar um pouco. No caso do HashMap padrão do Java, a estrutura transforma listas longas em árvores equilibradas, garantindo uma busca eficiente de O (log n). Já na implementação própria baseada em listas ligadas, a busca pode chegar a O(n) por operação, levando a um desempenho pior, especialmente na fase de carga, que poderia chegar a O (n^2).
Resumindo, a fase de carga e as operações de conjunto são todas realizadas em tempo proporcional ao número de pontos, usando estruturas de dados que mantêm o desempenho rápido mesmo para grandes volumes de dados.

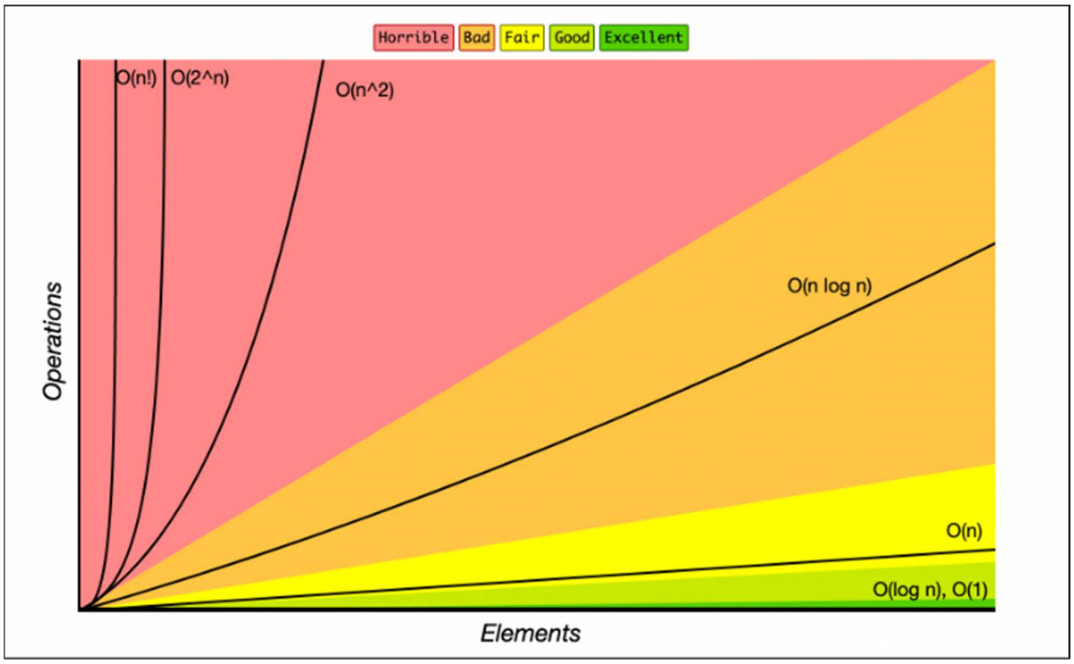

Gráfico 1: Big O Complexity Chart

As nossas implementações permanecem nas zonas verde e amarela do gráfico 1, o que confirma a eficiência dos algoritmos utilizados.

3. Avaliação Experimental

Para avaliar o desempenho das duas implementações propostas, realizamos um conjunto de testes com diferentes ficheiros de extensão .co, dados pelo enunciado (File1 e File2) para cada fase da aplicação produzidas nas duas implementações (load, union, intersection, difference).

Realizamos os testes no DESKTOP-TCVONLB com 16 GB de memória RAM e um processador Intel Core i7. A Figura 1 apresenta uma imagem do sistema, destacando as principais características de hardware e de sistema operativo empregues durante a experiência.

Figura 4: Características da máquina onde foram realizados os testes

Ficheiros atribuídos para fazer estes testes:

File1 - F1.co, F1r.co, F2.co, F2r.co, F3.co, F3r.co, F4.co, F4r.co, F7x.co e F8x.co

File2 - F6r.co

A tabela 1 a seguir ilustra os resultados médios de tempo de execução para cada implementação, em segundos (s):

File 1	File 2	Implementação	load	union	intersection	difference
F1.co	F6r.co	1	20.744684300s	2.920466100s	1.005242300s	1.047810100s
		2	21.099439700s	2.989751500s	1.066462600s	1.049916700s
F1r.co	F6r.co	1	27.982616400s	3.693852400s	1.147286s	1.243752801s
		2	27.678707700s	3.894753700s	1.090856s	1.186288600s
F3.co	F6r.co	1	26.619193500s	3.914387700s	1.174482100s	1.328764100s
		2	26.756572200s	3.843122600s	1.073123800s	1.214721600s
F3r.co	F6r.co	1	27.785498s	3.672224800s	1.044512100s	1.136816200s
		2	28.586071300s	4.805784300s	1.264120900s	1.216539900s
F4r.co	F6r.co	1	25.017580300s	3.631825200s	1.187052100s	1.174270300s
		2	25.653766600s	3.629805s	1.370028400s	1.406717700s
F8x.co	F6r.co	1	27.900882500s	3.923492600s	1.216723700s	1.284386200s
		2	28.107522300s	3.639043700s	1.312443200s	1.318015400s

Tabela 1: Resultados do tempo de execução de algoritmos da 1ª implementação e 2ª implementação

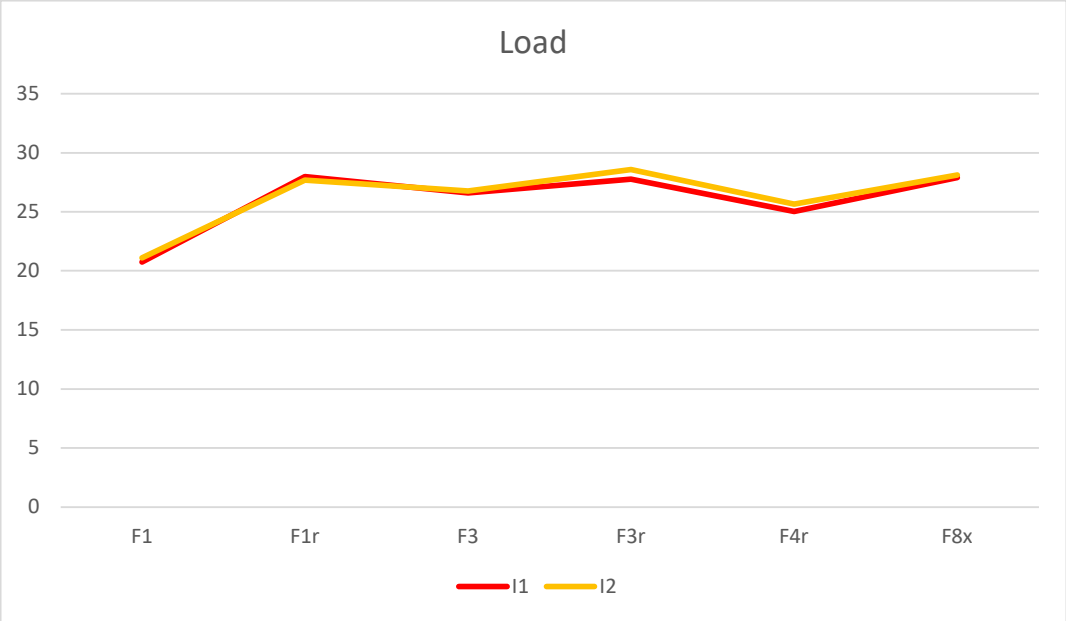

Gráfico 2: Comparação dos tempos de execução do algoritmo Load da implementação 1 e 2.

O gráfico 2, apresenta uma tendência crescente e suave, proporcional ao tamanho dos ficheiros. Confirma-se o comportamento O(n) esperado, tal como descrito na secção 2.3 do relatório. A Implementação 1 (linha vermelha) é ligeiramente mais rápida em ficheiros pequenos e médios, mas a partir do F3r, F4r e F8x (ficheiros maiores), a implementação 2 (linha amarela) começa a ser mais rápida, demostrando que a sua estrutura (AEDhashMap) tem uma excelente escalabilidade. Portanto este comportamento está alinhado com o previsto.

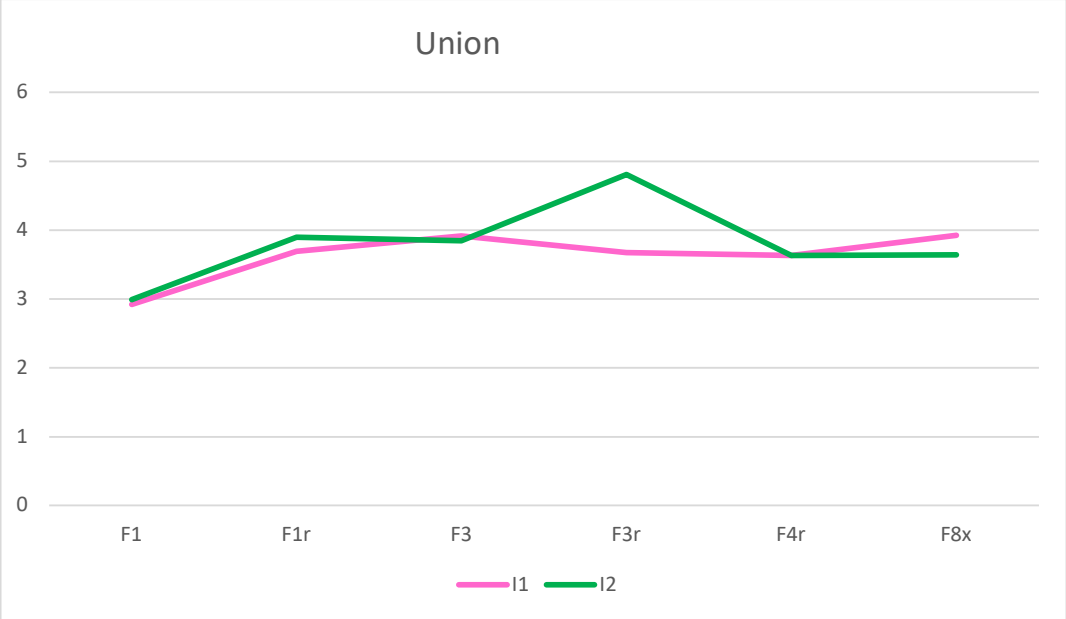

Gráfico 3: Comparação dos tempos de execução do algoritmo union da implementação 1 e 2

O gráfico 3 mostra que as diferenças entre as duas implementações são pequenas e praticamente constantes. Isso confirma que essa operação tem um custo O(n), pois basicamente envolve percorrer toda a estrutura e escrever no ficheiro. O tempo varia entre 3 e 5 segundos, refletindo o número de pontos diferentes presentes nos ficheiros.

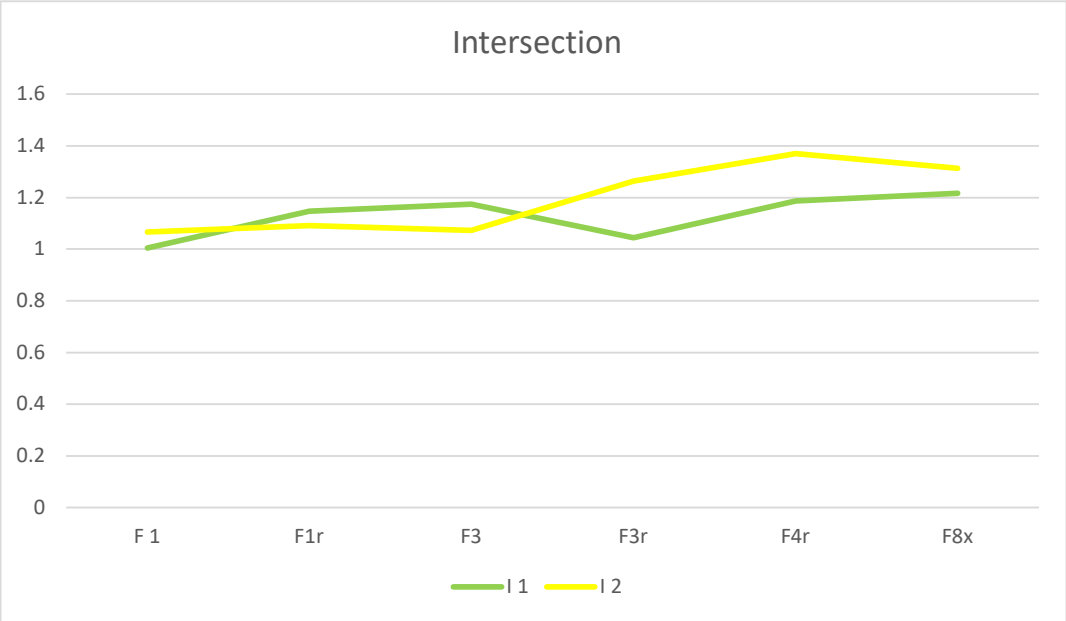

Gráfico 4: Comparação dos tempos de execução do algoritmo intersection da implementação 1 e 2

No gráfico 4, O tempo de execução para essa operação fica bastante estável e dentro do esperado. A tarefa de filtrar os pontos que pertencem a ambos os conjuntos tem um custo constante O (1) por ponto.

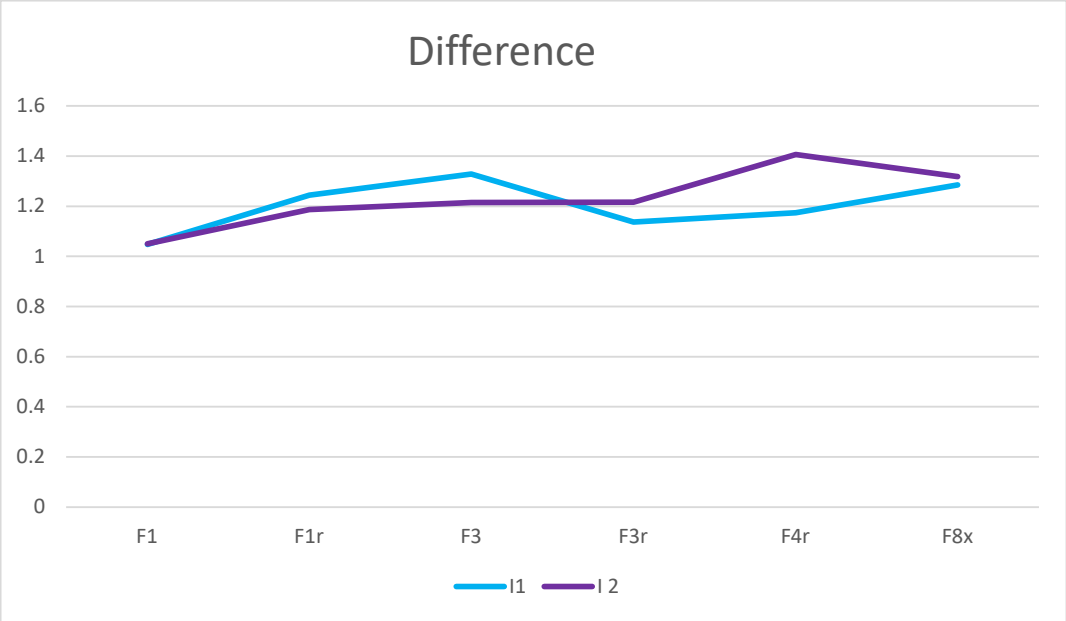

Gráfico 5: Comparação dos tempos de execução do algoritmo difference da implementação 1 e 2

Por fim, no gráfico 5 mostra o tempo para calcular a diferença que varia entre 1,0 e 1,4 segundos, crescendo de forma proporcional ao número de pontos nos ficheiros. Como a estrutura AEDHashMap usa listas ligadas para tratar colisões, elas podem influenciar um pouco o tempo de processamento. Por isso que no início a implementação 1 (linha azul) é mais rápida do que a implementação 2 (linha roxa).

Além disso, podemos perceber que os tempos para realizar as operações de união, interseção e diferença ficam sempre muito menores do que o tempo de carregamento dos dados (Load). Acontece isto porque as operações representam apenas uma pequena parte do custo total do processo. Isso também confirma que a divisão do programa em fases funciona bem e que a iteração sobre os mapas já preenchidos é bastante eficiente.
Embora os tempos das duas implementações sejam semelhantes nos ficheiros pequenos e médios, verifica-se que a AEDHashMap tende a ser mais eficiente em termos de memória e ligeiramente mais rápida em ficheiros com maior número de pontos, o que confirma a vantagem teórica da representação optimizada do valor.

Estes resultados validam a análise da secção 2.3 e confirmam que ambas as implementações apresentam um desempenho praticamente linear nas condições normais de execução, tal como esperado.

Nota: Cada experimento foi repetido 5 vezes para garantir maior precisão.

4. Conclusões

Neste trabalho, desenvolvi e testei duas implementações diferentes para resolver o problema denominado ProcessPointsCollections. Esta tarefa consiste em criar coleções de pontos no plano, realizando as operações de união, intersecção e diferença entre dois ficheiros que contêm esses pontos.

Na primeira implementação, optei por usar uma estrutura de dados padrão da biblioteca do Kotlin, um HashMap que associa cada ponto a uma lista de etiquetas que indica de qual ficheiro ele veio. Esta implementação foi rápida de montar e aproveitou as otimizações já presentes no Java/Kotlin, tornando as operações de acesso muito ágeis, com tempos praticamente constantes.

A segunda implementação construi a estrutura AEDHashMap (da questão 4 da parte 1 do trabalho), que armazena cada ponto ligado a um pequeno registo com dois booleanos, que indica a presença em cada ficheiro. Essa estratégia trouxe benefícios em relação ao uso de memória, pois o armazenamento é mais compacto, e oferece um controlo total sobre a dispersão dos dados, mantendo também tempos de acesso eficientes.

Nos testes que realizei com diferentes combinações de ficheiros, ficou bem claro que o tempo para carregar os dados aumenta de forma proporcional ao número de pontos únicos presentes nos ficheiros. As operações união, interseção e diferença têm custos constantes por elemento e se mantêm com tempos baixos, geralmente entre 1 e 5 segundos.

A versão do HashMap (implementação 1) foi um pouco mais rápida com ficheiros menores, enquanto a implementação com AEDHashMap (implementação 2) mostrou uma ótima escalabilidade, mantendo um bom desempenho mesmo com conjuntos maiores.

Esses resultados estão alinhados com o que esperávamos teoricamente e confirmam que as implementações adotadas são eficientes.

No fim, este trabalho desafiou-nos a refletir, analisar e dividir um problema em diversas partes de maneira a obter o programa ideal, isto é, buscar conhecimentos já obtidos e explorar os algoritmos mais apropriados para que o desempenho do programa não só executasse o desejado, mas que também fosse o mais estável e eficiente possível.

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.