ISEL
Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores

ALGORITMOS E ESTRUTURAS DE DADOS

2^o Série

Operações entre coleções de pontos no plano

51402 Gonçalo Silva

52537 Simão Almeida

52660 Rafael Carvalho

Turma: 22D

Docente: Maria Paula Graça

Algoritmos e Estruturas de Dados

2024 / 2025 verão

Conteúdo

1 Introdução

Neste problema foi-nos proposto desenvolver uma aplicação para realizar operações de união, interseção e diferença entre coleções de pontos no plano. Cada coleção de pontos está descrita num ficheiro de texto e cada ponto no ficheiro é descrito por um identificador de duas coordenadas: X e Y. Os principais objetivos da aplicação ProcessPointsCollections são: Receber como parâmetros dois ficheiros de texto (com extensão .co) Produzir um novo ficheiro (com extensão .co) que contenha os pontos, sem repetições, que ocorram em pelo menos um dos ficheiros de entrada Produzir um novo ficheiro (com extensão .co)que contenha os pontos comuns em ambos os ficheiros de entrada Produzir um novo ficheiro (com extensão .co) contendo os pontos únicos que existam em apenas um dos ficheiros de input

2 Operações entre coleções de pontos no plano

A secção 2.1 descreve o problema e a abordagem encontrada para o resolver. Na secção 2.2 apresentam-se as estruturas de dados utilizadas. Na secção 2.3 apresentamos uma análise lógica das funções relativas ao problema. Na secção 2.4 apresentamos a primeira e segunda implementações do problema.

2.1 Análise do problema

A aplicação deve ao iniciar carregar os documentos que têm a coleção de pontos para um único hash map estruturado de forma a permitir a consulta eficiente da mesma para responder às funções responsáveis por produzir os novos ficheiros de saída mencionados nos objetivos na secção 1.

2.2 Estruturas de Dados

2.2.1 Listas Ligadas

Listas ligadas (linked list) são estruturas de dados, sem tamanho fixo, ou seja, permitem inserções e remoções de elementos sem necessidade de realocação de memória e que se baseiam em sequências de zero ou mais elementos designados

por nós. Existem quatro tipos de listas ligadas: Lista ligada; Lista duplamente ligada; Lista duplamente ligada circular; Lista duplamente ligada circular com sentinela. A lista ligada simples é uma estrutura de dados linear composta por nós. Um nó contém um valor ou dado (key) e uma referência para o próximo nó da lista (next). A referência do primeiro nó é chamada de cabeça de lista (head), no caso de a lista estar vazia head = NULL pois não tem nenhum valor a apresentar e a lista termina quando o último nó também apresenta o valor NULL. O algoritmo List-Add insere o elemento desejado à cabeça da lista (1ª posição), este processo tem o custo de O(1) mesmo no pior caso. O algoritmo List-Contains tem como objetivo aceder a valores na lista o acesso, o que é feito através de uma pesquisa linear que vai procurar de elemento em elemento pelo primeiro elemento com a chave indicada e devolve a referência desse elemento. O algoritmo List-Remove serve para remover um elemento, para remover o elemento indicado este algoritmo usa o mesmo método de pesquisa que a função List-Contains. Pela forma como a pesquisa é feita, no pior caso, ambos os algoritmos têm custo de O(n).

Nas listas duplamente ligadas, cada elemento é composto não só pela key e pelo atributo next mas também por um atributo prev que como o nome indica armazena uma referência ao predecessor do elemento e a lista é identificada pela referência para a cabeça da lista. Os atributos next e prev funcionam da seguinte forma: dado um elemento x, x.next referencia o seu sucessor e x.prev o seu predecessor. Caso x.prev = NULL o elemento não tem nenhum elemento antes dele e portanto é cabeça da lista (head) e caso x.next = NULL não tem sucessor e portanto é o elemento final da cauda e por isso a cauda da lista (tail). Os algoritmos List-Contains e List-Add funcionam da mesma forma que numa lista ligada simples, no entanto o List-Remove funciona de uma forma um pouco diferente. O List-Remove retira o elemento da com a chave indicada da lista depois da procura, a diferença é que no elemento que precede o elemento a ser removido o .next se torna no .next do elemento removido e o .prev do elemento que sucede o elemento a ser removido passa a ser o .prev do elemento a ser removido. Estas mudanças não afetam o seu custo.

As listas circulares com sentinela usam um elemento dummy (sentinela) que é a cabeça da lista L.dummy e o elemento dummy tem todos os atributos dos outros nós. Numa lista circular, não existem referências a NULL pois são substituídas

por referências à sentinela, tanto com ligações à cabeça como à cauda da lista com L.dummy.next e L.dummy.prev respetivamente. O algoritmo List-Contains mantém-se praticamente igual usando a pesquisa com sentinela que iguala a chave à chave do dummy para evitar ciclos infinitos e procura o elemento desejado. O algoritmo List-Add funciona com o elemento a ser introduzido depois da sentinela e o algoritmo List-Remove é simplificado pois com a introdução da sentinela reduzem-se os casos especiais e evitam-se verificações de NULL.

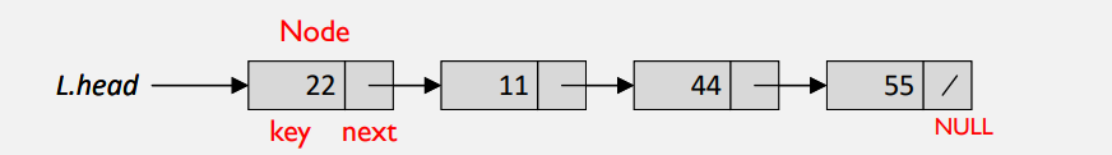

Figura 1: Lista ligada

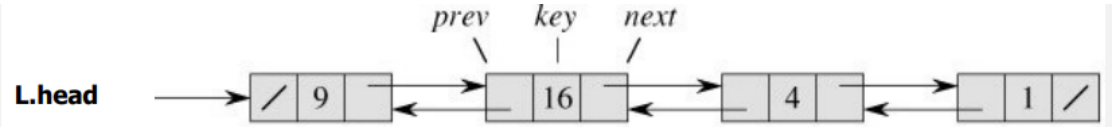

Figura 2: Lista duplamente ligada

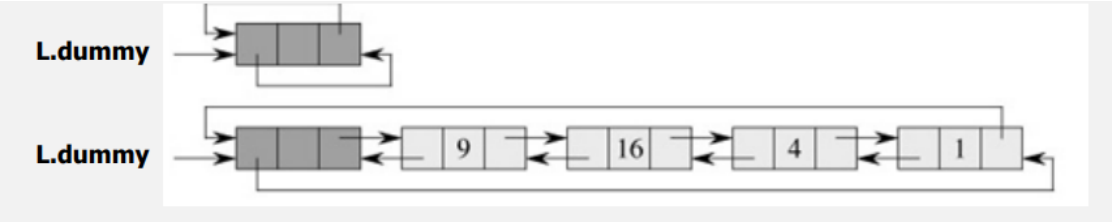

Figura 3: Lista duplamente ligada com sentinela

2.2.2 Hash Map

Um hashmap ou tabela de dispersão é uma estrutura de dados que nos permite armazenar de forma eficiente valores e chaves, este tipo de estrutura de dados é especialmente usada quando pretendemos inserir, remover ou procurar dados em tempo linear [O(1)]. O funcionamento do hashmap baseia-se numa função de dispersão (função de hash) que transforma uma chave num índice entre 0 e M-1,

sendo que M é o tamanho do array subjacente. O índice que resultar deste processo determina a posição onde o valor associado à chave será armazenado. Existe no entanto a possibilidade de chaves distintas resultarem no mesmo índice (colisão). Para resolver colisões existem dois processos possíveis: encadeamento externo (Separate Chaining) e encadeamento interno (Open Addressing). O encadeamento externo junta todos os elementos que colidam no mesmo endereço da tabela de dispersão e coloca-os na mesma lista ligada. Em cada índice do array da tabela de dispersão não armazena diretamente os valores mas sim uma referência/apontador para o início da lista ligada que contém os elementos que colidiram naquele índice. No encadeamento interno todos os elementos são introduzidos na tabela de dispersão. A tabela de dispersão neste caso contém em cada posição um elemento válido ou NULL caso não esteja nenhum valor nessa posição. Para arrumar os elementos nas posições do array há 3 técnicas: Linear Probing (Procura Linear); Quadratic Probing (Procura Quadrática) e Double Hashing (Dupla Dispersão). Caso a posição dada pela função de dispersão esteja já ocupada, segundo a procura linear, a chave é arrumada na próxima posição livre, procurando de uma posição em uma posição. Se usarmos a procura quadrática, procuramos uma nova posição que esteja livre através de uma função quadrática, primeiro na posição seguinte, depois duas posições depois da posição indicada pela chave, cinco posições, dez posições e por aí fora

Se optarmos pela dupla dispersão e a primeira função de dispersão dá-nos uma posição ocupada, calcula-se e aplica-se um deslocamento na segunda função de dispersão até que uma posição livre seja encontrada

À medida que o número de elementos cresce, o desempenho da estrutura deteriora-se. De modo a evitar essa deterioração da eficiência utiliza-se o conceito de load factor. Quando este ultrapassa um limite estabelecido o array é redimensionado e os elementos reorganizados, este processo é chamado rehashing. Um hashmap é uma estrutura de dados bastante eficiente mas não preserva a ordem de inserção dos elementos.

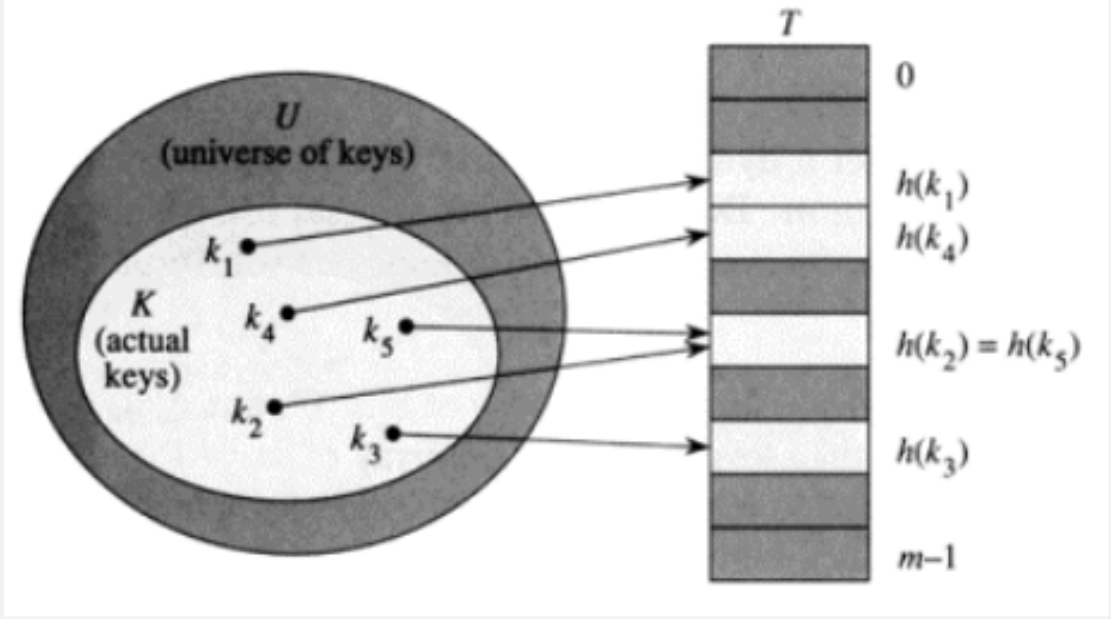

Figura 4: Tabela de Dispersão

2.3 Problema

Nesta secção, será realizada uma análise lógica das funções relativas ao problema. As funções reader, writer e a data class Point, são comuns a ambas as implementações.

2.3.1 reader

reader

```
fun reader(fileName: String): BufferedReader{
    return BufferedReader(FileReader(fileName))
}
```

Função para ler os ficheiros introduzidos, com recurso às bibliotecas java.io.FileReader e java.io.BufferedReader.

2.3.2 writer

.isEmpty()

```
fun writer(fileName: String): PrintWriter {
    return PrintWriter(fileName)
}
```

Função para escrever os ficheiros introduzidos, com recurso à biblioteca java.io.PrintWriter

2.3.3 data class Point

.offer(element:file): Boolean

```
data class Point (val x: Double, val y: Double)
}
```

Define os Points do Hash Map

As funções que serão descritas fazem parte do objeto ProcessPointsCollection1, relativo à Implementação I.

2.3.4 load(file1: String, file2: String)

load

```
override fun load(file1: String, file2: String) {
    // Inicializa a leitura dos dois ficheiros e carrega os pontos num HashMap

    val readfile1 = reader(file1)   // BufferedReader para o primeiro ficheiro
    val readfile2 = reader(file2)   // BufferedReader para o segundo ficheiro

    var line1: String? = readfile1.readLine()  // Lê a primeira linha do ficheiro 1
    var line2: String? = readfile2.readLine()  // Lê a primeira linha do ficheiro 2

    // Ignora linhas até encontrar uma que comece com "v" no ficheiro 1
    while ((line1 != null && !line1.startsWith("v"))) {
        line1 = readfile1.readLine()
    }

    // Ignora linhas até encontrar uma que comece com "v" no ficheiro 2
    while ((line2 != null && !line2.startsWith("v"))) {
        line2 = readfile2.readLine()
    }

    // Processa as linhas enquanto houver conteúdo nos ficheiros
    while (line1 != null || line2 != null) {

        // Processa linha do ficheiro 1 se começar com "v"
        if (line1 != null && line1.startsWith("v")) {
            val split = line1.split(' ') // Divide a linha em partes

            // Verifica se a linha tem o formato esperado e extrai coordenadas
            if (split.size == 4 && split[0] == "v") {
                val x = split[2].toDoubleOrNull() // Converte coordenada x para Double
                val y = split[3].toDoubleOrNull() // Converte coordenada y para Double

                if (x != null && y != null) {
                    val point = Point(x, y) // Cria um Point com as coordenadas

                    // Atualiza o HashMap com o ponto e a fonte (FILE1, FILE2 ou BOTH)
                    HashMap[point] = when (HashMap[point]) {
                        Source.FILE2 -> Source.BOTH
                        else -> Source.FILE1
                    }
                }
            }
            line1 = readfile1.readLine() // Lê a próxima linha do ficheiro 1
        }

        // Processa linha do ficheiro 2 se começar com "v"
        if (line2 != null && line2.startsWith("v")) {
            val split = line2.split(' ') // Divide a linha em partes

            // Verifica se a linha tem o formato esperado e extrai coordenadas
            if (split.size == 4 && split[0] == "v") {
                val x = split[2].toDoubleOrNull() // Converte coordenada x para Double
                val y = split[3].toDoubleOrNull() // Converte coordenada y para Double

                if (x != null && y != null) {
                    val point = Point(x, y) // Cria um Point com as coordenadas

                    // Atualiza o HashMap com o ponto e a fonte (FILE1, FILE2 ou BOTH)
                    HashMap[point] = when (HashMap[point]) {
                        Source.FILE1 -> Source.BOTH
                        else -> Source.FILE2
                    }
                }
            }
            line2 = readfile2.readLine() // Lê a próxima linha do ficheiro 2
        }
    }

    readfile1.close() // Fecha o BufferedReader do ficheiro 1
    readfile2.close() // Fecha o BufferedReader do ficheiro 2
}
```

Ao inicializarmos o Hash Map, carregamos os pontos de cada file após o "v". Depois, começamos a leitura de cada linha dos Hash Maps originais presentes em cada file;

Se encontrar uma linha válida no file, extrai as coordenadas 2 e 3, (números depois do "v"), cria um Point com as coordenadas lidas e atualiza o Hash Map mediante as condições: Caso já esteja no ficheiro oposto, atualiza para BOTH, caso contrário marca como o file.

2.3.5 union(output: String)

union(output: String)

```
override fun union(output: String) {
    // Cria um ficheiro de saída com todos os pontos (união dos dois ficheiros)

    val union = writer(output)  // Cria um PrintWriter para o ficheiro output

    for (point in HashMap) {
        // Escreve as coordenadas de cada ponto no ficheiro
        union.println("${point.key.x}, ${point.key.y}")
    }

    union.close()  // Fecha o PrintWriter
}
```

Permite produzir um novo ficheiro (com extensão .co) contendo os pontos, sem repetições, que ocorram em pelo menos um dos ficheiros de entrada (operação union).

Complexidade: $O(n)$

2.3.6 intersection(output: String)

intersection(output: String)

```
override fun intersection(output: String) {
    // Cria um ficheiro com os pontos que aparecem em ambos os ficheiros (interseção)

    val intersection = writer(output)  // Cria um PrintWriter para o ficheiro output

    for (point in HashMap) {
        if (point.value == Source.BOTH) {
            // Escreve as coordenadas apenas dos pontos comuns a ambos os ficheiros
            intersection.println("${point.key.x}, ${point.key.y}")
        }
    }

    intersection.close()  // Fecha o PrintWriter
}
```

Permite produzir um novo ficheiro (com extensão .co) contendo os pontos comuns entre ambos os ficheiros de entrada (operação intersection).

Complexidade: $O(n)$

2.3.7 difference(output: String)

difference(output: String)

```
override fun difference(output: String) {
    // Cria um ficheiro com os pontos únicos que aparecem apenas num dos ficheiros (diferença simétrica)

    val difference = writer(output)  // Cria um PrintWriter para o ficheiro output

    for (point in HashMap) {
        // Verifica se o ponto está só em FILE1 ou só em FILE2
        if ((point.value == Source.FILE1 && !(point.value == Source.FILE2)) ||
            (!(point.value == Source.FILE1) && (point.value == Source.FILE2))) {
            difference.println("${point.key.x}, ${point.key.y}") // Escreve o ponto no ficheiro
        }
    }

    difference.close()  // Fecha o PrintWriter
}
```

Permite produzir um novo ficheiro (com extensão .co) contendo os pontos únicos que estejam presentes apenas em um dos ficheiros de input (operação difference).

Complexidade: $O(n)$

Com isto, podemos observar que a Implementação I tem complexidade $O(n)$

As funções que serão descritas fazem parte do objeto ProcessPointsCollection2, relativo à Implementação II. Nesta segunda implementação, tínhamos de descrever o nosso próprio Hash Map, proviniente da questão I.4.

2.3.8 class AEDHashMap

class AEDHashMap

```
class AEDHashMap<K, V>(initialCapacity: Int = 16, val loadFactor: Float = 0.75f) : MutableMap<K, V>
```

A classe genérica de Hash Map com os tipo K para a chava e V para o valor. Os parâmetros passados sãp initialCapacity, que representa o tamanho inicial do array hash, e loadFacto, que representa o fator de carga para expandir a tabela.

2.3.9 HashNode<K, V>

HashNode<K, V>

```
class HashNode<K, V>(val key: K, var value: V, var next: HashNode<K, V>?)
```

Cada HashNode representa um par key-value. A classe também tem um ponteiro next implementado de modo a suportar colisões por meio de listas ligadas.

2.3.10 hash(HashCode: Int)

hash(HashCode: Int)

```
private fun hash(HashCode: Int): Int {
    // Calcula o índice do array para uma dada hashCode, garantindo número positivo e dentro do tamanho do array
    return HashCode.and(0x7fffffff) % table.size
}
```

Garante que o índice esteja dentro dos limites do array e que não seja negativo, através de and(0x7fffffff) para converter o hash num número positivo (elimina o bit de sinal).

2.3.11 get(key: K): V?

get(key: K): V?

```
// Busca o valor associado a uma chave, percorrendo a lista ligada na posição calculada
override fun get(key: K): V? {
    val idx = hash(key.hashCode())
    var currNode = table[idx]
    while (currNode != null) {
        if (currNode.key == key) {
            return currNode.value  // Retorna o valor se chave encontrada
        }
        currNode = currNode.next
    }
    return null  // Não encontrou chave
}
```

Recupera o valor associado a uma dada chave. Começa por calcular o índice onde a chave deveria estar, depois, percorre a lista ligada nessa posição. Se for encontrada uma chave igual à procurada, devolve-se o valor associado, caso contrário, devolve-se null.

Complexidade: $O(n)$

2.3.12 put(key: K, value: V): V?

put(key: K, value: V): V?

```
    // Insere ou atualiza um par chave-valor no mapa
    override fun put(key: K, value: V): V? {
        val index = (key.hashCode() and 0x7fffffff) % capacity  // Calcula índice positivo
        var node = table[index]

        // Verifica se a chave já existe e atualiza valor se for o caso
        while (node != null) {
            if (node.key == key) {
                val oldValue = node.value
                node.value = value
                return oldValue  // Retorna valor antigo
            }
            node = node.next
        }

        // Caso chave não exista, cria novo nó e insere no início da lista ligada
        val newNode = HashNode(key, value)
        newNode.next = table[index]
        table[index] = newNode
        size++  // Incrementa o tamanho

        // Verifica se precisa expandir a tabela
        if (size >= (capacity * loadFactor).toInt()) {
            expand()
        }

        return null  // Não havia valor antigo para retornar
    }
}
```

Adiciona uma nova entrada ou atualiza o valor de uma chave já existente. Se a chave já existir, atualiza o valor e devolve o antigo. Caso contrário, cria um novo HashNode e coloca-o no início da lista ligada correspondente, incrementa o size e, se este ultrapassar loadFactor*capacity, chama expand().

2.3.13 expand()

expand()

```
// Expande o array hash quando o fator de carga é ultrapassado, duplicando o tamanho e realocando os nós
private fun expand() {
    val oldTable = table
    capacity *= 2  // Dobra a capacidade
    table = arrayOfNulls(capacity)  // Novo array com capacidade aumentada

    // Reinsere todos os nós no novo array, recalculando o índice
    for (node in oldTable) {
        var currNode = node
        while (currNode != null) {
            val nextNode = currNode.next  // Guarda próximo nó antes de reinserir
            val idx = hash(currNode.hc)
            currNode.next = table[idx]
            table[idx] = currNode
            currNode = nextNode
        }
    }
}
```

Expande a capacidade da tabela. Recoloca todos os elementos da tabela antiga na nova (recalcula os índices com base no novo tamanho) e mantém a ordem relativa dos elementos.

Complexidade: $O(n)$

2.3.14 MapIterator:Iterator<AEDMutableMap.MutableEntry<K, V»)

MapIterator:Iterator<AEDMutableMap.MutableEntry<K, V»

```
inner class MapIterator : Iterator<AEDMutableMap.MutableEntry<K, V>> {
    var pos = 0  // Índice atual no array table
    var currNode: HashNode<K, V>? = null  // Nó atual a retornar
    var currNodeIter: HashNode<K, V>? = null  // Nó usado para iterar na lista ligada

    // Verifica se há próximo elemento no iterador
    override fun hasNext(): Boolean {
        if (currNode != null) return true  // Se currNode já está definido, há próximo
        while (pos < table.size) {  // Percorre o array até encontrar um nó
            if (currNodeIter == null) {
                currNodeIter = table[pos++]
            } else {
                currNode = currNodeIter
                currNodeIter = currNodeIter?.next
                return true
            }
        }
        return false  // Não há próximo elemento
    }

    // Retorna o próximo elemento do iterador
    override fun next(): AEDMutableMap.MutableEntry<K, V> {
        if (!hasNext()) throw NoSuchElementException()
        val toReturn = currNode
        currNode = null
        return toReturn!!
    }
}

// Cria um iterador para o mapa
override fun iterator(): Iterator<AEDMutableMap.MutableEntry<K, V>> {
    return MapIterator()
}
```

Percorre o mapa todos os elementos do map, ignorando potenciais posições vazias e avança pelas listas onde há colisão. Serve para ler todos os pares chave-valor de uma forma simples e organizada.

2.3.15 iterator()

iterator()

```
// Cria um iterador para o mapa
override fun iterator(): Iterator<AEDMutableMap.MutableEntry<K, V>> {
    return MapIterator()
}
```

Devolve um iterador que percorre todas as entradas no MutableMap.

As únicas alterações necessárias para a Implementação II, foi uma pequena mudança na função difference, de modo a respeitar a estrutura definida no AEDHashMap.

2.3.16 difference(output: String)

difference(output: String)

```
override fun difference(output: String) {
    // Cria um objeto writer para escrever no arquivo especificado por 'output'
    val difference = writer(output)

    // Itera sobre todas as entradas (pares chave-valor) no HashMap
    for (entry in HashMap) {
        // Verifica se o valor da entrada é igual a Source.FILE1 (provavelmente uma enum ou constante)
        if (entry.value == Source.FILE1) {
            // Escreve no arquivo a chave formatada com as propriedades 'x' e 'y' separadas por vírgula
            difference.println("${entry.key.x}, ${entry.key.y}")
        }
    }

    // Fecha o writer para garantir que o conteúdo seja salvo e liberar recursos
    difference.close()
}
```

Em vez de utilizarmos (point, source), que representam K, V no Hash Map do Kotlin, utilizamos entry, que representa K, V no AEDHashMap.

Com isto, podemos concluir que a Implementação II tem complexidade $O(n)$

3 Avaliação Experimental

Nesta secção serão realizadas as avaliações experimentais das nossas implementações. Estes testes serão realizados com três combinações de ficheiros, executados cinco vezes, para cada implementação. Os testes foram executados num computador com um Intel Core I7-10700F e 16Gb de DDR4 RAM.

Os tempos de execução entre implementações são semelhantes, devido ao facto dos algoritmos não apresentarem diferenças significativas entre si, e das suas complexidades serem ambas $O(n)$, algo que podemos observar com o facto do crescimento das funções (F7x e F8x têm menos elementos que F1r e F2r, e, F3 e F4 logo, o aumento é linear).

Dataset	Operação	Implementação I (ms)	Implementação II (ms)
F7x.co e F8x.co	Union	182.4	163.8
	Intersection	13.0	11.2
	Difference	72.4	64.2
F3.co e F4.co	Union	347.0	319.2
	Intersection	24.8	23.6
	Difference	120.2	111.4
F1r.co e F2r.co	Union	143.6	136.4
	Intersection	14.6	9.8
	Difference	73.8	66.4

Tabela 1: Tempos médios (ms) para cada operação (exceto `Load`) nas duas implementações

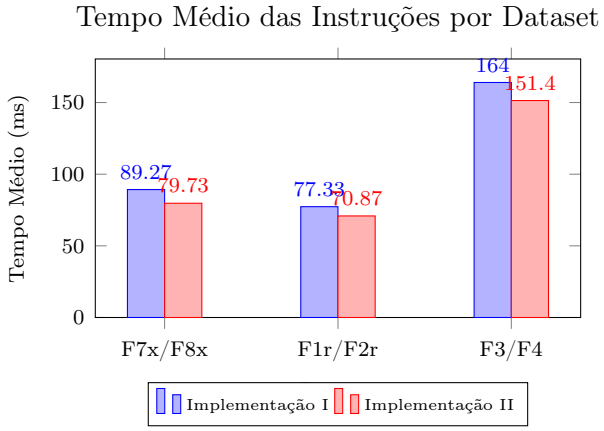

Figura 5: Comparação dos tempos médios entre as implementações, por dataset

4 Conclusão

O projeto permitiu aplicar conceitos fundamentais de estruturas de dados e algoritmos, em especial a utilização de listas ligas e tabelas de dispersão, para resolver um problema de operações entre coleções de pontos no plano. A implementação demonstrou ser funcional e escalável, permitindo lidar com ficheiros de dimensão arbitrária. Em suma, este trabalho proporcionou uma compreensão sólida dos mecanismos de gestão de tabelas de dispersão, reforçando a importância da escolha adequada das estruturas de dados em função dos requisitos de desempenho.

Também foi possível observar as diferentes formas de implementar uma tabela de dispersão, e as suas diferenças e semelhanças a um nível execucional.