Algoritmos e Estruturas de Dados

2ª Série

(Problema)

Operações com coleções de Pontos

52673 Guilherme Santos

52885 Daniel Viegas

53095 Francisco Inês

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2024/2025

18/05/2025

Índice

1. Introdução

Para o presente estudo, era pretendido desenvolver uma aplicação que fosse capaz de realizar operações entre coleções de pontos no plano, nomeadamente as operações de união, interseção e diferença. Cada coleção de pontos encontra-se descrita num ficheiro de texto onde cada ponto no ficheiro é composto por um identificador e duas coordenadas (X e Y).

Nas secções seguintes, vai ser apresentada uma descrição mais detalhada do problema e serão abordados os diversos aspetos importantes sobre a construção do algoritmo utilizado para a sua resolução.

2. Problema

Este capítulo apresenta todos os algoritmos e estruturas de dados utilizados na resolução do problema, assim como ambas as implementações solicitadas. Na secção 2.1, o problema é descrito com pormenor e define-se uma abordagem para a sua resolução. Na secção 2.2, são apresentadas as estruturas de dados utilizadas enquanto, em 2.3, os principais algoritmos. No último subcapítulo, encontram-se ambas as implementações da aplicação.

2.1 Análise do problema

O objetivo principal do projeto é o desenvolvimento de uma aplicação capaz de realizar as operações de união, interseção e diferença entre coleções de pontos no plano. As operações a realizar são escolhidas pelo utilizador, após este carregar para a aplicação duas coleções de pontos. As coleções de pontos encontram-se descritas em ficheiros de texto, cabe à aplicação a desenvolver extrair os pontos do ficheiro e originar um ficheiro final que contém resultado da operação solicitada. Para isso, assim que o utilizador carrega as coleções de pontos, todos os pontos são colocados numa tabela de dispersão onde, posteriormente, após indicação do utilizador sobre a operação a efetuar, os pontos são extraídos.

A aplicação, denominada de *ProcessPointsCollections,* deve apresentar as seguintes funcionalidades:

- Recebe como parâmetro dois ficheiros de texto (com extensão .co).
- 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*).
- Permite produzir um novo ficheiro (com extensão .co) contendo os pontos comuns entre ambos os ficheiros de entrada (operação *intersection*).
- Permite produzir um novo ficheiro (com extensão .co) contendo os pontos únicos que estejam presentes apenas em um dos ficheiros de input (operação *difference*).

Para iniciar a execução da aplicação *ProcessPointsCollections*, terá de se executar: *kotlin ProcessPointsCollections* e, durante a sua execução, a aplicação deve processar os seguintes comandos:

- *load* document1.co document2.co - Carrega os pontos dos dois ficheiros num único *hash map* ou tabela de dispersão, que deverá ser estruturado de forma a permitir a consulta eficiente para responder a todas as operações subsequentes.
- *union* output.co - Produz o ficheiro output.co contendo os pontos presentes em pelo menos um dos ficheiros de entrada, sem repetições.
- *intersection* output.co - Produz o ficheiro output.co contendo os pontos que ocorrem em ambos os ficheiros de entrada, sem repetições.
- *difference* output.co - Produz o ficheiro output.co contendo os pontos que ocorrem no ficheiro document1.co mas que não ocorram no ficheiro document2.co, sem repetições.
- exit - Termina a aplicação.

Por fim, requerem-se duas implementações deste algoritmo, uma utilizando estruturas presentes na *Kotlin Standard Library* e outra com estruturas construídas por nós.

2.2 Estruturas de Dados

2.2.1 - Tabela de Dispersão

A principal estrutura de dados utilizada neste projeto foram as tabelas de dispersão (ou *hash maps*). Uma tabela de dispersão é uma estrutura que permite associar chaves a valores, sendo particularmente eficiente em operações de inserção, remoção e pesquisa de elementos. As chaves são processadas por uma função de dispersão (ou função *hash*), que converte cada chave num índice numérico. Esse índice determina a posição num array interno onde o valor correspondente será armazenado. A Figura 1 ilustra um exemplo da construção de uma tabela de dispersão.

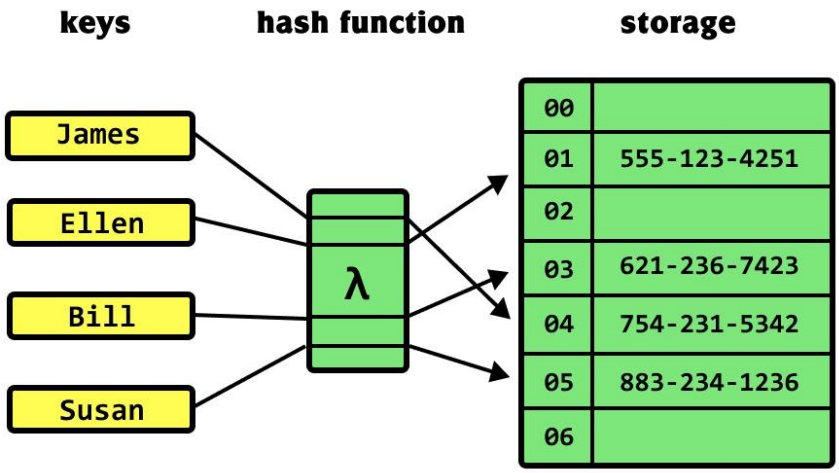

Figura 1: Exemplo de uma tabela de dispersão.

Nesta Figura, os nomes (*keys*) passam pela função de dispersão (*hash function)*, que retorna o índice numérico da tabela onde vão ser colocados os números telefónicos correspondentes (*values*). Embora no exemplo apresentado (na Figura) cada nome corresponda a um índice distinto, pode ocorrer que chaves diferentes sejam associadas ao mesmo índice. A estes casos chamamos colisões. Existem várias estratégias para lidar com colisões. Neste projeto, foi utilizado o método de encadeamento externo (*separate chaining)*,

onde todos os elementos que colidem no mesmo endereço da tabela são armazenados numa lista ligada. A posição endereçada na tabela aponta para a cabeça dessa lista, permitindo manter os dados organizados e garantindo operações eficientes. Mesmo em situações de colisão, a pesquisa continua a ser simples desde que a função *hash* seja bem construída.
Esta estrutura de dados é especial dada a sua eficiência na execução das operações. Um dos objetivos na construção de uma tabela de dispersão é certificar que as operações realizadas com recurso à tabela assumam complexidade constante O(1), na maioria dos casos.

2.3 Algoritmos e análise da complexidade

Neste subcapítulo serão analisados todos os algoritmos auxiliares utilizados na resolução do problema e as suas respetivas complexidades (em termos de tempo e espaço). Vai ser ainda abordada a forma como estes algoritmos afetam as estruturas indicadas no ponto anterior.

2.3.1 - *HashMap*

Este algoritmo é responsável pela criação da tabela de dispersão. Tem como entradas o valor da capacidade da tabela (inteiro) e o seu fator de carga (decimal). O fator de carga mede a relação entre o número de elementos armazenados e a capacidade total da tabela. Valores elevados deste fator levam a um maior número de colisões e, consequentemente, a pesquisas ineficientes. Por outro lado, valores muito baixos levam ao desperdício de espaço. Idealmente, o valor deste fator deve estar próximo de 0.75. Quando o número de elementos for superior à capacidade da tabela multiplicada pelo fator de carga, a tabela deve ser expandida, duplicando o valor da sua capacidade e reorganizando os seus elementos (garantindo escalabilidade à estrutura).

-> **Interface Genérica Utilizada:**

Para garantir que a tabela de dispersão pode operar com qualquer tipo de chave e valor, é utilizada uma interface genérica:

```
interface AEDMutableMap<K,V> : Iterable<AEDMutableMap.MutableEntry<K, V>> {
  interface MutableEntry<K, V>{
    val key: K
    var value: V
```

```
        fun setValue(newValue: V): V
    }
    val size: Int
    val capacity: Int
    operator fun get(key: K): V?
    fun put(key: K, value: V): V?
}
```

Esta interface define o comportamento e a estrutura do nosso algoritmo, incluindo os valores do tamanho (*size*) e capacidade da tabela (*capacity*) e as funções de inserção (*put*) e pesquisa (*get*). A interface interna *MutableEntry* representa um par chave-valor armazenado na tabela. Ela permite não só aceder à chave e ao valor, mas também alterar o valor associado à chave através da função *setValue*. Para a implementação da tabela, é ainda necessário um iterador, que permite percorrer todos os elementos de uma coleção, independentemente da sua implementação específica.

A implementação concreta do algoritmo começa por definir o valor inicial do número de elementos (*size*) a zero. Por omissão, a capacidade da tabela é definida a 13 (número primo, permitindo uma melhor distribuição dos elementos pela tabela), já o fator de carga é definido a 0.75.

-> **Estrutura de Nós (*HashNodes*):**

A tabela recorre ao método de encadeamento externo (listas ligadas) para a resolução de colisões, como mencionado anteriormente. Para isso, é definida a classe *HashNode*, cujas instâncias representam os nós da lista:

```
private class HashNode<K, V>(
        override val key:K,
        override var value: V,
        var next: HashNode<K, V>? = null
        ): AEDMutableMap.MutableEntry<K, V> {

            override fun setValue(newValue: V): V {
                value = newValue
                return value  }
        }
```

Cada *HashNode* armazena a chave (*key*), o valor associado (*value*) e uma referência para o próximo nó da lista (*next*).

-> **Inicialização da tabela:**

A tabela é representada internamente por um array de nós. Deve suportar o tipo nulo, dado que, inicialmente, a tabela de dispersão é preenchida com nulos, permitindo associar as listas a cada índice conforme o necessário.

```
private var hashTable = arrayOfNulls<HashNode<K, V>?>(capacity)
```

Esta abordagem torna a resolução de colisões eficiente e garante a escalabilidade da estrutura.

-> **Função de Dispersão (ou *hash function*):**

Esta função tem como objetivo transformar as chaves genéricas em índices do *array* interno da tabela, onde os dados são armazenados. Para isso, é aplicado à chave genérica o método *hashCode()* (presente na biblioteca do Kotlin), que origina um valor inteiro com base nesta chave. Em função de obter um índice válido do *array,* compreendido entre 0 e $capacidade - 1$, este algoritmo calcula o resto da divisão entre o resultado do método *hashCode()* e a capacidade da tabela e, ainda, aplica um método que garante que o valor resultante é positivo.

```
private fun hash(e: K): Int {
   val idx = e.hashCode() % capacity
   return if (idx < 0) idx + capacity else idx
}
```

Desta forma, é garantido que qualquer chave, independentemente do tipo ou valor, é convertida para um índice válido, adequado para o acesso eficiente ao *array*.

-> **Função de Pesquisa (get):**

Este algoritmo permite pesquisar o valor associado a uma chave. Primeiro, é calculado o índice do *array* interno correspondente a essa chave, através da função de dispersão. Com esse índice, acede-se à posição respetiva no *array* da tabela. Se essa posição estiver vazia (ou seja, não há nenhum nó na lista ligada), significa que essa chave não está presente na tabela e é devolvido *null*. Caso contrário, percorre-se a lista ligada dessa posição até ser encontrado o

nó com a chave correspondente. Se este for encontrado, é devolvido o valor associado à chave, se não, é devolvido *null*.

```
override operator fun get(key: K): V? {
   var node = hashTable[hash(key)]
   while(node != null){
      if (node.key == key) return node.value
      else node = node.next
   }
   return null
}
```

Esta abordagem tem complexidade média constante O(1), garantindo pesquisas na tabela eficientes, nos casos em que as colisões são mínimas.

-> **Função de Inserção (put):**

Esta função tem como parâmetros um par chave-valor e tem o objetivo de adicionar um novo elemento à tabela ou atualizar o valor associado a uma chave já existente. A função devolve ainda o valor anteriormente associado à chave. O algoritmo começa por calcular o índice correspondente à chave e obtém, através da função de pesquisa, o valor a ela atribuído, caso exista. Guarda-se ainda, o elemento da tabela que ocupa a posição do índice calculado. Seguem-se uma série de comparações que nos permite perceber se:

1. A posição da tabela correspondente ao índice da chave está vazia (ou seja, *node == null*). Neste caso, adiciona-se o par chave-valor recebido como argumento à tabela e é incrementado o número de elementos.

```
node == null -> {
        hashTable[idx] = HashNode(key, value, null)
        size++
}
```

2. A posição está ocupada, no entanto, não existe nenhum valor associado à chave (*node != null* e *oldValue == null*). Aqui, o par chave-valor é adicionado à lista ligada, fazendo referência ao nó existente nessa posição (é também incrementado o número de elementos).

```
        oldValue == null -> {
    hashTable[idx] = HashNode(key, value, hashTable[idx])
```

```
            size++
        }
```

3. A posição está ocupada e há um valor já associado à chave. Nesta condição, é percorrida a lista ligada dessa posição até ser encontrado o nó que armazena a chave correspondente. De seguida, troca-se o valor armazenado no nó com o valor recebido como argumento da função.

```
    else -> {
        while (node?.key != key) {
                node = node?.next
        }
        node?.setValue(value)
        return oldValue
        }
```

Aqui, já não é necessário incrementar o valor do número de elementos dado que não foi adicionado nenhum elemento, houve apenas uma troca no valor associado à chave.

Após estas condições, caso tenha sido adicionado um novo elemento, faz-se a verificação de expansão da tabela. Se o número de elementos ultrapassar o limite definido pelo fator de carga, então, a tabela é redimensionada, de acordo com as condições apresentadas anteriormente.

```
        if (size * loadFactor >= capacity) expand()
                return oldValue
```

Por fim, é retornado o valor anteriormente associado à chave.

-> **Função de Expansão:**

Quando o número de elementos ultrapassa o limite definido pelo fator de carga, é necessário a expansão da tabela de dispersão. A expansão da tabela consiste em:

1. Dobrar o valor da capacidade;
2. Inicializar um novo *array* interno (nova tabela) com o novo valor de capacidade;
3. Reinserir os elementos da antiga tabela na nova, recalculando os índices de cada elemento através da função de dispersão, pois o valor da capacidade mudou.
4. Substituir a tabela antiga pela nova.

```
private fun expand() {
    val oldTable = hashTable
    capacity *= 2
    hashTable = arrayOfNulls<HashNode<K, V>>(capacity)
    size = 0 // Redefinido o tamanho da lista porque a função put incrementa-o cada vez que
adiciona um elemento (O tamanho no fim é o mesmo)
    for (point in oldTable) {
        var element = point
        while (element != null) { // Percorre os nós da lista ligada
            put(element.key, element.value)
            element = element.next
        }
    }
}
```

A reinserção dos elementos na nova tabela é feita através da função *put()*, garantindo que os elementos são colocados nos índices corretos.

Esta função apresenta complexidade linear de O(n), dado que é necessário percorrer todos os elementos da tabela para os reorganizar.

-> **Iterador:**

O iterador vai ser responsável por permitir percorrer sequencialmente todos os elementos da tabela de dispersão, independentemente da sua posição ou de haver colisões. Para implementar o iterador, recorre-se à interface *Iterator()*, incluída nas bibliotecas do Kotlin. Esta interface contém as funções *hasNext()* (verifica se há um elemento seguinte a visitar) e *next()* (visita o elemento seguinte). Em código, a lógica do iterador passa por verificar se há uma posição seguinte a ser visitada e, se esse for o caso, permite à função *next()* visitar essa posição. Em termos práticos e no contexto das tabelas de dispersão, o iterador consiste em:

1. Percorrer o *array* interno da tabela;
2. Em cada posição, se estiver vazia, ignora-a, caso contrário, se tiver uma lista ligada (por conta das colisões), percorre a lista toda;
3. Vai devolvendo os nós com os elementos chave-valor, um por um.

O algoritmo do iterador tem o seguinte formato:

```
    private inner class MyIterator: Iterator<AEDMutableMap.MutableEntry<K, V>> {
        var currIdx = -1 // índice atual do array interno da tabela
        var currNode: HashNode<K, V>? = null // Nó da lista que vai ser retornado pelo next()
        var list: HashNode<K, V>? = null // Referência à lista ligada na posição atual (caso haja
colisão)

        override fun hasNext(): Boolean {
                if (currNode != null) return true
                while (currIdx < capacity) {
                        if (list == null) {
                                currIdx++
                        if (currIdx < capacity) list = hashTable[currIdx]
                } else {
                        currNode = list
                        list?.let { l -> list = l.next }
                        return true
                        }
                }
          return false
    }

        override fun next(): AEDMutableMap.MutableEntry<K, V> {
                if (!hasNext()) throw NoSuchElementException()
                val aux = currNode
                currNode = null
                return aux!!
                }
         }
```

Se *hasNext()* indicar que não há mais elementos, a função *next()* lança uma exceção. Caso contrário, *next()* devolve o elemento atual e define o *currNode* como null, indicando que já visitou essa posição. Este iterador permite que a tabela seja percorrida de forma eficiente mesmo em situações de colisão.

Todos os algoritmos mencionados neste subcapítulo estão disponíveis no projeto do GitHub, devidamente estruturados e comentados.

2.4 Implementações do Problema

2.4.1 - Implementação 1

Na primeira implementação, foi exigida a construção de uma solução baseada unicamente nas estruturas da *Kotlin Standard Library*. Isto significa que a tabela de dispersão foi inicializada com recurso às coleções fornecidas por essa biblioteca, e não às estruturas desenvolvidas anteriormente. É definida ainda a classe *Point*, que representa os pontos no plano. De seguida, apresentam-se as funções que permitem a execução das operações descritas no subcapítulo 2.1.

-> Inicialização do mapa:

Na tabela de dispersão utilizada para a resolução do problema as chaves são do tipo *Point*, já o seu valor associado é um *array* de *Booleans*.

- private var hashMap = HashMap<Point, Array<Boolean>>()

Caso a primeira posição do *array* seja *true*, significa que o ponto (chave) associado está presente no primeiro ficheiro. Se a segunda posição do *array* for *true*, então o ponto pertence ao segundo ficheiro. Pode acontecer de um ponto estar contido em ambos os ficheiros e, nesse caso, ambas as posições são definidas a *true*.

A ideia é que, à medida que os pontos são adicionados ao mapa, o *array* associado a cada ponto é atualizado. Esta é uma abordagem que visa sacrificar mais espaço de memória em função de ter um algoritmo mais eficiente. É criado um *array* para cada elemento da tabela, no entanto, as operações posteriores a realizar tornam-se mais simples, visto que é possível distinguir de onde foram extraídos os pontos diretamente do valor associado a cada um.

-> Operação Load:

A operação *load* tem como objetivo carregar, para uma tabela de dispersão, todos os pontos de ambos os ficheiros de entrada que a função recebe como argumentos. Como mencionado anteriormente, cada ponto é usado como chave e o seu valor associado corresponde a um *array* de *Booleans*, permitindo posteriormente identificar a origem de cada ponto. Segue-se o seguinte procedimento:

1. Ler cada ficheiro linha a linha, extraindo as coordenadas de cada ponto (assume-se que cada linha tem um único ponto).

2. Guardar o valor previamente associado à chave (ponto), permitindo perceber a sua origem caso já exista na tabela.

3. Criar um objeto *Point* com as coordenadas extraídas e associar-lhe o valor do *array* de *Booleans* (obtido ao analisar o valor anteriormente associado).

4. Inserir o ponto na tabela de dispersão através da função de inserção.

Esta abordagem permite construir uma estrutura que contém todos os pontos únicos e, em paralelo, identificar de forma eficiente os pontos comuns entre os dois ficheiros. A implementação concreta desta função tem o seguinte aspeto:

```
fun load(file1: String, file2: String){
   val reader1 = createReader(file1) // Abre os leitores dos ficheiros de entrada
   val reader2 = createReader(file2)
   var file1Lines = reader1.readLine() // Lê uma linha do ficheiro (file1)
   var file2Lines = reader2.readLine() // Lê uma linha do ficheiro (file2)
   // Lê continuamente as linhas dos ficheiros até chegar aos pontos (Salta a parte dos comentários dos ficheiros de teste)
   while (file1Lines.split(' ')[0] != "v" && file2Lines.split(' ')[0] != "v"){
      file1Lines = reader1.readLine()
      file2Lines = reader2.readLine()
      continue
   }
   // Caso a leitura das linhas do ficheiro resulte num "null", significa que o ficheiro não tem mais linhas para serem lidas.
   while (file1Lines != null || file2Lines != null) {
      if (file1Lines != null) {
         val point = getPoint(file1Lines) // Extrai o ponto da linha
         val oldValue = hashMap[point] // Guarda o valor anteriormente associado ao ponto (permite perceber a sua origem caso já exista na tabela)

         if (oldValue != null && !oldValue[0]) hashMap.put(point, arrayOf(true, true)) // Ponto está presente na tabela e no outro ficheiro
         else if (oldValue == null) hashMap.put(point, arrayOf(true, false)) // Ponto ainda não está na tabela
```

```
            file1Lines = reader1.readLine() // Lê a próxima linha
        }
        if (file2Lines != null) { // O procedimento é igual ao do primeiro ficheiro (em cima)
            val point = getPoint(file2Lines)
            val oldValue = hashMap[point]

            if (oldValue != null && !oldValue[1]) hashMap.put(point, arrayOf(true, true))
            else if (oldValue == null) hashMap.put(point, arrayOf(false, true))

            file2Lines = reader2.readLine() // Lê a próxima linha
        }
    }
    reader1.close() // Fecha os leitores dos ficheiros
    reader2.close()
}
```

Este algoritmo recorre a uma função auxiliar *getPoint*:

```
// Extrai o ponto da linha recebida como parâmetro.
private fun getPoint(line: String) = Point( line.split(' ')[2].toFloat(), line.split(' ')[3].toFloat())
```

A função *load* percorre todas as linhas de ambos os ficheiros para extrair os pontos, pelo que, assume complexidade linear $O(n)$.

-> Operação Union:

Esta operação tem o objetivo de escrever no ficheiro de output (dado como argumento) todos os pontos dos ficheiros de entrada, sem repetições. Para isso, basta transpor para esse ficheiro todos os pontos presentes na tabela de dispersão, já que esta contém todos os pontos de ambos os ficheiros de entrada, sem repetições.

```
fun union(file: String){
    val exitFile = createWriter(file) // Cria o ficheiro de output
    for (point in hashMap) {
        exitFile.println("${point.key.x} , ${point.key.y}") // Escreve cada um dos pontos
    }
```

```
    exitFile.close() // Fecha o ficheiro de output
}
```

Esta função tem complexidade linear O(n), dado que percorre toda a tabela de dispersão para escrever os pontos no ficheiro de saída.

-> Operação Intersection:

A operação *intersection* tem o objetivo de identificar e escrever no ficheiro de output (dado como argumento) todos os pontos que estão presentes, em simultâneo, em ambos os ficheiros de entrada. Para isso, percorre-se a tabela de dispersão e, para cada entrada, verifica os pontos cujo *array* de *Booleans* associado contém ambas as suas posições como *true*, o que indica que o ponto é comum aos dois ficheiros de entrada.

```
fun intersection(file: String){
        val exitFile = createWriter(file)
        for (point in hashMap) {
                if (point.value[0] && point.value[1])
                        exitFile.println("${point.key.x} , ${point.key.y}")
        }
  exitFile.close()
}
```

Este algoritmo assume complexidade linear $O(n)$, dado que percorre toda a tabela de dispersão para verificar se os pontos são comuns entre os ficheiros de entrada.

-> Operação Difference:

A operação *difference* pretende escrever, num ficheiro de saída (dado como argumento), todos os pontos presentes em apenas um dos ficheiros de entrada. Para tal, percorre-se a tabela de dispersão e, para cada entrada, verifica os pontos cujo *array* de *Booleans* associado contém apenas uma das suas posições como *true*, o que indica que o ponto está contido num único ficheiro de entrada. A verificação é feita através do operador lógico *xor*, que devolve `true` quando os valores são diferentes.

```
  fun difference(file: String){
      val exitFile = createWriter(file)
      for (point in hashMap) {
          if (point.value[0] xor point.value[1])
              exitFile.println("${point.key.x} , ${point.key.y}")
      }
      exitFile.close()
  }
}
```

Esta função tem complexidade linear $O(n)$, visto que é necessário percorrer toda a tabela de dispersão para verificar os pontos que são exclusivos a um dos ficheiros.

2.4.2 - Implementação 2

Na segunda implementação, era pedido que fosse implementada uma solução do problema com recurso às estruturas construídas por nós. O processo desta segunda implementação é muito similar ao da primeira implementação, com a única alteração de que, como agora é exigido que usemos as nossas estruturas, a inicialização da tabela de dispersão passa a ser dada por:

- `private var hashMap = AEDHashMap<Point, Array<Boolean>>()`

A lógica geral do algoritmo permanece a mesma à da primeira implementação. As operações *load*, *intersection*, *union* e *difference* continuam a utilizar a mesma estratégia. O uso da estrutura construída por nós pressupõe que esta implementa corretamente as operações fundamentais (*put*, *get* e *iterator*), garantindo o mesmo comportamento que a estrutura da biblioteca padrão. Assim, a complexidade dos algoritmos em ambas as implementações é a mesma.

3. Avaliação Experimental

O último passo do problema é avaliar experimentalmente as implementações dos algoritmos. Para isso, foram realizados testes com vários ficheiros de diferentes tamanhos de amostras de dados e registaram-se os valores dos tempos de execução dos algoritmos. Para cada combinação de ficheiros, repetiu-se o mesmo teste três vezes para cada implementação e fez-se uma média do tempo de execução do algoritmo, que foi colocada na tabela. Para a

realização da avaliação experimental, decidiu-se avaliar o tempo do algoritmo quando este executa todas as operações com as coleções de pontos. Estes testes foram executados numa máquina com processador *Intel Core i7-13700H* e 32G*B* de memória *RAM*. A Figura 2 apresenta uma tabela com os tempos de execução (em segundos) de ambas as implementações em função do tamanho da amostra de dados, assim como a respetiva ilustração gráfica.

	Tamanho do maior dos 2 ficheiros de entrada					
	12	70	89	121	290	--
Implementação 1	1,9	2,7	3,6	4,6	10	--
Implementação 2	2,1	3	3,7	4,7	10,5	--

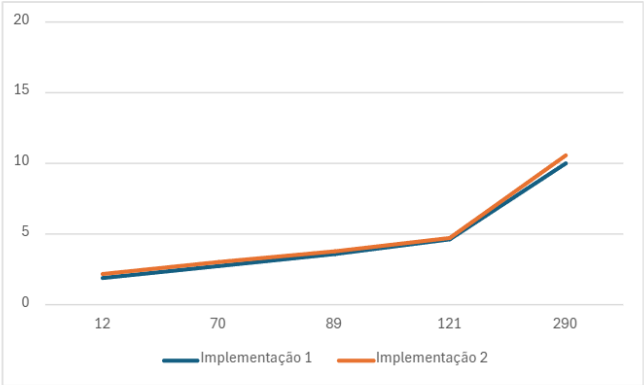

Figura 2: Resultados dos tempos de execução (em segundos) dos algoritmos considerando várias amostras.

Os resultados dos tempos de execução entre ambos os algoritmos não são assim tão díspares. Isto deve-se ao facto de ambas as implementações serem muito similares e também terem a mesma complexidade. O crescimento do tempo de execução dos algoritmos acompanha a tendência esperada da sua complexidade $O(n)$ (visível através da curva do gráfico e dos valores da tabela). A semelhança nos tempos de execução das implementações sugere que a estrutura construída é eficiente, cumprindo com os objetivos traçados para a solução do problema.

4. Conclusões

Este projeto teve como objetivo a construção de um algoritmo, com base numa tabela de dispersão, que fosse capaz de realizar operações entre duas coleções de pontos no plano, entre elas, as operações de união, interseção e diferença. As operações a realizar são escolhidas pelo utilizador, após este carregar para a aplicação duas coleções de pontos. O algoritmo começa por juntar todos os pontos de ambas as coleções num mapa de dispersão. Cada coleção de pontos encontra-se descrita num ficheiro de texto onde cada ponto no ficheiro é composto por um identificador e duas coordenadas (X e Y). A aplicação desenvolvida é capaz de ler as linhas dos ficheiros e extrair os seus pontos para o mapa de dispersão. Posteriormente, o utilizador seleciona uma das operações, que é executada com eficiência. O algoritmo da aplicação foi projetado, numa fase inicial, através das estruturas presentes na biblioteca do Kotlin e, numa segunda fase, com estruturas construídas por nós. Por fim, foram feitas avaliações experimentais onde foram comparadas as estruturas implementadas por nós e as estruturas do *Kotlin*. Os resultados dos testes revelam que a estrutura construída está bastante próxima à estrutura do *Kotlin*, dado que ambas as implementações assumem tempos de execução similares.

Referências

[1] “Disciplina: Algoritmos e Estruturas de Dados - 2425SV,” Moodle 2024/25. [Online]. Available: https://2425moodle.isel.pt

[2] OpenAI, *ChatGPT* [versão GPT-4], 2025. [Online]. Available: https://chat.openai.com

Anexos

Todos os algoritmos encontram-se no projeto do GitHub.