1. Introducción

El problema de maniobras de trenes consiste en reordenar una secuencia dada de vagones para obtener una secuencia deseada, utilizando para ello una estación de maniobras con vías auxiliares. Este informe describe el proceso de generación de la secuencia de movimientos necesarios para realizar esta reordenación. La tarea se centra en la implementación del algoritmo definirManiobra, el cual, dada una configuración inicial y una configuración objetivo del tren, determina la secuencia óptima de movimientos para transformar el tren inicial en el objetivo.

2. Descripción del Sistema

El sistema de maniobras de trenes se compone de los siguientes elementos principales:

- **Tren:** Un tren se representa como una secuencia ordenada de vagones. En la implementación, un Tren se modela como una lista (List) de elementos de tipo Vagon. Un Vagon puede representar cualquier tipo de carga o identificador único.
- **Estado:** Un estado del sistema describe la ubicación de los vagones en la estación de maniobras en un momento dado. La estación consta de tres vías: la vía principal, la vía auxiliar uno y la vía auxiliar dos. Por lo tanto, un Estado se representa como una tupla de tres Trenes: (Tren_Principal, Tren_Via_Uno, Tren_Via_Dos).
- **Movimiento:** Un movimiento describe la acción de mover uno o varios vagones de una vía a otra. Se definen dos tipos de movimientos:
    - Uno(n): Mueve n vagones entre la vía principal y la vía auxiliar uno. Si n es positivo, los vagones se mueven de la vía principal a la vía uno. Si n es negativo, los vagones se mueven de la vía uno a la vía principal.
    - Dos(n): Mueve n vagones entre la vía principal y la vía auxiliar dos. De forma similar a Uno(n), un valor positivo de n indica movimiento de la vía principal a la vía dos, y un valor negativo indica movimiento de la vía dos a la vía principal.

Estos componentes se modelan en el código Scala de la siguiente manera:

```scala
object Modelos {
  type Vagon = Any
  type Tren = List[Vagon]
  type Estado = (Tren, Tren, Tren)

  sealed trait Movimiento
  case class Uno(n: Int) extends Movimiento
  case class Dos(n: Int) extends Movimiento
}
```

Representación matemática de los trenes y el estado del sistema:

S = (P, T)

P: conjunto de posiciones en la vía.

T = {t1,t2,...,tn} conjunto de trenes.

Cada tren:

$$t_i = (id_i, p_i, d_i)$$

3. Descripción del Algoritmo definirManiobra

El algoritmo definirManiobra es el núcleo de la solución para generar la secuencia de movimientos. Su propósito es tomar un tren inicial (t1) y un tren objetivo (t2) como entrada, y producir una lista de Movimiento que, al aplicarse al tren inicial, lo transformen en el tren objetivo.

El algoritmo se implementa mediante la función recursiva construir, la cual tiene la siguiente firma:

def construir(actual: Tren, objetivo: Tren, uno: Tren, dos: Tren, acc: Maniobra): Maniobra

Donde:

- actual: Representa el estado actual del tren en la vía principal.
- objetivo: Representa el estado deseado del tren en la vía principal.
- uno: Representa el estado de la vía auxiliar uno.
- dos: Representa el estado de la vía auxiliar dos.
- acc: Es un acumulador que almacena la lista de movimientos generados hasta el momento. Los movimientos se añaden al principio de la lista, por lo que al final se debe invertir el orden.

La función construir opera de la siguiente manera:

- **Caso Base:** case (Nil, Nil) => acc.reverse
    - Si tanto el tren actual como el tren objetivo están vacíos, esto significa que se ha logrado reordenar todos los vagones. La función retorna la lista de movimientos acumulados (acc) en orden inverso.

- **Caso Coincidencia:** case (aHead :: aTail, oHead :: oTail) if aHead == oHead => construir(aTail, oTail, uno, dos, acc)
    - Si los primeros vagones del tren actual (aHead) y el tren objetivo (oHead) son iguales, no es necesario realizar ningún movimiento. La función se llama recursivamente con las colas de los trenes (aTail y oTail) y el mismo acumulador (acc).
- **Caso General:** case _ => ...
    - Este caso se ejecuta cuando los primeros vagones del tren actual y el tren objetivo no coinciden, lo que requiere determinar y aplicar un movimiento.
    - Se utiliza un for comprehension para encontrar el próximo movimiento óptimo. Este for actúa como una secuencia de operaciones flatMap y map:
        - **Determinación del Movimiento:**
            - mov <- actual.headOption.map(_ => Uno(1)) orElse { ... }
                - Primero, se intenta mover el último vagón del tren actual a la vía uno (Uno(1)). Esto se prioriza para liberar vagones en la vía principal.
                - Si el tren actual está vacío (actual.headOption.map(_ => Uno(1)) retorna None), se consideran otros movimientos:
                    - Si la vía uno no está vacía y su primer vagón coincide con el primer vagón del tren objetivo, se mueve el primer vagón de la vía uno a la vía principal (Uno(-1)).
                    - Si la vía dos no está vacía y su primer vagón coincide con el primer vagón del tren objetivo, se mueve el primer vagón de la vía dos a la vía principal (Dos(-1)).
                    - Si la vía uno no está vacía y ninguno de los casos anteriores se cumple, se mueven todos los vagones de la vía uno a la vía dos (Dos(uno.length)). Esta opción se utiliza para liberar espacio en la vía uno.
                    - Si ninguna de las condiciones anteriores se cumple, no se puede realizar un movimiento válido y se retorna None.
        - **Actualización del Estado:**
            - nextState = mov match { ... }

- Una vez que se ha determinado el movimiento (mov), se calcula el nuevo estado del sistema (nextState) aplicando el movimiento al estado actual.
- Se consideran los diferentes tipos de movimiento (Uno(1), Uno(-1), Dos(-1), Dos(n)) y se actualizan las vías principal, uno y dos en consecuencia.

- **Llamada Recursiva:**
  - case Some((a, o, u, d, aMov)) => construir(a, o, u, d, aMov)
    - Si se encontró un movimiento válido (Some(...)), se llama recursivamente a construir con el nuevo estado (a, o, u, d) y el movimiento añadido al acumulador (aMov).
  - case None => Nil
    - Si no se encontró un movimiento válido (None), se retorna una lista vacía, indicando que no se pudo encontrar una solución.

Definición de movimientos y función de transición

$$M = \{\text{Avanzar}, \text{Retroceder}, \text{Parar}\}$$

$$\delta : S \times M \to S'$$

4. Ejemplos de Ejecución

Para ilustrar el funcionamiento del algoritmo, se presentan algunos ejemplos de ejecución. Estos ejemplos se basan en los casos de prueba definidos en TestTrenes.scala.

- **Ejemplo 1: Trenes Iguales**
  - Tren Inicial: List(1, 2, 3)
  - Tren Objetivo: List(1, 2, 3)

- Secuencia de Movimientos Generada: List() (Lista vacía)
- Explicación: Dado que los trenes inicial y objetivo son idénticos, no se requiere ningún movimiento. El algoritmo identifica esto en el caso de coincidencia y retorna una lista vacía.

Secuencia de movimientos por tren

$$\mathcal{M}_i = [m_{i,1}, m_{i,2}, \ldots, m_{i,k_i}]$$

$$\mathcal{S} = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n\}$$

- **Ejemplo 2: Reordenamiento Simple**
  - Tren Inicial: List(1, 3, 2)
  - Tren Objetivo: List(1, 2, 3)
  - Secuencia de Movimientos Generada: List(Uno(1), Uno(-1), Uno(1), Uno(-1))
  - Explicación:
    1. Mueve el último vagón (2) a la vía uno (Uno(1)).
    2. Mueve el vagón de la vía uno (2) a la vía principal (Uno(-1)).
    3. Mueve el último vagón (3) a la vía uno (Uno(1)).
    4. Mueve el vagón de la vía uno (3) a la vía principal (Uno(-1)).
- **Ejemplo 3: Caso Complejo (Tomado de un test)**
  - Tren Inicial: List(3, 6, 2, 5, 8, 4, 1, 9, 7, 10)
  - Tren Objetivo: List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  - Secuencia de Movimientos Generada: (Secuencia larga omitida por brevedad, pero se puede obtener ejecutando el código)
  - Explicación: Este ejemplo demuestra la capacidad del algoritmo para manejar reordenamientos más complejos, utilizando las vías auxiliares uno y dos para almacenar y mover vagones según sea necesario.

5. Consideraciones de Diseño

- **Recursión:** Se utilizó recursión para implementar el algoritmo construir debido a su naturaleza intrínsecamente recursiva. El problema de reordenar un tren se puede descomponer en subproblemas más pequeños de reordenar sub-trenes. La recursión permite expresar esta lógica de forma concisa y elegante.

- **Estructura de Datos para el Estado:** La tupla (Tren, Tren, Tren) se eligió para representar el estado del sistema porque modela directamente las tres vías de la estación de maniobras. Las listas (List[Vagon]) son adecuadas para representar los trenes porque mantienen el orden de los vagones, que es crucial para el problema.
- **Priorización de Movimientos:** La priorización de mover vagones a la vía uno (Uno(1)) se implementó para intentar liberar espacio en la vía principal lo más rápido posible. Esto puede mejorar la eficiencia del algoritmo en algunos casos, al reducir el tamaño del tren principal y facilitar la comparación con el tren objetivo.
- **Alternativas y Mejoras:**
    - **Recursión:** Elegimos usar recursión en la función construir porque se nos hizo que era la forma más natural de resolver el problema. Como que cada paso de mover los vagones se parece mucho al paso anterior, pero con menos vagones, la recursión nos permitió escribir un código más corto.
    - **Estructura de Datos para el Estado:** Para representar el estado del sistema, usamos una tupla de tres listas ((Tren, Tren, Tren)) porque así podíamos representar directamente las tres vías: la principal, la uno y la dos. Cada lista representa los vagones que hay en cada vía. Nos pareció sencillo y fácil de entender.
    - **Priorización de Movimientos:** Decidimos que el algoritmo primero intentara mover vagones a la vía uno (Uno(1)) porque pensamos que así íbamos a liberar más rápido la vía principal. Si la vía principal se vacía rápido, es más fácil comparar con el tren objetivo. No estamos seguros de si es la mejor forma, pero funcionó para los ejemplos que probamos.

      Cálculo del puntaje total

      $$P = \sum_{i=1}^{n} |\mathcal{M}_i|$$

    - También pensamos que podríamos haber usado otras formas de decidir qué movimiento hacer en cada paso. Pero la que usamos nos dio buenos resultados, así que la dejamos así.

6. Conclusiones

En este informe, se presentó el proceso de generación de la secuencia de movimientos para la maniobra de trenes. Se describió detalladamente el funcionamiento del algoritmo

definirManiobra, abarcando su estructura, lógica interna y comportamiento en diversos casos de ejecución.

El algoritmo implementado es capaz de generar una secuencia de movimientos efectiva para reordenar un tren inicial y transformarlo en un tren objetivo, utilizando las vías auxiliares de la estación de maniobras.

El desarrollo de este algoritmo permitió aplicar y consolidar conceptos fundamentales de la programación funcional, como la recursión y el uso de estructuras de datos inmutables. Estas técnicas resultaron ser herramientas valiosas para abordar un problema complejo de manera clara y concisa.