Aircraft Scheduling Problem (ASP) - Milestone 03

The **objective** of milestone three is to develop a code that performs scheduling with a **cost equal to or lower** than the one achieved in milestone one. To accomplish this, a **brute-force algorithm** has been employed. This algorithm exhaustively tests all possible combinations to find the most efficient arrangement of aircraft on runways. By exploring every potential solution, the algorithm ensures the identification of the optimal scheduling arrangement for the given scenario.

Classes

The classes are utilized to determine the time **separation** between aircraft and to specify the types of aircraft each runway can **accommodate**. An **enum** has been created with six classes to categorize them accordingly.

```
enum ClassType:
    case Class1, Class2, Class3, Class4, Class5, Class6
```

Separation

The **Separation** contains three methods: **delay**, **minTime**, **separation**. And that functions provides a way to calculate the minimum needed time to separate two aircrafts based on their classes categories and the present schedule aircraft arriving on a runway.

```
object Separation:

    def delay(a: Aircraft, r: Runway): Int =
        minTime(a, r) - a.target
```

```
    def minTime(a: Aircraft, r: Runway): Int =
        r.aircrafts.foldLeft(0)((minTime, scheduled) =>
            val time = scheduled.getTime + separation(scheduled, a)
            if time >= minTime then time else minTime
        )
```

```
    def separation(al: Aircraft, at: Aircraft): Int =
        val cl = al.classType
        val ct = at.classType
        separation(cl, ct)
```

```
    def separation(cl: ClassType, ct: ClassType): Int = (cl, ct) match
        case (Class1, Class1) => 82
        case (Class1, Class2) => 69
        case (Class1, Class3) => 60
        case (Class1, _)      => 75

        case (Class2, Class1) => 131
        case (Class2, Class2) => 69
        case (Class2, Class3) => 60
        case (Class2, _)      => 75

        case (Class3, Class1) => 196
        case (Class3, Class2) => 157
        case (Class3, Class3) => 96
        case (Class3, _)      => 75

        case (Class4, _) => 60
        case (Class5, _) => 60

        case (Class6, Class1 | Class2 | Class3) => 60
        case (Class6, Class4 | Class5) => 120
        case (Class6, _) => 90
```

Simple Types

The **Simple Types** were created to be used in classes in order to validate values at the moment they are created. In total, we have four Simple Types.

AircraftId: Aircraft Identifier.

```
type AircraftId = String

object AircraftId:
    def apply(id: String): Result[AircraftId] = Right(id)
```

RunwayId: Runway Identifier.

```
type RunwayId = String

object RunwayId:
    def apply(id: String): Result[RunwayId] = Right(id)
```

NonNegativeInt: Positive Numbers and Zero.

```
type NonNegativeInt = Int

object NonNegativeInt:
    def apply(n: Int): Result[NonNegativeInt] =
        if n > 0 then Right(n)
        else Left(NonNegativeIntError("Number must be greater then zero"))
```

PositiveInt: Positive Numbers Only.

```
type PositiveInt = Int

object PositiveInt:
    def apply(n: Int): Result[PositiveInt] =
        if n >= 0 then Right(n)
        else Left(PositiveIntError("Number must be positive"))
```

Domain

The domain encompasses **Agenda**, **Aircraft**, and **Runway**. The `final case class` was employed to construct these classes, with a focus on including **only the mandatory** fields in their respective constructors. As the algorithm executes, the remaining fields are populated accordingly. The extensions serve to define and **provide methods** for the classes, ensuring their functionality and enhancing their capabilities.

Agenda

```
final case class Agenda(
    aircrafts: Seq[Aircraft],
    runways: Seq[Runway]
)
```

Aircraft

```
final case class Aircraft(
    iid: AircraftId,
    classType: ClassType,
    target: NonNegativeInt,
    maxTime: PositiveInt,
    time: Option[NonNegativeInt]
)
```

Runway

```
final case class Runway(
    id: RunwayId,
    classes: Seq[ClassType],
    aircrafts: Seq[Aircraft]
)
```

Solution Algorithm

For this milestone, three main algorithms were used:

Brute Force Algorithm

The `bruteForce` method implements a brute force algorithm to find the optimal solution for aircraft scheduling. It performs an exhaustive search by testing all possible assignments of aircraft to runways, considering constraints to optimize or reduce the search space, and minimizing the total scheduling cost.

```
def bruteForce(la: Seq[Aircraft], lr: Seq[Runway]): Option[Seq[Runway]]
```

The brute force algorithm starts with a list of aircraft `la` and a list of runways `lr`. It iterates over each aircraft in the list, attempting to assign it to each available runway.

For each aircraft-to-runway assignment, the algorithm checks if it violates any constraints, such as the aircraft's operation time window or compatibility between aircraft type and runway. If the assignment does not violate any constraints, the algorithm updates the runway list with the assigned aircraft and recursively calls the `bruteForce` method for the remaining aircraft.

The algorithm continues attempting to assign aircraft to runways until all aircraft have been allocated or no valid assignment is possible. It returns the sequence of runways that achieves the minimum total scheduling cost among all possible combinations.

Combination Algorithm

The combination algorithm is used in the `createAllPossibilities` method to generate all possible combinations of aircraft scheduling for a specific runway.

```
def createAllPossibilities(r: Runway, a: Aircraft): Seq[Runway]
```

This algorithm utilizes the concept of partitions of already scheduled aircraft on a runway. It iterates over the partitions, which consist of a sequence of already assigned aircraft and a sequence of remaining aircraft.

For each partition, the algorithm assigns the current aircraft (the one being considered) to the sequence of already assigned aircraft, generating a new combination. This combination is added to the list of possible runways, provided it does not violate any constraints.

The algorithm continues iterating over all partitions and constructs all possible combinations of aircraft scheduling. It returns the resulting list of runways containing all generated combinations.

Recursive Algorithm

The `bruteForce` and `loop` methods utilize recursion to systematically traverse the lists of aircraft and runways.

The `bruteForce` method is primarily responsible for recursively calling the `loop` method for each aircraft in the list. It passes the remaining aircraft list and the updated runway list as arguments. The recursion occurs until all aircraft have been assigned or no valid assignment is possible.

`loop`, on the other hand, receives as an argument the list of aircraft groups and the list of runways. It iterates over the aircraft groups and recursively calls the `bruteForce` method for each group, passing the updated runway list as an argument. The recursion runs until all aircraft groups have been processed.

```
def loop(aircrafts: Seq[Seq[Aircraft]], lr: Seq[Runway]): Result[Seq[Runway]]
```

The aircraft are divided into groups of 10 aircraft each. For example, if the original aircraft list has 60 aircraft, there will be 6 groups of 10 aircraft. This approach efficiently reduces the execution time of the brute force algorithm, making it more efficient and accelerating the search for a solution.

These algorithms are combined to perform an exhaustive search of all possible scheduling combinations, aiming to find the optimal solution that minimizes the total scheduling cost.

Tests

Testing an application is necessary to assure the software's quality, reliability, and usability. They enable the identification and correction of flaws, defects, and functional failures, as well as the verification that the application satisfies the set criteria. Testing also aids in improving the user experience by finding usability issues and optimizing the interface and interaction. Furthermore, they ensure the integrity of the data handled by the application and ease the maintenance and continual evolution of the software, giving developers confidence when making changes to the code. In summary, testing is necessary to assure a high-quality and dependable product for end customers.

Unit Tests

Agenda Test

The test suite includes several test cases that verify the behavior of the Agenda class when adding aircrafts and runways to an agenda, checking for errors when adding duplicate aircraft or runways, creating an agenda from aircrafts and runways, and retrieving the correct number of aircrafts and runways from an agenda.

```
test("Creating an agenda from aircrafts and runways") {
  val aircraft1 = Aircraft(aircraftId1, Class1, target, maxTime, time)
  val aircraft2 = Aircraft(aircraftId2, Class2, target, maxTime, time)
  val runway1 = Runway(runwayId1, List(Class1, Class2), List())
  val runway2 = Runway(runwayId2, List(Class2, Class3), List())

  val result = Agenda.from(Seq(aircraft1, aircraft2), Seq(runway1, runway2))

  val expected =
    Right(Agenda(Seq(aircraft1, aircraft2), Seq(runway1, runway2)))
  assert(result == expected)
}
```

Aircraft Test

The tests cover a variety of circumstances, such as when the aircraft's current flight time exceeds the goal altitude, when the aircraft is of Class2 type, and when the aircraft is of an invalid class type.

```
test("delay should calculate the correct delay when time is greater than target") {
  val aircraft = Aircraft("A1", Class1, target, maxTime, Some(time))
  assert(aircraft.delay == delay)
}
```

```
test("cost should be calculated correctly for Class2") {
  val aircraft = Aircraft("A2", Class2, target, maxTime, Some(time))
  assert(aircraft.cost == cost)
}
```

Runway Test

The tests cover a variety of scenarios, including creating a runway with no aircrafts, adding an aircraft to a runway, assigning a single aircraft to a runway, and updating a runway in a sequence of runways.

```
test("Creating Runway with empty aircrafts should have zero cost") {
  val runway = Runway(runwayId1, List(Class1, Class2, Class3, Class4), List())
  assert(runway.cost == 0)
}
```

```
test("Runway is compatible with an aircraft") {
  val a = Aircraft(aircraftId1, Class4, target, maxTime, time)
  val runway = Runway(runwayId1, List(Class1, Class2, Class4), List())

  assert(runway.isCompatible(a))
}
```

Separation Test

This code offers a test suite for the Separation class, which computes the distance between two aircrafts as well as the shortest time required to land on a runway.

```
test("separation between aircrafts") {
  val aircraft1 = Aircraft(aircraftId1, Class1, target, maxTime)
  val aircraft2 = Aircraft(aircraftId2, Class2, target, maxTime)
  assert(Separation.separation(aircraft1, aircraft2) == 69)
}
```

Property-Based Tests

Properties are general principles that describe the expected behavior of a program. Unlike conventional test suites, properties operate at a more abstract level and generically describe the desired result of an operation. They allow automatically validating whether a program fulfills these properties in a variety of scenarios, generating test cases based on them. This helps identify unwanted behavior or errors that may occur in different situations, providing more comprehensive coverage. Properties are widely used in property-based testing to improve program reliability and robustness.

This project includes defining numerous properties and generators to evaluate a scheduling approach for landing and taking off aircraft on runways.

The project defines generators for random aircraft instances, as well as implements property-based tests for all generations. Within the project, the 'AttributeGenerator' file is created, in which generators for various attributes used in the properties are defined. These properties and generators are used in conjunction with a property testing library such as ScalaCheck to produce and validate inputs for the specified properties.

Generators

AttributeGenerator

- The `genAircraftId` and `genRunwayId` function generates an Identifier type by constructing an Identifier from a string of alphanumeric characters of a certain length. If the construction fails, so will the generator.
- The `genClassType` method creates a ClassType type by selecting a random byte value from a given range and attempting to construct a ClassType from that value. If the construction fails, so will the generator.
- The `genAircraftTarget` function build NonNegativeInteger type by selecting a random integer value within a certain range. If the construction fails, so will the generator.
- The `genClassRunway` method generates a list of ClassType types by selecting a random length for the list and then using the `genClassType` function to generate that many ClassType values.

Properties

AgendaProperties

- This code creates a Scala object called `AgendaProperties` with the property `GeneratedAgenda`. The ScalaCheck library is used by the property to produce random test data for the `genAgenda` function.
- The `genAgenda` function generates a list of runways, as well as a list of aircraft, which are sorted by target time. The property ensures that neither runways nor aircraft lists are empty.

AircraftProperties

- The `genAircraft` method generates random `Aircraft` instances by using generators for its characteristics such as Identifier, ClassType, target and maxDelaTime.
- The property function defines a property-based test that determines whether the ClassType of the created `Aircraft` instances matches one of the ClassType values defined in the `Runway` instances.
- The forAll function is used to perform the test, which generates random `Runway` and `Aircraft` objects and validates the property for each combination. The produced `Aircraft` instance for each test case is included in the test output.
- The `genAircrafts` generates a list of aircrafts, generating a list of unique identifiers, and then generating a sequence of aircrafts using each identifier and a list of runways.

RunwaysProperties

- We have a `genRunway` function that generates a random `Runway` object.
- The property function implements a property-based test that determines whether the created `Runway` object has a list of runway classes that is not empty.
- When the test is run, ScalaCheck will build many random `Runway` objects with the `genRunway` generator and see if the property is true for all of them. ScalaCheck will notice a failure and provide a counter example if the property for any created `Runway` object fails.

ScheduleProperties

> In a valid schedule, every aircraft was assigned a runway

- The property test ensures that every aircraft is allotted a runway in every valid timetable. It accomplishes this by generating random sets of runways and aircrafts, using the schedule function to generate a schedule, and then ensuring that every aircraft in the original set gets allocated to a runway in the resultant schedule.

> Each aircraft should be scheduled for a runway which can handle it

- It generates random runway and aircraft agendas, then schedules the planes on the runways. Finally, it determines if all scheduled aircrafts are assigned to a runway capable of handling their class type.

> An aircraft can never be scheduled before its “target time

- It generates random scenarios of runways and aircrafts using the `genAgenda` generator, schedules the aircrafts on the runways beginning at time 0, and then verifies if all scheduled aircrafts have a target time that is less than or equal to their actual planned time. The test fails if any aircraft is scheduled before its planned time.

> Two or more aircraft on a runway should never be assigned simultaneous times

- This code defines a property that checks whether two or more aircraft on a runway are assigned simultaneous times. It generates test cases using a generator called `genAgenda` and then verifies that the scheduled aircrafts on the runways do not have overlapping times.

Functional Tests

The code you gave is a test suite for a functional domain model related to schedule creation. Let's go through the tests one by one:

```
test("generatePlan should return an error for an invalid agenda")
```

`generatePlan` should generate the following plan for a valid agenda:

This test determines whether or not the `generatePlan` function generates the required plan for a valid agenda.

It generates an `Agenda` object that has a list of planes and runways.

The `generatePlan` function is then called with the agenda and the outcome is saved. Finally, it uses the assert statement to compare the result to the intended plan.

```
  test("generatePlan should return an error for an invalid agenda")
```

For an invalid agenda, `generatePlan` should return an error:

This test determines whether the `generatePlan` function gives an error for an incorrect agenda.

It generates an `Agenda` object with a list of aircrafts and runways, where one runway has competing class types.

The `generatePlan` function is run with the agenda, and the outcome is saved. The assert statement is used by the test to assert that the result is an error (Left).

```
test("createGroups should divide the aircraft list into groups of size 10")
```

The aircraft list should be divided into 10 groups by `createGroups`:

This test validates the behavior of the function `createGroups`

It generates a list of aircrafts with 15 entries.

The aircraft list is passed to the `createGroups` function, and the result is saved. Using assert statements, the test asserts the length of the groups list (2) and the lengths of the individual groups.

```
test("createPartitions should correctly partition the aircrafts")
```

`createPartitions` should partition the aircrafts correctly:

This test verifies the functionality of the `createPartitions` function.

It uses the aircraft list to invoke the `createPartitions` method.

The assert statement is used to compare the expected partitions with the actual partitions and to verify the length of the partitions list (4).

```
  test("bruteForce should find the optimal solution")
```

The best solution should be found through `bruteForce`:

This check guarantees that the `bruteForce` function finds the best option for scheduling planes on runways.

The aircraft and runway lists are passed to the `bruteForce` function, and the result is saved.

The expected outcome is a series of runways with allocated aircrafts. Using the assert command, the test affirms that the result matches the expected result.

```
    test("createAllPossibilities should generate all possible combinations")
```

All potential combinations should be generated with `createAllPossibilities`:

This test validates the behavior of the function `createAllPossibilities`. It generates a runway and an aircraft object.

The runway and aircraft are passed to the `createAllPossibilities` function, and the result is saved.

The predicted combinations are defined as a series of runways with various aircraft assignment combinations.

Using the assert statement, the test asserts that the result matches the intended combinations.

Experience and Learning

Milestone 03 exceeded our expectations as an evolution from Milestone 01, providing a valuable opportunity for us to learn more about functional programming and the Scala language. The issues highlighted by the instructors were addressed and corrected based on their guidance. The feedback from the instructors was essential in sparking our curiosity and motivating us to seek new approaches to solve the challenges.

Developing the brute force algorithm was a major challenge. While the algorithm itself is relatively simple, we faced difficulties when realizing the need to create strategies to reduce the time required to reach a solution. We had to find ways to decrease the multiple branches of possibilities generated by the algorithm. We implemented approaches such as eliminating branches that attempted to schedule aircraft on incompatible runways or when an aircraft exceeded its time window.

Overall, the development of this activity was laborious but extremely rewarding, as it significantly boosted our knowledge of functional programming and Scala.

Final Considerations

A brute-force approach was created in milestone three to solve the Aircraft Scheduling Problem (ASP) by exhaustively trying all potential permutations to identify the most effective arrangement of aircraft on runways. The algorithm assures that the optimal scheduling arrangement is identified at a cost equal to or lower than the one reached in milestone one.

The brute force algorithm, the combination algorithm, and the recursive algorithm comprise the solution algorithm. The bruteForce approach conducts an exhaustive search by assigning aircraft to runways while taking constraints into account and minimizing total scheduling cost. The createAllPossibilities method generates all conceivable aircraft schedule combinations for a given runway. The loop method executes the bruteForce method iteratively for each aircraft group, splitting the aircraft into smaller groups to shorten execution time.

Overall, milestone three met the goal of creating code that executes scheduling at a cost equivalent to or less than milestone one. The brute-force algorithm, which is supported by the defined classes and algorithms, exhaustively investigates all possible combinations in order to discover the best solution to the Aircraft Scheduling Problem.