KU Leuven

Project Report

A Java program for genome alignment

Author:
Marco Salmistraro

Teacher:
Guy Baele

Advanced Master's in Artificial Intelligence

Course in Java Programming - I0S71a

December 2022

Contents

Chapter 1

A Java Program for gene alignment

1.1 Aim of the project

The task of the project is to design a Java-based program for managing and manipulating text-based data related to genome information. It is assumed that the user be provided with two input files; the first one, `hiv.fasta`, contains genome information related to 100 different individuals. Each entry is characterized by a unique `String`-type id as well as a corresponding `String` sequence of 2500 nucleotides. A first task is to implement the possibility of creating standard-type alignments, whereby single genomes are output in a sequential fashion to allow for instance-to-instance comparison. Nucleotides end up aligning along the vertical axis for corresponding positions on single sequences. A second demand is to also provide a similar implementation for SNiP-type alignments. A single genome is taken as a reference; then, all the other sequences are compared to this instance: for each single position, if the nucleotide is equal to the corresponding one in the reference sequence a `.` character is output. The second input file, named `team.txt`, is employed to instantiate the structure of a team of operators working on alignments. Since one of the requirements of the project is to only read input files once, one of the first commands in the `Main.main()` method is the creation of a team based on the given genome and structure information.

1.2 Structure of the program

The program is developed by splitting functionalities into three different packages:

- `Main` This package includes the `Main` class to which high-level execution of the program is delegated (see `main()`). It also provides a method for accessing external input information by means of the `config.properties` file.
- `Alignments` This package provides implementations for the two different types of genomic alignments. The abstract class `Alignment` constitutes a template for the two typologies, `StdAlignment` and `SNiPAlignment`. Moreover, `Alignment` implements the `Functional` interface: such a decision makes it easier to extend the program in a logical way, granting the opportunity for adding further alignment types while enforcing the implementation of all necessary methods.
- `Team` This package groups together all classes deputed to the creation and management of teams as well as their corresponding members. From a logical standpoint, a team consists of different members; all members share certain characteristics, as they all inherit the same abstract `Member` class.

1.3 The `Main` package

This package only includes the `Main` class, called upon for running the user interface. The first method is the standard `main()` call that needs to be present in all Java programs. It is employed here to display the different functionalities of the system: input files are read, objects instantiated and later method outputs returned to the user. The second method in this class, `loadProperties()`, is employed to retrieve genome- and team-related information from the file locations specified by `config.properties`. As the method accommodates a `String` argument, a different container can be chosen to retrieve input data.

1.4 The `Alignments` package

At the core of this package is the abstract `Alignment` class. It provides a template for the implementation of different types of alignment by granting higher-level, shared functionalities (two types of constructors, as well as the `getGenomeLibrary()` and `getScore()` methods). The latter is specified in the `Functional` interface and directly implemented at this level: different alignments only offer alternative visualizations for a given dataset, but carry the same inherent information. All alignments are based on a `LinkedHashMap` structure, which apart from creating unique assignments between ids (the keys) and genomes (the values) also enforces the `keySet` to be ordered. This turns out to be fundamental whenever adding new genomes or modifying the existing library; indeed, a simple `HashMap` structure would not provide any iteration order guarantees when accessing keys. The `Functional` interface lists all the methods that need to be implemented by any `Alignment`-type sub-objects. In so doing, it stipulates a contract for all present and future instantiations of a genome alignment. In other words, an alignment can only be considered as such (and thereby exist within the program) if and only if it is capable of performing a specified set of actions, namely those corresponding to the interface's methods. Finally, the two remaining classes, `StdAlignment` and `SNiPAlignment`, being both sub-classes of the abstract `Alignment` class, derive all of its methods and inherit the necessity of equally implementing the `Functional` interface. Single alignments can be visualized by printing them through the overridden `toString()` method. This is only specified at the level of sub-classes, as gene visualization is the only characteristic differentiating the two alignments.

1.5 The `Team` package

The abstract `Member` class implements all basic methods that are shared across all of its sub-classes. As with all abstract classes, no direct instantiation is possible; instead, objects are created at the level of the concrete classes that inherit it. Apart from detailing a set of getter methods, `Member` also provides an implementation for `toString()`; this is directly inherited from the topmost parent `Object` class and applied to the different sub-classes by leveraging the concept of polymorphism. Three concrete classes inherit `Member`: `TeamLead`, `BioInfo` and `TechnicalSupport`. Each one of these specifies a different role within a team. Most importantly, bioinformaticians only have access to their own standard alignment, stored as an object variable; operating on it is only possible for the respective bioinformatician, as actions are implemented through the `BioInfo` class. As for team leaders, they are able to access the `Repository` object for the team, storing the optimal `StdAlignment` and `SNiPAlignment` attributes. Necessary operations for this user typology are designed so that direct access to the optimal alignment is not necessary, nor possible at all. Finally, technical support personnel is meant to operate on the team's repository as well as on every member's alignment by backing-up, erasing and restoring information, as specified by the corresponding methods. Members of a team are brought together by the `Team` class, which stores an `ArrayList` object of `Member`-type instances. The different attributes for a repository are retrievable through the `Team` class; this, however, does not impact the degree of access that different users are granted. As an example, `TeamLead` members are not able to directly visualize the optimal alignment for their team, but can still employ it to overwrite any of the user's data.

Chapter 2

Extending the program

2.1 Limitations and improvements

A first possible improvement to the program could consist in adding a further class layer: an overarching structure where a project leader would be in charge of directing a group of teams, thus having access to multiple work bodies at the same time. Secondly, it is worth discussing the way bioinformaticians are uniquely identified: the current implementation make use of data attributes from the single `Member` objects to build identifiers that are later used as `key` arguments in `Map`-type structures. The `id` variable is set as `final` in order to avoid any modifications on single strings. On one side this solution ensures that back-up file information can be exported once and then re-employed at multiple runtimes, as identifier creation is fully deterministic. On the other side, the `getId()` method might require revisiting in case the number of team member increases: different objects should avoid sharing the same key. The default implementation of the `hashCode()` might also be an option, although it would possibly lead to reference collision. A third alternative could consist in employing the `UUID` class instead, along with its `randomUUID()` method. This would result in the creation of randomly generated strings: identifiers would be granted to be different across all newly instantiated `Member` objects, but would get a new value every time the program is executed. The same could happen with the `hashCode()` method, although at a generally less frequent rate.

Project report written by
Marco Salmistraro