Trace comparison project

Final project report

Gwendal Carpy, Maxime Schwartz

September 11, 2013
# Contents

**Introduction**  
3

1 **Presentation of the project**  
3

2 **Different researches and choices**  
3  
2.1 **Existing trace comparisons and analysis**  
3  
2.2 **Link with DNA comparison algorithms**  
4  
2.3 **Choices**  
4

3 **Description of the different algorithms**  
4  
3.1 **General comparison algorithm**  
5  
3.2 **“Local Alignment algorithm”**  
7  
3.3 **Global alignment algorithm**  
7  
3.4 **Algorithmic complexity**  
8

4 **First implementations**  
9  
4.1 **Prof of concept**  
9  
4.2 **Features implemented**  
10  
4.3 **Optimization**  
11  
4.4 **Structure of the first implementation (UML graph)**  
11

5 **Adding a new trace operation**  
11  
5.1 **Description of the problem**  
12  
5.2 **Proposed solution**  
13

6 **Final structure of the application**  
15  
6.1 **Structure of the trace application**  
15  
6.2 **Details on the implementation of the algorithms**  
15  
6.3 **Structure for the domain specific parts of the application**  
16  
6.4 **A few pieces of information about the views**  
16

7 **Final version of the application**  
17  
7.1 **Available algorithms**  
17  
7.2 **Final Features**  
17  
7.3 **Analyze of the results**  
17

8 **Future Work**  
17  
8.1 **A view for the “Local Alignment algorithm”**  
17  
8.2 **Cuting of traces**  
17  
8.3 **Limit of the DNA approach**  
18  
8.4 **Improvement of the matrix visualization**  
18  
8.5 **Comparisons between the results of the algorithms**  
19

A **Modelling of the first implementation**  
20

B **How to analyze results**  
20  
B.1 **First analysis**  
21  
B.2 **Second analysis**  
22

**References**  
23
Introduction

This report gathers all the work done during our internship period, from the research of information to the implementation of a trace comparison tool for domain specific traces (execution traces of Domain Specific Languages, for more details see next section).

The aim of this report is to describe all the researches and the choices made. This permits to explain and justify the final result. We present the project (Chapter 1), the research of information and the choices made for the implementation (Chapter 2). Then, we describe the chosen algorithms (Chapter 3) and we quickly present the first implementations (Chapter 4), which test these algorithms. We finish by presenting the final application (Chapter 6), the modification made on the framework providing the operations needed to make trace applications (Chapter 5), the results (Chapter 7) and the work remaining to be done (Chapter 8).

1 Presentation of the project

Our project is a part of the COREF Project, dealing with Domain Specific Languages (DSLs), i.e. languages built with concepts and technologies of a specific domain, in order to be used to solve problems of this specific domain. The advantage of these languages is that they are more adapted for a domain specific use than general purpose languages. Users do not need to deal with low-level implementation and can more rapidly implement and solve their problems. More details on these languages and the issues they raise can be found in [7].

In our case, we only consider executable DSLs, in order to have execution data, and in particular execution traces. Execution traces are a succession of data coming from a specific execution of a program (traces can be different from one execution to another). More information about traces and the way to record them can be found at [5].

Since the idiom of using DSLs is to get rid of low-level implementation, DSL execution traces follow that. In execution traces, data is domain specific and aims to be directly and easily understandable by the user. Data inside these execution traces can be used for debugging or to understand behavior of programs for example.

The COREF Team is, amongst other things, working on a framework allowing to work with traces coming from execution of DSL programs (for more details see [6]). The framework provides operations (called trace operations) that can be combined to make applications working with DSL execution traces. The aim of our project is to create such a trace application, able to make comparisons on DSL executions traces.

As it is written above, execution traces are domain specific, so one of the difficulty is to clearly separate in our work domain specific features, which have to be implemented for each specific trace, and features that are general enough to be implemented directly in the comparisons application. It also means that the chosen algorithms must be able to separate these two parts, in order for the user, which will implement features for a specific trace, to avoid too much work.

2 Different researches and choices

2.1 Existing trace comparisons and analysis

We need to find a way to make comparisons on these domain specific traces. First, it is important to see if there are already existing ways to compare or more generally analyze traces.

We have learned that there are many different ways to make a trace recording with many different purposes. Sometimes, specific hardware is used for trace recording [1]. What concerns us is only a software implementation to record traces.
3 DESCRIPTION OF THE DIFFERENT ALGORITHMS

A few pieces of information can be found in relation with trace comparisons such as [8] and [4], but there are more things about general trace analysis [3]. These analyses are in general based on general program execution data, such as variable values, threads, number of a certain type of actions.

A specific application of these analyses is profiling: profilers make statistics and center the focus on program efficiency. However, they work with very general data, which is not the case of our domain specific traces.

2.2 Link with DNA comparison algorithms

A totally different domain, which use comparison algorithms, is genetics. Several algorithms have been developed to find similarities between two DNA sequences, and the way these algorithms are designed can have an interest for our domain specific traces [9].

Execution traces can be considered as sequences of objects, each object representing the state of the execution of a program at a certain moment. In one sequence, these objects have a similar structure, but from one execution trace to another, the structure can be different. Actually, these objects are the domain specific features.

If now we consider DNA sequences, they can also be seen as sequences of objects, where objects are, in our case, the amino acids. DNA sequences are comparable to a domain specific trace. Moreover, the fact that DNA suffers mutations (insertions, deletions and substitutions of amino acids) made people create algorithms designed to find similarities, differences and global correspondence between DNA sequences.

This philosophy can be useful in trace comparisons. Program executions are not deterministic: some random events could happen, or some different execution choices could create differences inside execution traces (sort of substitutions), but globally the program behaves similarly between executions and the execution traces have many similarities. DNA algorithms finding similarities can be used here to estimate correspondence between two execution traces.

2.3 Choices

According to the descriptions above and to the fact that DNA comparison algorithms are well described and developed, we have chosen to try to adapt these algorithms to domain specific execution traces.

3 Description of the different algorithms

All algorithms are based on the fact that the comparison between two amino acids gives a score, and the main parts of the algorithms use these scores to obtain results.

| Sequence 1 | A | T | G | C | A | A | C | G | T |
| Sequence 2 | A | T | C | G | A | G | T | G | A |

Score : 3 3 0 0 3 1 1 3 0

Figure 1: An example of comparison of amino acids.
In the figure 1, amino acids are represented by their first letter in their names: A, T, G and C, respectively for Adenine, Thymine, Guanine and Cytosine. The score function gives a score of 3 when two amino acids are the same, a score of 1 for A/G or C/T combinations, and a score of 0 for other combinations. This type of evaluation can be done to find similar sequences. If A can more easily be substituted with G than with T or C, it is interesting to give a small score of 1 to these combinations instead of 0.

If now we replace the amino acids by other objects, we have something comparable to a domain specific trace, and if we are able to compute scores between two of these objects, we are at the same stage with DNA and traces: the only remaining work is to use these scores to display similarities and differences. The following algorithms, used in genetics, are suitable for that purpose. People can find more information at the sources which inspired our work: [9], [2].

3.1 General comparison algorithm

The first interesting thing to do is to compare all amino acids (or trace parts) of one sequence with all the amino acids of the other sequence and collect all the results inside a matrix, named $M$. That allows to show the similarities and the differences between two DNA sequences or, by extension, between two DSL traces.

The algorithm needs results given by the evaluation function mentioned above. This evaluation function is DSL specific, i.e. it is totally independent from the general application and specific to each traces (result of the execution of the DSL). The evaluation function has to return a positive score, the result of the comparison of two amino acids (or, more generally, the comparison of two traces parts in the trace) against each other. This evaluation is the base of all algorithms used in the application.

The general comparison algorithm (or matrix point algorithm) has two versions: the “Two by Two” algorithm and the “Bloc” algorithm. These two algorithms differ about the way to fill the matrix $M$, which will be used later in the same way by the display. These two algorithms are described below:

3.1.1 Two by Two algorithm

This algorithm is the more simple version of the general comparison algorithm. It uses results of the evaluation function without any other transformations or computations.

Each point $i, j$ in the matrix $M$ contains the score returned by the comparison of the amino acid in the position $i$ in the first sequence and the amino acid in position $j$ in the second sequence. Some other information such as the maximum score and the total score in the matrix are also collected (they will be used later for the display of the matrix).

This algorithm is quite primitive. It shouldn’t be used to visualize the similarities between two large DNA sequences, because it does not have an overall vision of the two sequences. However, it could be useful to compare short sequences (or a short part of a long sequence) or for the detailed view. It’s also used by the global alignment algorithm (described later).

The figure 2 represents the matrix $M$ after the comparison of two very short DNA sequences, which are exactly similar. This matrix is also used by the evaluation function to give a score for each couple of amino acids. We give a score of 3 when amino acids are equals, a score of 1 for the pairs (A-G) and (T-C) (this amino acids are different but their chemical behavior are quite similar) and finally a score of 0 for the other couples (A-T), (A-C), (T-G) and (G-C).
3 DESCRIPTION OF THE DIFFERENT ALGORITHMS

3.1.2 Bloc algorithm

The “Bloc” algorithm is the advanced form of the general comparison algorithm. In each square of the matrix, it uses computes the results given by the evaluation function for a certain number of amino acids (described below) and sum these scores.

The algorithm uses a comparison window. This window has a size, say $W$. The first sequence is cut in segments of $W$ (and the second sequence is browsed successively to these segments).

If the window is positioned at the position $i$ in the first sequence and at the position $j$ in the second one, the point $i, j$ in the matrix $M$ will receive the sum of the evaluation function between each amino acids in the window:

$$M_{i,j} = \sum_{k=0}^{W-1} F(A_{i+k}, B_{j+k})$$

Where $F(A_i, B_j)$ represents the score of the evaluation between the amino acids at the position $i$ and $j$ respectively in the two sequences.

The “Bloc” algorithm allows to analysis the two sequences with an overall view, depending on the size of the window, $W$. We can see here that the “Two by Two” algorithm is a specific case of the “Bloc” algorithm, with $W = 1$.

Moreover, thanks to an important improvement add on the “Bloc” algorithm (with uses a queue to stock successive evaluations), the execution time does not depend on $W$. For this reason it’s recommended to use the “Bloc” algorithm instead of the “Two by Two” algorithm.

The figure below (Figure 3) represents the evaluation of the point $i = 5, j = 2$ in the matrix ($M_{i,j}$) during the comparison of two DNA sequences with $W = 4$. $M_{5,2}$ contains 4, sum of the scores given by the evaluation function for the couples (G-C), (G-G), (T-A) and (G-A).

![Figure 3: Comparison of two sequence with $W = 4$, $M_{5,2} = 4$](image)
3.2 "Local Alignment algorithm"

In genetics, the “Local Alignment algorithm” is frequently used to find the best match of a specific gene into a larger DNA sequence, such as an entire chromosome. The aim of the algorithm is not really to compare the sequences but to find the position of the smallest sequence inside the largest, which is the most relevant (a maximum number of identical amino acids). Then it will be possible to focus on this particular part of the largest sequence and to use a more precise comparison algorithm.

The “Local Alignment algorithm” uses the same “Bloc” method of the “General Comparison algorithm”. The “Bloc” algorithm here requires a fixed window, equal to the size of the smallest sequence. Then the algorithm behaves similarly to the “General Comparison algorithm”, except that the result will not be a matrix but the position in the sequence where the maximum score is found. It is possible to find more than one best local alignment, and of course it depends on the evaluation strategy used by the algorithm.

This algorithm should be used if the size of the smallest sequence is significantly smaller than the size of the largest one. In fact, the “Bloc” algorithm is not really precise for long sequences, more especially if the evaluation function returns large scores. Moreover, in this conditions the algorithm has a quite linear behavior (concerning execution time) and will be faster than the “General Comparison algorithm”. The complexity of this algorithm will be described later.

The figure 4 represents the result of the “Local Alignment algorithm” for two DNA sequences. As you can see, the first sequence is very small compared to the other sequence. To find the best alignment, the algorithm use the “Bloc” method with \( W = 5 \), the size of the shortest sequence, as described above.

![Figure 4: Local alignment of two sequences](image)

3.3 Global alignment algorithm

DNA sequences suffer constantly a lot of structural alterations: insertions, deletions or substitutions of one or more amino acids. These mutations can be active (the function of the molecule is different) but they can also be silent (the function of the molecule remains the same). Therefore two DNA sequences could be structurally different and have the same function and effect.

The “Global Alignment algorithm” is able to find the best global alignment between each amino acid. It allows to detect insertions and deletions on sequences. It uses the Needleman and Wunsch algorithm, a dynamic programming algorithm.

This algorithm requires the result of the “General Comparison algorithm”, and more especially the “Two by Two” algorithm (equivalent to a “Bloc” algorithm with a size of window \( W \) equal to 1), named the matrix of similarities \( S \). This matrix is browsed from the right bottom corner to the top left corner, line by line from the left to the right in order to create another matrix, named \( F \) obeying to the following expression:
∀i ∈ [1, n], ∀j ∈ [1, m], F_{i,j} = S_{i,j} + \max \left( \frac{S_{i+1,j+1} + \max_{k \in [i+2, n]}(S_{k,j+1} - P(k, j + 1))}{\max_{l \in [j+2, m]}(S_{i+1,l} - P(i + 1, l))} \right)

where n and m are the size of the two sequences respectively and P(x, y) the penalty for the insertion of |y - x| amino acids.

The algorithm implemented uses two different penalty functions. These two functions use two positive coefficients which are domain specific.

“Constant penalty” function: \( P = C \) where C is the “Constant penalty” coefficient. This function allows to make an improvement on the algorithm (about the computation of maximum scores which could be stocked) but it does not limit the number of successive insertions in the same sequence.

“Linear penalty” function: \( P(x, y) = L * |y - x| + C \) where L is the “Linear penalty” coefficient and C the “constant penalty” coefficient. This function makes the algorithm slower but it limits the number of successive insertions in the same sequence.

The coefficients L and C have to be chosen by the developer of the DSL. They have to be in adequacy with the values returned by the evaluation function. For example, if L is very small compared with the maximum value returned by the evaluation function, a lot of successive insertions can be allowed (and it’s not the aim of this penalty function).

Once the matrix F has been computed, we can compute the global alignment between the two sequences. We start from the position \( (i, j) \) where \( F_{i,j} \) is maximum and i and j are minimum. Then we create the alignment sequence by backtracking from the start point to the bottom left corner of the matrix. We have to follow three rules:

- If \( F_{i+1,j+1} \geq \max(F_{i,j+1}, F_{i+1,j}) \) then the amino acids at the position i and j (in the two sequences respectively) are aligned.
- Else, if \( F_{i+1,j} \geq F_{i,j+1} \) then there is an insertion in the first sequence.
- Else, there is an insertion in the second sequence.

As you can see this backtracking method is deterministic: the diagonal browsing is privileged and if it’s not the case the insertion in the first sequence is privileged. The algorithm of Smith and Waterman tries to solve this issue, but we didn’t implement it in the application.

This step is named processing stage. The result of the backtracking is stocked into an array which is transformed during the rendering stage to be displayed into the application.

The figure below (Figure 5) represents the result of the “Global Alignment algorithm” between two DNA sequences. The first alignment is given by the “Constant penalty algorithm” and the second one is given by the “Linear penalty algorithm”. You can see that there are two successive insertion in the alignment given by the first algorithm while there is only one successive insertion in the second alignment (insertions after the first amino acid and before the last amino acid are not considered when we talk about “successive insertions”)

3.4 Algorithmic complexity
4 FIRST IMPLEMENTATIONS

The different algorithms (and their variants) have different algorithmic complexity. It’s important to choose the right algorithm depending on the size of the two traces. The following results are given in function of the number of call to the evaluation function (which is domain specific) and in function of the length of the two evaluated traces, \( n \) and \( m \) respectively (we will suppose \( n \leq m \)). The final complexity (given in function of the number of elementary operations) will mainly depend on the complexity of the evaluation function, which is not necessarily constant or majored.

<table>
<thead>
<tr>
<th>Name of the algorithm</th>
<th>Average complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Two by Two”</td>
<td>( O(n \times m) )</td>
</tr>
<tr>
<td>“Bloc”</td>
<td>( O(n \times m) )</td>
</tr>
<tr>
<td>“Local alignment”</td>
<td>( O(n \times m) )</td>
</tr>
<tr>
<td>“Constant penalty”</td>
<td>( O(n \times m) )</td>
</tr>
<tr>
<td>“Linear penalty”</td>
<td>( O(n \times m \times \max(n, m)) )</td>
</tr>
</tbody>
</table>

4 First implementations

In this chapter we will describe the work done before making the link between the comparison application and the framework.

4.1 Prof of concept

After two weeks of research about trace comparison in general and DNA comparison, it was necessary to know if the algorithms used for the comparison of DNA sequences were viable and if they were really applicable for execution traces. We already had a sample of trace, named SHOT (in reference to the eponymous game) given by Mr. Manders.

We first implemented a very simple version of the “Bloc” algorithm, together with a graphical view to display the results. The first results were convincing: the algorithm was able to show main similarities between DNA sequences, and its behavior was reasonable for very long sequences (concerning execution time).

\(^1\)This result is quite particular: This algorithm calls the evaluation function \( n \times m \) times. The third factor comes from the browsing of the matrix during the computation of the maximum value. If we consider that the complexity of the evaluation function is majored by a real value \( k \) (in order to simplify the expression), the real complexity of this algorithm (given in function of the number of elementary operations) is \( O(n \times m \times k + n \times m \times \max(n, m)) \). The result above remains true if \( k \) is significantly small compared with \( n \) and \( m \).
We also had to choose a structure which might be adaptable for each DSL trace. For each trace it was necessary to implement two functions (an evaluation function and a display function) of an abstract class "Sequence". It was possible to make a distinction between the main application, which was general, and the evaluation function, which was domain specific.

We decided to validate the concept (the link between trace comparison and DNA comparison) at the end of this first implementation.

4.2 Features implemented

4.2.1 Other algorithms

The first algorithm implemented was the "General Comparison algorithm". In order to complete the panel of features in the application, we implemented three other algorithms:

**Local alignment algorithm:** This algorithm is described in Chapter 3.2. It allows to align two sequences.

**Advanced local alignment algorithm:** This algorithm cuts the shortest sequence and tries to align each segment with the other sequence as best as possible. The algorithm was operational but the results was not satisfying. This algorithm is not implemented in the final version of the application.

**Global alignment algorithm:** This algorithm is described in Chapter 3.3. It allows to find the best global alignment between two sequences. At first the “Linear penalty” algorithm was the only algorithm implemented.

4.2.2 Rendering tools

In order to visualize the results of the main algorithm, we made a feature able to collect and transform data contained in the matrix $M$ and visualize it.

The rendering object collects the matrix $M$ which contains scores between 0 and the maximum value of the matrix. These scores are transformed to RGB codes, between 0 and 255 included thanks to an extrapolation function. These RGB scores represent various shades of grey, from black (RGB = 0) to white (RGB = 255). The points with a high score (close to the maximum) are transformed into black dots (RGB score of 0) and the points with an small score (close to the minimum) into white dots. The extrapolation function uses an “exponential profile” with a variable coefficient. The higher this coefficient is, the more the differences between high score points and low score points are important.

The rendering operation also collects the average score in the matrix. Using the maximum score and the average score, we are able to delete point whose score is lower than a certain percentage of the maximum or the average score (for example, 50% of the maximum or 70% of the average). In the final implementation we decided to keep only the elimination using the maximum value (and no longer with the average value).

4.2.3 Resizing tools

In order to visualize the result of general comparison algorithm for very long sequences, we had to create resizing tools able to conserve as much as possible the results of the algorithm. As we described above, the “General Comparison algorithm” returns a matrix of scores which is transformed into a matrix of RGB scores (between 0 and 255), which could be considered as an image. For this reason, the resizing stage uses algorithms used for image processing. We chose to implement two resizing methods:

**Average method:** The average methods is often used for image processing. It computes the average RGB score of a square (whose size depends on the factor of resizing chosen) to create a new pixel of the image.
Threshold method: This method uses an array of thresholds, which contains RGB scores associated with different values. For each square of the old image, we count how many points have a RGB score between each threshold. For each threshold (from the smallest to the largest), if the sum if higher than the value associated with this threshold, the pixel will have the same score as the threshold. For example, if the thresholds are \{0, 100, 255\} associated with \{10, 10, 10\}, if there are in the square more than 10 points whose RGB score is between 0 and 100 the pixel will be black. Else, if there are more than 10 points whose RGB score is between 100 and 200, the pixel will have the a RGB score of 100. Else, the pixel will have the a RGB score of 255 (white dot).

4.3 Optimization

In order to improve our application we proceeded to two major algorithmic improvement to reduce the execution time:

Bloc algorithm (General comparison): Thanks to a new browsing strategy (diagonal by diagonal and no longer line by line) and a queue, we achieved to delete the dependence of the algorithm with respect to the size of the window, $W$. The algorithm is a little bit slower when $W$ is equal to 1, but it reduces considerably the execution time when $W$ is high.

Constant Penalty algorithm (Global alignment): The global alignment algorithm needs the computation of a maximum value inside one row and one column for each point in the matrix (see Chapter 3.3 for more information). If the penalty function used by the algorithm is constant, we can store during the execution the values of the maximum for each row and each column and modify them when it’s necessary. This optimization allows to suppress one algorithmic dimension to the algorithm (the constant algorithm is quadratic whereas the linear algorithm is cubic).

4.4 Structure of the first implementation (UML graph)

See Appendix A for a UML description of the application.

5 Adding a new trace operation

For the final implementation, the objective was to create a trace application as it is described in [6]. The two questions developed in this paper are (see section 1: Introduction):

1. How can we define tools that manipulate execution traces?
2. Can we make these tools general enough to manipulate execution traces of arbitrary DSLs?

In the case of the trace comparison application, we can adapt these two questions:

1. Can we use the framework’s description of trace applications to implement our trace comparison tool?
2. Can we make a tool for trace comparisons general enough to manipulate execution traces of arbitrary DSLs?

In the precedent implementation, we tested the second question: Are the DNA algorithms described above able to be adapted for trace comparisons of arbitrary DSLs? Now that the precedent implementation proved that it was possible to isolate domain specific features and display interesting results for different domain specific features, we have to answer the first question: Is the current definition of trace applications provided by the framework complete enough to implement our trace application, or have we to add a new trace operation to facilitate the implementation.
5 ADDING A NEW TRACE OPERATION

In the philosophy of the framework, trace applications are combinations of *trace operations*. For example, a visualizer of traces is a trace application, which can be decomposed into a trace operation providing access and browsing inside the trace, and a trace operation providing the visualization. The framework provides standard trace operations, which can be combined to create trace applications. The objective is now to use these operations, and if necessary, to add a new trace operation. For more details on the framework, see [6].

5.1 Description of the problem

In first tests with the framework operations, we discovered that the provided trace operations were well designed to make comparisons between two requested trace parts, but the difficulty was to request the good trace parts. A domain specific trace (in the way it is designed by the framework) has a very general definition:

- A trace is a set of *trace parts*, and a *static model* part.
- The set of trace parts is expected to have an entry point (a trace part expected to be the first requested).
- Each trace part has a successor and a predecessor, except the first and the last ones, respectively.

A simple example following this description is a sequence (such as a DNA sequence): The entry point is the first element of the sequence, and for each elements, the next / previous element is the next / previous in the sequence position. But sequences are not the only possible structures able to follow the description of the framework.

The entry point of the trace can be obtained in the framework with a *reset request* and the successor / predecessor of a trace part can be obtained with a *next trace part request* / *previous trace part request*.

![Diagram of a simple trace application merging two input trace parts.](image)

Figure 6: A simple trace application merging two input trace parts.

If now we consider a simple trace application, which aims to merge two input trace parts, the requests will be made by the trace operation positioned after the merger (see figure 6). But if we make a reset request, it will be translated by a reset request for both input trace parts, and it will be the same for the other types of requests. We will not be able to make for example a reset request for one trace part, and a next trace part request for the other one. That will be problematic, because we need these sort of requests for our comparisons (if for example we want to compare the first amino acid of a DNA sequence with the third one of the other sequence).
In figure 7, we consider a more complicated trace application merging two input trace parts and trying to solve the precedent problem. Here, requests will be made by the two trace operations between the accessors and the merger. It will here be possible to make a reset request for one trace accessor and a next trace part request for the other. However, for each request, the merger will collect a trace part, and that is our next problem, because we want sometimes to make artificial requests only to change the position of the cursor inside the trace accessor, and these requests have not to be forwarded to the merger. If we try to make these requests without blocking them, the merger will send to the application useless results, and these requests will have an influence on the other requests (the real ones).

5.2 Proposed solution

The proposed solution is to implement a new trace operation designed to browse inside the trace application, and allows to make these sort of artificial requests to move the cursor of the accessors without executing actions.

We have implemented this trace browser operation, with a specific controller, which can be used directly by the trace applications to access to the functionalities of the trace browser operation. Section 5.2.1 gives a description of the implementation.

5.2.1 ITraceBrowser interface

```java
    boolean canRequestNextTracePart(void)
    boolean canRequestPreviousTracePart(void)
    boolean canRequestReset(void)
    ITracePart getCurrentTracePart(void)
```

These methods have exactly the same functionality as in the other trace operations. The first three methods ask to the providers if the corresponding request can be made, and the last one is to get the trace part requested.

```java
    ITraceBrowserController getTraceBrowserController(void)
```

This method is to get the controller of the trace operation (the description of this interface is below).
5 ADDING A NEW TRACE OPERATION

These methods have nearly the same functionality as the corresponding methods of the other trace operations: they make a request to the providers. The input boolean offers the possibility to forward the result of requests to the consumers or not. If it is `true`, the result of the request will be forwarded to the rest of the application. It corresponds to the behavior of the other trace operations. If now the input parameter is `false`, the request will be artificial, and the trace operations placed after the browser in the application will not be aware of the request. This functionality solves our problem described above, in section 5.2.2.

![Figure 8: A request, which is not forwarded. The only activated part is the green one](image)

```java
boolean isCurrentTracePartForwarded()
```

This method permits to know whether the current request will be forwarded to the rest of the application or not.

An implementation, working with a boolean to know whether the requests will be forwarded or not, is available in the framework.

That permits to solve our precedent problem. An example of solution is given in figure 8. The trace parts, which are not expected to be forwarded are blocked by the trace operation.

5.2.2 `ITraceBrowserController` interface

This interface provides the functionalities to access to the browser operation, and to have a feedback on the requests made by the browser (forwarded or not). Here is a quick description of the methods.

```java
void requestNextTracePartExecuted(ITraceBrowser)
void requestPreviousTracePartExecuted(ITraceBrowser)
void requestResetExecuted(ITraceBrowser)
```

These methods give a feedback to requests and can be used to execute actions on these requests.

```java
void setITraceBrowser(ITraceBrowser)
```

This setter will be called by the framework at the creation of the application to give the trace browser operation to the controller.
6 Final structure of the application

Now that the trace framework has been updated with a new trace operation able to solve our browsing problems, we can create the final structure of the application. The application follows the MVC design pattern. The access to traces, the computation of scores and the main process of the algorithms are gathered in the model part, and all the view part communicate with the model thanks to the controller, which corresponds in our case to the class ApplicationController in the package trace.framework.comparisons.impl.

6.1 Structure of the trace application

In terms of trace operations, we have a structure similar to the small test examples described in Section 5. For more details on the trace browser, see Section 5, and for more details on the other trace operations, see [6].

![Figure 9: Structure of the final application in terms of trace operations.](image)

Figure 9 illustrates the final application. The two input traces are linked to the application thanks to the two trace accessors. Each of these accessors is controlled by a trace browser (Operations with a “B” in the figure. For more details on this operation, see Section 5). A merger is then used to make the comparisons. The merger calls the domain specific evaluation function, and transforms the two input trace parts into a trace part containing the score. A trace browser is then added to make general browsing inside traces.

The results are collected using a trace action (operation with a “C” in the figure). This trace action does not make any modification inside the trace, but only sends the scores to the algorithms for computation.

The last operation, the trace consumer is only here to terminate the application. The methods of the operation are not used by the application, but can be used for a debugging purpose.

6.2 Details on the implementation of the algorithms

In this part is described the implementation of the algorithms. The developer should try first to understand clearly these structural points before looking at the implementation.

6.2.1 General structure of the algorithms

To activate an algorithm, several things have to be done:
The collector related the algorithm (see Collector class in its implementations in the package trace.framework.comparisons.impl.collector) has to be set as a listener of the trace action designed for that purpose (see TraceCollectorActionListener class in the package trace.framework.comparisons.impl).

When a collector receives a score, it implicitly knows from which trace parts it comes, because the browsing inside trace parts and the order of comparisons is predetermined and especially designed for each algorithm. That permits to optimize algorithms, and to minimize requests made to the trace accessors. For example, the MatrixCollector knows in which order the scores will be received and can compute alone the next position to fill inside the result matrix. This order is also known by the object, which is responsible of the browsing inside the application. The description of this object is in the ApplicationBrowser class in the package trace.framework.comparisons.impl. This class contains one browsing method for each algorithm and manage the trace browser operations to collect the scores in the correct order.

All these things are performed by the controller object. For each algorithm, there is a corresponding method inside the ApplicationController class (the “scenario” methods).

### 6.2.2 Comparator strategies

In the matrix view, there is a configuration point called “configuration strategy”, which correspond to the simple matrix comparison, and the comparison with blocs. These strategies can be found inside the trace.framework.comparisons.impl.comparator package. They are implemented with the design pattern strategy: the strategy is a parameter of the collector.

### 6.3 Structure for the domain specific parts of the application

In terms of implementation, the domain specific parts of the application are the following:

1. An XML configuration file
2. The implementation of the trace accessor
3. The trace strategies

The way to implement these features is described in the user manual, in Section 3. The interface to implement for the trace accessor is ITraceComparisonsAccessorListener in the package trace.framework.comparisons.impl. The class describing the way to implement strategies is EvaluatorStrategy in the package trace.framework.comparisons.impl.evaluator. The method loading the XML configuration file can be found in the LoadConfig class in the trace.framework.comparisons.config package.

### 6.4 A few pieces of information about the views

To make the link between the display and data received from the algorithms, each view has an object to make the transformation. These objects are renderers and can be found in the trace.framework.comparisons.renderer package. More precisely, they convert data coming from the algorithms into data directly usable by the corresponding view.

The views are always extensions of the SWT Composite widget, which are panels, in which people can put anything he wants. A few methods have been added in order to be used by the FrameManager class, which provides the windows and the basis structure for the views.
The **DetailedView** is a particular view. Its content depends on the interaction with the other views. That is why this view is always in parameter inside the view, in which a detailed display is required. For the content of the detailed view, since it depends on the interactions with the other views, two renderers have been implemented (see the `DetailedRenderer` class, and its subclasses in the package `trace.framework.comparisons.renderer`).

7  **Final version of the application**

The final version of the comparison application is linked with the general framework and uses the trace definition formalism as described in Chapter 4. This version reuses a great deal of features developed in the first implementation. Some useless algorithms (such as the “Advance local algorithm”) and non improved algorithms (such as the first versions of the “Bloc” algorithm) have been abandoned. The details of this features are described above.

7.1  **Available algorithms**

The three main algorithms are implemented in the final application, and they work successfully with the trace definition given by the framework.

**“General comparison algorithm”** The two version, “Bloc” and “Two by Two” are implemented and can be used in the final version of the application.

**“Local alignment algorithm”** The algorithm is implemented, but can’t be used in the final version of the application. A view has to be made in order to display the results of the algorithm.

**“General comparison algorithm”** The two version, “Constant penalty” and “Linear penalty” are implemented and can be used in the final version of the application.

7.2  **Final Features**

The details of all the features implemented in the framework are described in the user manual.

7.3  **Analyze of the results**

A few pieces of information about the way to analyze results in the algorithms is available in Appendix B.

8  **Future Work**

This section contains a list of things, which could be added, modified or improved inside the application. Some things described below are already half implemented.

8.1  **A view for the “Local Alignment algorithm”**

The “Local Alignment algorithm” is implemented inside the application, but no view is provided to visualize it. We expected to make a view similar to the “Global Alignment algorithm”, with a configuration panel, a visualization and a detailed view with interactions with the main visualization.

8.2  **Cuting of traces**

The goal of the “Local Alignment algorithm” is to find the position of a short sequence inside a large one. When it comes to traces, the interest could be to find a specific behavior inside a trace. However, if these behaviors come from another execution trace, users will need to “cut” the trace and consider only a subpart of it.
That is why a module done for the cutting of traces could be practical. The structure for this module is already partly implemented. The interface ITraceComparisionsAccessorListener of the package trace.framework.comparisons.impl already provides three methods for that purpose. In the DNA project, these methods have been implemented to provide an example.

```java
void setVirtualCut(int indexStart, int indexEnd)
```

This method aims to cut the trace and consider only a subpart of it, between the two input indexes. Then, a reset request will correspond to a request for the trace part in position `indexStart`.

```java
void removeVirtualCut(void)
```

This method aims to remove the precedent cut. Concretely, the starting index is again the first index inside the trace, and the ending index corresponds to the last trace part.

```java
boolean isCut(void)
```

This method permits to know whether the trace has been cut or not.

### 8.3 Limit of the DNA approach

The DNA comparison algorithms permit to make comparisons between sequences. In general, DSL execution traces can be seen as sequences, but the trace application framework is a bit more permissive for the structure of traces. For example, it is possible to implement “circular” traces.

The problem with these traces is that the comparison application will not be able make comparisons with particular traces. The constraints required by the application are the following:

- The algorithms use many reset requests to make the optimized browses inside the traces. If it is possible, users should try to let the reset requests available at each moment. If this is not possible, the algorithms could compute and display wrong results.

- The traces are also supposed to be finite. Sometimes, the algorithms go to the next trace parts until this type of request is no longer possible. If one trace is not finite (a “circular” trace for example), it will involve infinite loops.

If these constraints cannot be respected, or more generally if the trace has not the structure of a finite sequence, a solution could be to implement and use the “cut of traces” described above. This functionality could provide a way to linearize the trace or one part of it, and that could permits to use the comparison algorithms.

### 8.4 Improvement of the matrix visualization

Several improvements could be made for the visualization of the matrix. The first one is to make a clear separation between the main part of the algorithm and the border part, which gives approximated results (see Appendix B).

Another thing, which could be changed is to visualized in a better way the part of the matrix, which is displayed in the detailed view. For the moment, it is only a red point. Moreover, the borders of the matrix cannot be displayed by the detailed view, and that could also be improved.
When it comes to large traces, resizing tools have been implemented, but it does not work very well. Improvements could also be done for that part.

8.5 Comparisons between the results of the algorithms

It could be interesting to compare the results of the algorithms between themselves if these results are obtained thanks to different ways. For example, it could be interesting to compare the matrix corresponding to the comparison of traces, depending on the chosen strategy. However, for the moment, nothing is provided to store the results and reload them.

Another thing, which makes the comparisons of two different matrix difficult, is that one level of grey can correspond to a different scores inside the two matrix. The view renderer searches the maximum inside the matrix and makes this score corresponding to the color black. For the color white, it is the same behavior with the lowest scores. But, of course, these maximum and minimum can be different at each comparison, so the matrix cannot be compared between themselves easily.
A Modelling of the first implementation

![UML Diagram](image)

Figure 10: UML modeling of the first implementation.

B How to analyze results

In this section we will explain how to analyze the results given by the “General Comparison Algorithm”. We will describe in a first time the result of the comparison between two quite similar traces (here DNA sequences) and then we will analyze the result of the comparison between two very similar traces.


B.1 First analysis

The figure above represents the result of the “Bloc” algorithm between two sequences which are quite similar. In fact, we can see two main similarities (long black lines) which represent equal (or very similar) sub sequences (see part 1 in the figure above). We can see that these two similarities begin and end at the same horizontal position. It means that there is a repetition of this sub sequence inside the second DNA sequence.

There is also a great deal of equals sub sequences. We can observe the similarities almost everywhere inside the matrix. Very short similarities are quite normal: there are only four different possible states so the number of different combination is quite low. For this reason similarities can appear randomly.

Among this similarities, some of them are interesting. As you can see in the part 2 of the figure above, there are some important similarities witch are quite redundant in the two sequences. Moreover, in the square you can see a large similarity and two others, smaller, in the diagonals above and below. It means that this sub sequence is composed with a redundant scheme, with  is repeated two times approximately.

It is possible to remove very short similarities by increasing the value of $W$. The number of random similarities will decrease significantly. You can also use graphical features to adapt the view and remove useless points or non necessary similarities.

The blue part of the figure (part number 3) is due to the improvement of the “Bloc” algorithm. The algorithm uses a queue which is filled during the browsing of the two sequence (the matrix is browsed diagonals by diagonals). The aim of the improvement is to compute one time only the evaluation function between two amino acids, and that for any value of $W$. So we fill the queue and when it is completed (i.e. there are $W$ values in the queue) the real algorithm begins. But before, the fill is not totally completed and the results are not correct. In fact, if for example $W = 15$, the first point of each diagonal contains the score of the evaluation function between two states, the second point in the diagonal contains the sum of the scores given by two calls

Figure 11: Comparison of two DNA sequences (250 amino acids) with the “Bloc” algorithm ($W = 15$)
to the evaluation function etc... Theses values are in general lower than in a regular point, which contains the sum of $W$ values. However these $W$ first points could be interesting so we extrapolate the results of this points but sometimes (and it is particularly true when $W$ is high) the extrapolation is not very effective.

B.2 Second analysis

![Matrix](image)

Figure 12: Comparison of two DNA sequences (250 amino acids) with the “Bloc” algorithm ($W = 15$)

The figure above represents the result of the “Bloc” algorithm between two sequences which are extremely similar. We can observe that the central diagonal is globally black, with denotes important similarities between the two sequences (The evaluation function returns maximum values when the two amino acids are equals). We can find only two white parts in this sequence, which represent differences in the two sequences. This blank parts are quite sort, and if we increase the value of $W$, they will be less visible. Contrariwise, if we decrease $W$, this differences will be more visible. A priori there are no large redundancies in these two sequences, but we can observe short repetition of sub sequences, visible near to the main diagonal. They may be caused by “random” redundancies.
References


