Petit projet de recherche

B.A.B.A.

Basic-Algorithms-of-Bioinformatics Applet

Report

IFT6291 - Fall 2003

by
Norman Casagrande

1. Introduction

1.1. Motivations

Understanding algorithms is one of the main tasks of a bioinformaticist, but that does not mean that is easy. Reading papers can give a general idea of the behaviour and possibilities offered by the algorithms proposed, but to deeply understand them it is often necessary to get pen and paper, or better a computer and a keyboard, and program them.

If you don't have time you have two ways:

- Find some other sources of information. But this takes also a lot of time, and does not assure that you will fully understand what happens, for example, when you change a parameter.
- Play with the existing code. But if the code is not elegant or if you are not a good coder this will also take time. Besides the code is not always available.

But what if there is a program that shows itself step-by-step allowing the user play with its parameters? The Columbus' egg!

The problem with these software is that they are often programmed for a specific algorithm. This means that there is a lot of work for reinventing the wheel every time a new program needs to be programmed. Out of all these wheels we find:

- Graphic interface.
- Invalid entered values checks.
- Algorithm framework.

Maybe because of these problems current works available (if they exist) are generally ugly, and limited.

My goal was to create a program that could serve as basic framework for creating algorithm demos based on dynamic programming (plus some algorithms), and publish the code under a public license, so that everybody could contribute to it.

1.2. Algorithms

On the framework some algorithms have already been programmed as demonstration and basis for further extensions.

These algorithm are:

- Basic Dynamic Programming Table (without scores matrix) [1]
- Smith & Waterman [2]
- Needleman & Wunsch (local search) [3]
- Four Russians [4]

The first three are the basic dynamic programming algorithms.

Four Russian was chosen for two reasons:

1. Provide an algorithm which would not be just a simple extension of the dynamic programming table.
2. Provide a full description of an algorithm that is really poorly documented on the net.

1.3. Previous works

As mentioned before, on the net it is very hard to find other interactive works. If somebody is looking for the explanation of an algorithm, he can find some step-by-step pages, but rarely the explications will be available in an interactive format.
For the algorithms presented in the current work, I made a wide research among the internet. Here is what I found:

Basic Dynamic Programming

- Smith Waterman Applet[1] by Dennis Kibler and Ray Klefstad (University of California):
  Despite the name, this applet runs a simple dynamic programming table.
  Pros: Source code available.
  Cons: The result is shown as text. No step-by-step.

Needleman & Wunsch

- Alignment Applet[2] by the German Cancer Research Center of Heidelberg:
  This nice applet can run a N&W algorithm step-by-step.
  Pros: It is possible to change the score table, but only from a list of previously edited tables.
  Cons: The gap penalty is fixed. Source code not available.

Smith Waterman

- Local Similarity Applet[3] by Thierry Lecroq:
  This nice applet for local alignment.
  Pros: Allow the user to set many parameters of the local search algorithm.
  Cons: No score table. Just two steps. No source code. Cannot choose the starting local point.

Four Russians

- Nothing found. Even the static explanations are really poor, and all refers to the book of Gusfield.

1 http://www.ics.uci.edu/~kibler/SW/SW.html

2 http://www.dkfz-heidelberg.de/tbi/bioinfo/PracticalSection/AliApplet/

3 http://www-igm.univ-mlv.fr/~lecroq/seqcomp/node6.html

2. The program

2.1. Basic Information

BABA has been programmed with java. It takes full advantage of the object oriented philosophy of this language to make further improvements very easy.

It has also been developed to be run on the net, and therefore the starting class is an Applet, but this is not a mandatory requirement.

2.2. The Framework

The whole program fit in the package norman.baba.
Below this level other packages exist. The most important are:

- algorithms:
  Where all the algorithm – behaviour, graphic interface, etc.. – are programmed.
- grids:
  With all the elements for managing the matrixes used in the dynamic programming. This includes also the graphic objects that display the matrixes.

Other packages take care of basic user interface elements, thread synchronization, special data structures, etc..

Here is a summary scheme of the whole structure:

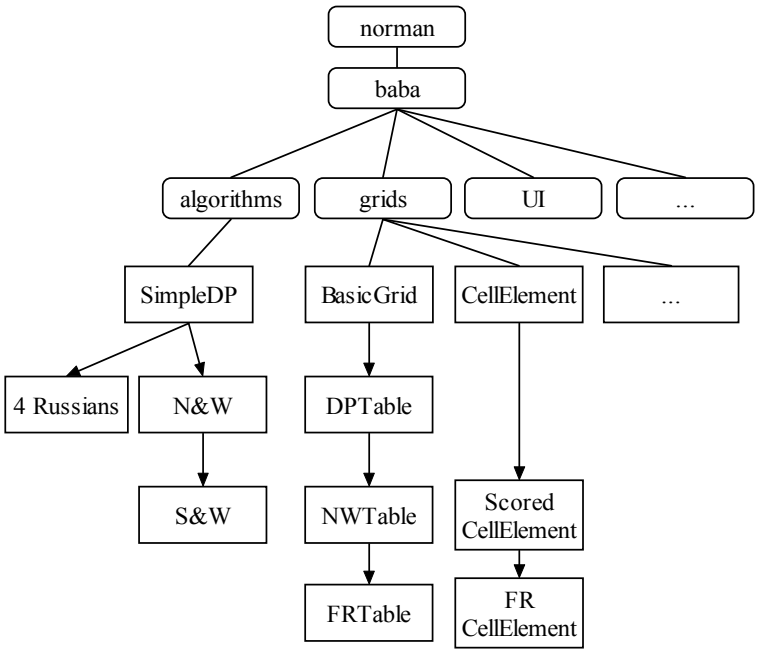


Figure 1 - A summary scheme of main packages and classes

The basic algorithm of dynamic programming has been put in the *SimpleDP* class.

This class manage several tasks:

- Set up the interface for the algorithm.
- Instantiate the grid graphic object.
- React to the user input.
- Run the algorithm. This is done by a special function called *forwardStep*. In this function a *switch* selects the next step according to the current phase. There is a basic list of available phases that can be overridden or extended.

Clearly the most interesting task is the one that runs the algorithm. It is also the one that likely will be overridden when a programmer adds his own algorithm.

As example let's consider the simple dynamic programming (SimpleDP) algorithm.
When the user hit the forward button (see Figure 2) the *forwardStep* function is called. The *switch* of this function selects between these options:

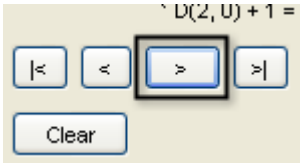


Figure 2- The forward button

- If current phase is *grid computing*
  - If it is the last cell: set the phase as *backtracking*. Call backtracking function.
  - Else: call grid filling function.
- If current phase is *backtracking*
  - If it is the end: exit.
  - Else: call backtracking function.

On the Needleman & Wunsch algorithm, the only part changed in this process is the *grid computing* function that had to be changed in order to take in consideration the scores. This algorithm is on its turn extended by Smith & Waterman to take care of the local search.
Four Russians does not have scores, thus it extends only *SimpleDP*.

On the other side, the graphical representation of the algorithm is a concern of the classes available in the *grid* package. The basic grid class is an extension of a panel where the grid is drew. As already said the grid object is instantiated in the algorithm class.
A basic grid offers a set of tools like:

- Grid drawing.
- Values storing and drawing.
- Cell highlighting.
- Cell interaction methods.
- In general all the graphical operations.

The grid can be seen as an extended graphic matrix. Each cell of the grid is a *CellElement* object. In this class are stored data like value, color, pointers to other cells (any number), etc..
Depending on the algorithm, both grids and cells have been extended to handle the requirements. For example the basic grid does not have the pointer arrows: they are defined in *DPTable*.

2.3. Implementation of the algorithms

Simple Dynamic Programming

When the program start a default 10x6 empty matrix is created. It is resized when the user has set the two strings.
All the parameters that the user can set have a syntax check on them. It is impossible for example to enter a letter in the gap field.

Two main phases exists in the DP algorithm:

- Grid calculation: fill the cell using the standard dynamic programming algorithm. It also set the pointers to the previous cells.
- Backtracking: by following the pointers previously set it gives the alignment. The interaction with the user is handled by an interface between the grid and the algorithm (note: interface meant as java *interface*).
  The backtracking policy (when the user presses the forward button and do not select the cell) is set in the algorithm class. The *CellElement* object representing the current cell, has a method to get the next pointer with a given policy. The available policies are:
  - Clockwise
  - Counter Clockwise
  - Random
  - Any (differs from random because it takes the first inserted pointer)

Needleman and Wunsch

Extends the *SimpleDP* class, and does override, as previously said, only the *grid computing forward* function to take account of the scores.
The scores available are:

- PAM 250 [5]
- BLOSUM 52 [6]
- VTML 160 [7]
- Penalise mismatch

The latter scores matrix uses a score scheme where match = 1 and mismatch = -2.

Each scores matrix is a hashtable with the two characters as key, and the score as value. A special hashtable class has been created to handle this.

Smith and Waterman

This algorithm is an extension of the previous, thus it inherits also all the characteristics of the simple dynamic programming.
The main differences are two:

- It adds a fourth choice (0) while computing the table.
- It adds another phase to the other two: the local selection. This was done by overriding the *forwardStep* function.

Four Russians

Once the basic framework was done, programming the algorithms was easy. Four Russians nevertheless needed some extra work because of its particular pre-processing phase.
First of all, the bloc size (parameter *t*) is available as a parameter for the user. But only valid *t* values are accessible. The automatic selection is done by a simple procedure that checks if the module of the *t* values and both grid height and width equal to zero. The minimum *t* value allowed is 2, the maximum is set to 127 (this limit is due to the usage of *byte* arrays for the *restricted bloc function* parameters, in order to save memory), but normally 6 is well enough.
Before precomputing the blocs, the alphabet is encoded to another alphabet of size:

$$|encoded\ \Sigma| = \min(2(t-1), |\Sigma|)$$

where $|\Sigma|$ is the current alphabet size.

This is done to save more memory, and have a reduced complexity because every combination of the alphabet must be computed on the blocs.

Explication:
Consider a bloc of 3*3 (*t*=3), and an alphabet of {*A*, *B*, *C*, *D*, *E*}:

		A	*B*
C			
D			

Even by having all different characters on each position the fifth character *E* is not used. Therefore we can avoid using it.
It can also happen that the alphabet size is smaller than 2(*t*-1). In this case is the size of the resulting encoded alphabet is the original alphabet itself.
When the *restricted bloc function* applies (see Gusfield [4]) it is enough to encode the current bloc alphabet to get back the bloc.

To save further RAM the computed bloc is stored in a class called *MinimalistMatrix*. Each element of this matrix is a *MinimalistCellElement*. This last one stores the value as byte and has just three pointers to the other cells.

When the user selects a *t* value with the appropriate slider, the program calculates the number of blocs that will need to be computed. This number comes from the formula:

$$3^{2(t-1)}|\Sigma|^{2(t-1)}\ [4]$$

But because the size of the encoded alphabet size could differ depending on the bloc size and the original alphabet size, in the program I used the formula:

$$3^{2(t-1)}|encoded\ \Sigma|^{2(t-1)}$$

During the precomputation, the blocs are stored into an hash table that accept as key the four parameters of the *restricted bloc function* (string 1, string 2, top row, left column – the upper left corner is not considered because is always 0), and the matrix as value.

Another array is filled with the keys, to keep the order of insertion. Thanks to this array, the user is informed of which table has been used for computing the bloc, and therefore is able to retrieve it in the precomputed blocs list (see chapter 2.4 – section Four Russians, for an example).

If the user tries to precompute a large amount of blocs (> 60'000) he is warned with a message box. If he continues and the memory is not enough, he will quickly get a "out of memory" error, likely even before reaching 2%. This is because to save time the algorithm automatically allocates the space for the hashtable and the array at the beginning.
Note: unfortunately when this kind of error appears, the applet becomes unstable, even if the references to the big objects are detached and the garbage collector is called.

Once all the precomputed blocs are ready, the user can run the algorithm and see the blocs being selected using the four parameters. These parameters are encoded (in the way previously explained) and used to retrieve the bloc in the hashtable. Then the values and pointers of the bloc are copied in the main table and showed to the user.

An extension of the *NWTable* grid was necessary to handle the half cells, the score at different positions and the index drawing.

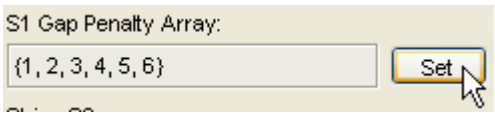
2.4. Explanation of main features

Basic common features

The basic features available for every algorithm which extends the dynamic programming algorithm are:

- Check on the entered strings: no empty string allowed, auto upper-case.
- Check on the entered gaps: syntax check, no characters allowed.
- Interactive backtracking.
- Basic console.
- Info label.
- Zoom.
- Anti-aliasing.

Simple Dynamic Programming Table



Once the user has set the strings, it can change the penalty gap with the special button (see figure on the right).
At this point the user can manually enter the gap sequence, or select one of the buttons (see Figure 3).

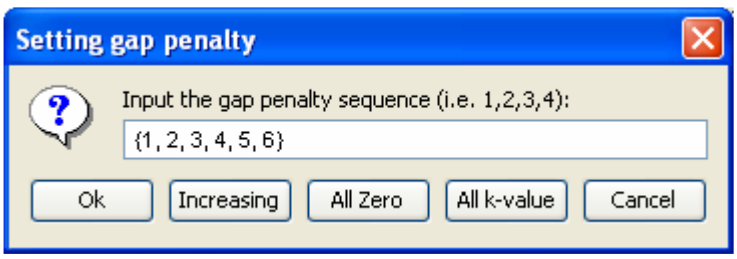


Figure 3 - The gap penalty sequence dialog

The available options are:

- Increasing: given the length of the string it will create an increasing array of the same size.
- All zero: it will set all the gaps to zero.
- All k-value: it will pop up a dialog asking for a value to be inserted at each gap position.

If the user insert a non valid sequence depending on the values it could be considered totally wrong (and then resetting to the previous sequence), or set a zero on the invalid positions. To clearly explain the flow of the dynamic programming algorithm, the program shows three important features:

1. The cells that are currently involved in the calculation (arrows from the current cell – see Figure 4).
2. The cells that are pointed (yellow highlight and black arrows – see Figure 4).
3. The details of the calculation (see Figure 5).

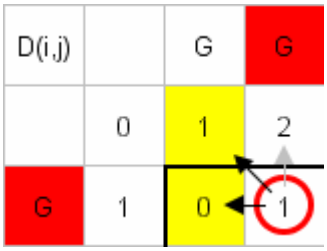


Figure 4 - The current cell and the pointers

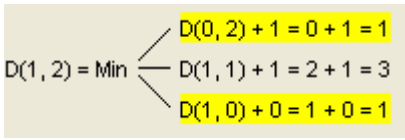


Figure 5 – Details of the calculation

If the characters of the current cell are equal the two are highlighted in red, otherwise in blue.

Needleman and Wunsch

This program differs from the simple dynamic programming by the scores that can be set for each alignment or mismatch.

Once the user has entered the two strings, the program automatically set a penalty gap score of -6.
Note: On this algorithm the penalty gap is considered to be the score of the gap (and in fact the interface labels changes accordingly).

The score can be set by pressing the button with the same name. Then a dialog will appear (see Figure 6) with the characters used in the two strings. If a character is not present in the selected scores, a ? will appear at its place.

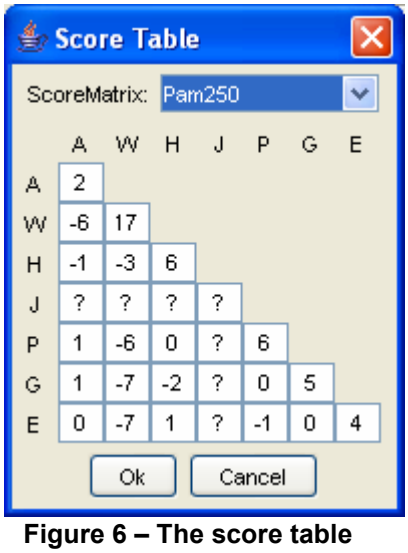


Figure 6 – The score table

The scores are showed on the grid as a little number at the upper right corner of each cell (see cell example on the right).

The details of calculation are changed accordingly.

Smith and Waterman

Like on Needleman & Wunsch, also this program features the scores.
But when it calculates the value of the current cell, it has a fourth choice apart from the typical dynamic programming: the zero. This is shown on the details of calculation (see Figure 7).

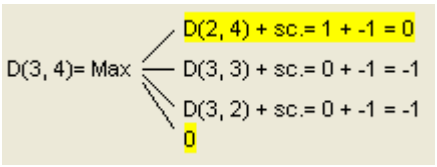


Figure 7 - Details of calculation showing the fourth choice

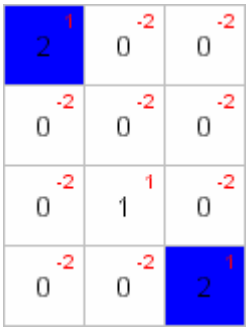


Figure 8 - Selecting the starting local

At the end of the *grid calculation* phase the user is requested to select a starting local (see Figure 8). If the user keeps pressing the *forward* button on the console, the program will select the most bottom-right one.

Four Russians

The Four Russians algorithm is based on a simple dynamic programming, therefore it does not have scores. This is because in order to decrease the number of preprocessed blocs it relies on the assumption that the values variation from cell to cell is just an offset between -1 and 1.
This explain why when entering the gap penalty array, the program refuses to accept inputs with offset different from -1, 0 and 1.

The gaps are showed as main value in the cells, while the real value is in the position where on N&W was the score:

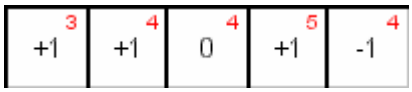


Figure 9 - Example of sequence with gaps

Once the strings have been set, the program automatically calculates the valid subdivision values.
These values represent the size of the blocs (the parameter *t* in the Four Russians algorithm), and are showed on a slider right on the bottom of the console (see example on the right).

When the user changes this value the grid is partitioned, and the informative label (upper right of the window) shows the number of blocs to precompute.

At this point if the user presses the forward button, the program starts precomputing the blocs:

Figure 10 - Precomputing the blocs

When the program is finished it shows an alert.

If the user tries to start a precomputation which will need a huge amount of memory, he will be warned by a special message box:

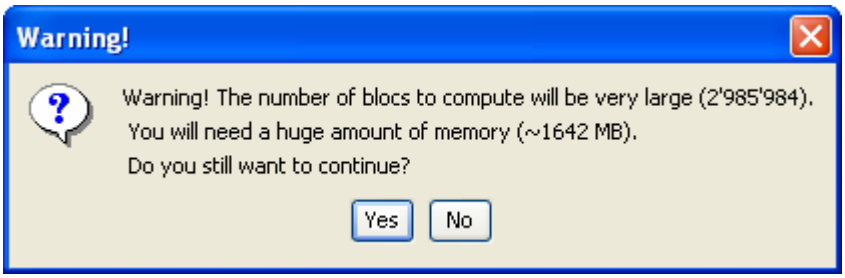


Figure 11 - The precomputation warning

Once the computation is done the user can see the resulting blocs by pressing the “show” button. Then a window will open with all the precomputed blocs available for review (see Figure 12).

Because the precomputed blocs must have normal values, and not offset, the showed blocs draw the values on the center of the cells, and the offset on the upper right angle (the opposite of what happen in the main grid).

By pressing again the forward button, the algorithm starts using the precomputed blocs to fill the grid. The index of the precomputed bloc used is showed in the center of the bloc, so that the user can find it by entering the range in the precomputed bloc list (see Figure 13).

The blocs are found by using the values highlighted in green. These are the parameters of the *restricted bloc function*.

When all the blocs have been assigned, the normal backtrack procedure applies.

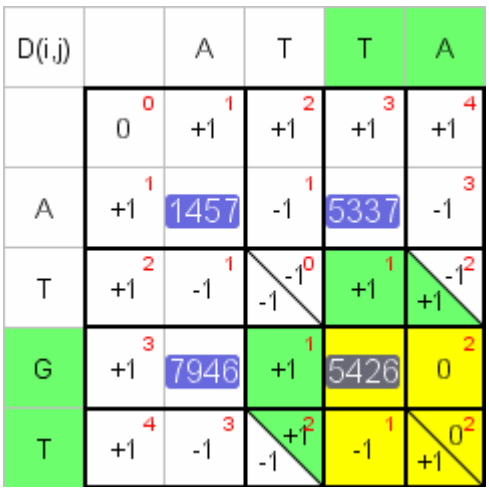


Figure 13 - The blocs on the table

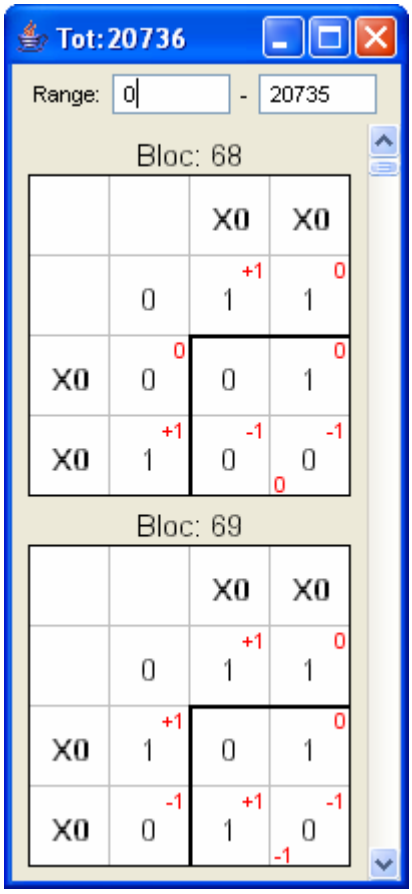


Figure 12 - The blocs list

3. Conclusion

While programming this software I had the opportunity to study in depth the problematic of creating a framework for algorithms demo, and review in every aspect the presented algorithms.
But this does not want to be a conclusion. The program will be published under the GPL[4] licence and will be available both on my webpage (soon ready) and on SourceForge[5], making it a first step for a further development available for everybody.

4. References

[1] “Algorithmes fondamentaux pour la comparaison de séquences”, Nadia El-Mabrouk. Pages 5 – 8.
[2] “Algorithmes fondamentaux pour la comparaison de séquences”, Nadia El-Mabrouk. Page 11.
[3] “Algorithmes fondamentaux pour la comparaison de séquences”, Nadia El-Mabrouk. Pages 13 – 14.
[4] "Algorithms on Strings, Trees and Sequences - Computer Science and Computational Biology", Dan Gusfield, Cambridge University Press, 1997. Pages 302 – 307.
[5] Schwartz RM, Dayhoff MO. "Matrices for detecting distant relationships." (In) *Atlas of Protein Sequence and Structure*, 5 suppl. 3:353-358 (1978), Nat. Biomed. Res. Found., Washington D.C.
[6] Henikoff S, Henikoff JG. "Amino acid substitution matrices from protein blocks". *Proc Natl Acad Sci U S A.* 1992 Nov 15;89(22):10915-9.
[7] T. Müller, R. Spang, and M. Vingron. "Concepts for the estimation of amino acid substitution models". Mol. Biol. Evol., 2001.

[4] http://www.gnu.org/copyleft/gpl.html
[5] http://sourceforge.net/