Finding global similarities between proteins using vector algorithms[1]

Sylvie Hamel

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, CP.6128 Succ. Centre-Ville
Montréal, Québec, Canada, H3C 3J7
phone:+1 514 343-6111(3504) fax:+1 514 343-5834
sylvie.hamel@umontreal.ca

Abstract

Global similarity between proteins was first discussed by Needleman and Wunsch in 1970 [7]. The classic solution computes an $n \times m$ table, where n and m are the length of the two proteins being compared, in $\mathcal{O}(nm)$ time. This article presents new efficient vector algorithms for the *global string similarity problem*. The key idea is to first reduce the computation to a Moore automaton, as done in [1] for the approximate string matching problem using edit distance, and then to obtain the output of this automaton in a bounded number of steps, regardless of the length of the input. This yields a very efficient algorithm since the operations are bit-wise operations widely available in processors.

Keywords: Global similarity, vector algorithms, automata, proteins

1 Introduction

The edit distance is often used to formalize the relatedness of two strings. On the other end, when dealing with proteins or biological sequences, a preferred way to formalize the relatedness of strings is to measure their *similarity*. Several similarity tables between amino acid are currently used and all of them are based on the idea that chemically closed amino acid are more similar, and substitutions between them should appear more often during evolution. The dominant similarity matrices used to align proteins are the PAM matrices of Dayhoff[3] and the BLOSUM matrices of Henikoffs[4].
Global alignment of proteins is important since, for instance, it can help us decide if two proteins are member of the same family, which in turn gives us insight about the functions of the proteins. The classic way to find a global alignment between two proteins, referred to as the Needleman-Wunsch algorithm[7], is to compute an $n \times m$ table, where n and m are the length of the two proteins being compared, in $\mathcal{O}(nm)$ time. Here, we want to present new efficient vector algorithms for the *global string similarity problem*, where the key idea is to first reduce the computation to a Moore automaton. Now, given a deterministic finite Moore automaton, and an input sequence $e_1 \dots e_m$, we are interested in the output sequence $r_1 \dots r_m$ of *visited states*. Since executing one transition is usually considered to be a constant time operation, the output sequence can be obtained in $\mathcal{O}(m)$ time.
One way to accelerate the computations is to exploit the parallelism of vector operations. For example, in [2] and [5], bit-vectors are used to code the set of states of a non-deterministic automaton. Another approach, developed in [6] and generalized in [1], uses bit-vectors to code both the input and output sequence, and computes the output with a *bounded* number of bit-wise operations on the input. This is this approach that we will generalize here to the global string similarity problem.
The article is organized as follow. I will first describe the global similarity problem in general and the classic solution to solve it. I will then show how we can restate the problem in terms of a Moore automaton, which we will use to derive our vector algorithm. I will finally discuss the complexity of the algorithm when different similarity matrices are used.

2 Global similarity

I will begin this section by given the basics of global string similarity and the classic algorithm, referred as Needleman-Wunsch algorithm [7], to compute the optimal alignment of two strings, given a similarity matrix. I will then show how we can restate the problem using an automaton and use vector operations to accelerate the com-

[1]with the support of NSERC and FQRNT

putation.

2.1 Basics of global string similarity

The *global string similarity problem* is to find a *maximal score* alignment between two strings, given a similarity matrix s on the input alphabet. For the purpose of this article, we will consider the elements, in the similarity matrix s, to belong in the interval $[-2c, M]$, where c is the cost of a deletion or an insertion in an alignment, and M is the maximal element of s. More formally, we have the following definitions.

Definition 2.1 *Let x and y be two strings on an input alphabet Σ. Let $\Sigma' = \Sigma \cup \{-\}$ denote the alignment alphabet, where "$-$" correspond to the insertion or deletion of a letter in one of the strings. Then, for any two letters a and $b \in \Sigma'$, $s(a, b)$ denotes the* **score** *obtained by aligning a with b.*

Definition 2.2 *Let $x = x_1x_2 \dots x_m$ and $y = y_1y_2 \dots y_n$ be two strings on an input alphabet Σ. An* **alignment** *A of x and y is given by two strings x', y' on the alphabet Σ', of length $l \geq max(m,n)$. Each column of the alignment have one of the following forms:*

$$\begin{bmatrix} x_i \\ y_j \end{bmatrix} \quad or \quad \begin{bmatrix} - \\ y_j \end{bmatrix} \quad or \quad \begin{bmatrix} x_i \\ - \end{bmatrix}.$$

The **score** *of the alignment A is then defined as $\sum_{i=1}^{l} s(x'(i), y'(j))$.*

Definition 2.3 *Given a pairwise scoring matrix s over Σ', the* **similarity** *of two strings x and y over Σ is the maximal score of any alignment of these two strings.*

The classic solution, to compute the global similarity of two strings x and y, and an associated optimal alignment, required $\mathcal{O}(nm)$ times. It can be computed by dynamic programming with the following recurrence, where $V(i, j)$ is the score of the optimal alignment of the prefixes $x[1..i]$ and $y[1..j]$ and c is the cost of an insertion or a deletion,

$$V(i,j) = \max \begin{cases} V(i-1, j-1) + s(x_i, y_j) \\ V(i, j-1) - c \\ V(i-1, j) - c \end{cases} \quad (1)$$

with initial conditions $V(i, 0) = -ci$ and $V(0, j) = -cj$. The entry $V[m, n]$ of the computed table V then give the maximal score of an alignment between x and y.

2.2 Restating the problem using an automaton

Classically the computation of each column of the table $V[0..m, 0..n]$ takes $\mathcal{O}(m)$ steps. In order to recast the computation as an automata computation, one needs to restrict the possible outputs to a range that is independent of m. This is done by noting that the differences between two adjacent horizontal or vertical cells in the table V are bounded.

Lemma 2.1 *$V(i, j) - V(i-1, j)$ and $V(i, j) - V(i, j-1)$ are in the interval $[-c, c + M]$, where M is the maximal element in the given similarity matrix s for the alignment.*

Proof. The relations clearly hold for the first line and the first column of the matrix V, since $V(0, j) = -cj$ and $V(i, 0) = -ci$. Suppose now that they hold for the pairs $(i-1, j)$ and $(i, j-1)$. We will show that they also hold for the pair (i, j). By Equation 1 we have that $V(i, j) - V(i-1, j) =$

$$\max \begin{cases} V(i-1, j-1) - V(i-1, j) + s(x_i, y_j) \\ V(i, j-1) - V(i-1, j) - c \\ -c \end{cases}$$

Thus $V(i, j) - V(i-1, j) \geq -c$. We have now to prove that $V(i, j) - V(i-1, j) \leq c + M$. By hypothesis, $V(i-1, j-1) - V(i-1, j)$ is in the interval $[-c - M, c]$ so it is $\leq c$ and, since by definition of M, $s(x_i, y_j) \leq M$, we certainly have $V(i-1, j-1) - V(i-1, j) + s(x_i, y_j) \leq c + M$. Finally, $V(i, j-1) - V(i-1, j) - c$ can be written as $(V(i, j-1) - V(i-1, j-1)) + (V(i-1, j-1) - V(i-1, j)) - c$. Since by hypothesis $V(i, j-1) - V(i-1, j-1) \leq c + M$ and $V(i-1, j-1) - V(i-1, j) \leq c$, we have that $V(i, j-1) - V(i-1, j) - c$ is also $\leq c + M$. A similar argument holds to show that $V(i, j) - V(i, j-1)$ is in $[-c, c + M]$.

Now, with the notations of Equation 1, define:

$$\begin{aligned} \Delta v_{i,j} &= V(i-1, j) - V(i, j) + c + M \\ \Delta h_{i,j} &= V(i, j) - V(i, j-1) + c \end{aligned}$$

From Lemma 2.1, $\Delta v_{i,j}$ and $\Delta h_{i,j}$ are in the interval $[0, 2c + M]$, and if the successive values of $\Delta h_{m,j}$ are known, then the value of the maximal *score*, $V[m, n]$, can easily be computed by the recurrence, $V[m, j] = V[m, j-1] + \Delta h_{m,j} - c$, and initial condition $V[m, 0] = -cm$.
With elementary arithmetic manipulations, Equation 1 translates as:

$$\Delta h_{i,j} = \max \begin{cases} \Delta v_{i,j-1} + s(x_i, y_j) - M \\ \Delta v_{i,j-1} + \Delta h_{i-1,j} - (2c + M) \\ 0 \end{cases} \quad (2)$$

with initial conditions $\Delta v_{i,0} = 2c + M$ and $\Delta h_{0,j} = 0$. We can thus define an automaton $\mathcal{B}$ that will compute the sequence $\boldsymbol{\Delta h_j} = \Delta h_{1,j} \dots \Delta h_{m,j}$ given the sequence of pairs $\boldsymbol{(\Delta v_{j-1}, S(x, y_j))} = ((\Delta v_{1,j-1}, S(x_1, y_j)) \dots (\Delta v_{m,j-1}, S(x_m, y_j))$, where $S(x_i, y_j) = M - s(x_i, y_j)$.
The states of $\mathcal{B}$ are $\{0, \dots, 2c + M\}$, with initial state 0, and the transition function F of $\mathcal{B}$ is given by:

$$F(state, (\Delta v, S)) = \max \begin{cases} \Delta v - S \\ \Delta v + state - (2c + M) \\ 0. \end{cases} \tag{3}$$

for an event $(\Delta v, S)$ in the Cartesian product $\{0, \dots, 2c + M\} \times \{0, \dots, 2c + M\}$.
After the computation of $\boldsymbol{\Delta h_j}$ by the automaton $\mathcal{B}$, $\boldsymbol{\Delta v_j}$ can be computed using the relation $\Delta v_{i,j} = \Delta h_{i-1,j} + \Delta v_{i,j-1} - \Delta h_{i,j}$, which leads to the vector equation

$$\boldsymbol{\Delta v_j} = \uparrow_0 \boldsymbol{\Delta h_j} + \boldsymbol{\Delta v_{j-1}} - \boldsymbol{\Delta h_j},$$

where $\uparrow_0 \boldsymbol{\Delta h_j}$ stands for a right shift of the vector $\boldsymbol{\Delta h_j}$ with the value 0 filled in in the first position.

2.3 From automata to vector algorithms

In the last section, we have reduce the computation of $V[m, j]$ to a few vector operations and one automata computation, which still takes $\mathcal{O}(m)$ steps. Now, we will recall from [1], that for a special class of automata, the *shellable* automata, there is a general way to produce a vector algorithm that computes the output - regardless of its length- in $\mathcal{O}(q)$ steps, where q is the number of states of the automaton. (Here the output is the sequence of states that the automaton will visit on a given input.) We will then show that the automaton $\mathcal{B}$, described by the transition function of Equation 3, is shellable.

Definition 2.4 *Let $\mathcal{A}$ be a finite automaton on alphabet of events Σ, with states Q and transition function $Q \times \Sigma \xrightarrow{F} Q$. A state s is* **shellable** *if and only if for all transitions $x \in \Sigma$, the fact that there exist a state $s' \neq s$ such that $F(s', x) = s$ implies that x is a reset to s (for all states $q \in Q$ we have $F(q, x) = s$).*

The **indicator set** I_s of a shellable state s is the set of events $e \in \Sigma$ such that $\forall t \in Q,\ F(t, e) = s$, i.e the set of resets to s. Let the function $Ind(s, \boldsymbol{X}, \boldsymbol{Y})$ be defined as:

$\boldsymbol{X} \vee [\boldsymbol{Y} \wedge (\neg \boldsymbol{X} +_{\boldsymbol{b}} \neg(\boldsymbol{X} \vee \boldsymbol{Y}))]$ if s is initial
$\boldsymbol{X} \vee [\boldsymbol{Y} \wedge \neg(\boldsymbol{X} +_{\boldsymbol{b}} (\boldsymbol{X} \vee \boldsymbol{Y}))]$ otherwise.

Proposition 2.1 (Bergeron-Hamel [1]) *If s is shellable, and if L_s is the set of event looping on s but not in I_s, then*

$$\boldsymbol{(r = s)} = Ind(s, \boldsymbol{e \in I_s}, \boldsymbol{e \in L_s}),$$

where $\boldsymbol{r} = r_1 \dots r_m$ is the sequence of states visited by the automaton on the input sequence $\boldsymbol{e} = e_1 \dots e_m$ and $\boldsymbol{(r = s)}$, is the boolean vector $(r_1 = s, \dots, r_m = s)$. (For the proof, see [1].

Let $\mathcal{A}$ be a complete deterministic automaton and $\mathcal{A} \setminus \{s\}$ be the automaton obtained from $\mathcal{A}$ by removing state s, and all its pending arrows. Then if s is shellable, $\mathcal{A} \setminus \{s\}$ is still a complete automaton on the alphabet $\Sigma \setminus I_s$, since $F(r, a) \neq s$, if a is not in I_s.

Definition 2.5 *An automaton $\mathcal{A}$ is* **shellable** *if it has one state, or if it has one shellable state s, and $\mathcal{A} \setminus \{s\}$ is shellable.*

When an automaton $\mathcal{A}$ with d states is shellable, there is an induced ordering on its states, starting from the first shellable state, and then the next, and so on. We can thus relabel the states of $\mathcal{A}$ and assume that $Q = \{0, 1, \dots, d - 1\}$.

Theorem 2.1 (Bergeron-Hamel [1]) *If an automaton $\mathcal{A}$ is shellable, then there exists a vector algorithm for $\mathcal{A}$.*

Proof. Since the proof of this theorem can be found in [1], let us just give here, for further use, a vector algorithm for a shellable automaton $\mathcal{A}$ in a code-like style. Let $Q = \{0, 1, \dots, d - 1\}$ be an ordering of the d states of $\mathcal{A}$ such that $\mathcal{A} \setminus \{0, 1, \dots, k - 1\}$ is shellable for state k, and let i be the initial state of $\mathcal{A}$. Given the input sequence $\boldsymbol{e} = e_1 e_2 \dots e_m$, we will compute the output $\boldsymbol{r} = r_1 r_2 \dots r_m$ by recursively computing the boolean vectors $\boldsymbol{(r = k)}$ for k in $\{0, 1, \dots, d-1\}$. Define the vector $\boldsymbol{K}$ as $\boldsymbol{(r < k)}$. We first compute $\boldsymbol{(r = 0)}$, and initialize $\boldsymbol{K}$ to this result:

$$\begin{aligned} \boldsymbol{(r = 0)} &\leftarrow Ind(0, \boldsymbol{e \in I_0}, \boldsymbol{e \in L_0}) \\ \boldsymbol{K} &\leftarrow \boldsymbol{(r = 0)} \end{aligned}$$

Then, for the successive values of k in $\{1, \dots, d - 2\}$, we execute the three instructions:

$$
\begin{array}{rcll}
\boldsymbol{N} & \leftarrow & [(\uparrow_{i<k} \boldsymbol{K}) \wedge (\boldsymbol{F}(\uparrow_i \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})] & \\
 & & \vee\ [\neg \boldsymbol{K} \wedge (\boldsymbol{e} \in \boldsymbol{I_k})] & (I1) \\
(\boldsymbol{r} = \boldsymbol{k}) & \leftarrow & Ind(k, \boldsymbol{N}, \boldsymbol{e} \in \boldsymbol{L_k}) \wedge \neg \boldsymbol{K} & (I2) \\
\boldsymbol{K} & \leftarrow & \boldsymbol{K} \vee (\boldsymbol{r} = \boldsymbol{k}) & (I3)
\end{array}
$$

The value of $(\boldsymbol{r} = \boldsymbol{d-1})$ is finally easily computed as the complement of the last vector $\boldsymbol{K}$.
It should be clear, from this presentation, that apart from the test $\boldsymbol{F}(\uparrow_i \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k}$, the algorithm requires $\mathcal{O}(d)$ elementary steps, since all the vectors of the form $(\boldsymbol{e} \in \boldsymbol{S})$ can be precomputed. In the next section, in the particular case of global similarities, we will give a simple recursive definition of the test $(\boldsymbol{F}(\uparrow_i \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})$.

3 A vector algorithm for the global similarity problem

In this section, we will first show that the automaton $\mathcal{B}$, described in section 2.2 is shellable. We will then derive from that fact, a vector algorithm for the global similarity problem.

Theorem 3.1 *The automaton $\mathcal{B}$, described by the transition function of Equation 3, is shellable.*

Proof. We will show that, for $k \in \{2c+M, \ldots, 0\}$, k is shellable in $\mathcal{B}_k = \mathcal{B} \setminus \{k+1, \ldots, 2c+M\}$, with the set of transitions $(\Delta v, S)$ such that $\Delta v - S = k$. First note that, since $\Delta v - S > k$ implies that $F(state, (\Delta v, S)) > k$, the only remaining transitions in $\mathcal{B}_k$ are those that satisfy $\Delta v - S \leq k$. If $\Delta v - S = k$, then $\max(\Delta v - S, \Delta v + state - (2c+M), 0) = k$, since $state \leq k$ for states in $\mathcal{B}_k$. If both $\Delta v - S < k$ and $state < k$, then the maximum is certainly less than k, and $F(state, (\Delta v, S) < k$.

Since automaton $\mathcal{B}$ is shellable, Theorem 2.1 can be used to produce a corresponding vector algorithm. Recall that the core of the general algorithm has three main instructions ((I1),(I2),(I3)), where i represent the initial state. Here, $\boldsymbol{K} = (\boldsymbol{r} > \boldsymbol{k})$, since the shellable states are in decreasing order in $\mathcal{B}$ ($2c+M$, is the first shellable state and 0 is the last). In order to adapt the algorithm to automaton $\mathcal{B}$, we need to derive suitable expressions for $(\boldsymbol{F}(\uparrow_i \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})$, $(\boldsymbol{e} \in \boldsymbol{I_k})$, and $(\boldsymbol{e} \in \boldsymbol{L_k})$. Theorem 3.1 gives an elementary test for recognizing the sets I_k. Indeed, since k is shellable with the set of events such that $\Delta v - S = k$, then, under the hypothesis that $(\boldsymbol{r} \leq \boldsymbol{k})$, we have:

$$(\boldsymbol{e} \in \boldsymbol{I_k}) = (\boldsymbol{\Delta v} - \boldsymbol{S} = \boldsymbol{k}).$$

Looping events are easily detected, since if $k = s > 0$, we need either $\Delta v = 2c+M$ or $\Delta v - S = k$ to loop on k. Since events of the form $\Delta v - S = k$ are in I_k, the only looping events not in I_k are those for which $\Delta v = 2c + M$. Thus, the test $(\boldsymbol{e} \in \boldsymbol{L_k})$ is independent of k and is given by the vector $(\boldsymbol{\Delta v} = \boldsymbol{2c} + \boldsymbol{M})$.
Finally, in order to compute the values of $(\boldsymbol{F}(\uparrow_{\boldsymbol{c}} \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})$, we define the following vector:

$$\boldsymbol{V_k} = \boldsymbol{\Delta v} + \boldsymbol{max}(\uparrow_{\boldsymbol{0}} \boldsymbol{r}, \boldsymbol{k}) - (\boldsymbol{2c} + \boldsymbol{M})$$

Clearly, $\boldsymbol{V_{2c+M}} = \boldsymbol{\Delta v}$, and we have the following relation between $\boldsymbol{V_k}$ and $\boldsymbol{V_{k-1}}$:

Proposition 3.1

$$\boldsymbol{V_{k-1}} = \boldsymbol{V_k} - \uparrow_{\boldsymbol{0}} \neg(\boldsymbol{r} > \boldsymbol{k})$$

Proof. First, it's easy to see that we have the following identity (where $(a \leq k-1)$ is the truth value 1, if $a \leq k-1$ and 0, otherwise):

$$\max(a, k-1) = \max(a, k) - (a \leq k-1) \quad (4)$$

By definition, we have $\boldsymbol{V_{k-1}} = \boldsymbol{\Delta v} + \boldsymbol{\max}(\uparrow_{\boldsymbol{0}} \boldsymbol{r}, \boldsymbol{k-1}) - (\boldsymbol{2c} + \boldsymbol{M})$. Using Equation 4, we get

$$
\begin{array}{rcl}
\boldsymbol{V_{k-1}} & = & \boldsymbol{\Delta v} + \boldsymbol{\max}(\uparrow_{\boldsymbol{0}} \boldsymbol{r}, \boldsymbol{k}) - \\
 & & (\uparrow_{\boldsymbol{0}} \boldsymbol{r} \leq \boldsymbol{k-1}) - (\boldsymbol{2c} + \boldsymbol{M}) \\
 & = & \boldsymbol{V_k} - (\uparrow_{\boldsymbol{0}} \boldsymbol{r} \leq \boldsymbol{k-1}) \\
 & = & \boldsymbol{V_k} - \uparrow_{\boldsymbol{0}} \neg(\boldsymbol{r} > \boldsymbol{k}).
\end{array}
$$

Thus, computing the vectors $\boldsymbol{V_k}$ is elementary. The next proposition shows that we can replace the computation of the vector $\boldsymbol{N}$ (Instruction I1) in the algorithm with an expression involving only $\boldsymbol{V_k}$ and known vectors. Note that the vector $\boldsymbol{N}$ can be written as

$$
\begin{array}{rcl}
\boldsymbol{N} & \leftarrow & [(\uparrow_{\boldsymbol{c}} \boldsymbol{r} > \boldsymbol{k}) \wedge (\boldsymbol{r} = \boldsymbol{k})] \\
 & & \vee\ [(\boldsymbol{r} \leq \boldsymbol{k}) \wedge (\boldsymbol{\Delta v} - \boldsymbol{S} = \boldsymbol{k})]
\end{array} \quad (5)
$$

by using the fact that $(\boldsymbol{F}(\uparrow_i \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})$ is equal to $(\boldsymbol{r} = \boldsymbol{k})$, and the identity $(\boldsymbol{e} \in \boldsymbol{I_k}) = (\boldsymbol{\Delta v} - \boldsymbol{S} = \boldsymbol{k})$, if $(\boldsymbol{r} \leq \boldsymbol{k})$.

Proposition 3.2 *The vector in Equation 5 is equal to*

$$
\begin{array}{c}
[((\uparrow_{\boldsymbol{c}} \boldsymbol{r} > \boldsymbol{k}) \wedge (\boldsymbol{V_k} = \boldsymbol{k})) \vee (\boldsymbol{\Delta v} - \boldsymbol{S} = \boldsymbol{k})] \\
\wedge (\boldsymbol{r} \leq \boldsymbol{k}).
\end{array} \quad (6)
$$

Proof. We will prove this by recalling the definitions of $\boldsymbol{r}$ and $\boldsymbol{V_k}$ component-wise:

$$r_i = \max(\Delta v_i - S_i, \Delta v_i + r_{i-1} - (2c + M), 0)$$

$$V_{ki} = \Delta v_i + \max(r_{i-1}, k) - (2c + M)$$

(6) true ⇒ (7) true: Suppose that $r_{i-1} > k$, and $r_{i-1} > k$, and $r_i = k$, than equation (6) is true and we will show that this implies $V_{ki} = k$ or $\Delta v_i - S_i = k$, which will make (7) true. Since, by definition, $r_i = \max(\Delta v_i - S_i, \Delta v_i + r_{i-1} - (2c + M), 0)$ and by hypothesis, $r_i = k$, we must have $\Delta v_i - S_i = k$ or $\Delta v_i + r_{i-1} - (2c + M) = k$. If $\Delta v_i - S \neq k$, then $\Delta v_i + r_{i-1} - (2c + M), 0) = k$, which implies that $r_{i-1} = k - \Delta v_i + (2c + M)$. And so, since $r_{i-1} < k$, $V_{ki} = \Delta v_i + r_{i-1} - (2c + M) = k$.

(7) true ⇒ (6) true: Now, suppose that $r_{i-1} > k$, $V_{ki} = k$ and $r_i \leq k$ so that (7) is true. Then $V_{ki} = k = \Delta v_i + r_{i-1} - (2c + M)$. By definition, we have $r_i \geq \Delta v_i + r_{i-1} - (2c + M)$, then $r_i \leq k$ implies $r_i = k$ and (6) is true.
Putting all these together, we have the following

Theorem 3.2 *There exist a vector algorithm to compute the global similarity of two proteins*

Proof. Putting everything we have done together, we have the following vector algorithm:

$$\begin{array}{rcl} \boldsymbol{(r = 2c + M)} & \leftarrow & \mathit{Ind}(2c + M, \boldsymbol{(\Delta v - S = 2c + M), \Delta v = 2c + M}) \\ \boldsymbol{K} & \leftarrow & \boldsymbol{(r = 2c + M)} \\ \boldsymbol{V} & \leftarrow & \boldsymbol{\Delta v} \end{array}$$

For the successive values of k in $\{2c + M - 1, \ldots, 0\}$, we execute the four instructions:

$$\begin{array}{rcl} \boldsymbol{N} & \leftarrow & [((\uparrow_0 \boldsymbol{K}) \wedge \boldsymbol{(V_k = k)}) \vee \\ & & \boldsymbol{(\Delta v - S = k)}] \wedge \neg \boldsymbol{K} \\ \boldsymbol{(r = k)} & \leftarrow & \mathit{Ind}(k, \boldsymbol{N}, \boldsymbol{\Delta v = 2c + M}) \wedge \neg \boldsymbol{K} \\ \boldsymbol{V} & \leftarrow & \boldsymbol{V} - \uparrow_0 \neg \boldsymbol{K}) \\ \boldsymbol{K} & \leftarrow & \boldsymbol{K \vee (r = k)} \end{array}$$

This vector algorithm allows the computation of a column of the table V, described by Equation 1, in $\mathcal{O}(2c + M \cdot \frac{m}{\omega})$ steps, where m is the length of the column and ω is the word size of the machine. The whole table can then be computed with $\mathcal{O}(n \cdot 2c + M \cdot \frac{m}{\omega})$ steps.

4 Conclusion and comments

We are currently driving tests to compare the time efficiency of our vector algorithm versus Needleman-Wunsch and Sellers algorithms. Unfortunatly, no results are ready yet. However, since our vector algorithms depends on M, the maximal element of the similarity matrix used, and c the cost of inserting or deleting a letter in the alignment, we can already state that our vector algorithm will be more efficient when used with a BLOSUM matrix. Here is a table of M and c values for different PAM and BLOSOM matrices:

Similarity matrix	M	c
PAM 120	12	4
PAM 250	17	4
BLOSUM 45	15	2
BLOSUM 62	11	2

Thanks: Many thanks to Anne Bergeron for showing me how fun and interesting the area of biocomputing is.

References

[1] A. Bergeron and S. Hamel, *Vector Algorithms for Approximate String Matching*, International Journal of Foundations of Computer Science, 13-1, (2002), 53–66.

[2] R. A. Baeza-Yates and G. H. Gonnet, *A New Approach to Text Searching*, Communications of the ACM, 35, (1992), 74-82.

[3] M.O.Dayhoff, R.M. Schwartz and B.C. Orcutt, *A model of evolutionary change in proteins*, Atlas of Protein Sequence and Structure, 5, (1978), 345-352.

[4] S. Henikoff and J.G. Henikoff, *Amino acid substitution matrices from proteins blocks*, Proc. Natl. Academy of Science, 89:10, (1992), 915-919.

[5] J. Holub and B. Melichar, *Implementation of Nondeterministic Finite Automata for Approximate Pattern Matching*, in Automata Implementation, LNCS 1660, Springer-Verlag, (1999), 92-99.

[6] G. Myers, *A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming* , J. ACM, 46-3, (1999), 395-415.

[7] S.B. Needleman and C.D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, J. Mol. Biol., 48, (1970), 443–453.

[8] P. H. Sellers, *The Theory and Computation of Evolutionary Distances*, J. of Algorithms, 1, (1980), 359-373.