MULTIPLE STRING ALIGNMENT

Matteo Barigozzi

Paolo Pin

Introduction

Every DNA molecule can be described, avoiding its tridimensional structure, as a string (*genome*) of elements from a set of cardinality only four, whose elements (*basis*) can be listed as A, C, G, T.

Sequences of DNA are repeated many times through the genome without yet understood biological function. DNA is in every living cells and whenever a cell duplicates itself every new offspring get a complete copy of the original DNA. During these replication events mismatches may happen, due to insertions, deletions or substitutions (actually also finite replications may happen, but these can be seen as multiple insertions).

Biological experimental research has actually shown that not every region of the DNA has the same probability to be object of a change. This is due to the supposed purpose of every single region and to evolutionary reasons (a change in a region that does not effect macroscopical properties would be more likely). We will anyhow avoid all the considerations of this kind.

This simple algebraic representation can also be applied to other common biological structures such as amminoacids, with different sets of base elements. In order to highlight the similarities and differences among the instances of such strings we want to define a good method of comparison. To do so we start from comparison between two strings, but first of all we need some definitions.

Note that Σ is any given alphabet and Σ^* is the set of every finite string on it.

Definition 1 . MULTIPLE SEQUENCE ALIGNMENT MSA*: given a set of strings $S = \{s_1, \dots, s_n\}$ with $s_i \in \Sigma^*$, a Multiple Sequence Alignment is a set of strings $A = \{a_1, \dots, a_n\}$ with $a_i \in (\Sigma \cup \{-\})^*$, and $- \notin \Sigma$ is called 'gap' character. Moreover $\forall\ a_i \in A : |a_i| = k$ and removing all gaps from a_i we obtain s_i.* $\diamondsuit$

Definition 2 . *$\mathcal{A}$ is the set of all possible MSA from a given S.* $\diamondsuit$

Definition 3 . Scoring function *F is every function from $\mathcal{A}$ to $\mathbb{R}$.* $\diamondsuit$

Definition 4 . Optimal MSA *is the $\bar{A} \in \mathcal{A}$ such that $F(\bar{A}) = max F(A)$.* $\diamondsuit$

The MSA problem is then to find the optimal MSA. As seen in the definitions we can use every alphabet.

To practically face this problem we must define a scoring function f between the characters. Let's consider the following trivial scoring scheme for the binary alphabet $\{0, 1\}$:

f	$'0'$	$'1'$	$'-'$
$'0'$	1	0	0
$'1'$	0	1	0
$'-'$	0	0	0

In principle every scoring function is possible, but for our purpose we'll accept only those in which: $f(\sigma, \tau) \in [0, 1]$ $\forall \sigma, \tau \in \Sigma \cup \{-\}$, $f(-, -) = 0$ and $f(\sigma, \sigma) = 1$ $\forall \sigma \in \Sigma$, it's moreover clear that a symmetric function has to be preferred.

We have assumed as equally probable the operations of changing of characters (in both the ways), deletion and insertion, if not we can easily modify the scoring table (as well as could be done using a larger alphabet).
From such a table a very common scoring function, for an MSA of n strings, is Sum of pairs SP: sum of the values of the $\binom{n}{2}$ couples on every column, and summatory of them all.

Note that a column of all gaps gives no contribution to the global score. Note also that the table we used makes the calculation of this scoring function very easy.

Example 1 *Consider the following alignment where $\Sigma = \{0, 1\}$:*

$$\begin{matrix} 1 & 0 & - & 0 & 1 & 0 & - \\ - & 0 & - & 1 & 1 & 0 & 1 \\ 1 & - & 0 & 0 & 1 & 1 & 1 \end{matrix}$$

This MSA will have a score of 8 *for the SP method.* $\diamondsuit$

The MSA problem for only two strings is polynomial, and the most used algorithm is given by Needleman and Wunsch (NW) [7], it's running time is $O(k^2)$, where k is the length of the strings after the alignment (it's bounded by the sum of the lengths of the original strings). The inputs for this algorithm are clearly the two given strings, of hypothetical length h and l. Then it will construct an $(h+1) \times (l+1)$ matrix on which it will find a path (as we'll see in the next chapter). The output will be the optimal MSA, or eventually only its score s and length k.

The MSA problem for n strings is instead an NP-complete problem as first shown by Wang and Jiang (1994) [9]. More precisely it can be proved that, using a generalization of NW dynamic programming, it's possible to find an optimal MSA. Anyway the running time is $O((2k)^n)$, so it is really a hard problem when the number of strings considered is big, which is actually the case for typical problems of molecular biology (usually the length of a protein is $k \simeq 350$ and a typical protein family has hundreds of members ($n \geq 100$)). We'll give independently an idea of the NP-completeness of the MSA problem reducing it to the Traveling Salesman one (but this reduction is not complete and does not fully solve it).

To get an approximated solution one common algorithm suggested by literature [4] is the Center Star Alignment Algorithm CSA: NW is applied on all the possible couples of strings s and t, being $F(s,t)$ their score. For each string t we define a number: $\sum_{s \in S - \{t\}} F(t,s)$. We can now take the string that minimize this number. Every other string is compared with this minimal one with NW. Adding then proper gaps to them all we obtain the final MSA. This algorithm runs in $O((nk)^2)$, where n is the number of strings and k as before (here k is bounded by the sum of the lengths of all the strings). It gives a score that is upper bounded by the double of the optimal one.

We'll adopt instead another algorithm, for which we need to define a distance between all the original strings in $S = \{s_1, \ldots, s_n\}$.

Definition 5 . Distance $d(s,t)$: *given a set of strings S, let's apply NW to all the $\binom{n}{2}$ possible couples, for every couple (s,t) we'll have a score F and a length k, then the distance d is defined: $d(s,t) = k(s,t) - F(s,t)$.* $\diamondsuit$

It's easy to prove that this is actually a distance (it satisfies also triangular inequality).

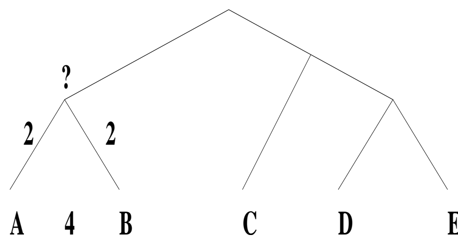
Traveling Salesman Approach TSA: as before NW is applied on all the couples to construct a complete graph, to whom is applied the TSP,

considering the distance defined above. This is justified by the research of the tree in which all the strings are leaves and the nodes are common unknown ancestors. As before a proper choose of gaps will lead to MSA.
Here is an example showing this process:

Example 2 *Consider the five sequences of amminoacids A, B, C, D, E which lead to the following MSA:*

```
A: RPCVCP---VLRQAAQ--QVLQRQIIQGPQQLRRLF-AA
B: RPCACP---VLRQVVQ--QALQRQIIQGPQQLRRLF-AA
C: KPCLCPKQAAVKQAAH--QQLYQGQLQGPKQVRRAFRLL
D: KPCVCPRQLVLRQAAHLAQQLYQGQ----RQVRRAF-VA
E: KPCVCPRQLVLRQAAH--QQLYQGQ----RQVRRLF-AA
```

These sequences can be represented as leaves of one hypothetic common ancestor (the root) and their order depends on their mutual distances as represented in the phylogenetic tree given below.



From the MSA given above we see that the distance between strings A and B is 4 *so they have both a distance of* 2 *from their hypothetic common ancestor. The same reasoning can be applied to all other strings and in this way the phylogenetic tree is built.* ♢

This algorithm has the same running time of the previous, and the efficiency of many algorithms on the classical traveling salesman problem, with a distance satisfying triangular inequality, will lead to good results. The advantage of this approach is strong in evolutionary studies when considering phylogenetic trees as it is explained below and in the previous example.
We'll also try an heuristic approach with Simulated Annealing (SA) to compare the result of our TSA approach, and to try to improve it.

1 Needleman-Wunsch algorithm

The Needleman-Wunsch (NW) is a dynamic programming algorithm which has as input two strings a, b of length h and l and will try to maximize the

score between these two strings giving as output their optimal global alignment. NW involves an iterative method of calculation in which all possible pairs of characters are represented in a $(h+1) \times (l+1)$ matrix and all possible alignments are represented by pathways through this matrix. There are three main steps in dynamic programming:

1. Initialization.
2. Matrix fill (scoring).
3. Traceback (alignment).

1.1 Initialization step

First of all we create an $(h + 1) \times (l + 1)$ matrix, using a given scoring scheme. One possibility is represented by the scoring scheme f, defined in the introduction. In general f has the following properties:

- match score $f(\sigma, \sigma) = 1 \ \forall \sigma \in \Sigma$;
- mismatch score $f(\sigma, \tau) = 0 \ \forall \sigma, \tau \in \Sigma$, with $\sigma \neq \tau$;
- gap penalty $f(-, -) = 0; f(\sigma, -) = w(\sigma) \ \forall \sigma \in \Sigma$

If there are no gap penalties ($w = 0$), which will be our case, the first row and the first column can be initially filled with 0. Then a numerical value is assigned to every position in the matrix depending on the matching or mismatching of the two characters as given by f.
This last step can be omitted and included in the next one as we have done practically writing our program in the function *nw*.

1.2 Matrix fill

Since all the elements of the first row and of the first column are defined (they are all 0), we can fill all the other positions of the matrix. For each position we want the maximum score $M_{i,j}$ for an alignment ending at that point. Starting from position $(1, 1)$ we proceed as follows:

$$\begin{aligned} M_{i,j} \ &= max[\\ &M_{i-1,j-1} + f(a_i, b_j), \quad \text{(match/mismatch in diagonal)} \\ &M_{i,j-1} + w, \quad \text{(gap in string a)} \\ &M_{i-1,j} + w \ ] \quad \text{(gap in string b)} \end{aligned}$$

This is the general formula with the possibility of gap penalties given by w which in our case is always 0. Repeating this procedure for all the positions we fill the whole matrix and so we are ready for the final step.

1.3 Traceback

The traceback steps begins at position h, l in the matrix (the lower right hand corner), which is the position which leads to the maximal score denoted by $F(a, b)$. Traceback takes the current cell and looks to the neighbor cells that could be direct predecessors. It looks to:

- the neighbor to the left (gap in string b);
- the diagonal neighbor (match/mismatch);
- the neighbor above (gap in string a).

The algorithm chooses as next position the one with the maximum score among all the possible predecessors and, searching for the quickest way, moves upper left until it reaches position $(0, 0)$ where it stops. It could be noticed that there are more alternative solutions each resulting in an optimal alignment with the same maximal score $F(a, b)$. We wrote the functions *nw* and *function* to run the NW algorithm. As output we get the global optimal alignment between two strings, each of them now is of length k and their distance is defined as $d(a, b) = k(a, b) - F(a, b)$. Anyway for the solution of MSA we need to know just the distance between two strings which is computed by the function *nwdist*. The codes of these functions are given in the appendix.
Finally, as already noted in the introduction, the running time of NW is polynomial, more precisely it runs in time $O(k^2)$.

We can consider the following example obtained with the function *nw*:

Example 3 *First of all from an alphabet of 4 characters (like DNA), we get the two random strings using the function 'genstr' (see the appendix):*

`a) TAGTCCTCA`

`b) TCCAGCCCCAGGA`

Then executing the first two steps (initialization and matrix fill) we get the following matrix:

$b \backslash a$	-	T	A	G	T	C	C	T	C	A
-	**0**	0	0	0	0	0	0	0	0	0
T	0	**1**	1	1	1	1	1	1	1	1
C	0	**1**	1	1	1	2	2	2	2	2
C	0	**1**	1	1	1	2	3	3	3	3
A	0	1	**2**	2	2	2	3	3	3	4
G	0	1	2	**3**	**3**	3	3	3	3	4
C	0	1	2	3	3	**4**	4	4	4	4
C	0	1	2	3	3	4	**5**	**5**	5	5
C	0	1	2	3	3	4	5	5	**6**	6
C	0	1	2	3	3	4	5	5	**6**	6
A	0	1	2	3	3	4	5	5	6	**7**
G	0	1	2	3	3	4	5	5	6	**7**
G	0	1	2	3	3	4	5	5	6	**7**
A	0	1	2	3	3	4	5	5	6	**7**

The last step (traceback) follows the highlighted path in the previous table and leads to the following optimal alignment:

```
a) T--AGTCCTC-A---
b) TCCAG-CC-CCAGGA
```

From this example we get also that $F(a, b) = 7$ *and* $k(a, b) = 15$ *so the distance between the two strings is* $d(a, b) = 8$*, which is exactly the number of mismatching characters* $\diamondsuit$

The Needleman-Wunsch procedure could be applied with the same criteria to any given number n of strings. It would construct an n-dimensional matrix and search a path on it. Clearly it would lead to a running time of $O((2k)^n)$ (as said in the introduction), that becomes soon useless as n increases (which unluckily is exactly the situation faced in solving practical problems).

2 Traveling salesman approach

To solve the MSA problem we used a new approach based on the Traveling Salesman problem. This way is suggested by G.Gonnet et al. of ETH in Zurich [3].
We split the problem in two parts:

1. ordering of the strings which is equivalent to the TSP;

2. gap insertion which gives the MSA.

2.1 Circular order of the strings

We can define a tree $T(S)$ with as leafset the strings $\{s_1 \dots s_n\}$ to be aligned and with internal nodes representing the common ancestors of the strings, these ancestors are usually unknown. The distance between two strings evaluates the probability of evolutionary events on the edges that link the strings, the shorter is the distance the higher is the probability that the strings have a common ancestor. To build the tree in the correct way we must put the sequences in the right order, i.e. we must minimize the distance between all the couples. This is almost equivalent to solving the TSP; as the n cities we have the strings with the $\binom{n}{2}$ distances between them that can be computed with NW. Of course also the TSP is NP-complete but there are many good approximated algorithms given in the literature to solve the problem.
One of these methods is suggested by Rivest et al. in [2] and is the following:

- build a minimum spanning tree among the n cities; once the first (root) is chosen the next are added one by one in such a way to take always the one nearest to the ones already checked;
- the traveling salesman path is given moving around the minimum spanning tree from the leaves to the root and from the left to the right.

Anyway it must be noticed that if, the distance doesn't satisfy triangular inequality, good approximation algorithms cannot be found in polynomial time, unless P = NP (see [2]). So to solve TSP we must use a distance which satisfies the triangular inequality; in particular the distance $d(a, b) = k(a, b) - F(a, b)$ satisfies all the properties of a real distance:

1. $d(a, a) = 0$, $\forall a \in S$;
2. $d(a, b) > 0$, $\forall a, b \in S$ with $a \neq b$;
3. $d(a, b) = d(b, a)$, $\forall a, b \in S$;
4. $d(a, b) \leq d(a, c) + d(c, b)$ $\forall a, b, c \in S$;

This definition of distance is very natural, indeed it counts the number of characters in which two strings differ from each other.

Instead of searching for a minimum spanning tree we used the function *order* which creates an ordered list of strings as follows:

- given the matrix of all the distances among the strings it looks for the two strings whose distance is maximum and put them at the beginning and at the end of the list;
- recursively, having a list, it searches the nearest string to the list, which is not already in it;
- the found string is added in just after the most nearest string among those already in the list;
- once the list is filled it runs Simulated Annealing (see next section for a general introduction to this algorithm) in such a way to minimize the distance between adjacent strings.

To use the previous function it is necessary to know all the $\binom{n}{2}$ distances between all the couples of strings, i.e. to build the matrix of distances. This is done using *nwdist* iteratively and it will take $O(n^2k^2)$ time.

This time is however much small if compared with the last Simulated Annealing step (as will be clearly seen in next section), especially increasing the number of the strings. In any case also the Simulated Annealing approach is almost linear with this number n.

2.2 Gap insertion

Now the n strings are ordered in the circular order explained in the previous section but they still have to be aligned. We assume that the first h strings are already aligned and an MSA $A' = \{a_1 \dots a_h\}$ is built.
We used an algorithm written in the function *insgap* which has four steps:

1. take string s_h and calculate an optimal pairwise alignment (using NW) with the next string s_{h+1}. The strings, in this alignment with the gaps, are called t_h and t_{h+1}.

2. insert all gaps from t_h that are not already present in a_h into all previously aligned strings $(a_1 \dots a_h)$.

3. take all gaps that were present in a_h before the previous step and insert them in both t_h and t_{h+1} except for the gaps that are already present in t_h.

4. at this point $a_h = t_h$ and $a_{h+1} = t_{h+1}$. We add a_{h+1} to the alignment and increment h by one.

The running time for this gap insertion step is $O(nk^2)$, indeed it just runs NW $n-1$ times. Once the alignment is complete the total score can be computed easily using the usual SP method as defined in the introduction.

The TSP approach presented above is a polynomial algorithm. It has a running time of $O(n^2k^2)$ to compute all pairwise alignment at the beginning plus a running time of $O(nk^2)$ to insert all the gaps (where k is the length of the longest string). Of course the TSP is an NP-complete problem (exponential in n), but as already said many approximation algorithm which run in polynomial time are possible, so that the most expensive part remains the pairwise alignment necessary to build the distance matrix.

As in the previous section we present the following example which is obtained using the function *insgap*:

Example 4 *The following ten strings randomly generated with 'genstr' are given:*

```
0) GTACTACG
1) CGAAATGG
2) TCCAGCCCCAGG
3) TCTAGTCCT
4) ACTTGTATCT
5) TCATCGCACG
6) TCGTCTCCTA
7) CTGCCCTGAGCC
8) CTTATAGG
9) GCACAACAAC
```

Then as explained before using iteratively 'nwdist' and then 'order' we get the strings correctly ordered:

```
0-0)GTACTACG
1-5)TCATCGCACG
2-8)CTTATAGG
3-6)TCGTCTCCTA
4-3)TCTAGTCCT
5-4)ACTTGTATCT
6-1)CGAAATGG
7-9)GCACAACAAC
8-7)CTGCCCTGAGCC
9-2)TCCAGCCCCAGG
```

Finally using 'insgap' we obtain the following MSA:

```
-----G---T-----------A-C-TACG-
---------T---------C-ATCGCACG-
-----C---T---------T-AT---A-GG
----TC--GT--------CTCCT---A---
----TCTAGT--------C-C-T-------
----ACTTGTA------TC---T-------
-----C--G-A-AA---TG---G-------
----GC----ACAAC--AA---C-------
-C-TGC-----C--CTGAG---CC------
TCCAGC-----C--C-CAG----G------
```

This MSA has an SP score of 105. ◇

TSA leads to very good results if we change our scoring functions, that is if, instead of considering all the possible couples of strings when computing a column score, we consider only subsequent ones (the first with the second, the second with the third ... and finally the last one with the first one).

In this case this approach (assuming that the Traveling Salesman problem was exactly solved) would give a result which is $\frac{n-1}{n}$ the optimal one (where n is as usual the number of the strings). It's not the best possible score because the last string and the first one are not considered.

3 Simulated annealing approach

The method of Simualted Annealing (SA) is an heuristic algorithm, a very useful one for finding the global extremum of a large scale system. It's a random search technique which exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure. The essence of the process is slow cooling (annealing) which allows time for the redistribution of the atoms as they lose mobility and is essential for ensuring that a low energy state is reached.

SA is a generalization, for other than thermodynamic systems, of the Metropolis algorithm [6] which is used to compute the Boltzmann average of any thermodynamic quantity. SA was proposed in 1983 (see [5]); it starts with a Metropolis Monte Carlo simulation at a high temperature and after a sufficient number of Monte Carlo steps, or attempts, the temperature is decreased. This process is repeated until the final temperature is reached. To apply SA one must provide the following elements:

1. a description of all possible system configurations;

2. a generator of random changes in the configuration, which we call moves, each of them is accepted with Boltzmann probability distribution as in Metropolis algorithm [6];

3. an objective function E (analog of energy) whose optimization is the goal of the procedure. In our case we look for a minimum of distances in the function *order* and for a maximum of score in the function *simann*;

4. a control parameter T (analogous to temperature) or its inverse β and an annealing schedule which tells how to increase β, i.e. after how many moves each upward step in β is taken and how large it is.

In the function *simann* we proceed as illustrated below.

1. Each possible MSA is a configuration of the system. The initial configuration can be either created at random or can be the output of the function *insgap* described in the previous section. The initial β depends on the configuration chosen: for random configuration we chose $\beta = 0.1$, whereas for a configuration given by *insgap* we can start from higher β given that the initial MSA is already nearly optimized.

2. The goal is to maximize the total score of the MSA using the SP scoring method.

3. Once the initial configuration and initial β are given, each move consists on finding a gap at random and changing it with the nearest non gap character in the same string. Each move gives a change ΔF in the SP score of the MSA:

   - if $\Delta F > 0$, the move is accepted for sure and the score is increased by ΔF ;
   - if $\Delta F < 0$, the move is accepted with the Boltzmann probability, i.e. we draw a uniform random number $r \in [0, 1]$ and
     - if $e^{-\beta(\Delta F)} > r$ the move is rejected and the score doesn't change;
     - if $e^{-\beta(\Delta F)} < r$ the move is accepted and the score is increased by ΔF.

4. The annealing schedule is such that the number of moves at each β-step must be large enough to explore the regions of search space that are filled with non gap characters; so we increase it with number of strings and the mean length of them (e.g. for 100 strings of 10 elements $2 \cdot 10^5$ moves are enough). The β-step chosen is linear ($\beta_{k+1} = \beta_k + 0.1$) according to the fact that the final score is improved with slower cooling rates, at the expense of greater computational effort. An exponential scheme ($\beta_{k+1} = c\beta_k$ where $c > 1$) could also be possible.

5. The algorithm stops when it's seen that no significant improvement happens increasing β further on.

In our work we choose experimentally the final β and the number of moves for each β-step so that increasing them the result cannot be appreciably improved anymore. Here is an example on how *simann* works:

Example 5 *As usual we consider the following random strings ($N = 10$) generated by 'genstr':*

```
0)      TCCAGCCCCAGG
1)      TCTAGTCCT---
2)      ACTTGTATCT--
3)      TCATCGCACG--
4)      TCGTCTCCTA--
5)      CTGCCCTGAGCC
6)      CTTATAGG----
7)      GCACAACAAC--
8)      GTGCGTTCGGG-
9)      ACGCTATGCCCT
```

The initial configuration is generated randomly spreading the elements of each string and building the following matrix:

```
TCC--AG-CC-CC-AGG-------
-----TCT-AGTCC--T-------
-A-C--TT-GTA----T-C---T-
-T-CAT-C-G--C-AC-G------
-T--CGT--C-TCCT--A------
-C----TGCCCTGAG----C--C-
----CT-T-A--TAG-G-------
G--C--AC--A-AC--AAC-----
GT-GCG---TTC--G-G---G---
A--C----GCT-AT----G-CCCT
Score: 63
```

The annealing schedule is set with $\beta_{init} = 0.1$ *and each step consisting of* 10^5 *random moves after which* β *is increased of* 0.1. *It stops at* $\beta_{fin} = 0.9$ *as it can be seen after this point increasing* β *will no more increase the score. Here are the results for some steps of SA:*

```
beta=0.1
T-C-----C-A-GC--CC--C-AGG
T---C---T--A-G-T-C----C-T
--A--C---TTG-T--A-T-C-T--
TC-A--T-C---G-C-A--C----G
TC-G---T--C--T--CC----T-A
-C--T-G-----C-C-C-T-GAGCC
--C-TT----A-T-A---G--G---
G-C-A-C--A-A----C---A--AC
GT-GC----G----T-T--CG-G-G
-A-C--G---C-TATGC--C--C-T
Score: 54
```

```
beta=0.3
-T-C-----CA-G-C-CCCA-GG-
T--C----T-A--G--T-C-C-T-
A--C---T---TG----TA-TCT-
T--C-ATC--G-C-A-C------G
--T-C--GT---CT-C--C-T-A-
-C-T-GCC-C--TG--AG---C-C
-C----T-T-----A--T-A--GG
G--C-A-CA--A---C-AA--C--
G-T-----G--CGT--T-CG--GG
-A-C-GC---TA-TG---C-CCT-
Score: 74
```

```
beta=0.5
T-CCA-GC-C---C---C-AGG-
-T-CTAG---T----C---C--T
--AC-TT---G-T--A-T--C-T
T--CA-T--C--GC-A----CG-
T--CGT-C---T--C--CT-A--
---CTG-CCCTG--A---G-C-C
---CT-T-A--T---A--G--G-
G--C-A-C-A---A-C--A-A-C
G---TG-C--GT-T-C--G-GG-
A--CGC-T--A--TGCC--C--T
Score: 104
```

```
beta=0.7
-T-C---C-A-G--CCCC-AG--G
-T-C----TA-GT--C-C--T---
--A---CTT-G-T-A-TC----T-
-T-C-A--T-C--G-CA--C---G
-T-C--G-T-C-T--C-C--T--A
CTGC---C--C-TGA-GC----C-
---C-T--T-A-T-A-----G--G
G--C-A--C-A---A--CAA--C-
GTGCG---T---T--C-G--G-G-
-A-CG--CT-A-TG-C-C-C-T--
Score: 163
```

```
beta=0.9
-T-C--C-A-GC-C-C-CAG-G
-T-C--T-AGT--C---C--T-
A--C--T---T-GTA-TC-T--
-T-C----A-TCGCA--C---G
-T-CG-TC--T--C---C--TA
CTGC--CC--T-G-A-GC---C
---CT-T-A-T---A----G-G
G--C-A-CA--A-CAA-C----
GTGC-GT---T--CG--G-G--
A-CGCT-A-T-GC-C-C--T-
Score: 201
```

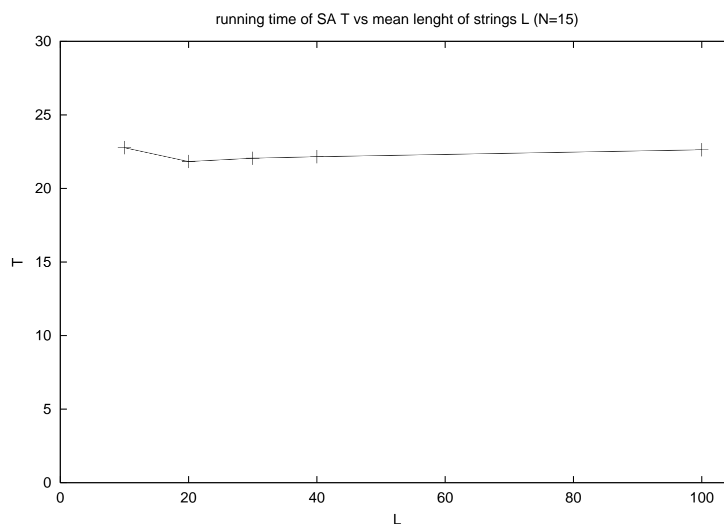
```
beta=1.0
-T-C--C--A--G-CC-C-CAGG-
-T-C-TAG--T---C----C--T-
A--C---T--T-GT-A-T-C--T-
-T-C---A--TCG-CA---C--G-
-T-C-G-TC-T---C----C--TA
CTGC--C-C-T-G--AG--C---C
---C-T-T-AT----A--G---G-
G--CA-C--AA---CAA--C----
GTGC-G-T--T---C-G-G---G-
-A-C-GCT-AT-G-CC---C--T-
Score: 201
```

The last result can be compressed leading to the final MSA:

```
-T-C-C-A--GCCCCAGG-
-T-CTAG-T--C--C--T-
A--C--T-T-GTATC--T-
-T-C--A-TCGCA-C--G-
-T-CG-TCT--C--C--TA
CTGC-C-CT-G-AGC---C
---CT-TAT---A-G--G-
G--CAC-AA--CAAC----
GTGCG-T-T--C-GG--G-
-A-CGCTAT-GCC-C--T-
Score: 201
```

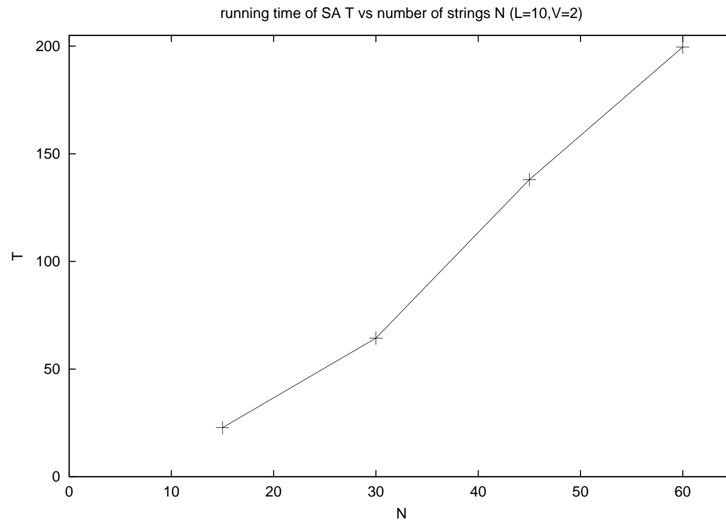
In this example the compression doesn't affect the total score but in general it can slightly improve it. ♢

We saw how Needleman-Wunsch algorithm is linear with the length of the strings. Simulated Annealing applied experimentally to 15 random strings of different length (10, 20, 30, 40 and also 100) proved itself to be almost linear in time. The parameters used (different β's and number of trials for each of them) seemed to be equally good in these four cases. Surely increasing moreover L (the length of the strings) those parameters would be too low, so that the running time will grow proportionally.



A very different behavior comes out maintaining the length (which is actually randomly between 8 and 12) and increasing the number of the string. We tried experimentally with 15, 30, 45 and 60 strings. The running time

grew almost exactly linearly with the number of the strings. In this cases however this is due to the fact that the parameters used had to be changed linearly in order to obtain reliable results.

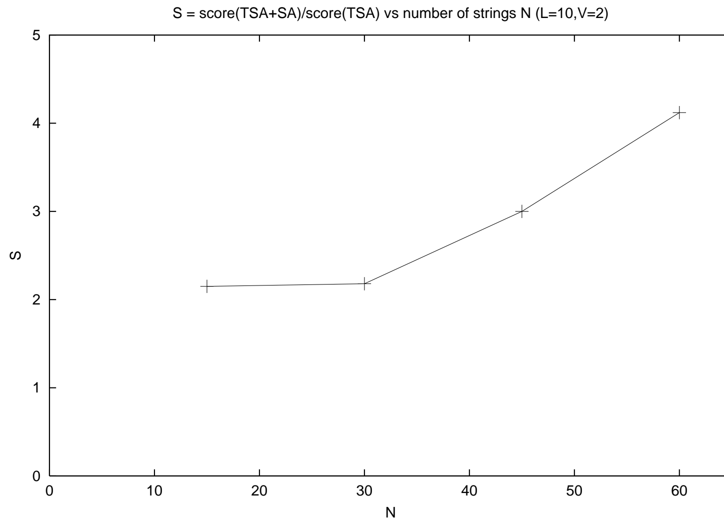


4 Experiments and conclusions

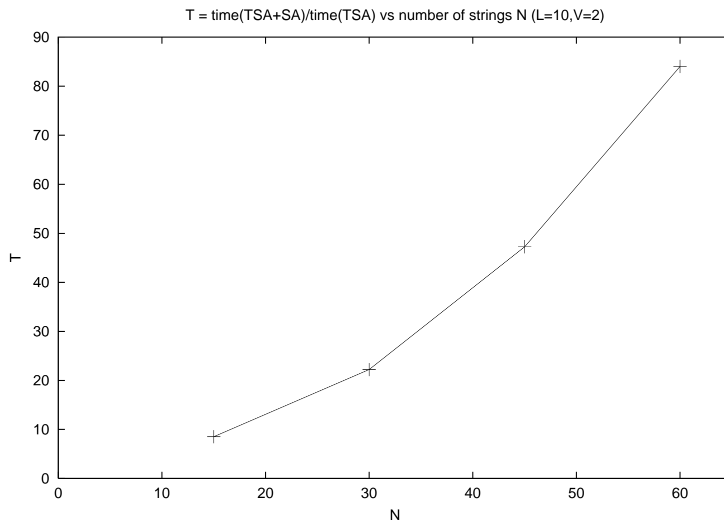
The last step of our research was based on experimental trials with three different algorithms: the already known Traveling Salesman Approach (TSA) and Simulated Annealing (SA), and a mixture of both of them (TSA+SA). To be more accurate we first ran TSA and then refined it with SA, but starting with a higher β than the one used in simple SA.
As all the parameters in single SA, also this initial β has been chosen, case by case, in order to obtain the best results in any of them (e.g. for 100 strings we started with $\beta = .3$ and ended with $\beta = 1.2$, while in the original SA starting β was 0.1).

We have then tested these three algorithms on four different numbers of strings (15, 30, 45 and 60), maintaining fixed their length (between 8 and 12 as in previous section).
In choosing properly the initial β for the combined TSA+SA, our main goal was to obtain an equal score for simple SA and TSA+SA (with an error that has always been less than $\pm 1\%$).
In such conditions we've been able to compare the score and the running time of TSA+SA with those of TSA, and also its execution time with the one of SA.

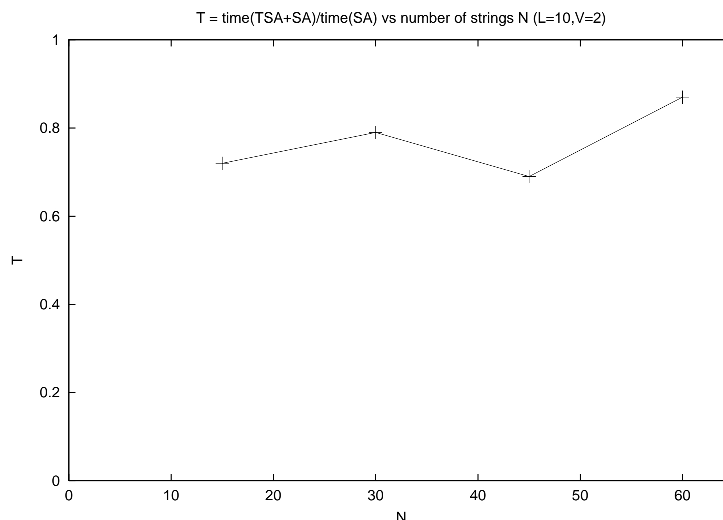
We have seen experimentally that the proportion of the results (i.e. the scoring function of the best MSA) of TSA+SA and TSA, increases almost linearly with the number of strings.



In despite of this improvement in score, the proportion of the running times increases almost exponentially.



Finally the comparison of execution times for TSA+SA and SA (maintaining fixed and equals their final score) remains more or less around 0.8, but it seems to approach asymptotically 1 increasing the number of strings.



This final experimental analysis has brought us to three mainly considerations:

1. The length of the strings influences linearly both TSA and SA.

2. What happens increasing their number?
   TSA is a very quick algorithm, but its results are very poor. Even for as few as 15 strings SA (whose running time is in this case 11 times bigger) gives a result that is twice as good.
   This disproportion gets much worse with more strings: with 60 the result is $\frac{1}{4}$ and the running time is $\frac{1}{100}$.
   So, increasing number of strings, TSA, compared with SA, becomes exponentially faster but linearly less accurate.

3. On the other hand this two approaches can be used together and TSA can be a good tool in improving SA. Nevertheless its utility slowly decreases (linearly?) with growth of number of strings.

Concerning the actual dimensions of real genetic problems (hundreds of DNA sequences with millions of basis each). TSA can be used only to slightly improve a pure heuristic SA approach. Moreover it may be dangerous to trust too much on the TSA result: it could be near to a local maximum which is very far from the global one.

TSA is however and so far the best polynomial algorithm for the MSA problem (due to very good polynomial algorithms on the classical Traveling Salesman), and it is very powerful when facing phylogenetic problems (i.e. the research of common ancestors).

Appendix

We present here briefly the functions that we wrote to implement the algorithm to solve MSA.

- *main*

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "nrutil.h"

#define C 4                             // number of characters in the alphabet
#define N 15                            // number of strings
#define L 10                            // average number of characters in a string
#define V 2                             // variance on this number

// random processes
#define LOOP  0                         // random loops before spreading the strings
#define LOOPG 120                       // random loops before generating the strings
#define LOOPS 10                        // random loops before Simulated Annealing

// Simulated Annealing
#define BETA .1                         // gap between betas in Simulated Annealing
#define INIZ 5                          // first beta is INIZ*BETA
#define FINE 11                         // last  beta is FINE*BETA
#define GIRI 1e5                        // number of trials for every beta

// Simulated Annealing on Traveling Salesman
#define BETATS .1                       // gap between betas in Simulated Annealin
#define INIZTS 10                       // first beta is INIZ*BETA
#define FINETS 25                       // last  beta is FINE*BETA
#define GIRITS 1e5                      // number of trials for every beta

int M[L+V+1][N];                        // matrices to use    [trasposed!]
int G[L*N/2][N];

int w[N];                               // array for original string order


void genstr (int c, int n, int l, int v, int m[][N], int loopg);

void nw     (int c, int l, int v, int str1[], int str2[], int gnw[][2]);

void order  (int c, int n, int l, int v, int m[][N], int w[], float beta, int iniz, int fine, int giri);

int  nwdist (int c, int str1[], int str2[]);

void insgap (int c, int n, int l, int v, int m[][N], int g[][N]);

void simann (int c, int n, int l, int g[][N], float beta, int iniz, int fine, int giri, int loops);

void stampa (int c, int n, int l, int m[][N]);

// MAIN PROGRAM

main () {

  float ran2(long *idum);

  long h = -1;                          // parameter for random number generator

  int i,j,k,g;                          // cycle variables
  int str1[L+V+1];                      // strings on which run nw
  int str2[L+V+1];
  int gnw[2*L+2*V+1][2];
  int lun;
  int ch;
  int col = L*N/4;


  genstr (C, N, L, V, M, LOOPG);
  stampa (C, N, L+V, M);                // print the matrix

  printf ("\nTraveling Salesman Approach (1), Simulated Annealing (2) or both (3).");
  scanf ("%d", &ch);


  if (ch == 2) {
```

```
    for (j=0; j<LOOP; j++) ran2(&h);     // random loops to change results

    for (i=0; i<N; i++) {                // generates random spread matrix
      lun = 0;
      while (M[lun][i] != C ) lun++;
      g = 0;
      for (j=0; j < col; j++) {
if  ( ((lun-g)!=(col-j)) && ( (M[g][i]==C) || ((ran2(&h)) > (4.8/N)) ) )  G[j][i] = C;
else { G[j][i] = M[g][i]; g++; }
      }
    }

    stampa (C, N, col, G);             // print the matrix

    simann (C, N, col, G, BETA, 1, FINE, GIRI, LOOPS);
  }
  if ((ch == 1)||(ch == 3)) {

    order  (C, N, L, V, M, w, BETATS, INIZTS, FINETS, GIRITS);

    for (i=0; i<N; i++) {                // print the matrix (with previous positions)
      printf ("%d-%d)\t", i, w[i]);
      for (j=0; j<L+V+1; j++){
if (M[j][i] == C) printf ("-");
else printf ("%d", M[j][i]);
      }
      printf ("\n");
    }
    printf ("\n");

    insgap (C, N, L, V, M, G);

    stampa (C, N, L*N/2, G);              // print the matrix
  }
  if (ch == 3) {

    g=0; i=0;                             // count g(i)'s length
    while (g==0) {
      g=1;
      for (j=0; j<N; j++) { if ( G[i][j] != C ) g=0; }
      i++;
    }
    simann (C, N, i, G, BETA, INIZ, FINE, GIRI, LOOPS);
  }
}
```

- *genstr*

  It generates the strings to be aligned. It can be done totally random or, given one string, all the others are created with just a finite number of random changes such that the output simulates strings related by some evolutionary change. It is also possible to vary the mean length of the strings, the variance of such length, the number of characters used (i.e. the alphabet) and of course the number of strings.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

// Subroutine to generate N strings of L characters

void genstr (int C, int N, int L, int V, int M[][N], int LOOP) {


        float ran2(long *idum);

long h = -1;                          // parameter for random number generator
int i,j,m;                            // variables for loops
int l;                                // variable for the length of the strings
int r;                                // random choice
int p;
int ch;
int or[L];                            // original string
int k[L+V+1];                         // string with the modifications
```

```
printf ("How to generate strings?\n Similar (1), Random (2).");
scanf ("%d", &ch);

if (ch == 1){

  for (i=0; i<LOOP; i++) ran2(&h);

  for (i=0; i<L; i++) or[i] = floorl ((ran2(&h)) * C);

  printf ("\nOriginal string:\n\t");
  for (i=0; i<L; i++) printf("%d", or[i]);
  printf ("\n");

  for (i=0; i<N; i++) {

    for (j=0; j<L; j++) k[j] = or[j];
    while ( j <= L+V ) { k[j] = C; j++; }
    l = L;
    p = floorl (L/2);

    for (j=0; j<p; j++) {                  // make L/2 changes

      r = floorl ((ran2(&h)) * 3) ;         // 0 insert, 1 cancel, 2 change

      if ((r==0) && (l<L+V)) {
p = floorl ((ran2(&h)) * l) ;

for (m=0; m<=L+V-p-1; m++) k[L+V-m] = k[L+V-m-1];   // insert one element in position p of k
k[p] =  floorl ((ran2(&h)) * C);
l++;
      }

      if ((r==1) && (l>L-V)) {
p = floorl ((ran2(&h)) * l) ;

for (m=0; m<=L+V-p-1; m++) k[p+m] = k[p+m+1];   // cancel one element in position p of k
k[L+V] = C;
l--;
      }

      if (r==2) {
p = floorl ((ran2(&h)) * l) ;
        k[p] =  floorl ((ran2(&h)) * C);               // change one element in position p of k
      }
    }

    for (j=0; j<l; j++) M[j][i] = k[j];               // insert the characters
    while ( l <= L+V ) { M[l][i] = C; l++; }          // put all gaps in the end
  }
}

else{
  for (i=0; i<LOOP; i++) ran2(&h);

  for (i=0; i<N; i++) {

    l = L - V + floorl ((ran2(&h)) * ( 2*V +1 )) ;

    for (j=0; j<l; j++) M[j][i] = floorl ((ran2(&h)) * C);  // insert the characters

    while ( l <= L+V ) { M[l][i] = C; l++; }               // put all gaps in the end
  }
}

return;
}
```

- *function*

  Calculates the scoring between two characters.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "nrutil.h"

//Subroutine to calculate the Needleman-Wunsch function to build the matrix
```

```
int function (int C, int n1, int n2) {

  if ( (n1 == n2) && (n1 != C) ) return 1;
  //if ( (n1 == n2) && (n1 == c) ) return 0;

  else return 0;
}
```

- nw

  Align two strings using NW algorithm.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "nrutil.h"

int function (int C, int n1, int n2);

// Subroutine of Needleman-Wunsch to find the optimal pairwise alignment

void  nw  (int C, int L, int V, int str1[], int str2[], int gnw[][2]) {

int i,j;                                // variables for loops
int mem[2];                             // memory of the path
int k;                                  // final length of the two strings
int g1;                                 // gap in the first row
int l1, l2;                             // length of the two strings
int *v1;                                // vectors to represent the two strings
int *v2;
int **mnw;                              // NW matrix (dimensions unknown yet)

l1 = 0; l2 = 0;                         // compute the length of the two strings
while (str1[l1] != C) l1++;
while (str2[l2] != C) l2++;

        i  = 0;
v1 = ivector(1, l1+l2);                 // define vectors (both of length l1+l2) and write them
while (str1[i] != C) { v1[i+1] = str1[i]; i++; }
i++;
while ( i <= l1+l2 ) { v1[i]   = C; i++; }  // put all gaps in the end
i  = 0;
v2 = ivector(1, l1+l2);
while (str2[i] != C) { v2[i+1] = str2[i]; i++; }
i++;
while ( i <= l1+l2 ) { v2[i]   = C; i++; }  // put all gaps in the end

mnw = imatrix(0,(l1+1),0,(l2+1));        // define dimensions of the NW matrix

// Define the matrix

for (i=0; i<=l1; i++) mnw[i][0]=0;       // first column equal 0

for (i=1; i<=l2; i++) mnw[0][i]=0;       // first row equal 0

for (i=1; i<=l1; i++) {                 // loop on the rows
    for (j=1; j<=l2; j++) {             // loop on the columns

      mnw[i][j] = mnw [i-1][j-1] + function (C, v1[i], v2[j]);   // give the value of the cell in direction\
                                                                 // to which add the value of f in that position

      if (mnw[i-1][j] > mnw [i][j]) mnw[i][j] = mnw [i-1][j];   // check the neighbors | and -
      if (mnw[i][j-1] > mnw [i][j]) mnw[i][j] = mnw [i][j-1];
    }
}

// Look for the path

i=l1; j=l2; k=1; g1=0;                          // start from the lower right hand side corner k at first step

while ((i != 1) || (j != 1)) {                  // stop in upper left hand side corner

     if ((mnw[i-1][j-1] >= mnw [i-1][j])&&(mnw[i-1][j-1] >= mnw [i][j-1])&&(i!=1)&&(j!=1)) { // try to go \

               v1[l1+l2+1-k] = v1[i];             // write both the values in the string with gaps
       v2[l1+l2+1-k] = v2[j];

                mem[0]=i; mem[1]=j; i=i-1; j=j-1; k++; }
```

```
     else {
       if (((mnw[i][j-1] > mnw[i-1][j]) || ((mnw[i][j-1] == mnw [i-1][j])&&(j>i)) || ((i==1))&&(j!=1))) { // try to go -

 if (mnw[i][j-1] < mnw[i][j]) { v1[l1+l2+1-k] = v1[i]; mem[0]=1; } // choose if to put a gap in first row
 else  v1[l1+l2+1-k] = C;

 v2[l1+l2+1-k] = v2[j]; mem[1]=j;        // write second row

 j=j-1; k++; g1++; }

       else {                          // in this case go |

                 v1[l1+l2+1-k] = v1[i];         // write the first row

                 if (mnw[i-1][j] < mnw[i][j]) { v2[l1+l2+1-k] = v2[j]; mem[1]=j; } // choose if to put a gap in second row
 else  v2[l1+l2+1-k] = C;

 i=i-1; k++; }
     }
}

if (mem[0] > 1) v1[l1+l2+1-k] = v1[1];     // write the last characters
else  v1[l1+l2+1-k] = C;
if (mem[1] > 1) v2[l1+l2+1-k] = v2[1];
else  v2[l1+l2+1-k] = C;

for (i=0; i<l1+g1; i++) gnw[i][0] = v1[i+l2-g1+1];
while ( i <= L*2+V*2 ) { gnw[i][0] = C; i++; }  // end with all gaps

for (i=0; i<l1+g1; i++) gnw[i][1] = v2[i+l2-g1+1];
while ( i <= L*2+V*2 ) { gnw[i][1] = C; i++; }  // end with all gaps
}
```

- *nwdist*

  Completely analogous to *nw*, but returns only the distance and doesn't insert gaps.

- *order*

  Orders the input strings using the circular order method, then it improves it with Simulated Annealing.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "nrutil.h"

int  nwdist (int C, int str1[], int str2[]);

// Subroutine to order the strings using the Traveling Salesman Approach

void order  (int C, int N, int L, int V, int M[][N], int w[], float BETA, int INIZ, int FINE, int GIRI) {

  float ran2(long *idum);

  long h = -1;                     // parameter for random number generator

  int app [L+V+1][N];              // vector for the strings to be ordered
  int str1[L+V+1];                 // strings on which calculate the distance
  int str2[L+V+1];
  int md[N][N];                    // matrix of distances
  int md2[N][N];                   // same matrix on which changes are made
  int i,j,m,n,p;                   // variables for loops
  int min;                         // variable to find minima
  int col;                         // variable for the searched column
  int row;                         // variable for the searched row
  int q[N];                        // vector to sign the position of strings
  int score;                       // score of the alignment
```

```
  float beta[50];                   // vector for simulated annealing data
  int delta;                        // variation of the score


  for (i=0; i<N; i++) {             // calculate the matrix
    md[i][i] = 0;
    for (j=0; j<i; j++) {
      for (h=0; h <= L+V; h++) { str1[h] = M[h][i]; str2[h] = M[h][j]; }
      md[i][j] = nwdist (C, str1, str2) ;
      md[j][i] = md[i][j];
    }
  }

  for (i=0; i<N; i++) {             // md2 = md
    for (j=0; j<N; j++) md2[i][j] = md[i][j];
  }

  min = -1;                                   // looks for the maximum in md to choose the extremals
   for (i=0; i<N; i++) {
     for (j=0; j<i; j++) if (min<md[i][j]) {
       min=md[i][j]; row=i; col=j;
     }
  }

  for (i=0; i<N; i++) md[i][row] = -1;        // exclude the corresponding columns
  for (i=0; i<N; i++) md[i][col] = -1;

  for (i=0; i<N; i++) q[i] = -1;              // at the beginning there are only row at first place and col at the last
  q[row] = 0; q[col] = N-1;
  w[0] = row; w[N-1] = col;

  for (h=1; h<N-1; h++) {                     // align the strings

    min = 1e5;

    for (i=0; i<N; i++) {                     // find the minimum among the remaining strings

      if ( (q[i]>-1) && (q[i]!=N-1) ) {

for (j=0; j<N; j++) {

  if ((md[i][j] < min) && (md[i][j]>0)) { min = md[i][j]; col=j; row=i;}
}
      }
    }
    for (i=0; i<N; i++) { if ((q[i]>row)&&(q[i]<N-2)) q[i]++; }       // put in w the position of the found string
    q[col] = q[row] + 1;
    for (i=0; i<h-q[row]-1; i++) w[h-i] = w[h-i-1];
    w[ q[row] + 1 ] = col;

    for (i=0; i<N; i++) md[i][col] = -1;   // exclude the corresponding column

  }

  score = 0;
  for (i=0; i<N-1; i++) score = score + md2[ w[i] ][ w[i+1] ];

  for (m=INIZ; m<=FINE; m++) {            // --->  start of Simulated Annealing <---

    for (n=0; n<50; n++) { beta[n] = exp ( BETA*m*n*(-1) ) ; }  // Calculate beta[], with the values of e^(-bD)

    for (n=0; n<=GIRI; n++) {

      i = 1 + ( floorl ((ran2(&h)) * (N-2)) );            // looks randomly for two different strings (not the first neither the last)
      j = 1 + ( floorl ((ran2(&h)) * (N-2)) );
      while (j==i) j = 1 + ( floorl ((ran2(&h)) * (N-2)) );
      if (j<i) {p=i; i=j; j=p;}                           // always  j>i

      delta = md2[ w[j] ][ w[i-1] ] -  md2[ w[i] ][ w[i-1] ] ;                    // evaluate the move
      delta = delta + md2[ w[i] ][ w[j+1] ] -  md2[ w[j] ][ w[j+1] ] ;
      if (j != i+1) delta = delta + md2[ w[j] ][ w[i+1] ] + md2[ w[i] ][ w[j-1] ] - md2[ w[i] ][ w[i+1] ] - md2[ w[j] ][ w[j-1] ] ;

      if ( delta<0 ) { p=w[i]; w[i]=w[j]; w[j]=p; score = score+delta; }    // do the move if improving

      else {                                  // Monte Carlo step
if ( delta > 49 ) delta = 49;         // calculate delta (49 as maximum value)
if (( ran2(&h)) < beta[delta] )  { p=w[i]; w[i]=w[j]; w[j]=p; score = score+delta; } // do it
      }
    }
  }


  for  (i=0; i<N; i++) {      // returns the strings aligned and the vector w of the changes done
```

```
    for (j=0; j<= L+V; j++ ) app[j][i] = M[j][ w[i] ];
  }

  for  (i=0; i<N; i++) {
    for (j=0; j<= L+V; j++ ) M[j][i] = app[j][i];
  }
}
```

- *insgap*

  Inserts the gaps as explained in section (2.2).

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "nrutil.h"

// Subroutine to insert the gaps in the N strings once ordered

void insgap (int C, int N, int L, int V, int M[][N], int G[][N]) {

  int gnw[2*L+2*V+1][2];         // 2 vectors with strings with gaps
  int str1[L+V+1];               // 2 strings on which execute NW
  int str2[L+V+1];
  int gap[N*L/2][2];             // 2 vectors with strings with gaps to make comparisons and insert in G

  int i,j,h;                       // variables for loops
  int p;                         // counters on the strings
  int col = (N*L)/2;               // columns of G
  int l;                         // length of the considered string
  int c, c1, c2;                 // counters of the element of the strings

  for (i=0; i<=L+V; i++) G[i][0] = M[i][0];          // move first string from M to G
  while ( i<=col ) { G[i][0] = C; i++; }

  for (i=1; i<N; i++) { // loop to insert strings in gap[] using NW and extending gaps

    for (j=0; j <= L+V; j++) { str1[j] = M[j][i-1]; str2[j] = M[j][i]; }  // NW between the last inserted and the next
    nw ( C, L, V, str1, str2, gnw);

    for (j=0; j<2; j++) {
      for (h=0; h<=2*L+2*V; h++)  gap[h][j] = gnw[h][j];          // move gnw[2] into gap[2]
      while ( h<=col ) { gap[h][j] = C; h++; }
    }

    l = 0;                          // count number of characters of line M[i-1]
    while (M[l][i-1] != C) l++;

    c=0; c1=0; c2=0;                   // initialize counters (c counts non gap elements)

    while (( G[c1][i-1] != gap[c2][0] ) || (c != l)) {           // check the 2 vectors until the end of both

      if ( G[c1][i-1] == gap[c2][0] ) {                       // case 1: they are equal -> proceed

if ( G[c1][i-1] != C ) c++;                         // if they are not gap increase c
c1++; c2++;
      }
      else {
if ( G[c1][i-1] == C ) {   // case 2: gap above

  for (j=0; j<2; j++) {    // insert one gap into position c2 of gap[0] and gap[1]
    for (h=0; h<col-c2-1; h++) gap[col-h-1][j] = gap[col-h-2][j];
    gap[c2][j] = C;
  }
  c1++; c2++;
}

if ( gap[c2][0] == C ) {   // case 3: gap below

  for (j=0; j<i; j++) {     // insert one gap into position c1 of G[0]...G[i-1]

    for (h=0; h<col-c1-1; h++) G[col-h-1][j] = G[col-h-2][j];
    G[c1][j] = C;
  }

  c1++; c2++;
}
      }
```

```
    }

    for (j=0; j<col; j++) G[j][i] = gap[j][1];          // move gap[1] into G[i]
  }
}
```

- *simann*

Finds the MSA using Simulated Annealing.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "nrutil.h"

int function (int C, int n1, int n2);

void stampa (int C, int N, int L, int M[][N]);

// Subroutine which executes Simulated Annealing on the strings

void simann (int C, int N, int col, int G[][N], float BETA, int INIZ, int FINE, int GIRI, int LOOP) {

  float ran2(long *idum);

  long h = -1;                  // parameter for random number generator

  int p [col];                  // vector for the scores
  int gg[col];                  // vector for columns with all gaps
  int i,j,k,g,d,y,u,s;          // variables for loops
  int lun;                      // counter for the length of the single string
  int t[2];                     // variable to evaluate the single move
  float beta[40];               // vector for the data of Simulated Annealing
  int delta;                    // variation of score


  for (j=0; j<LOOP; j++) ran2(&h);

  for (i=0; i<col; i++) {       // calculate the score vector
    p[i] = 0;
    for (j=0; j<N-1; j++) {
      for (k=j+1; k<N; k++) p[i] = p[i] + function ( C , G[i][j], G[i][k] );
    }
  }

  for (u=INIZ; u<=FINE; u++) {           // ---> Start Simulated Annealing <---

    for (d=0; d<40; d++) { beta[d] = exp ( BETA*u*d*(-1) ) ; }  // calculate vector beta[], with the values of e^(-bD)

    for (d=0; d<=GIRI; d++) {

      i = floorl ((ran2(&h)) * N);           // look randomly for a gap
      j = floorl ((ran2(&h)) * col);
      while (G[j][i] != C) {
i = floorl ((ran2(&h)) * N);
j = floorl ((ran2(&h)) * col);
      }

      k = j; g = 1;                          // look for the nearest non gap element
      y = ( floorl ((ran2(&h)) * 2)) *2 - 1;  // y is 1 or -1
      while ( G[k][i] == C) {
k = k + ( y * g  *  (((g%2)*2)-1) ); g++;
if ( (k<0) || (k>=col) ) { k = k + ( y * g * (((g%2)*2)-1) ); g++; }
      }

      g = G[k][i]; G[k][i] =  G[j][i]; G[j][i] = g;     // do the move

      t[0] = 0;                              // evaluate the move
      for (y=0; y<N-1; y++) {
for (g=y+1; g<N; g++) t[0] = t[0] + function ( C , G[j][y], G[j][g] );
      }

      t[1] = 0;
      for (y=0; y<N-1; y++) {
for (g=y+1; g<N; g++) t[1] = t[1] + function ( C , G[k][y], G[k][g] );
      }

      if ( t[0]+t[1] >= p[j] + p[k] ) { p[j]=t[0]; p[k]=t[1]; }    //  do the move because it's better
```

```
      else {                                  // Monte Carlo step
delta = - (t[0]+t[1]-p[j]-p[k]);      // computes delta (19 is the maximum)
if ( delta > 19 ) delta = 19;
if (( ran2(&h)) < beta[delta] )     { p[j]=t[0]; p[k]=t[1]; } // do the move

else { g  = G[k][i]; G[k][i] =  G[j][i]; G[j][i] = g; }  // reject the move
      }
    }

    printf ("\nbeta = %.3f", BETA*u );

    stampa (C, N, col, G);                 // prints the matrix after each SA step
  }

  // Compression of the result

  for (i=0; i<col; i++) {
    y=0; g=1;
    while ( ( y==0 ) && ( i+g < col ) ) {
      y=0;
      for (j=0; j<N; j++) {
if ( ( G[i][j] != C ) && ( G[i+g][j] != C ) ) y = 1;
      }
      if (y == 0) {
for (j=0; j<N; j++) {
  if ( G[i+g][j] < C ) { G[i][j] =  G[i+g][j];  G[i+g][j] = C; }
}
      }
      g++;
    }
  }

  stampa (C, N, col, G);                  // prints the final matrix
}
```

- *stampa*

  Prints the list of strings and the matrix given by MSA.

- *ran2, nrutil* just see [8]

References

[1] H. Carillo and D. Lipman,
"The multiple sequence alignment problem in biology",
SIAM J. Appl. Math., **48**(5):1073-1082 (1988).

[2] T.H. Cormen, C.E. Leiserson and R.L. Rivest,
Introduction to algorithms
(MIT Press, 1990)

[3] G. Gonnet and C. Korostensky,
http://www.wr.inf.ethz.ch/personal/gonnet/papers/spirefinal.ps
SPIRE (String Processing and Information Retrieval) proceedings (1999)

[4] D. Gusfield,
"Efficient methods for multiple sequence alignment with guaranteed error bounds",
Bull. Math. Biol., **55**, 141-154 (1993).

[5] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi,
Science, **220**, 671-680 (1983).

[6] N. Metropolis, A.W. Rosenbluth, M.N.Rosenbluth, A.H. Teller and E. Teller,
"Equation of state calculations by fast computing machines",
J. Chem. Phys., **21**, 1087-1092 (1953).

[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**, 443-453 (1970).

[8] W. Press, S. Teukolsky, B. Flannery and W. Vetterling,
Numerical recipes in C
(Cambridge University Press, 1992)

[9] L. Wang and T. Jiang,
"On the complexity of multiple sequence alignment",
Journal of Computational Biology, **1**, 337-348 (1994).