BIOINFORMATICS

Giacomo Castagnetti - Genomics 2023

INDEX

        - Gap affine penalty
- Database search
    - Fasta
        - Hash tables
            - Build a hash table
            - Chaining
                - Implementation
                - What can we use as a container?
            - Open addressing
        - Filtering
        - Alignment
        - Conclusions
            - Official Fasta site
            - Results
    - Blast
        - Neighbourhood
        - Alignment
            - Parameters
        - Conclusions
            - Output
    - Statistical evaluation

INTRODUCTION

Course organisation

Theory and lab (which will start on March 15). This course will talk about bioinformatics applied to biological sequences.

Book as reference: Genome-scale algorithm design Veli Makinen

The grade will be assigned with a theoretical and a practical exam, the final grade is weighted by the number of credits (8 for theory and 4 for lab).

Exam for theory module:

- Homeworks: 4 assignments, up to 6 points in total)
- Written exam (up to 24 points)
- An optional oral exam

Date (oral 5 days later):

- June 15
- June 30

Syllabus

- Biological sequence comparison
    - Sequence pairwise alignment
    - Sequence multiple alignment
    - Database sequence search
- Genome assembly
- Genome mapping
- Gene expression estimate
- Sequence statistical modelling
    - Genome modelling
    - Gene finders
- Phylogenetic inference
- Pattern matching

What is bioinformatics?

Bioinformatics consists in applying computational sources to biology. With this bachelor course we will become bioinformaticians specialised in genomics data.

Bioinformatics is an interdisciplinary field that develops methods and software tools for understanding biological data, in particular when the data sets are large and complex. With bioinformatics we want to find the solution to a biological question by working with data. Bioinformaticians come from many different fields.

Applications

- **Biological sequence analysis**: billions of biological sequences are nowadays available, and obviously computational methods for analysis and organising them are essential. Computational analysis is the bottleneck.
- **Functional annotation**: gene finding, database search, functional patterns and protein domains identification.
- **Phylogeny**: reconstructing the evolutionary history and relationships of groups of different species from molecular data.
- **Structural bioinformatics**: tridimensional structures of proteins and molecules.
- **System biology**: study of biological systems as a whole (borrows concepts from engineering). Qualitative or quantitative models can be created, with quantitative models, simulations can be run.
- **Synthetic biology**: engineering of biosystems in order to make them do something that they couldn't do before.
- **Text mining**: extract information from biological papers.
- **Ontologies**: orientation and classification of concepts, biological concepts.
- **Databases**: since biology creates data, they have to be collected and stored in databases, with some interface that allows retrieval of information in a possibly easy way.
- **Omic sciences**: with the advent of omic data, the requirements of computational methods have become essential. Omic sciences require computational analysis, they would not exist without it.

History on bioinformatics: on virtuale.

(It started in 1070, when the first sequence of proteins became available. Margaret Dayhoff was the first to understand that parallel to the advancement of experimental techniques a parallel advancement of computational methods was needed; she is considered the mother of bioinformatics.)

To perform bioinformatics multiple skills are needed:

- Biology background.
- Mathematics, statistics and physics background.
- Programming skills in python and higher performance languages (C++).
- Knowledge of unix-based OS (linux) and shell scripting.

SEQUENCE COMPARISON

Most modern bioinformatics is based on sequence comparison.

It is useful for:

- Finding functional relationships.
- Identifying gene and protein families.
- Building phylogenetic trees.
- Identifying protein characteristics, such as their organisation in domains functional sites, regulatory sites.
- Building a protein three dimensional model: if the structure of a protein is still not experimentally known, it can be predicted, the mechanism at the base is sequence comparison.

Comparing is important to extract information from the huge amount of sequencing data.

The advent of massive parallel nucleic acids sequencing provided us with an unprecedented amount of data. In turn, this required the development of specific and efficient resources algorithms , software, statistics, databases) for analysis, organisation and storage.

Nowadays, bioinformatics is fundamental in any project involving genome analysis, and the computational pipelines are often the bottleneck of these projects, i.e. the part of the project requiring more time and from which the project outcomes more strictly depend.

(note that also the companies that provide sequencing have huge computers, illumina consists in a constant scanning of an image, this procedure requires a huge computational power and the data is usually pre processed before leaving the facility).

Genomics: branch of molecular biology concerned with the structure, function, evolution, and mapping of genomes. We cannot do this without bioinformatics.
The branch of molecular genetics concerned with the study of the genomes, specifically the identification and sequencing of their constituent genes and the application of this knowledge in medicine, pharmacy and agriculture.

Sequence comparison (alignment): To compare two sequences we have to align them (even though they are not the same exact thing).

Comparing sequences is useful for:

- Reconstructing the sequence of the genome of a species or an individual
- Identifying genome features, such as genes, regulatory regions etc.
- Analysing high-throughput sequencing data for measuring gene expression, protein-DNA or protein-RNA interactions, the 3D shape of chromosomes, identifying genome variants.

What is an alignment?

Align two sequences, A and B, means to highlight their similarity, measuring how much they have in common. It basically consists in finding which residues of A and B match, which means which residues are equivalent (but not necessarily identical) in the two sequences.

Why should two sequences be similar, what's the reason for their similarity? Two sequences are similar if they evolved from the same sequence. Compare sequences means comparing two strings, which are the result of evolution.

We need to compare sequences in different organisms taking into account how they evolved. In the laboratory often we do experiments on model organisms. if we study disease in mice, we usually don't want to cure a mouse but we want to transfer what we understand to humans.

Without a way to compare sequences, we wouldn't have model organisms. Transferring information between sequences would be really difficult.

Why is the mouse a model of human processes? Humans and mice evolved from the same ancestor, this means that both have some common (conserved) characteristics. Also yeast have some common processes with humans.

A common ancient gene (X) could give two similar genes in two different species (X' and X''). Each gene in the two species is subjected to different mutations, we have to find a way to realise that X' and X'' are related.

Evolution is included in the way in which we evaluate sequence similarity. To do this we have to understand how sequences evolve.

When we compare sequences we have to take into account their common origin.

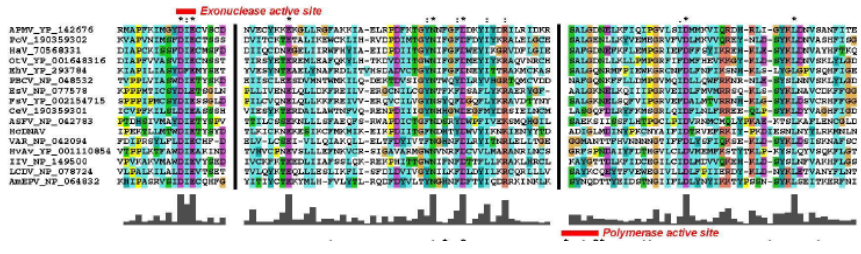
Each pair of modern species originated from an extinct ancestor species. From the divergence from this common ancestor, the genomes of the species that originated from it started accumulating mutations. Mutations that hit genes or other important genomic positions and that might affect their functions are counter selected by natural selection, and are not passed to the progeny. As such, it is possible, and also likely, that genes of modern species are still similar to those of their ancestor, and therefore also similar to each other.

Modifications in the ancestor sequence are usually spread in a population only if they give an advantage to the organism. When we compare sequences we have to take into account their common origin.

Comparing sequences is also important to identify residues with important functions, since they are less tolerant to mutation and under stronger evolutionary constraints. While the mutations that are inherited usually affect non-important genes.

We know that modifications in the sequences are caused by mutations, if a residue is conserved there must be a reason. The residue is important for function or structure. We don't know the function but still this is a starting point for understanding the function.

Conservation may not be perfect. Similar amino acids can be interchanged, two polar amino acids may be exchanged without having a big impact on the function of the protein.



Most conserved positions correspond to higher columns

Parsimony

Imagine these two sequences:

Mouse: ACGATCGAGCCGACTGATCGA

Human: ACTATCGAACTCACCATCGA

Sequence comparison and alignment is done by comparing the two strings, the goal is to associate common characters of the two sequences. We have to find a way to accommodate the two sequences in a way that reflects their evolution from the latest common ancestor.

```
ACGATCGAGCCGACTGA---TCGA
ACTATCGA----ACTCACCATCGA
```

If we align two bases we are saying that they are the same residues, they come from a common ancestor base. If they have a non correspondence it means that only one of the two is equal to the ancestor, the other one has mutated. If there is a dash it suggests that a deletion or an addition has occurred.

These comparisons are based on a sort of **inference of evolution of the two sequences**, the evolution can be much more complex than what we have inferred.

To know exactly what type of evolution occurred we need the ancestor sequence.

Example:

```
Modern species 1   ACGATCGAGCCGACTGA---TCGA
Ancestor           ACGATCGA----ACTCACCATCGA
Modern species 2   ACTATCGA----ACTCACCATCGA
```

In this case we would be able to assign exactly the modification that happened in each species.

The problem is that we never have the ancestor sequence. We can never be sure about our predictions.

When the two residues are equal, we usually state that the same residue was present also in the ancestor, but we are not completely certain about this. Each species could have mutated independently to the same base, while the ancestor had a different gene.

We need to simplify all of these problems: we apply the principle of **parsimony**. We try to give the simplest explanation of something. We are trying to explain the differences that we observe with the minimum number of mutations that can happen. This is not the absolute truth but it's the most convenient to apply.

Since it is impossible to reconstruct the true evolutionary history of a genome we need to use parsimony. Parsimony is a principle that we need to apply basically to most bioinformatics tools, any time we compare something we have to seek for the simplest explanation and solution.

Aligning two sequences is a way to reconstruct their possible evolutionary history from the latest common ancestor. Nevertheless, we have no way to really know what happened during the evolution that led to the modern species sequences. We seek the most simple explanation for the modern sequences, using a principle of parsimony meaning that we wish to minimise the number of events used to explain the differences between two nucleotide sequences. Parsimony reflects a belief that mutation events have low probability, thus in searching for solutions that minimise the number of events we are implicitly maximising their likelihood.

Homology

Comparing sequences in order to identify relationships, is called homology. Homology is a concept that can be applied to many biological entities, sharing a common evolutionary character. For sequences, we consider them as originated from the same ancestor sequence, if sequences sufficiently similar are homologous.

Homology can be applied also to macroscopic elements, e.g. arms of human, cat, whale and bat have different morphology but they have the same function. They have a common origin.

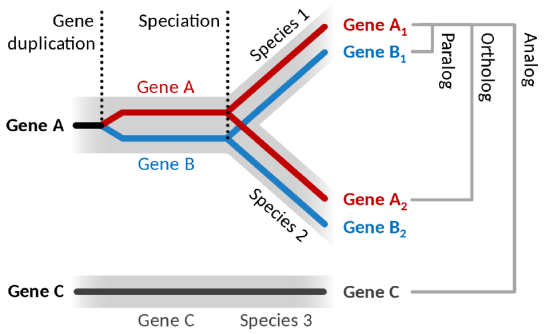
Genes might preserve their original functions and cellular roles. If the two modern species are distant, meaning that their latest common ancestor is very ancient, it is possible that the sequences of their genes did not remain similar, but they might still have the same function. In these cases, it is possible that, while the gene sequences diverged and cannot be recognised as similar anymore, the three dimensional structure of the proteins encoded by these genes are still similar, since the 3D structure tends to be resistant to mutations (i.e. mutations can accumulate without altering the protein folding).

We assume that two entities with common origin have a common function, even though a common function could be misleading.

We assume a relationship between sequence similarity, structure similarity and function similarity.

Homology can be even more complex. Different types of homology:

- Two traits (e.g. genes or proteins) are homologous if they share a common evolutionary origin
- Orthologous traits in two species evolved from the same trait of their most recent common ancestor, and they often (but not always) have the same biological function.
- Homologous traits that originated within a single species are instead called paralogs, and they are the result of duplication or higher order genome amplification events.
- Homologous genes found in the same genome can also be the result of horizontal gene transfer between two different species; in this case the homologous genes are called xenologs.



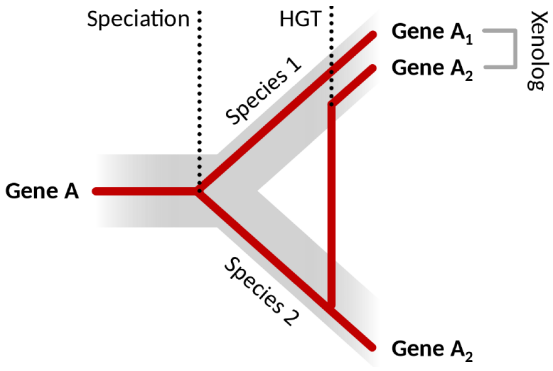
What we described until now is orthology. Paralogs occur very frequently in genomic sequences: a piece of a genome could be duplicated, creating two paralogs. This is a form of homology that happens inside the same organism. Paralogs are genes with common origins, formed by duplication.

When a selective pressure is applied they start to differentiate. If a mutation occurs in an homolog it doesn't matter, the organism still has the other one, for this reason, they tend to diverge faster than other sequences.

If the numerous mutations cause the loss of function of a paralog, a pseudogene is created. If instead of abolishing the function each paralog gets a slightly different but useful function, a gene family is created (e.g. hemoglobins).

There are even more complex cases:

In prokaryotes there can be transfer of genetic material. One bacteria has gene A, from this, two species are generated. Species 2 passes material to species 1, in this way species 1 will have an extra copy of a portion of its gene.



In species 1 these two genes are both orthologs and paralogs, they are called xenologs.

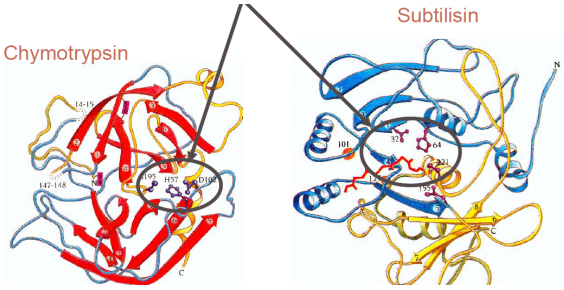
Convergence and analogy

Sometimes all these assumptions may not be correct, when we consider two similar structures as having the same origin we are not considering convergence.

Convergence is the development of structure similarity from different ancestors. Convergence can happen randomly or because of adaptive convergence.

An example are the wings of birds and bats, which have no common evolutionary origin. These structures are not homologs; they are called analogous, since they have the same role while not having a shared origin. In a certain sense analogy is the opposite of homology.

This phenomenon occurs also in proteins.



Proteases, in bacteria and eukaryotic, have very different structures but they have the same active site. In the catalytic site, there are the same residues, with the same relative position.

Nature could find the same solution from different starting points. Maybe the solution is really efficient or maybe it happened by chance.

Mutations and selection

Imagine an ancestor gene X that has two slightly different versions in two species (X' and X''). We would like to compare modern sequences, taking into account that they are the result of complex evolution. We want to reconstruct evolutionary history even though we are sure that we won't know all the true steps. We should apply parsimony, trying to explain differences in the simplest possible way.

An important concept to keep in mind during sequence comparison is that not all mutations are the same. The selective pressure on a mutation depends on where the mutation occurs.

Mutations in **intergenic regions** won't have any function repercussions. While mutations happening in important parts of the genome will be subjected to selective pressure. They won't necessarily have a negative effect but they will have some effect.

Another factor to consider is the repercussions of genomic modifications on the proteome, the nucleic information is translated to proteins through the **genetic code**.
The genetic code is redundant, different codons code for the same amino acid. For example there are 4 codons that code for alanine; they change in the last base. So a mutation on a base that doesn't modify the encoded amino acid will likely be tolerated.

Sometimes even if encoded amino acid is the same, the mutation might have some effect. The abundance of tRNA for a specific codon or other factors may influence and apply a selective pressure.

We can classify **mutations** as:

- Synonym mutations: mutations that do not lead to a change in the amino acid encoded by the codon to which the mutated nucleotide belongs to.
- Non-synonym mutations: cause a change in the protein sequence.

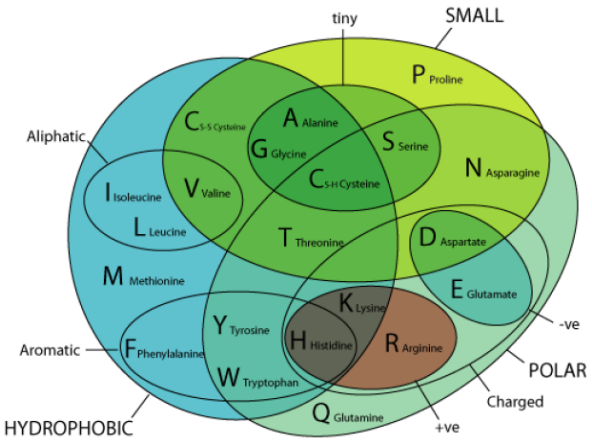
If we compare the sequence of homologous elements in different species, we can count their differences. Different elements will have very different evolution rates: some will be free to change while others change slowly.

To express these rates we can count the synonym and non synonym mutations: a protein like histone 3 is identical in different species, it has a KA (non-synonym mutation rate) of 0, this is because histones are fundamental for life, any minimal mutation in their structure is discarded. Some other proteins accumulate more differences (e.g. globines).

Once again these differences are the result of selective pressure, if we see mutations it means that they have survived selective pressure, they might cause a neglectable effect.

Non-synonym mutations

Why do some mutations survive and others not? It depends on the environment and on the mutation. Amino Acids with similar characteristics may allow a non synonym mutation. It may have no effect.



Lysine is similar to arginine in many ways, so a mutation of such type may cause no effect. But if the encoded protein relies on a property of lysine which is not shared with arginine, this could cause a problem and the mutation could be counterselected.

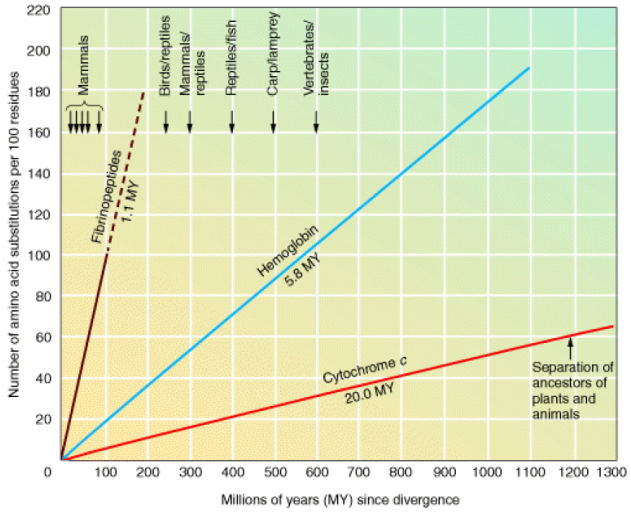
Generally, any mutation in the core of protein (inside) is less likely to survive. It is more likely to affect folding and stability of protein (especially when the size of the residue changes). On the surface there is more freedom (not completely free) for mutations, there is a higher chance to see insertions or deletions. If we add or remove residues in the core (where the amino acids are tightly packed) it is difficult to accommodate the addition of material. On the surface there is more chance (actually this considers only the stability of the protein, not its function).

Rate of mutation

We saw that different protein families can have different rates of changes, usually we assume that the accumulation of differences happens linearly in time. What varies among different sequences is the rate of modification.

If we plot the sequence difference in time, the slope can differ depending on the selective pressure, but the relation is nearly the same.

From this assumption the molecular clock theory was developed: the molecular changes depend on time.

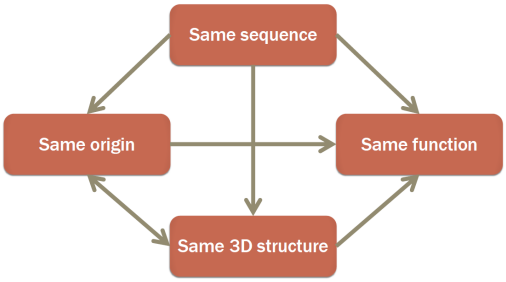


Sometimes this kind of relation is not true, we are simplifying.

Recap

Sequence evolution is much more complex than what we have described so far, but we don't care. We just set some general rules that will help us during the course:

1. If two sequences are sufficiently similar or identical they have the same origin. Note that this relation is unidirectional, if two sequences have the same origin we cannot exclude that they have too many differences such that we cannot recognise them as similar.
2. If two sequences are sufficiently similar or identical they have the same 3D structure. Again we cannot state that if two proteins have the same structure they have the same origin.
   If we see the same sequence we can be sure that the protein will fold in the same way. If we see proteins that fold the same way they might not have the same sequence and the same origin. Dehydratase and hydrolase look similar, they fold in the same way, but they have different sequences and different functions.
3. If two sequences are sufficiently similar or identical they have the same function.
4. If two elements have the same origin they have the same structure and function.
5. If two elements have the same structure they have the same function.



All these rules have exceptions. Sometimes setting up rules is useful to simplify understanding.

Algorithms

Our objective is to compare sequences taking into account how they might have evolved. We need to create an automatic way to align sequences, which has to be also biologically meaningful.

If we want to set up an algorithm to compare differences we have 2 problems:

1. The algorithm itself that carries out the comparison.
2. The weight of each difference, we have to create a scoring function.

These algorithms are optimization problems. We have a scoring function, we want to find among many different alignments, which alignment gives the best score.

Biological sequences have some fundamental characteristics:

1. They are formed by a well defined alphabet.
2. They have a clear polarity, from the 5' to the 3' end for DNA, and from the N-terminus to the C-terminus for proteins.

3. They are linear, meaning that there are no branching points

These are all characteristics of strings, biological sequences are well representable as strings, even though we always have to remember that they are more than just strings.

Python strings

Strings in python are an ordered container of characters.

s = ‘ACGT’
len(s) #get the length of the string (4)
s[0] #indexing (A)

The offset is the difference in between the index corresponding to a certain base and the actual position of that base in the DNA sequence.

For example, considering the first base of a sequence: in a biological sequence the first character is number 1. In python it is 0. In this case the offset is 1. If the sequence starts from base 1000 the offset will be 1000

t = ‘GGTT’
v = s + t #concatenation (ACGTGGT)
s[2:5] #slicing (GTG)
s[:2] #prefix(AC)
S[5:] #suffix (GT)

The prefix and suffix will be really useful, during sequence assembly we usually look for similarity of the prefix suffix type. The suffix of one read is similar to the prefix of another. There are specific algorithms for comparing sequences looking only for prefix-suffix similarities.

s.upper()
s[::-1]
s.count()
s.split()
s.reverse()

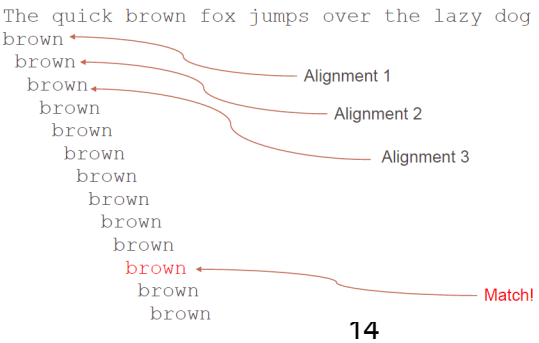
We will compare strings, using different algorithms and methods. Some algorithms were developed from mathematical theorems.

We need to define the different ways in which strings can be compared, and how to evaluate, numerically and statistically, their comparison. In Bioinformatics, many classic string matching algorithms used for text analysis are commonly employed, while other methods were developed or adapted specifically for taking into account characteristics of biological sequences

String matching and alignment

Suppose we have P and T. with P shorter than T. P is a pattern, while T is text. We have to check for occurrences of the pattern in the text, with exact match.

Exact matching can have some applications but usually we look for patterns that are similar (we will have to define similarity). Exact string matching is usually not very meaningful, if we take



into account how evolution works, we understand that exact is not a frequent option.

String matching: find all places in which P occurs as a substring of T. Each such place is called a match or occurrence.
String alignment: a way of putting P's characters opposed to T's characters. May or may not correspond to a match.

Brute force algorithm

To create this algorithm we can use a **brute force approach**.

Brute force: algorithm that tries to solve a problem by trying all possible solutions, choosing the best one. This type of approach cannot be often applied to real cases, because it is too inefficient.

The easiest brute force algorithm consists in shifting the pattern along the text.

How many **alignments** are needed in brute force? With alignment we mean placing a string upon another and checking if they are identical.

m - n + 1

Where m is the length of the text and n is the length of the pattern.

How many **character comparisons** are needed? For alignment we need to compare all the characters of the pattern (n characters).

n (m - n + 1)

```
def naive(p, t):
    occurrences = []
    for i in range(len(t) - len(p) + 1):
        match = True
        for j in range(len(p)):
            if t[i+j] != p[j]:
                match = False
                break
        if match:
          occurrences.append(i)
    return occurrences
```

Worst case scenario: thanks to the break statement, if a letter of the pattern is different from the text the program jumps to the successive alignment. For this reason the worst case scenario occurs when both the pattern and the text are made of the same character repeated n and m times.

Best case scenario: the best case scenario is the one leading to the smallest number of comparisons. In this case it occurs when the first character of the pattern never occurs in the text, at each alignment the algorithm performs only one comparison (m-n+1 iterations).

Boyer-Moore algorithm

A better approach to the problem is the Boyer-Moore algorithm; we won't go into details because we are not interested.

Approximate string matching

The approaches described so far check for an exact matching. In bioinformatics, we are generally more interested in approximate string matching. We want to find all substrings of T that are identical or slightly different from P. The motivation for this is once again evolution.

We need to define what slightly different means, using a **scoring function** that, given two strings, returns a score based on how many characters are different or identical.

Scoring function

We need a function that can scan the alignment producing a score. Once we decide how to score, we need an algorithm able to optimise the function (we need the minimum or maximum value of the function depending on the function).

Given any scoring function the algorithm will find the best alignment. Even if the scoring function has no biological meaning. If we tell an algorithm to do something which is biologically meaningless it will still produce an output.

Finding a good scoring function is not easy, we need to understand how sequences evolve. We will never get a score function able to recapitulate the exact evolutionary history of a sequence.

There are two kind of scoring function:

1. **Distance scoring functions** assign a cost to all differences that they find. If two residues are the same the cost is 0 if they differ the distance is a positive number, depending on how many differences there are a final score is assigned.
2. **Similarity scoring functions** assign a cost to similarities. If there are identical characters we assign some positive score, if the characters are different we assign a score penalty (0 or negative number).

Both scoring functions find usefulness in bioinformatics: distances are used for clustering and phylogenetic analysis, whereas similarities are used for the identification of evolutionary relationships and functional inference. The algorithms are quite similar.

Similarity scoring functions

If we compare sequences for evidence of homology, usually a similarity function is more intuitive, because the score grows the more similar sequences there are. It is easier to understand.

If we use a distant function the amount of similarity is not measured, it is just measured the amount of differences. If two sequences of 10 bp are identical they will have score 0; if two sequences of 1000bp are identical, they will still have score 0, even though they are much more related than the other ones.

Distance doesn't consider how long is the tract of identity of two strings, while similarity grows on how similar and long they are.

Since the alphabets composing biological sequences are composed by a small set of characters, it is likely that two sequences might be similar just by chance, but the longer these sequences are, the less likely is that their similarity is a random occurrence.

Distance scoring functions

Distance functions are used in applications for clustering. They are better at organising multiple sets of objects. They allow the grouping of objects such that the distance within a group is minimised.

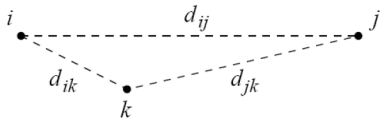
Conditions for distance scoring functions:

- Identity: the distance between an item A and itself is 0.
- Symmetry: the distance between A and B is the same between B and A.
- Triangle inequality: given 3 items A, B and C, the distance between A and C is smaller or at most equal to the sum of the distance between A and B and the distance between B and C.

A distance measure that satisfies these conditions is called a metric.

Metric conditions

$$
\begin{aligned}
d_{ij} &> 0 \qquad for\ i \neq j \\
d_{ij} &= 0 \qquad for\ i = j \\
d_{ij} &= d_{ji} \qquad \forall i, j \\
d_{ij} &\leq d_{ik} + d_{kj} \qquad \forall i, j, k \qquad (triangle\ inequality)
\end{aligned}
$$



When we want to do clustering, scoring function on distance is essential, because we work with triplets and more. trying to form a group in which the intra cluster similarity is minimised and the inter cluster similarity is maximised.

Hamming distance

The simplest measure of distance is Hamming distance, it is defined as the number of positions at which two strings are different. It counts how many different characters there are. The distance is linearly proportional to the number of differences.

Hamming is useful in particular applications, we are comparing only strings with identical length. Can be useful because it is very simple to compute (usually we need something more flexible).

Implementation:

```
import sys
def hamming(str1, str2):
        count = 0
        for i in range(len(str1)):
                if (str1[i] != str2[i]):
                        count += 1
        return count
if len(sys.argv) != 3:
        print(“Usage: %s string1 string2” % sys.argv[0])
```

```
        sys.exit()
str1 = sys.argv[1]
str2 = sys.argv[2]
print(hamming(str1, str2))
```

sys module allows to interact with the system. If we want to run a python program we have to use the command line of our pc, through the command we can also pass arguments to the program (usually bioinformaticians work in linux).

sys.argv is a list with all the elements that were inserted in the command line. The first argument is the program to run. The second and third elements are the two input strings needed for the algorithm: the pattern and the text. After having retrieved the two strings the hamming function is run.

The function iterates through the length of the shortest string, only one alignment is evaluated. It doesn't matter how long the two sequences are, we compute distance only in the first part. This might not be biologically meaningful, since often we don't know the exact start and end point of the sequences that we want to compare.

We could compute hamming distance for each alignment, shifting the smaller sequence within the longer. Remember that with the hamming distance we are only considering substitutions, usually in sequences there are also additions and deletions but with this kind of algorithm we cannot detect them.

Edit distance

With this kind of score function, we can take into account the existence of multiple types of differences among two strings, and weight them differently.

The edit distance is defined as the **minimum number of changes that we need to apply to a string to transform into another**. The possible changes are substitutions, deletions and insertions.

When comparing, we need an algorithm that can transform one string into the other.

Ex.

TGCATA~~T~~ → delete last T
TGCAT~~A~~ → delete last A
ATGCAT → insert A at the beginning
ATCCAT → substitute G with C
ATCCGAT → insert G

In this case the distance is 6 if we give an identical cost to each operation.

We can assign whatever cost to substitutions, insertions and deletions. If we assign the same cost to each edit operation we will find the minimum number of edit operations to transform one string into the other.

As you can imagine, there are multiple ways to transform a string into another, when computing the distance, we want to consider the sequence of changes that yields the highest similarity scores. All these algorithms are optimization problems, we want the best solution, which depends on the scoring function.

The most common variant of the edit distance is the **Levenshtein distance**, which assigns the same score to each substitution, so the distance corresponds to the minimum number of changes that are needed to go from a string to another. When computing an edit Levenshtein distance we have to implement it in an efficient way.

The score of the Levenshtein distance represents both similarity and distance, for this reason it is a good way to compare biological sequences. It takes into account the three ways in which biological sequences can evolve: substitution, addition and deletions.

When computing this kind of distance we are looking for the minimum number of changes, we have to apply the parsimony principle. The edit operations will be an approximation of how sequences have evolved over time. We cannot know which is the actual path that they followed during evolution, we are just approximating it. We are giving the simplest explanation that might correspond to evolution or not.

Aas we said before, there are many ways for transforming a string into another. For the same strings we can have different solutions. We have to select the lowest amount of changes.

Ex. the same sequence of before, can undergo the following changes.

ATGCATAT → insert A at the beginning
ATGCATAT → delete T
ATGCGAT → substitute A with G
ATCCGAT → substitute G with C

Levenshtein distance

The most common edit distance algorithm is the Levenshtein distance. It can be a good measure to compare biological sequences.
This distance measurement can be implemented with a recursive algorithm, with complexity equal to the…..

Pros and cons of the Levenshtein distance

Pros:

- It takes into account edit operations that can describe biological sequence evolution: substitution, insertion, deletion.
- It can be converted into an alignment.
- Less complex algorithms are available.

Cons:

- The employed scoring function is not flexible, and the cost of each edit operation is generally the same.
- The distance itself cannot be easily interpreted in biological terms.

The similarity scores are better to understand evolutionary relationships.

Similarity scores

A similarity score is computed for any alignment of 2 biological sequences, it grows with the increase of their similarity.

There are different ways in which we can define a similarity score: we will explore how to build meaningful score function, on the basis of parsimony and actual biological data. The way in which we measure similarity is based on an estimate on how the sequences might have evolved over time.

In the simplest instance, we can align the identical characters of the two sequences and we consider them as coming from a common ancestor (parsimony), the count of the identical character is a measure of similarity (similar to hamming distance).

How to compute similarity?

The first way to implement a similarity score is to use a brute force algorithm: we can shift a string over the other, trying all possible alignments, we register the score (based on the matches) of each alignment and then select the alignment with the highest score.

The complexity of this algorithm is not too bad, we can simply have an intuition over it. If we have a string of 6 characters and a string with 5 characters we have 10 possible alignments (6+5-1) and we make 30 character comparisons (5*6).

The problems of this kind of similarity are similar to the problems of the Hamming distance: we cannot handle strings with different length and we are completely ignoring insertions and deletions (no gaps in the alignment).

Considering insertions and deletions

Can we approach the problem of comparing two strings, including possibility of addition and deletion, with a brute force algorithm? In principle yes, but the number of alignments is almost exponential, it doesn'ìt have applications in biological sequences that are very long. Unfortunately sequence comparison requires speed, in many cases we have to compare thousands of subsequent characters.

Before trying to find a way to evaluate dashes, remember that we cannot differentiate an insertion from a deletion; we just know that one string has more characters than the other. For this reason we refer to this kind of modification as **indels**.

How could we add a way to score gaps?

In general, the similarity scoring function should be more complex, taking into account not only matching positions, but also the cost of introducing gaps, and the cost of having mismatches. We need a way to maximise this scoring function, finding, among all the many possible alignments between two sequences, the one with the highest score.

We can arbitrarily decide that a match has score 2, a mismatch has cost 1 and a gap has score 1 for each deleted character.

We could add penalty scores to penalise differences (differences are substitutions and gaps), and then find the alignment that explains the differences in the simplest way possible, with the smallest number of evolutionary events.

If we don't penalise differences we wouldn't be able to minimise the number of evolutionary events. If we need a lot of gaps to align two sequences in the right way, we have to keep this into account. Even if two sequences have a lot of matches, they may have a lot of gaps too, by

establishing a criteria we can give a weight to each difference and establish the degree of similarity.

The biological relevance depends on how good the scoring function is. How well the scoring function describes evolution. These score functions can be based on real evolution data of sequences, in this way we will have the most objective representation of evolution. In real bioinformatics, we are not interested in arbitrary schemes, we can base our score functions on the reality of how sequences evolve.

Dynamic programming algorithms

Given a meaningful similarity score function, how can we compute the best possible alignment?

We are going to describe the algorithms that are considered the best, they were created many years ago but we are still using them to align.

These kinds of algorithms can be applied in all cases in which we want to solve a problem and the **problem can be decomposed to smaller subproblems** until we reach a problem which has an immediate solution. olution is immediate.

Dot matrix or dot plot

A dot plot is a simple visual way for comparing strings (biological sequences). It is a matrix in which on 1 axis there are the characters of one string and on the other axis the characters of the other string. Each cell corresponds to the combination of one character of one string and a character of another string. The dots on the plot correspond to the matches between the characters of the two sequences. We are just highlighting the matches, we mark all calls with identical characters

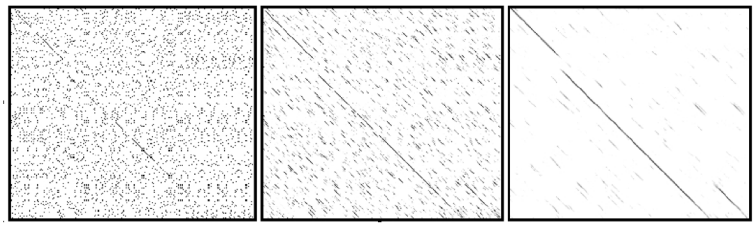
At the end, usually we can see diagonal tracks of labelled cells. Usually the diagonal lines are parallel and they are separated by jumps. Each jump needed to connect two diagonal tracks corresponds to a gap in the alignment, while the diagonal portions are the matches (or substitution if a dot is missing)

Clearly we will also have alignment of identical pairs by chance, these are the interspersed dots around the diagonals or everywhere else in the plot. This is called background noise.

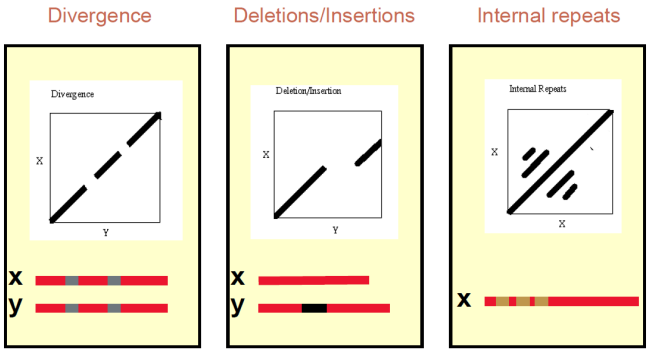
This kind of representation has a huge **background noise**. Usually to simplify the representation we can decide to highlight only the substrings in which there is a contiguous occurrence of matches. We could set the threshold to couples of matches or to a more stringent condition. The higher is the threshold (required a longer match to show the line) the fewer lines will show up.

This threshold is called window size. By increasing the window size, the similarity between the two sequences is much more evident, with much less background noise. Note that, if we increase the window size too much, also the main diagonals may disappear.

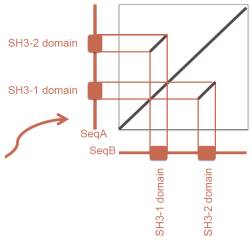
Window = 1 Window = 5 Window = 15



What do we look for?



- Diagonal tracks represent sequences that are identical.
- Two diagonal tracks on the same line with interruptions indicate a substitution (divergence)
- Two diagonal tracks that are parallel but on different lines indicate a gap.
- If there are symmetric segments surrounding the main diagonal it means that the sequences have multiple copies of the same substring. The biological meaning of this phenomenon is once again evolution.
  Proteins have modular domains that can carry out a function on their own, so they are conserved and used in multiple proteins.
  Also genomes have a lot of repetitive sequences.



To compute the optimal alignment, given a similarity scoring function, a class of algorithms generally called **dynamic programming** are used.

A dynamic programming algorithm is an algorithm that solves a problem by finding the optimum solution of a subproblem for which the solution is obvious then extends the solution to the whole problem. If we can decompose a problem into subproblems (easier problems), it is said that the problem has an optimal structure (this doesn't apply to all problems). Usually a matrix representation of the problem is used, the solution of the problem in a restricted area of the matrix allows to use that information to fill all the rest.

1. Needleman and Wunsch: aligns and computes the optimum global alignment. It includes all chars from the beginning to the end.
2. Smith and Waterman: we find the substring which is most similar between two strings.

These algorithms are based on two assumptions:

- Each column of the alignment is independent from the other columns.
- If one can extend in the optimal way a partial alignment of two subsequences that were already aligned in the optimal way, the resulting alignment is optimal (this assumption holds true, since it can be proved that the alignment problem has an optimal substructure).

Needleman and Wunsch

This algorithm exploits the concept of dot matrix: visual representation in which one string is on the x axis and the other string on the y axis. Each character is a column or a row. Each cell is a particular pair of characters. In this matrix we can identify the perfect alignment.

In this matrix all possible alignments of the two strings are possible and can be represented by different paths. Each path from the first cell to the last corresponds to a different global alignment.

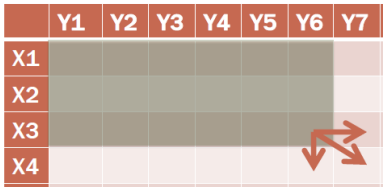
How can we walk in the matrix? The path must follow rules that make sense: we can move in the cell on the right, to the cell below or to the cell on the bottom-right (diagonal step); all the other directions are meaningless.

- Diagonal step: adds to the alignment two new characters, we are adding a perfect match or a substitution to the alignment.
- Horizontal step: adds only a character from the string on the y axis, we are adding a gap on the x sequence
- Vertical step: adds only a character from the string on the x axis, we are adding a gap on the y sequence

The number of paths is almost exponential. Each path, hence each alignment, has its score, depending on the path itself and the chosen scoring scheme. We need to find the best one. We don't want to generate one of them and then choose the best. We need a way to reach it without trying all of them.

Here the algorithm takes advantage of optimal substructure.

If we have two sequences of m and n characters and we know the optimal alignment of a substring up to i on the first string and j on the second string. Then we know how to extend the optimal sub-alignment of one step in an optimal way (optimal extension of the sub-alignment), by assumption, what we get is still optimal. It might not seem particularly useful but this is the basis to solve the whole problem



Let’s say we know how to align optimally up to i=3 and j=6 (we know the perfect alignment).

How can we extend this?

We can follow the rules of the matrix paths and we can extend the optimal sub alignment by moving from the bottom right corner of the green part by the three possible extensions. Each extension has a clear meaning. (explained above).

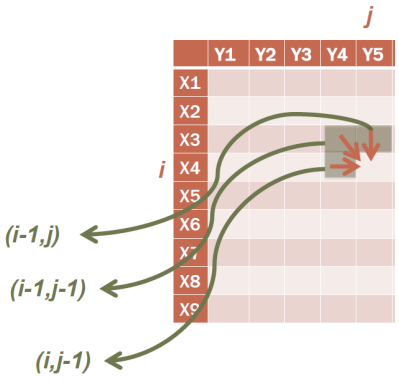
Each of these 3 cases we will apport a modification to the value of the score function. Two of them will correspond to a gap penalty and the diagonal movement corresponds to the alignment of two characters (identical or not).

Which is the best way of extending the sub alignment? whichever case has the best score. This is true and can be demonstrated to be true for each cell of the matrix.

Unfortunately knowing the score of the movement is not enough, we want an optimisation.

We do not need to check only the score of the step, we have to evaluate a cumulative score depending on how we reached the cell from which we are moving and then add to that score the score of the next step. At that point we can choose the direction that yields the best score.

Let's say we are working on 4,5.



How can we compute optimal value? If we compute optimal values of the previous cells, in the last cell the score corresponds to the score of the whole path.

Each cell can be reached from 3 different positions. Each of these 3 positions corresponds to 3 different pairs of prefixes of the two strings.

(i,j-1)=(4,4); (i-1,j)=(3,5); (i-1,j-1)=(3,4).

Suppose we know the best alignments up to each of these positions.

To find the best subsequent step we have to take the scores of the optimal alignments of the 3 cases and add to it the score corresponding to the movement.

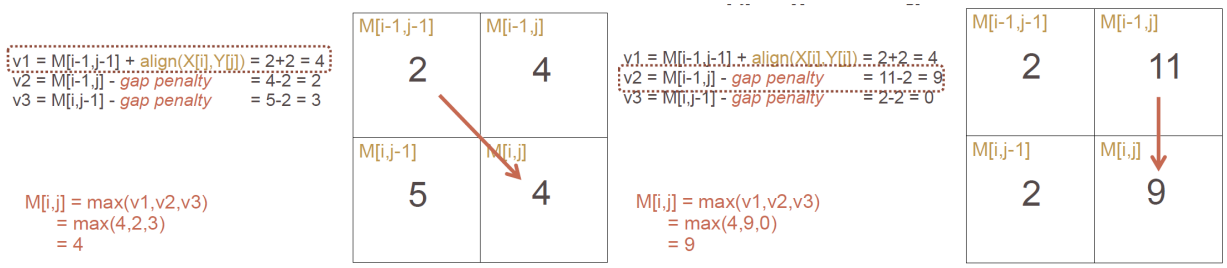
In the case of vertical and horizontal movements, we will add the gap penalty, while in the case of a diagonal movement we may add the substitution penalty or the addition corresponding to the match. If the step is a match this movement will be always preferred among the others. In the other cases, if the score to reach the cell above is much more than the score of the diagonal cell, the algorithm will select the vertical movement and will introduce a gap, even if the penalty is higher than the substitution penalty.

In a formal way:

$$M[i,j] = \max \begin{cases} M[i,j-1] + GAP \; \begin{matrix} 1\ldots i \\ 1\ldots j-1 \end{matrix} + \begin{matrix} - \\ j \end{matrix} \\ M[i-1,j-1] + Score[i,j] \; \begin{matrix} 1\ldots i-1 \\ 1\ldots j-1 \end{matrix} + \begin{matrix} i \\ j \end{matrix} \\ M[i-1,j] + GAP \; \begin{matrix} 1\ldots i-1 \\ 1\ldots j \end{matrix} + \begin{matrix} i \\ - \end{matrix} \end{cases}$$

We have to pick the highest value among these 3. For now, we are not interested in how we computed the scores up to the previous cells, we just have to select the biggest one.

Given that we know the values up to the previous cells, the computation is just a matter of similarity score function. We may decide that the gap penalty is -2, the score for the match is 2 and the penalty for a substitution is -1.



This is a **progressive optimisation**, the last cell will contain the score of the best possible alignment.

Initiation

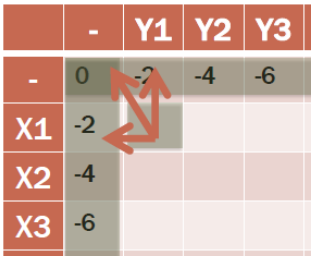
How to start?

Since we are using dynamic programming and the problem has an optimal structure, we can decompose the problem to smaller subproblems. The alignment of two characters by knowing the previous results is a subproblem, we solved this, we are able to extend something optimal.

We need to keep decomposing the problem in simpler subproblems until we reach a point in which the solution is obvious. We need a starting point from which we can compute the score. From this point we can progress in filling the matrix.

The problem is that the first character has no neighbours, we can solve this by reasoning on what usually happens at the start of an alignment.

To obtain the initial scores we add a dummy column and a dummy row. If we start the alignment by position (1,4) we have the first 3 characters not aligned, they are coupled with a gap in the other string. The score of this starting position is just 3*gap_penalty, to this score we can add the alignment of the first two characters.



We can insert the gap penalties corresponding to the initial gaps in the dummy line and column. The first cell now has two neighbours, they are 0, -2, -2. Now we can compute the value of the alignment. We decomposed the problem into smaller ones. The first row and column can be computed as sums of gap penalty.

Traceback or backtracking

At this point we should have reached the optimal value of the similarity score function, but we had just gone through some numbers in a matrix, we do not have the two sequences aligned one on top of the other in the most convenient way. Without the alignment the algorithm is useless, we want to understand the evolutionary history of the biological sequences.

We need to trace back from the last cell to the beginning, we need to reconstruct the path that we have travelled to reach the last cell. If the score in the last cell was computed from the cell on the left we have to write the alignment with a gap.

We start from the bottom and go to the first cell. At each step of the traceback we move from 1 cell to the cell from which we computed the best alignment.

It can happen that during traceback we find out that there are multiple best alignments, more than one alignment with the same score. In that case the best thing is to consider one of them but to be more accurate we would have to follow all possible best alignments.

Recap of Needleman and Wunsch algorithm:

- Initialisation: we add 1 additional first column and 1 additional first row and we fill them with the gap penalty sum.
- Iterating in the matrix: we compute the values of the score function for the best alignment by knowing the previous one.
- Traceback: write the optimal alignment.

Complexity

The complexity of the Needleman and Wunsch algorithm is the product of the length of the two strings: O(n^2). This is the very simplest algorithm for alignment. This kind of complexity is fine if we want to align two sequences, but when we have to align 100,000 or millions of sequences this algorithm is too inefficient.

This is an **exact algorithm**, that is to say that we will **surely get the optimal solution**.

Smith and Waterman

This Needleman Wunsch alignment computes the optimal global alignment, global means that the alignment considers all the characters of all sequences. That's fine but sometimes a local alignment might be better, a local alignment is an alignment that doesn't include all characters, it can start from whatever cell and end to whatever cell.

Biologically a local alignment is better when:

- There is large divergence between two sequences. We take the part where two strings are more similar, ignoring the rest.
- There are repeated elements between different sequences. Biological sequences are modular, often the same element is present in multiple sequences that may have little to no correlation in the rest of the sequence (transportation during evolution).

How to compute a local alignment?

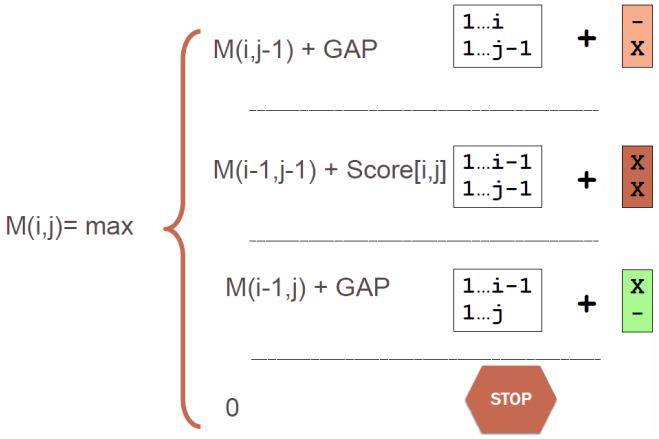
This algorithm derives from the Needleman and Wunsch algorithm, so there are only few differences. We still have a matrix, reporting two strings on the two axis, we still need to initialise the matrix and compute the optimality, then we will have to trace back.

Differences:

- The initialisation of the matrix is performed by adding a dummy row and column filled with zeros
- When computing the score, we reset the matrix every time a strong divergence is observed, any time we try to compute the optimum alignment and only negative score are produced, this indicates that up to that point the sequences are very different.
- Start the alignment from the cell containing the maximum score, it corresponds to the end of the alignment of two substrings, in particular the sub alignment that has produced the highest similarity score.

How can we modify the prev algorithm for local alignment? We have to avoid regions of strong divergence. We have to mark in some way in the matrix the cell with the highest score. And we need to rethink the traceback to start from the maximum value.

In practice we still need to compute the value of the 3 neighbouring cells and add to them the score of the last step to select the best one. In order to avoid regions of strong divergence (cells with negative scores) we have to add a 4 condition to the system: it is enough to add a constant value, corresponding to zero, that can be picked together with the others.



If 3 other conditions have a negative score we place 0 in the cell (i,j). We still take the maximum of the cases but now the cases are 4.

This is no longer a progressive optimization.

Another difference with respect to the previous algorithm is the way in which we initialise the matrix: during initialisation we insert a column and a row and we fill them with zeros.

We don't want negative scores. The whole idea of local alignment is to avoid negative scores.

With the initial values we can fill the whole matrix, and, when we reach the last cell, we just need a traceback. We start from the cell of the matrix having the highest similarity function score. We do not consider the values after the highest one because they are all steps that decrease the score; the sequences from that point on will contain differences.

The traceback is stopped when we reach a cell for which all 3 neighbours contain 0.

We can also perform multiple tracebacks, if there are more regions of similarity we might be able to get all of them. We just reset the cells of the best alignment and consider the second best.

Biologically meaningful alignments

The most accurate algorithms to maximise a similarity score function are the ones that we just described, we can't do better than this. As we said many times, the biological correctness of the alignment depends only on the score function, the algorithms themselves are correct.

If we want to improve the biological meaning, we need to set scores based on how sequences truly evolve over time, we cannot rely on an arbitrary score function.
Up to this point we supposed scores for match, gap and mismatch. Now we want to find a way to evaluate differences and identities with scores that are biologically meaningful.

Amino acid sequences

We will base the creation of similarity score functions on empirical evidence coming from real mutation rates in biological sequences. This process works better if the biological sequence under study is a protein sequence, we can observe that usually aminoacids do not change randomly into one another, some mutations are more favoured than others. There are similar amino acids and very different ones. Two different mismatches shouldn't have the same score.

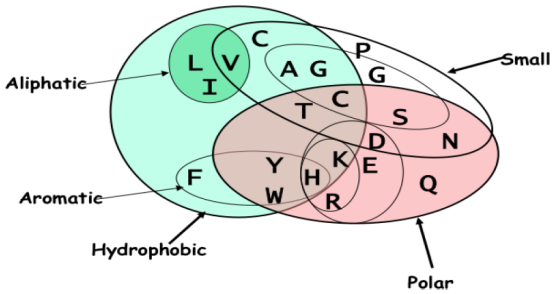
Tryptophan is a rare amino acid, it usually doesn't change much during evolution; alanine on the other hand is more common and it has several amino acids with similar characteristics, so it is more likely to change.

There are some amino acids which are more likely to be mutated, and others that are more likely to be conserved.

Classification

To observe this phenomenon and to give a different weight to each mutation we can start by classifying amino acids into groups. In the same group we place amino acids as similar as possible.
Then we modify the scoring function s.t. the alignment of residues from equal groups has a different score with respect to the alignment of residues from different groups.

However the classification is not easy to determine. For example, tyrosine has an aromatic ring and an OH group, it is difficult to place it in an unique group. Scientists have come up with the following classification.



Given that this classification is a good representation of the actual likelihood of each mutation, we would still need an arbitrary scoring scheme to assign a small similarity score to mutations

across groups and bigger scores to mutations within the same group. But the whole point of this classification was to find a more biologically meaningful score function.

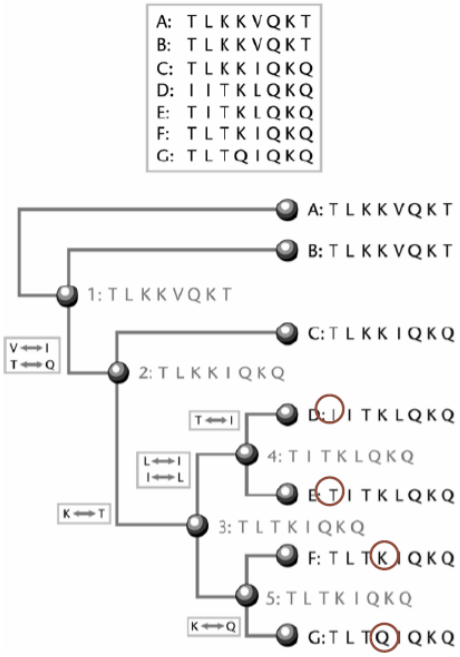
Substitution matrix

The way in which this problem was solved was to compute substitution rates for each pair of amino acids by using data coming from real protein sequences.

Margaret Dayhoff (considered the first bioinformatician) was the first to have this intuition and built the first **substitution matrix**. The matrix contains all aminoacids in the rows and columns, 20x20, it reports in each cell the substitution rate of the pair of amino acids.

What we need is the cost of substituting an amino acid with another, by counting its **substitution frequencies**, i.e. the number of times that, during evolution, a given amino acid was mutated into another, and this mutation was tolerated by selection. The matrix will also contain the conservation frequencies, in correspondence of the diagonal (A-A, W-W pairs) which are the number of times that A has not changed during evolution. The matrix will contain 400 cells and it will be symmetric.

The data to fill the matrix were collected from protein sequences coming from the same protein family; these sequences were very conserved orthologs that changed very little during evolution. Once they gathered the sequences, the group of Dayhoff counted the differences by hand, they aligned the several sequences by eye (very easy task since they were mostly identical) and then they registered every substitution.



Note that this was the only way in which they could register the substitution scores for the first time. They were trying to build a system to increase the biological correctness of the alignments, so they could not use old similarity score functions to align the sequences automatically. Thanks to the huge similarity among the chosen sequences, they were able to compute the substitution scores by hand.
In order to keep track of all substitutions in a set of sequences, each possible pair of sequences among the group has to be aligned and analysed.

All the differences between the sequences correspond to mutations that survived selective pressure. The substitution rates can give an estimate of which change with respect to another has the chance to survive. We give a higher score of similarity to changes that are common and that survive easily selective pressure, while we give a lower score to rare changes that do not happen often (if they happen it means that the two sequences have been separated for a long time).

These scores will be inserted in the cells of the substitution matrix. Now we have to define a mathematical way to compute a score starting from a substitution rate.

They converted each substitution frequency (Aij) into a probability (pij). Pij is the probability of having a mutation between amino acids i and j. It is computed as a ratio between the number of times the substitution between i and j was observed, over the total substitutions of i observed.

$$p_{ij} = \frac{A_{ij}}{\sum_n A_{in}}$$

To avoid working with probabilities we can directly convert it to a score and fill the matrix. The score is computed as the logarithm of the probability over another normalisation factor: the frequency of amino acid i times the frequency of amino j. The expected number of substitutions depends on how frequent the two amino acids are.

$$s_{ij} = \log \frac{p_{ij}}{p_i * p_j}$$

Finally the score is rounded to the nearest integer number and it is inserted in the matrix.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**2**	-2	0	0	-2	0	0	1	-1	-1	-2	-1	-1	-4	1	1	1	-6	-4	0
R	-2	**6**	0	-1	-4	1	-1	-3	2	-2	-3	3	0	-5	0	0	-1	2	-4	-3
N	0	0	**2**	2	-4	1	1	0	2	-2	-3	1	-2	-4	-1	1	0	-4	-2	-2
D	0	-1	2	**4**	-5	2	4	1	1	-2	-4	0	-3	-6	-1	0	0	-7	-4	-2
C	-2	-4	-4	-5	**12**	-6	-6	-4	-4	-2	-6	-6	-5	-5	-3	0	-2	-8	0	-2
Q	0	1	1	2	-6	**4**	3	-1	3	-2	-2	1	-1	-5	0	-1	-1	-5	-4	-2
E	0	-1	1	4	-6	3	**4**	0	1	-2	-3	0	-2	-6	-1	0	0	-7	-4	-2
G	1	-3	0	1	-4	-1	0	**5**	-2	-3	-4	-2	-3	-5	-1	1	0	-7	-5	-1
H	-1	2	2	1	-4	3	1	-2	**7**	-3	-2	0	-2	-2	0	-1	-1	-3	0	-2
I	-1	-2	-2	-2	-2	-2	-2	-3	-3	**5**	2	-2	2	1	-2	-1	0	-5	-1	4
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	**6**	-3	4	2	-3	-3	-2	-2	-1	2
K	-1	3	1	0	-6	1	0	-2	0	-2	-3	**5**	0	-5	-1	0	0	-4	-5	-3
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	**7**	0	-2	-2	-1	-4	-3	2
F	-4	-5	-4	-6	-5	-5	-6	-5	-2	1	2	-5	0	**9**	-5	-3	-3	0	7	-1
P	1	0	-1	-1	-3	0	-1	-1	0	-2	-3	-1	-2	-5	**6**	1	0	-6	-5	-1
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	**2**	1	-3	-3	-1
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	**3**	-5	-3	0
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-4	-4	0	-6	-3	-5	**17**	0	-6
Y	-4	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-5	-3	7	-5	-3	-3	0	**10**	-3
V	0	-3	-2	-2	-2	-2	-2	-1	-2	4	2	-3	2	-1	-1	-1	0	-6	-3	**4**

The scores can be positive or negative, the score is positive if the substitution frequency is higher than what we may expect to see by chance, the substitution is tolerated (the substitution happens very commonly, so the two sequences that differ for this particular substitution are likely highly related). The negative scores are less frequent, they are counter selected by evolution (if two sequences differ for one of these substitutions it is likely that a lot of time passed since they started diverging).

This matrix will be part of the score function, we just need to look at the pairs we are trying to align and fetch the score in the substitution matrix.

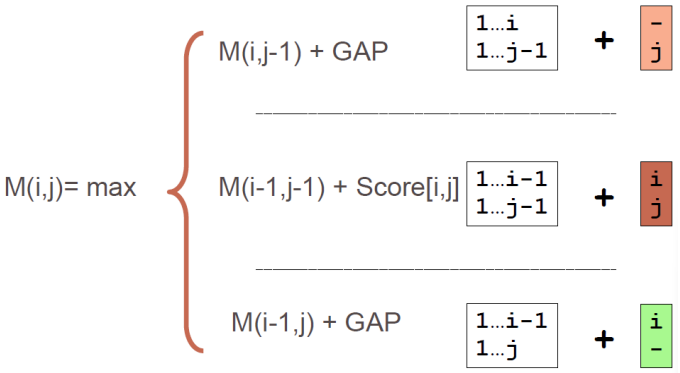
50 years ago, they didn't have many protein sequences available so they computed the scores with few samples. Over time the substitution scores were recomputed on larger data and, surprisingly, more or less the results were the same. This implies that these substitution rules are more or less universal, if we change a set of proteins we end up with similar scores.

The values of the matrix are symmetric, we can never know if a k changed to a q or a q changed to a k.

Some peculiar aminocaids, which are rare, usually have higher conservation scores and lower substitution scores: for threonine, the lowest substitution score is of -8 and the conservation score is of 17. When there is a threonine it usually means that it has an important function and it cannot be substituted. Others have a smaller conservation score, as alanine, which can be substituted by many similar amino acids with no modification of the function.

The conservation scores are computed similarly to the substitution score, but, instead of starting from the frequency of substitution of the amino acid, we start from the frequency of conservation, which is counted as the number of times that amino acid has an identical correspondence in the other sequence. This value is then transformed to a score.

If we want to include the matrix it in the algorithm we just change the computation of the score for a substitution.



Score[i,j] is the cell of the substitution matrix corresponding to the amino acid pair under investigation.

PAM

Since the original substitution matrix was computed on a set of very similar protein sequences, with very clear evolutionary relationships, the matrix scores are not suitable for aligning more divergent sequences.

The original matrix was computed on protein identical at 99% of their sequences, it is called PAM1, where 1 indicates the percentage of divergence of the sequences under alignment and PAM stands for **Percent Accepted Mutation**.

Scientists proved that the variations on protein sequences remain the same in very different species and protein families, the only caveat is that we need to consider very conserved proteins (that can be easily aligned by hand). These are proteins under strong selective pressure (histones).

The scores that we obtain are consistent but they are representative only of rates for conserved proteins. When computing the alignment of other kinds of protein, under weaker selective pressure, are these values correct?

It would be better to have multiple substitution matrices one for each degree of divergence of the sequences under alignment. How to compute these substitution matrices at different degrees of divergence and selective pressure?

We need to be able to align sequences which are very different, but we don't have an easy way to compute a meaningful sequence alignment if we don't have a good scoring system. This was the very same problem found by Dayhoff during the analysis of conserved sequences; in this case, it couldn't be solved by hand alignment, because the sequences were too different.

They used a shortcut, a mathematical way for simulating a higher degree of divergence. They started from the original substitution matrix (1% of divergence) and computed the other matrices for less conserved protein families.
If the PAM1 matrix is multiplied for itself, the resulting values might correspond to substitution rates between sequences differing by 2%. If we continue doing this, we are simulating further evolutionary steps. With **evolutionary steps**, we refer to the number of differences every 100 residues.

The scores should be similar from good alignment of divergent protein sequences. Then we get different matrices. We get a sequence of matrices with values that become lower.

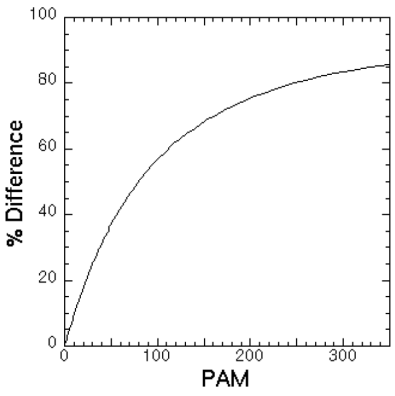
This mathematical trick worked, different matrices were produced, one for each divergence degree. If we use them to align proteins which are very different from each other we obtain a different alignment with respect to the results produced using PAM1.

PAM	0	1	30	80	120	200	250
% identity	100	99	75	60	50	25	20

If two sequences are phylogenetically distant, one should use to align them a PAM matrix of higher degree, and vice versa, the most commonly used PAMs are PAM120 and PAM250, which can be used for the alignment of protein sequences having around 50% and 20% of protein identity, respectively.

The series of matrices that we obtain has values that become smaller in absolute value. If we align divergent sequences we aspect many variants not distributed randomly, so the rate of substitution should decrease.

How do we know that it is biologically meaningful? If we see that there are alignment that make sense, if we see that the important amino acids are aligned, we can inspect some domains of the sequence with known function and if they are aligned we are right (e.g. active sites).



This curve reports the % of difference between the two sequences for which we have to use a particular PAM matrix.

The workflow is the following: we look at sequences, we estimate by eye their difference (we don't need to guess the perfect matrix, we just need to roughly choose a good matrix), we choose the right PAM and we use it in the score function.

Other substitution matrices were developed over the years, and by changing the protein families on which the matrix is computed, the scores were more or less the same. The

substitution rates seemed to be universal for whatever protein family.

Exceptions: some matrices computed on eukaryotes and prokaryotes seem to be different.

Criticism: we are counting the differences in very well conserved proteins, these by definition are less incline to change, few regions of these proteins allow substitutions and we are analysing these regions. Usually the regions of conserved protein that are free to change are not not important for the protein and whatever substitution may be accepted.
To have more information, we would have to look at the substitutions in the conserved regions, since these are strongly subjected to selective pressure; each change could disrupt the protein fold and the less harmful substitutions are more frequent than others.

BLOSUM

20 years later the Blosum substitution matrices were computed; they are the most commonly used matrices nowadays. Ideally we would like to count differences, from which we compute substitution scores, in conserved regions, where each change might have bad consequences.

In order to reach our goal we need to identify the hypervariable regions in the multiple alignments and then ignore every substitution happening in those regions.

We still have a sequence alignment, possibly multiple, made of conserved proteins. How can we understand which part of the alignment corresponds to which part of the protein just by looking at them?

Not a naive answer: use software to predict the 3d structure of the protein.

Simpler solution: some structure features appear with specific patterns, if a certain region of the multiple alignment has many gaps, this more likely corresponds to the surface of the protein. This is because on the surface there is more freedom for deletion and insertions, the added portions can fold independently without affecting the stability of the protein (as opposed to indels in the core of the protein. To compute the matrix they simply considered only the substitutions in the parts of the alignments with no gaps.

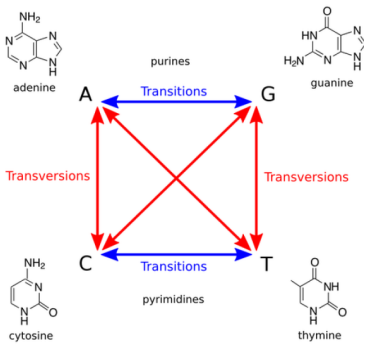
Blosum stands for **BLOck SUbstitution Matrix**, the blocks refer to parts of the sequence. We count the differences in the blocks without substitution. As for PAM matrices, we still need multiple matrices useful for sequences with different degrees of similarity, to produce them, we can classify each block on the basis of the % of difference. Then compute the scores from the ones with the same degree of difference.

The nomenclature of the BLOSUM matrices is the following: BLOSUM+%similarity. For example BLOSUM62 is the matrix that has to be used when comparing blocks with 62% of identity. The similarity degree is stated by eye.

It is difficult to have a benchmark or to have statistical evidence to state which matrix is better. However scientists agree that BLOSUM matrices perform better.

Nucleotide sequences

Substitution matrices for nucleotides don't work as well as the protein ones, it is possible to observe patterns in nucleotide substitutions but this is not enough to build a matrix.



We can have mutations:

- Transitions
- Transversions

Some kinds of variances are more frequent (transitions) are more frequent, but it is difficult to convert this empirical observation to scores. The rate is usually also significantly different among organisms. It is impossible to derive some universal rule for mutations of nucleotides, so we rely on arbitrary scoring schemes.

Meaningful gap penalty

The last piece we need to build a meaningful scoring function is a smart way to define the gap penalty. Usually gap penalties are large negative values that increase linearly with the size of the gap. This linear relation is the simplest, but there are more complex and smarter ways (the downside is that we will also increase the complexity of the algorithm for the alignment).

Gap affine penalty

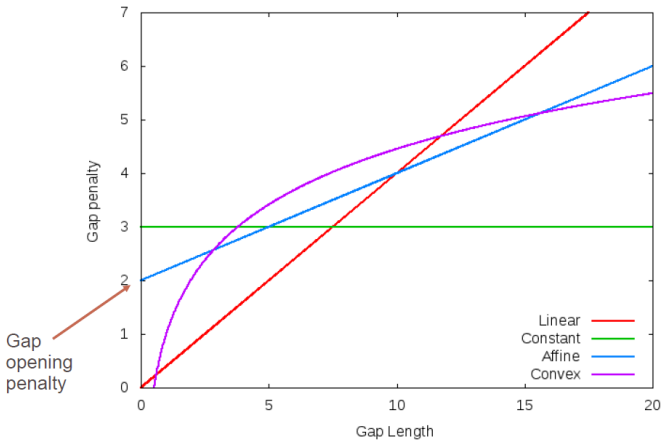
In this gap scoring scheme we assign two penalty values to each gap: gap opening penalty and gap extension penalty.

The gap opening penalty is assigned just 1 time for each gap, it usually is a large negative value. The gap extension penalty takes into account how long the gap is, usually it is smaller than the gap opening penalty; this is because the occurrence of a gap can be considered as one single evolutionary event, no matter how large it is. This is more coherent with the parsimony principle.

With longer gaps we will have a “small” increase on the basis of the length of the sequence.

There are some other gap penalty systems, one that seems to be better is the convex gap penalty. It consists in a gap opening penalty plus the extension penalty times the logarithm of the length of the gap. For the convex gap penalty there is no algorithm that is simpler than cubic, so it is difficult to apply this kind of function in the real world.

Nowadays, the common way to align two sequences is using BLOSUM substitution matrices and affine gap penalty.



DATABASE SEARCH

Up to now we only described pairwise alignment, we compared a string against another. From this kind of alignment we can convince ourselves that the two strings are homologs. But this kind of procedure can be performed only when we already have some sequences with some evidence of similarity.

In many other cases we just have an unknown sequence and we want to find its homologs in other species, by comparing it against a whole set of known and annotated sequences (database). We have one sequence of interest, and a collection of sequences from the other species, we want to search which are the most similar sequences. Maybe in the database there are multiple homologs, each with a different part conserved.

The brute force approach to solve this problem is to align the unknown sequence to all the sequences in the database, but if the database is huge this would require **too much time**. Moreover if we compare a sequence against a vast collection of sequences, the **majority of the comparisons will be useless**. Imagine we have 20 000 proteins for 100 species, we would have 2 million protein sequences, among these, maybe less than 100 will be homologs, we would perform 1900000 useless comparisons. Moreover **by chance there might be similarity** in the database, there might be a lot of matches that might seem good or significant by chance.

This problem can be approached in another way: we can select from the database some sequences that might be related to the sequence of interest, s.t. when we perform the alignment we do it only on promising proteins. We spend computational efforts only on sequences that might be related. We select from the database a small **subset of sequences**.

This selection must be based on **reasonable assumptions**, which can be quickly verified.

Forming an assumption of this kind leads to uncertainty: we might discard some true homologs but that's the cost to speed up the process. We do not have the guarantee that the result will be optimal. These kinds of algorithms based on reasonable assumptions are called **heuristic algorithms**. They are used instead of exact algorithms that are too complex to solve a problem.

An heuristic algorithm can solve a very complicated problem without guaranteeing that the solution is optimal, but it gives a good solution.

Heuristic algorithms are based on assumptions, the **usefulness of the solution is dependent on assumption**. If the assumptions do not hold, the algorithm will give bad results. Heuristic algorithms are common in bioinformatics and other fields, for all problems for which exact algorithms are too complex.

To have an idea of how far the solution proposed by the algorithm is from the optimal one, often solutions are supported by statistical evaluation.

The problem of database search in formal terms is:

We have a database containing K sequences, we have a query sequence Q, we want to find all sequences for which we have an evolutionary relationship. We want all the sequences with a similarity score higher than a threshold T.

The optimal solution can be found by comparing Q with all the sequences of the database and then selecting the best one.

The faster solution is to select a subset of the database and make the comparisons only on the sequences belonging to the subset. The risk is that in the filtering we lose something, if the filtering is not smart we will discard the homologs of Q, and if the true homologs are discarded in the first step they won't undergo the alignment and they will be lost.

How big is this problem? It depends on the kind of sequence that we start with (Q) and on how many homologs of Q are present in the dataset, some common sequences might have a lot of homologs or related sequences in the datasets.

Usually if the problem is to characterise a sequence, we don't need to find all its homologs, it is enough to find some of them. We don't need the optimal solution, we want a good solution that gives us information to characterise the sequence Q with unknown function.

There exists two algorithms that exploit this approach

- Fasta
- Blast: this is the most famous and important bioinformatic algorithm. It is the basis of modern biology, it allows the usage of model organisms.

Fasta

Fasta stands for fast for all kinds of sequences, it was invented many years ago and it is still used nowadays for nucleic acid sequences, it works better than blast which is mostly used for proteins.

The filtering procedure of fasta is based on assumptions that are stronger than the blast ones. Blast is similar in concept, it maintains the core idea of fasta.

After the filtering the Needleman Wunsch algorithm is applied to find the best alignment.

Which is the assumption? If two sequences are homologs (homology of any kind), no matter how distant these sequences are in evolutionary terms, they have maintained some homology in their subsequences.

We take a seq from the database, we decompose it to sequences of length k: **kmers**.

If the kmers have some homology with sequence Q then we are going to perform alignment or they will be discarded.

There will be sequences that will show similarity by chance (often happen in nucleotide sequences), false positives, but it doesn't matter because when we perform alignment, the majority of the string will be different so the alignment will get a low score.

How often we are going to perform the alignment depends on: number of possible homologs, length of identical substrings.

The parameter k is set by the user. If k is large, then we are going to avoid all identities by chance only, but we will lose many of the true homologs. If we reduce k, we will slow down the procedure, aligning many sequences that have nothing to do with the sequence we are interested in. In this case the result is a more complete output and the execution is heavier.

Balancing the value of k is not easy, the default value is fine for most cases.

The filtering step is crucial and must be done quickly. If the step is slow there is no point in losing precision. We want to trade precision for speed.

How can we identify equal substrings in a very fast way when we have a huge dataset? Using hash tables.

Hash tables

Hash tables are a way for storing variables, the data is stored in an array, each item is associated with some form of identifier s.t. we can fetch from the table the value corresponding to the key.

The advantage of a hash table is that the retrieval of a value from the table is performed in constant time.

In python to fetch the value associated with a key, we could store the variables in a dictionary, then iterate through the keys and find the corresponding one. This is a linear time procedure, the computational time depends on the number of elements.

With hash tables this is done in constant time. It doesn't depend on the size of the dict, it is immediate.

The idea is: we have a collection of strings that possibly can be many. This database can be preprocessed to store all the substrings in an hash table, the database is decomposed: all sequences are preliminarily divided in substrings of some length.

In fasta servers, all of the sequences are already stored in a hash table, so that to perform the database search, it is enough to create the hash table of the query sequence.

When the user asks for a key, the value is already there. Such that the retrieval is as fast as possible.

Say we have this sequence: “GTGCGTGTGGGGG", these are all the possible 3mers:

3-mers of T	Offset of 3-mer
GTG	0,4,6
TGC	1
GCG	2
CGT	3
TGT	5
TGG	7
GGG	8,9,10

We shift 1 for each 3mer, each one is a substring of 3 chars. Together with the Kmer we have to store its relative position in the string. A common way to store them is to use key:value pairs: the key is the Kmer and the value is the offset with respect to the string. It can be just one number or more if it appears more than once.

To query for a 3mer in python we can iterate through the keys and check each of them until we found one that is equal to the query 3mer. This is not an efficient way, not even in python (it has

a more optimised procedure), it requires linear time, if the dataset is huge it will require a lot of time.

We want something quicker than linear, we would like a constant procedure, to achieve this, hash tables come into play.

In python hash tables are the internal organisation of the dictionaries. A hash table can be imagined as an array, a vector of values, where each index is associated to a key through a function. When we want to retrieve a value associated with a key, we don't have to iterate, we just know in constant time which is the value of the list associated with that key.

How? The magic is performed by the hash function.

```
def hashString(st):
    s = 0
    for c in st:
            s = (128*s + ord(c)) % (2**120+451)
    return s**2 % (2**120+451)
```

Ord is a python method that, given a char, returns its unicode. Each char has its code, similarly to the ascii code.

A **hash function is a function that, given a string, returns a number**. What the function does is to scan for each character, get the unicode representation, perform some operation and then returns a number. There exist infinite possible hash functions.

How are these functions used? To build hash tables

Build a hash table

We start from the first pair key-value; the key, that in the case of database search is a string representing the Kmer, is processed by an hash function that returns an integer number. This number is going to be the index position at which the value will be stored.

Imagine the key is key_1, it is processed by a hash function which returns the number 5. 5 is the index position at which the value will be stored in the array.

This is done for all key-values of the dictionary. Everytime we add a value the key is processed and determines the index of the array where the value will be placed. Once all the values are stored, if we want to retrieve a value associated with a key, we have to process the key through the same hash function that was used to create the table. This function is going to tell us the position at which the value corresponding to the key is stored.

That's it, this process is as fast as the hash function is able to process the key, usually it is really fast.

Python has its own hash function, which can take as input only immutable objects, it is a pretty complex function. The hash of character "c" is 235723564236. When we create a dictionary a very large empty array is created, it is large enough to store many values, we need to store all values of the dictionary, having large numbers as indexes.

Unfortunately, we cannot guarantee that the hash function is going to give a different number for every different key. If we have different strings and the hash function returns the same number, different keys may point to the same object in memory, this is a collision. It is impossible to

completely avoid collisions, there will always be some keys for which the same number is created.

Some hash functions can be worse than others, example:

```
def h(k):
        return k%7
```

The function above can be considered a hash function, it returns the remainder of a division by seven, if we use the key 2, the value will be 2 (0*7+2=2), so we will store the key in memory at position 2. For key 3, the value will be 3 and for key 7 the value will be 1 (to retrieve the value stored at key=7, we will just put seven in the hash function and we will gate the place in memory where the value is stored). The problem is that if we have a key equal to 9, we will have again 2 as value, this is a collision, the key 2 and 9 will be stored in the same place in memory.

There is no perfect hash function, in a perfect hash function for each different input there is a different output.

How to solve collisions?

Chaining

One simple way to handle collisions is simply to let them happen and store multiple objects in the same index of the array. We are going to place the values in the same cell where it is saving them, maybe in a container (a list). When we have to fetch these values, the hash will point to the cell which will be a list of values. We will need a way to determine which of the multiple values corresponds to which key.

All values associated to diff keys for which we have collision are put together in some form of container (bucket).

Implementation

When we insert a value in the hash table we apply the hash function to the key of the value; each empty unit of the array will contain an empty list, each value is going to be appended to the list stored at a certain index of the array. Each list contains one or more elements.

If we want to retrieve a value, we have to process the correspondent key through the hash function to obtain the index of a container. This container will have one value or more, then we can scan through the container until we find the correct value.

Since we have to iterate in lists of variable length, the overall complexity for fetching values cannot be considered constant. It depends on how many collisions there are, for each collision we have to iterate. The worst case scenario would be a hash function that for any key returns the same value, we will have an element of the array that contains the container with all the values of the table. The complexity would be n, but fortunately, we never have such a simple hash function.

When we want to delete a key-value pair we need to fetch it and then remove the value from the container. The time required for all these operations depends once again on the number of collisions.

What can we use as a container?

Since all when handling collisions we have to **insert**, **fetch** and **remove** elements from the containers, the efficiency of the algorithm depends on the efficiency of these operations.

Python lists are good for some operations, s.a. appending; but they are very inefficient for others, s.a. Inserting an element at the beginning of the list (in this procedure, the element is inserted and all the others are shifted to the right).
For this reason usually chaining **containers are not just lists**, they are more complex data structures. They usually are **linked lists**, it is not just an array, it is a series of values and pointers. This is better because some operations, especially **insert operations**, are more **efficient**; this small advantage that becomes significant with large lists.

Linked lists are made of nodes, containing two fields: one is a value and the other is a pointer to where the next element of the link list is found in memory. Also the last node has two fields, the value and a pointer to None.

Why do we use linked lists? Some operations are more efficient.

With linked lists if we want to insert something we don't need to shift all the rest. We just change the pointer of the previous element to the new element and point the pointer of the new element to the next element.

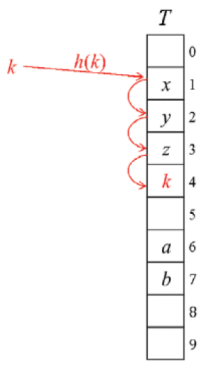
Comparison with python lists:

- Insert operation: in python standard lists the insert operation is done more or less in linear times (it depends also on where we want to add the item). In linked lists, the complexity does not depend on how long the list is. It can be considered constant time.
- Remove operation: in a standard python list to remove an item all other items have to be shifted forward of one position (linear) while in a link list we just remove a node and modify a pointer (constant).
- Searching operation: searching has linear complexity both in python lists and in linked lists. In both cases, we have to traverse all of the elements and search for the one we are interested in.
- Also the space used to store the data is comparable between lists and linked lists.

There are more complex and efficient ways for handling collisions (this doesn't mean that they are solved, collisions are always present).

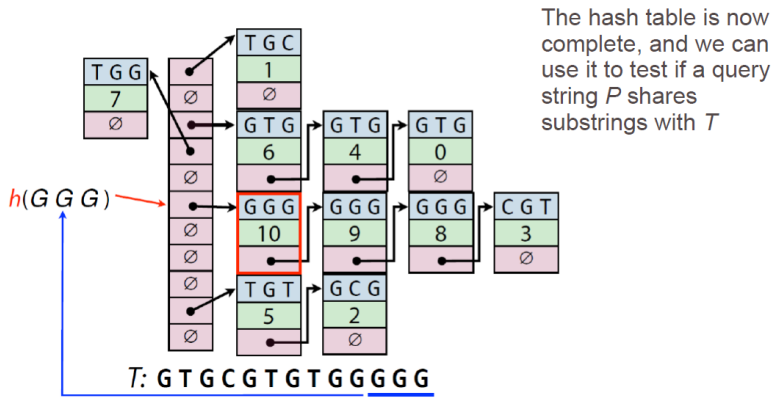
Open addressing

Python dictionaries use open addressing. The idea is that if the hashing function returns an equal integer number for two different keys and the slot is occupied, open addressing starts looking within the hash table for an empty slot, then places the value there. Now an additional position of the hash table is occupied, if the hash function will return a value corresponding to the slot occupied by the offset key, the second element will be shifted to another slot (even though there is no actual key that returns the same value in the hash function, it is just due to the offset produced by another collision).
The problem of this approach is that something has to store the offset and it is more complex than chaining.

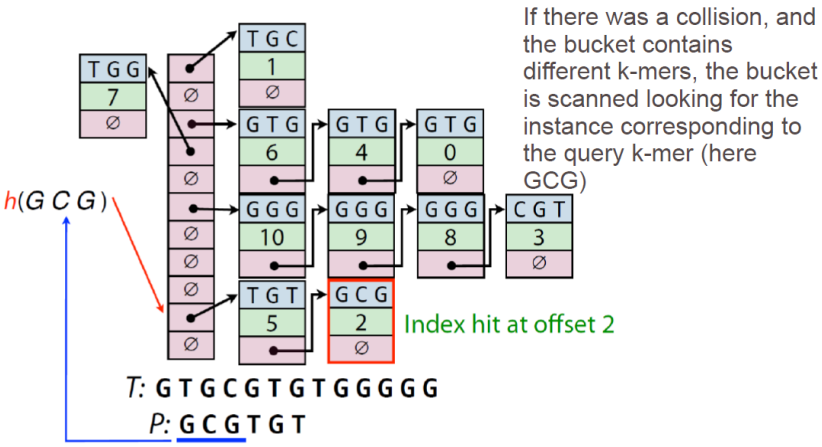


Chaining and open addressing can be more or less efficient depending on the data.

Now given a collection of sequences we store them in a hash table:



The linked lists also store the number of the position of the substring in the string. Also if a single hash table contains multiple strings the linked list can contain the identifier of the string.



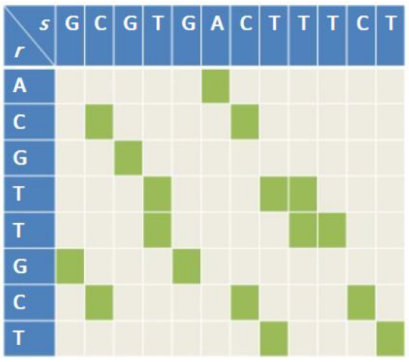
Filtering

All the sequences in the database are hashed , and the hash tables for different values of k are stored in the machine running FASTA. When the user inputs a query sequence q and selects the desired value of k , q is split into overlapping k mers , and each one of them is hashed and used to fetch from the corresponding hash table all the database sequences containing the k mer.

Additional steps before deciding whether performing a true alignment: depending on the length of the substring I require to be identical, many of these strings may not be related. Aligning them would be a waste of time.

We want to do the alignment only when there is strong evidence that two sequences are related in some way. For example, the user could impose that at least m k mers of q must be found in a sequence i from the database in order to let sequence i proceed to the next step of the algorithm.

Example with dimers:



Length-2 subsequences of *s*	Positions
AC	6
CG	2
CT	7, 11
GA	5
GC	1
GT	3
TC	10
TG	4
TT	8, 9

1. **Identify all the common kmers** between the query string and a sequence of the database: the diagonal green squares are the identical substrings of at least 2 characters. There are many positions in which the two sequences have dimers in common.
2. **Merge consecutive matches**: if we have a dimer followed by another, they can become a trimer. On the graph these multiple k-mer matches are on the same diagonal and they are consecutive (a continuous diagonal longer than 2 cells).
3. **Extend the matches**: we want to join matches which are not exactly consecutive. Merge dimers that are near by accepting some **substitutions**. On the graph these extensions are made between two segments on the same diagonal, we join two diagonals with a transparent cell (the mismatched base).
   If we are working with protein sequence, we can use substitution scores.
4. **Joining**: we can try to see if some of the segments can be joined with a gap. On the graph this results in joining different diagonals that are close to each other.
   In this case we can use gap penalties to evaluate the pseudo similarity.

All these operations (merging, extension and joining) are done without aligning the two sequences, this is still **filtering!!!**

We can stop the procedure at any point. If we have no long diagonal we could stop, or if we cannot connect any diagonal we could stop. The single k-mer identity was due to chance.

Parameters that can be decided by the user:

- **Word length** (substring size, K-tup): part of the initial filtering step, when we require that the sequences in the database have to share substrings of length k with our sequence. The longer the subsequence, the smaller the number of sequences that we carry forward; it is less likely to carry forward sequences that have identity by chance, but there is the case that we discard sequences that may be related. If it is too short we will retrieve all true homologs but also many false positives.
  Raising the k-tup leads to: lower background noise, shorter running time, higher risk of missing distant homologs.
- Number of best scoring matches to keep (default is 10): we select the best diagonals of a preliminary alignment matrix and we work only with them in the next steps. The score of these 10 best matches is called init1.

- Initn: threshold for the init1 score. The partial alignments that are above these threshold are expanded through extension and joining.
- Band width for the final alignment

Alignment

If we are satisfied with the preliminary similarities that we found we can run the alignment algorithm. The fasta algorithm uses an exact dynamic programming alignment algorithm.

To save even more time, sometimes the alignment algorithm is not computed completely, some cells are skipped. Furthermore, all of the cells connected during filtering are already part of the solution, they do not have to be recomputed. Finally we can exclude some areas of the matrix, we could keep just the cells belonging to a limited band around the biggest diagonals.

Conclusions

Fasta is not the most used algorithm, Blast works better. Also, when using Fasta, running the algorithm is not enough, we have to prepare the database (it is not just a collection of sequences, a hash table has to be created). The database has to be prepared offline and the hash tables will be saved in memory, since they can be very large we need a lot of memory.

Official Fasta site

Online we have the database made available by the company, they are the standard and most important databases (Uniprot and others).

When we select a provided database we can see that there are different versions: Clusters 100%, 90%, 50%. These are subsets of uniprot in which redundancy is removed to some degree.

Redundancy: all biological datasets may contain entries that are very similar if not identical.

Also these datasets are built by different groups of scientists, and the same sequence could be present multiple times, coming from multiple research groups. If there are two or more entries identical or very similar, each one of them will be processed during the query without giving any additional information. We do not have to check both the gorilla and human genome, they are basically the same.

We can reduce redundancy by creating clusters. We compare each sequence of the database against the other and we group all of the sequences that show a certain degree of similarity (given by the percentage next to the name of the database).
Each cluster is going to contain sequences from the database that are similar. From each group once we have performed clustering we take a representative member, based on some specific criteria, called the cluster centroid.

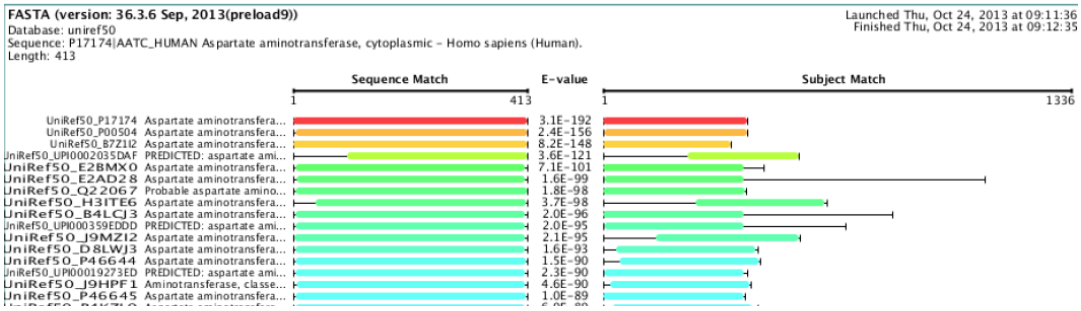
Once we select a database we upload the sequence and run the algorithm.

Results

The different colour is related to the e-value: statistical evaluation of the significance of the match, it means how frequently or likely is to get an alignment of the same score from two unrelated sequences (not a probability). It answers the question: i have an alignment with a

score, if we align one of the two sequences with a random sequence, how likely is to get that score?

Red is highly significant, while blue is poorly significant. The smaller the e-value is, the more significant the result.



We can also get a summary table with the result and the multiple alignments of all the matching sequences.

Why is it not popular anymore? Limit: it requires exact string matching, perfect identity between substrings of two sequences. We are using the hypothesis that sequences have identical substrings, no matter how far apart they are.

Blast

Blast performs better because the initial criteria is more relaxed, it requires just similarity. Blast considers similar substrings, not identical. This is not easy to implement.

These algorithms have to balance between informative output and speed. If we have a long sequence to query on a large database we can't search by brute force the similar substrings.

We described the data structure where all sequences are stored for the Fasta algorithm: through hash tables, Fasta allows querying an identical substring in constant time.

If we want to move to similarities, then hash tables cannot be used anymore, we need to evaluate these similarities, we need something similar to an alignment or a distance.

How to solve this?

The idea is to perform a simple lookup in a hash table, searching for identity, but instead of all possible substrings of the query sequence, blast generates variants of the query sequence, to simulate how it may have evolved during evolution.
Once we have generated them we could put them in a hash table and query the database for each of them.

Starting from a query sequence , BLAST Basic Local Alignment Search Tool) employs a seed and extend search strategy , similar in some parts to that followed by the FASTA algorithm. The basic idea still is to identify all target sequences sharing some substring of length W with the query, and carry on in the procedure only these , while discarding all the others. The main difference between BLAST and FASTA is that the initial matching does not require identical substrings, but also similar substrings are accepted, their similarity defined through a substitution matrix.

Overview:

The BLAST algorithm can be divided into the following steps:

1. Split query into overlapping words of length *W* (the W-mers)
2. Find a “neighborhood“ of similar words for each word
3. Lookup each word in the neighborhood in a hash table to find the location in the database sequences where each word occurs. Call these the *seeds*, and let *S* be the collection of seeds.
4. Extend the seeds in *S* until the score of the alignment drops off below some threshold *X*
5. Report matches with overall highest scores

Neighbourhood

Suppose this is the query sequence:

Query sequence: LSHERGAAVEKS

Word 1	LSH
Word 2	SHE
Word 3	HER
Word 4	ERG
	.
	.
	.

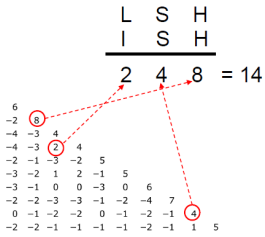
We can divide it in overlapping substrings of length W called words or W-mers (these could be used in fasta).

Then, for each word, additional w-mers are generated, they are called neighbors or variants of the word.

The variants are created by applying substitutions which are reasonable, which are evolutionary plausible. They are based on how that sequence could be found in another species, in an homolog.

Then the process can continue, generating variants of the neighbours and so on.

The amino acid pairs with lower scores in the substitution matrices are the ones that vary more often. We have the measurement of how aminoacids evolve, it is the substitution matrix. We can score each variant and keep only the ones above a certain threshold.



The neighbours represent how we expect to see the sequence in an homolog. Even though they might not exist.

If conservation score is low it means that frequently it changes, so it makes sense that the neighbour having this residue conserved is unlikely. All neighbours must be above a threshold.

With the neighbours we are going to look for identical substrings in the matrix.

After the matches we compute the alignments as in fasta. The similarity of the substring itself is already kept into account by the neighbours

Alignment

Blast is faster than Fasta because it never performs a true alignment. What blast does is to extend the matches resulting from the hash research of the neighbours.

The initial match is where the query and the database sequence contain a similar substring. This is called seed.

Blast just looks at residues before and after the seed. It evaluates the residues of the database at the borders of the seed with the residues in the query string. If they have a good substitution or conservation score, then they are added to the initial match. The score of the aligned segment is computed again and when the score starts to decrease the algorithm stops.

The extended seed is then called High-scoring Segment Pair (HSP).

In this way there cannot be gaps, we are searching for gapless alignments. More recent versions of Blast allow searching for alignment that have gaps.

Also this type of alignment are local alignments. Even in the most recent versions of Blast the alignments are just an extension of a seed.

All matches can be also evaluated statistically, we can compute the probability of finding similar alignments by chance only.

Parameters

- Word size: : Large W would result in fewer spurious hits, thus making it faster, at the cost of having a large neighborhood of slightly different query sequences, a large hash table, and too few hits. On the other hand, if W is too small, we may get too many hits which pushes runtime costs to the seed extension/alignment step
- Word similarity threshold: If T is higher, the algorithm will be faster, but you may miss sequences that are more evolutionarily distant. If comparing two related species, you can probably set a higher T since you expect to find more matches between sequences that are quite similar.
- Minimum HSP substitution cut-off score X: Its influence is quite similar to T in that both will control the sensitivity of the algorithm.

Given values of W, T and X, the algorithm then looks for a segment pair with a score of at least X that contains at least one word pair of length W with score at least T

While W and T affect the total number of hits we get, and hence affect the runtime of the algorithm dramatically, setting a really stringent X despite less stringent W and T, will result in runtime costs from trying unnecessary sequences that would not meet the stringency of X. So, it is important to match the stringency of X with that of W and T.

After the original BLAST algorithm , an update called BLAST2 was introduced, which included a modification of the seed extension in order to consider also gaps. The produced alignments are therefore local gapped alignments.
BLAST2 is more sensitive and faster.

In blastn (the BLAST version for nucleotide sequences), since no substitution matrix is used, we cannot generate words that are similar to the query W mers. Identification of seeds is

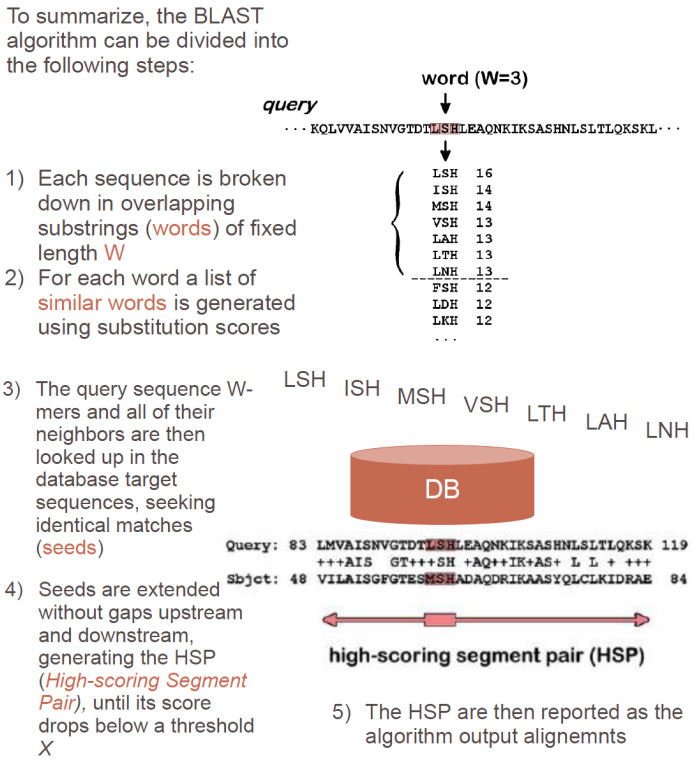
therefore only based on sequence identity, similarly to what FASTA does. The scoring scheme used for the seed extension employs only a match score and a mismatch penalty

All these procedures work only for proteins. For nucleic acid there is no substitution matrix. Then we cannot generate the neighbours. For proteins we use this strategy that allows us to search for similarity and not identity. Blastn is simpler, it just looks for identity. After the match we extend.

In order to allow longer extension, the scoring scheme is different, tolerating mismatches, if following characters are identical the alignment proceeds.

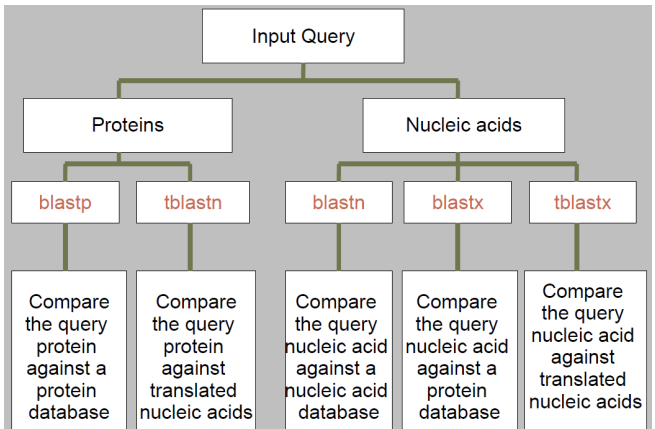
The original idea of blast, which was looking for similarity and not identity, cannot be done for nucleic acids. Fasta can be preferred for nucleotides. Even if it is slower at least at the end we have a true alignment.

Conclusions



Blast is used all the time, modern bioinformatics and biology is based on this.

There exist many different implementations of blast. Some of them can be used remotely through web browsers. Depending on which database and on how the sequences are treated.



Even if protein databases are growing overtime containing more and more, determining the sequence of a protein is still more difficult than nucleic acids. The number of known protein sequences is much lower with respect to nucleic acids. If we start from a protein with a new sequence, there might be no homologs.
But maybe there are genomes sequenced that encode for the same protein. If we run the blast search on proteins we might find little information. We may compare protein against nucleic acids. We have to find the right reading frame but then we could find it, we have to examine 6 sequences in total.

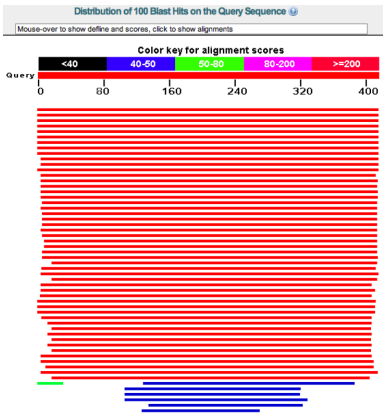
If we start from nucleic acids we can compare it against nucleic acid (blasn) or we can compare the sequence against a protein database by translating the sequence. In general terms proteins are more conserved, two genes with many substitutions may still give the same proteins.

BLASTN: compares a DNA query sequence to a DNA sequence database;	$q_{DNA} \leftrightarrow s_{DNA}$
BLASTP: compares a protein query sequence to a protein sequence database;	$q_{prot} \leftrightarrow s_{prot}$
TBLASTN: compares a protein query sequence to a DNA sequence database (6 frames translation);	$q_{prot} \leftrightarrow s_{t_1(DNA)}\ s_{t_1^c(DNA)}\ s_{t_2(DNA)}\ s_{t_2^c(DNA)}\ s_{t_3(DNA)}\ s_{t_3^c(DNA)}$
BLASTX: compares a DNA query sequence (6 frames translation) to a protein sequence database.	$q_{t_1(DNA)}\ q_{t_1^c(DNA)}\ q_{t_2(DNA)}\ q_{t_2^c(DNA)}\ q_{t_3(DNA)}\ q_{t_3^c(DNA)} \leftrightarrow s_{prot}$
TBLASTX: compares a DNA query sequence (6 frames translation) to a DNA sequence database (6 frames translation).	$q_{t_1(DNA)}\ q_{t_1^c(DNA)}\ q_{t_2(DNA)}\ q_{t_2^c(DNA)}\ q_{t_3(DNA)}\ q_{t_3^c(DNA)} \leftrightarrow s_{t_1(DNA)}\ s_{t_1^c(DNA)}\ s_{t_2(DNA)}\ s_{t_2^c(DNA)}\ s_{t_3(DNA)}\ s_{t_3^c(DNA)}$

On the Blast site, we have to enter the query sequence in fasta forma, the reference database, and the algorithm to use, then we click on BLAST (we can also change parameters).

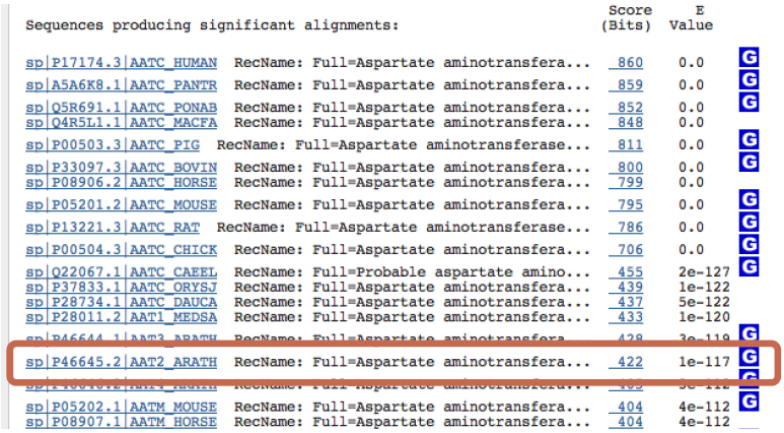
Output

Visual representation of all the matches that we found:



From this representation we get the length of the alignments. The color coding corresponds to the score of the alignment: red = good score, black = bad score.

Then we have a list of matches with the score of alignment, the statistical significance and the relative gene associated to the sequence. The gene ID can be searched on NCBI.



Sequences producing significant alignments:		Score (Bits)	E Value
sp|P17174.3|AATC_HUMAN	RecName: Full=Aspartate aminotransfera...	860	0.0
sp|A5A6K8.1|AATC_PANTR	RecName: Full=Aspartate aminotransfera...	859	0.0
sp|Q5R691.1|AATC_PONAB	RecName: Full=Aspartate aminotransfera...	852	0.0
sp|Q4R5L1.1|AATC_MACFA	RecName: Full=Aspartate aminotransfera...	848	0.0
sp|P00503.3|AATC_PIG	RecName: Full=Aspartate aminotransferase...	811	0.0
sp|P33097.3|AATC_BOVIN	RecName: Full=Aspartate aminotransfera...	800	0.0
sp|P08906.2|AATC_HORSE	RecName: Full=Aspartate aminotransfera...	799	0.0
sp|P05201.2|AATC_MOUSE	RecName: Full=Aspartate aminotransfera...	795	0.0
sp|P13221.3|AATC_RAT	RecName: Full=Aspartate aminotransferase...	786	0.0
sp|P00504.3|AATC_CHICK	RecName: Full=Aspartate aminotransfera...	706	0.0
sp|Q22067.1|AATC_CAEEL	RecName: Full=Probable aspartate amino...	455	2e-127
sp|P37833.1|AATC_ORYSJ	RecName: Full=Aspartate aminotransfera...	439	1e-122
sp|P28734.1|AATC_DAUCA	RecName: Full=Aspartate aminotransfera...	437	5e-122
sp|P28011.2|AAT1_MEDSA	RecName: Full=Aspartate aminotransfera...	433	1e-120
sp|P46645.2|AAT2_ARATH	RecName: Full=Aspartate aminotransfera...	422	1e-117
sp|P05202.1|AATM_MOUSE	RecName: Full=Aspartate aminotransfera...	404	4e-112
sp|P08907.1|AATM_HORSE	RecName: Full=Aspartate aminotransfera...	404	4e-112

Again we can get the multiple alignment of the sequences that showed a similarity.

BLAST vs FASTA

BLAST employs a substitution matrix from the beginning to generate sequence variants, while FASTA only seeks sequence identity.
A consequence is that in general, BLAST works better for proteins, while FASTA works a bit better for nucleic acids.
BLAST is better at finding ungapped alignments, FASTA is better at finding more divergent sequences with lots of indels.
BLAST is much faster

Statistical evaluation

In the output of Fasta and Blast output we can see the E-value, it is a sort of statistical evaluation of each match. It is not a probability. The range of the E-value is from 0 to above 1, the smaller the E-value is, the more significant the alignment is.

It basically answers this question: having an alignment with this score, is it a true indication of some relationship between the two sequences? Or this score can be obtained by chance?

We could align our sequence with another sequence with no evolutionary relationship and see how likely it is to get a similar score.

Remember that we are searching for homologs in a database. With this statistical evaluation we are classifying each sequence of the database as homolog or not. It is a sort of inference.

It is a binary prediction, that can be positive or negative:

- Positive: see an alignment, and we consider it as an homolog
- Negative: the two sequences are not homologs

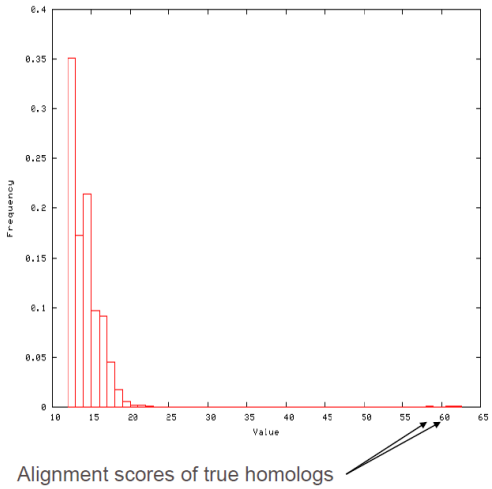
We might be right or wrong, there is no way of knowing it.

There are 4 possible outcomes:

- Positive prediction
    - Correct: true positive
    - Wrong: false positive = we see in output a sequence that in reality is not a homolog.
- Negative prediction
    - Correct: true negative
    - Wrong: false negative = a homolog sequence is discarded and it doesn't show up in the output.

During these predictions we have to decide whether to minimise an error or the others. If we minimise the false positives we will have false negatives.

False negagives might be not a huge problem. This is because during database search we just need to find some homologs in the database that can give us some information (maybe also functional information). On the other hand, many false positives means that we will have in output many non homologous sequences, that could mislead our phylogenetic and functional assessment.



This is a normal distribution of alignment scores, the majority of random alignments have low scores and very few have super high scores, they are the homologs!

We would like to know if they are truly homologs, trying to reduce at minimum the number of positive errors.

Is the obtained score significantly higher than a random one for two equally long sequences using the same scoring matrix?

We will have to perform statistical testing. Hypothesis testing: we test the probability that an hypothesis is true.

- Null hypothesis: the score that we see is due to randomness.
- Alternative hypothesis: the score is due to some kind of relationship.

Then, depending on the data and the hypothesis, one can compute the probability of the H0, which is called the p-value. If the p-value is very small, one can reject the H0 and implicitly accept the Ha.

Once we have set up the null hypothesis and the alternative we have to find the distribution of the test statistic. We want to work on the distribution of the score alignments under the null hypothesis: what is the score that we would obtain by comparing the input sequence against a random sequence.

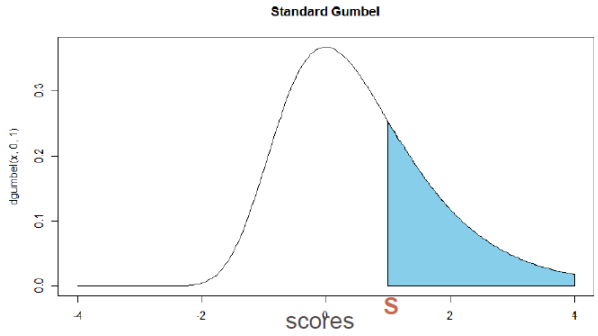
From this distribution, we would like to estimate how surprising a given alignment score is, if the two compared sequences have no evolutionary relationship.

We have to set a threshold of significance and then compute the probability. If the probability is lower than a threshold we may discard the alignment. We have to compute the p value: the probability of obtaining that alignment or a less likely one.

Which is the score distribution that we can obtain between unrelated sequences?

If we compare one input sequence against a large collection of sequences the alignments have a gaussian distribution of scores. On this distribution, we compute the probability for which the score that we have in output, as a positive result, would have been generated by this distribution. If the p value is low, we can consider the alignment as non-random.

In blast we get not a normal distribution, we get a Gumbel distribution, it is not symmetric:



We don't actually have this distribution, we would have to compare the sequence with the database each time, this is against idea of speeding up the process. We want to know beforehand which is the background distribution.

We can solve this by assuming that in the BLAST output most matches come from unrelated sequences, and the true homologs are few. We consider all the results of blast as the distribution of H0 and we compare our result with it.

We consider the average score of the BLAST output as the average score of the background distribution. How far is our alignment score from the average score? If the score is further from the average the better. We also have to take into account the variance of the distribution, how large it is. The smaller variance the better, there are less alignments by chance. We have to normalise the distance from the mean by the variance.

$$\text{Z-score} = \frac{\text{Score S} - \text{average score}}{\text{standard deviation}}$$

In Z-scores is very large, the observation is very far from the average,

This approach relies on the assumption that in BLAST output we have only garbage. This may create other problems.

So they computed background distribution for large sets of input sequences against large databases, trying to understand if it was possible to estimate the background distribution without running the comparison. They found a dependence on scoring schemes.

Intuitive: if we are running a blast for a protein sequence. We have to choose a substitution matrix. Then we will have the highest score of alignment. If we chose a pam substitution matrix

with low degree, in which substitutions are penalised a lot, we should see that many sequences in the database are going to be discarded and only sequences in the database. If we use pam matrices with high degree, substitutions will be less penalised, they tend to lower scores.

For more relaxed parameters (divergent proteins) we expect a wider distribution. On the other hand stringent parameters produce a narrower distribution. These dependencies between scoring scheme and distribution of scores were converted in a way to compute the parameters of the background distribution: K and λ, scale and location, which describe the Gumbel distribution.

They are computed on the basis of the substitution matrix used and on the gap penalty. This means that before blasting, once we have selected the matrix and the gap, from these parameters we can compute K and λ. The distribution can be described just by these two parameters.

This approximation was quite good and it was one of the reasons for the popularity of blast.

From these two parameters, a p-value can be computed:

$$P(x \geq S) \approx Ke^{-\lambda S}$$

If probability is low we can reject the null hypothesis and say that the alignment is due to relation. Otherwise we accept H0 and we consider the alignment as due to chance.

But as we said at the beginning of the section, Instead of p-value, in the output of FASTA and BLAST, is reported the E-value.
The E-value is computed from the p-value, it takes into account the size of the database. The number of sequences in the database and how long they are. This is because sometimes we compare the query sequence with a very large database.

If we set a significance threshold at 0.01. It means that there is a sort of risk in considering false positive alignments 1% of the time. It means that 99% of the time we will make the right choice.

1% of errors in a database containing millions of sequences may contain a lot of false positives. The larger is the database, the larger is going to be the number of errors of this kind that we will make.

This means that p value, computed independently from the size of the database may not be enough. Usually p value is scaled by size of database.

If we have m sequences in the database and a query sequence of length n, we can multiply the p-value for m*n to scale it accordingly to the database. We will get a bigger p-value, for this reason it is not a probability. The new value is still considered with respect to the p-value threshold; the p-value is so small that it remains small even after the multiplication.

$$E(x \geq S) = Kmne^{-\lambda S}$$

The minimum of the E-value is 0 but there is no upper value. Usually E-values that are considered significant are smaller than 10^{-5}.