CMSC701: Computational Genomics

Mihai Pop

Contributions from:

Chris M. Hill	Raul Guerra	Sam Huang
Mark Daly	Derrick Wood	Wikum Dinalankara
Geet Duggal	Ted Gibbons	Ferhan Ture
Koyel Mukherjee	Emre Sefer	
Hyongtae Cho	Stephen Altschul	

Last modified on December 2, 2013

Table of Contents

1 Introduction

This course covers a number of topics related to biological string alignment. As you will see below, many biological objects can be seen as strings of letters, and string algorithms are one of the most important components of modern biological research. Many of the ideas described here apply to any strings (web pages, books, computer programs, etc.), thus this class should be useful even if you are not interested in biology.

It shouldn't be a surprise to find out that string processing algorithms have been around pretty much since computers were invented. Many very smart scientists have tackled a vast number of problems related to the alignment and analysis of strings, and this course is just meant to give you a teaser of what you will encounter if you decide to delve deeper into the wonderful world of strings. This course will also provide you with the basics of stringology, allowing you to understand the main concepts related to string analysis.

2 Quick introduction to biological sequences

Many biological objects can be interpreted as strings. DNA, for example, is usually encoded as a string built from a 4-letter alphabet {A,C,T,G}, corresponding to the four nucleotides forming the DNA double-helix. Proteins can also be seen as strings build from a 20 letter alphabet, corresponding to the 20 different amino-acids commonly used by living organisms. This course will primarily focus on concepts related to the analysis of such biological strings. First, we'll introduce the context within which these strings exist in life. This quick introduction to molecular biology is neither comprehensive, nor fully accurate, rather it is just meant to quickly introduce you to some of the key terms in the field. You should spend some time searching some of the terms on the internet, or reading various introductory textbooks (the Cartoon Guide to Genetics by Larry Gonick – ISBN 978-0062730992 – is a surprisingly good first read).

2.1 The central dogma of molecular biology

The central dogma of molecular biology attempts to explain the flow of information in self-replicating biological systems. In this model, genetic information is encoded in DNA. A molecular process, called *transcription*, copies the genetic information into (nearly) identical RNA molecules, which are then converted into proteins through a process called *translation*. Proteins are the biological objects that perform all the functions of the cell, including the actual transcription and translation processes, as well as the DNA replication function. This general model can broadly accurately explain how living organisms operate, though in recent years scientists have uncovered substantially more complex features of the mechanisms of life, such as the fact that genetic information can be transmitted not just through DNA alone but also through 'annotations' along the DNA (e.g., methylation status) or even the way in which the DNA is physically organized within a cell.

The terms transcription and translation are well ingrained in the vocabulary of biologists and are taught as part of Biology 101, thus, it may come as a surprise that these terms were actually proposed by John von Neumann, a pioneer of computer science. Von Neumann proposed this two stage process, comprising a blind replication step followed by the interpretation/execution of the instructions, as a way to achieve self-replicating automata. It is remarkable that his intellectual predictions turned out to so closely model the actual operating principles of life.

2.2 DNA and self-replication

The DNA has a particular structure that is essential for its role as a 'magnetic tape' storing our genetic information. Specifically, the DNA is a polymer, a molecule composed by the chaining of four different small molecules called *nucleotides* or *bases*. These four nucleotides (adenine – A, guanine – G, cytosine – C, thymine – T) can be chained in an arbitrary order along a sugar backbone, allowing the encoding of essentially arbitrary information. The particular sequence is commonly represented by a string comprised of the first letters of the nucleotides (A, G, C, T). Furthermore, most commonly, the chromosomes of organisms consist of two such chains (or strands) that are twisted around each other in a double-helix. The two strands are held together by chemical bonds between 'complementary' nucleotides. Specifically, the nucleotides fall into two classes: purines (A, G) and pyrimidines (C, T) based on their chemical composition (purines are larger and contain two rings while pyrimidines only contain one ring). Their chemical structure allows the specific pairing of the purine adenine with the pyrimidine thymine (A-T) and of the purine guanine with the pyrimidine cytosine (G-C), and these bonds hold together the double-helix structure.

A by-product of this paired structure of DNA is redundancy in the genetic information encoded in an organism's chromosomes. Specifically, the sequence of one strand can be determined from the sequence of the other by simply reconstructing the complementary structure of its bases. This property is key in the ability to

copy the genetic material of one cell and distribute these copies to its daughter cells during cell division. Specifically, the double helix is unwound, and a chemical reaction synthesizes a sister strand for each of the two unwound strands, leading to the creation of two double helices, each an identical copy of the original.

It is important to remember that this process of replication always occurs in the same direction along the DNA strand – a direction termed 5' to 3' due to the chemical nomenclature for the ends of the DNA polymer (for more details check out a biochemistry manual or online resources). By convention, the string of letters representing a DNA molecule is always written 5' to 3'. Converting the sequence of one strand to that of the other, thus requires one to both change all letters to their complement, and also to inverse their order, a procedure called *reverse complementation*. For example, see the two strands represented below and the correct representation of the DNA composition of the second strand.

strand 1 5' – ACAGTGCATGCCTGA – 3'
strand 2 3' – TGTCACGTACGGACT – 5'

string for strand 2 - 5' TCAGGCATGCACTGT 3'

2.3 RNA – just another type of DNA

Ok, this is just a computer scientist's view and biologists will strongly disagree with this statement. RNA is really a very similar molecule to DNA, where the thymine nucleotide is changed to uracil (U). Unsurprisingly U and A form a complementary bond, matching the situation found in DNA. Where RNA differs from DNA is in it's structure and function within the cell. Above we described a situation where RNA is simply a 'crib sheet' used to copy part of the DNA as a step towards generating a protein. In this setting RNA lives as a single stranded molecule. An important feature of single stranded RNA is that it can fold onto itself and form base pairings along the same strand, as seen below.

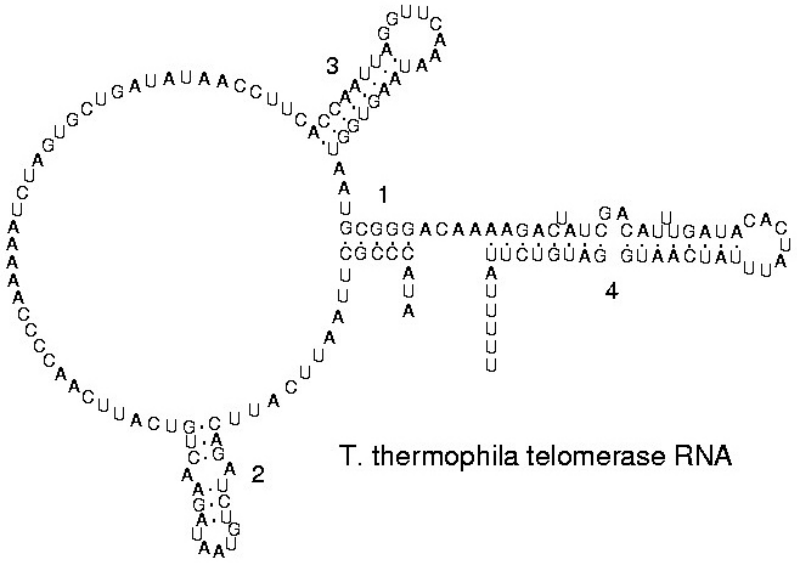


image from wikipedia Commons: https://commons.wikimedia.org/wiki/File:Ciliate_telomerase_RNA.JPG

This folded structure makes the RNA molecule rigid and allows it to be used as a structural element in other cellular objects. A good example is the ribosome – a complex of proteins and structural RNA that is the molecule which synthesizes proteins from the template provided by a gene (more on that below). The folded RNA structure (also called *secondary structure*) also plays a role in gene expression, for example by stopping certain biochemical processes such as transcription or translation (sort of like a knot on a rope preventing it from going through a hole).

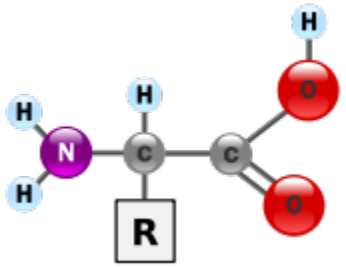
It is important to note that such self folding is not only restricted to RNA and single stranded DNA molecules

can also self-fold. Furthermore, as we'll discuss in a bit more detail below, most biological strings (including double-stranded DNA) can and do fold up in complex structures that ultimately affect their function in the cell.

Also note that RNA can also exist in a double-stranded form. Such a form is often used by viruses to encode their chromosomes, thus many organisms have developed mechanisms to destroy double-stranded RNA as a defense against invading viruses. Similar mechanisms can also be used on purpose to destroy RNA produced within the cell as a way to regulate the amount of RNA and/or protein produced in the cell. Such processes are part of the fascinating field of gene regulation.

2.3.1 Proteins – the sequence is just the beginning

Proteins are the cellular molecules that 'do stuff' inside the cell – they are generally responsible for the machinery of life. Proteins are, like DNA, polymer chains composed of smaller molecules called *amino acids*. Amino acids have a same *backbone* (with the generic chemical formula $H_2NHCROOH$) which is decorated with a side-chain (the R in the earlier formula) which determines the type of amino acid (see below)



From wikipedia

The amino acids are chained together through chemical bonds between the extreme nitrogen and carbon atoms. There are 20 amino acids generally used in most organisms, thus, proteins can be viewed as strings of letters over a 20-letter alphabet.

Viewing proteins as strings of amino acids is, however, overly simplistic as these molecules owe their functionality to their three-dimensional folded structure (see below). The task of inferring this three-dimensional shape (also called *tertiary structure*) is the subject of the subfield of *protein folding*, and is one of the big unsolved problems in bioinformatics.

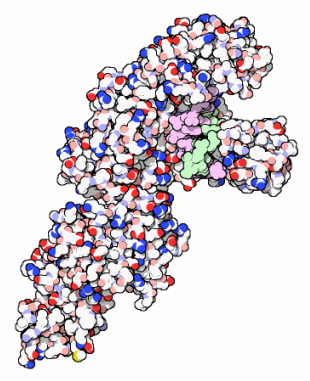


Figure 1: Structure of the DNA polymerase, the enzyme that synthesizes a strand of DNA using another one as a template. From: www.rcsb.org

2.4 Translation – how RNA becomes protein

As mentioned above, the central dogma of life posits that the information contained in DNA (strings over an alphabet of 4 letters) gets copied into RNA (also strings over an alphabet of 4 letters) and then converted into

proteins (strings over an alphabet of 20 letters). For this translation to be possible, each amino acid must be encoded by a pattern of three or more nucleotides ($4^2 = 16 < 20$, $4^3 = 64 > 20$). Biological systems use a *genetic code* of exactly three letters. Note that this code is redundant (multiple codes yield to a same amino acid, see below)

Standard genetic code

1st base	2nd base U			2nd base C			2nd base A			2nd base G			3rd base
U	UUU	(Phe/F) Phenylalanine	UCU	(Ser/S) Serine	UAU	(Tyr/Y) Tyrosine	UGU	(Cys/C) Cysteine	U
	UUC		UCC		UAC		UGC		C
	UUA		UCA		UAA	Stop (*Ochre*)	UGA	Stop (*Opal*)	A
	UUG		UCG		UAG	Stop (*Amber*)	UGG	(Trp/W) Tryptophan	G
C	CUU	(Leu/L) Leucine	CCU	(Pro/P) Proline	CAU	(His/H) Histidine	CGU	(Arg/R) Arginine	U
	CUC		CCC		CAC		CGC		C
	CUA		CCA		CAA	(Gln/Q) Glutamine	CGA		A
	CUG		CCG		CAG		CGG		G
A	AUU	(Ile/I) Isoleucine	ACU	(Thr/T) Threonine	AAU	(Asn/N) Asparagine	AGU	(Ser/S) Serine	U
	AUC		ACC		AAC		AGC		C
	AUA		ACA		AAA	(Lys/K) Lysine	AGA	(Arg/R) Arginine	A
	AUG	(Met/M) Methionine	ACG		AAG		AGG		G
G	GUU	(Val/V) Valine	GCU	(Ala/A) Alanine	GAU	(Asp/D) Aspartic acid	GGU	(Gly/G) Glycine	U
	GUC		GCC		GAC		GGC		C
	GUA		GCA		GAA	(Glu/E) Glutamic acid	GGA		A
	GUG		GCG		GAG		GGG		G

(from Wikipedia)

Within a chromosome only specific segments become proteins – these are called *genes* and are delimited by special signals in the DNA sequence. Simplistically, genes start with a *start codon,* usually AUG (or ATG in DNA) which also encodes the amino acid methionine. Genes end with one of three different *stop codons* (UAA, UAG, UGA). It is important to note that the start and stop codons must be *in frame*, i.e., the length of the string of DNA separating them must be a multiple of 3, as the cellular machinery (a protein – RNA complex called the ribosome) reads the RNA in groups of 3.

In higher organisms the genes are often broken up in the chromosome into a collection of *exons* which are separated by *introns*. During the transcription into an RNA molecule, the exons are *spliced* together and the introns are discarded, and the translation into protein proceeds from the spliced *transcript*. It is possible that the same set of exons and introns can produce different transcripts, and different proteins, through a process called *alternative splicing*, process which splices together different combinations of the starting exons and introns. Through this process the fairly small number of genes found in the human genome (~26,000) yields the tremendous complexity embodied in our organism (many simpler organisms have more genes than we do).

2.5 DNA sequencing – how we 'read' DNA

The ability to read the DNA sequence of an organism's chromosomes has revolutionized modern biological research, and has inspired much of the research described in this course. At the same time, modern DNA sequencing technologies have emerged from a meaningful and deep collaboration between biologists, biochemists, engineers, and computer scientists. Future transformative developments in bioinformatics and other fields will only be achievable through similarly multi- and inter-disciplinary interactions. In other words: learn to understand and work with others!

Existing DNA sequencing technologies take advantage of the natural machinery that replicates DNA, which they modify so that they can 'spy' on the DNA as it is being synthesized. For example, *Sanger sequencing*, a technology that dominated the field for over 30 years (and which yielded Frederick Sanger, its inventor, a Nobel prize), uses modified nucleotides that block the DNA replication process. By carefully tuning the ratio of normal and modified nucleotides the biochemical reaction that replicates DNA ends up generating a population of DNA sequences of different lengths (imagine each of them stopped at a random location in the replication process). These sequences are then sorted by length within a gel and the DNA sequence is being reconstructed by reading the terminating base (which is usually tagged with a colored probe). This approach is a *first generation* technology, and is fairly low throughput – the most advanced instruments can only read about 384 sequences at the same time.

A new breed of sequencing technologies, the so called *second generation* technologies, have emerged in the early 2000s and use a slightly different approach to read DNA. Generally, millions of single stranded DNA molecules are immobilized on a surface, then an imaging device 'spies' on the DNA replication process as it proceeds simultaneously along all these molecules. The technologies differ primarily through the way in which the incorporation of new nucleotides is detected, usually through the emission of light. This approach achieves much higher throughput (tens to hundreds of millions of sequences are being read simultaneously) but read shorter segments than the Sanger technology (hundreds as opposed to thousands of letters).

Recently, *third generation* technologies have emerged that are able to sequence longer DNA segments (up to tens of thousands of letters) as well as to avoid the need for DNA amplification technologies. The latter introduce noise in the data, and are essential in the first and second generation technologies in order to ensure sufficient signal is generated during the sequencing process.

Here I will not go into more details about the specific technologies available today but encourage you to search online the following terms: Sanger sequencing, Roche/454, Illumina/Solexa, ABI Solid, Ion Torrent, Pacific Biosciences, nanopore sequencing.

2.6 Representing sequences computationally

There are two major formats commonly used to represent biological sequences. The oldest and widest used is the *FASTA format*. It is an entirely ASCII text format that follows the pattern:

```
>seq1 some more information
ACCGGTAGCATAGA
CGGATAGACTTAGT
GCATT
>seq2 etc
...
```

Specifically, each sequence (either DNA or amino-acid) is prefixed by a one line header starting with a greater-then sign. The header than contains some sequence identifier (generally similar in syntax to what you would use as an identifier in a programming language – no white space, characters, digits, and some symbols allowed) and also some additional free-form information. The latter can be more precisely defined for specific applications. Each header is then followed by one or more rows of sequence information, separated by

newlines. There are no specific requirements on the width of these rows, other than an unofficial standard keeping them a 60 characters (to fit on an old 80x24 CRT terminal). Of course, the last line of each record can and often is shorter.

Recently, a new format – *FASTQ* – has been proposed for encoding DNA sequences, and very specifically the sequences produced by modern sequencing experiments. This format is also split into a collection of records, one for each sequence, and each of which uses up four rows as follows:

```
@seqid other info
ACCAGTACGTCCGTG
+seqid other info     (optional)
!+30qr-130!@+-@
```

Just like the FASTA file, the first and third lines encode header information including a sequence identifier, prefixed by the 'at' sign (first line) and the plus sign (third line). The information on the third line is the actual DNA sequence, while the last line in the record encodes the quality information for each nucleotide. The latter is information provided by the sequencing instrument that describes the uncertainty in the specific call (essentially the noise in the signal processing used to convert the signal into a DNA sequence).

The quality information merits a few additional qualifications. First, the quality concept was introduced in Sanger sequencing and is supposed to be $-10 \log_{10} p_{error}$ where p_{error} is the estimated probability of error. For example, a quality value of 20 indicates a 1 in 100 probability of error. Second, the letter string encoded in the FASTQ format is a translation of the actual quality values into printable ASCII characters. For example, an offset of 48 would convert the number 0 into the character '0' (the zeroth position in the ASCII table is the unprintable NUL character).

While the FASTQ format is quite widely used it has several major limitations. First, it is unclear why datasets commonly comprising tens of millions of sequences need to be encoded in a human readable format. Second, the header information is unnecessarily verbose (can even be duplicated in the first and third lines of the record) and is often of roughly the same length as the sequence itself. Finally, the function translating integers into printable characters is not standardized nor obvious from the file itself. There are, in fact, several competing and incompatible formats that cannot be immediately distinguished from each other. A much better option would have been the development of a binary file format and associated utilities for viewing, editing, and converting this information.

2.7 Exercises

1. How many open reading frames are present in the following DNA sequence (assume the start codon ATG and the stop codon TAA)? Please highlight where all these ORFs occur.
ATGCATCATGGATGTTAATGTAACCGTCTAACTAA

2. A "palindrome" is a DNA sequence that is equal to its reverse complement. Please construct a palindrome of length 8 that uses all of the 4 DNA letters.

3. Assume you are trying to find out more information about a particular gene (say gene oprD) and the corresponding protein product. Which online biological database would you start with? (note: wikipedia is not an appropriate answer)

4. Identify the longest open reading frame (ORF) in the following DNA sequence (note: start codon ATG, stop codons TAG, TGA, TAA)
ATGATATGATGTCACAGGATGTAGATGCTCCGACATAAGCTTAG

Pick one position in the ORF found above and indicate which type of mutation (insertion, deletion, or mutation) would be called a "frame-shift mutation". Briefly explain why.

3 Exact matching

One of the most basic string operations is finding an exact copy of one string (often called query or pattern) within a longer string (text or reference). One can easily come up with simple examples of this operation in pretty much all text related applications, such as finding a word or short phrase within a webpage or document. Similarly, there are many such examples in biology. For example, restriction enzymes (proteins that 'cut' the DNA at specific locations in the genome) recognize exact patterns within a genome. Finding all such restriction sites is, thus, an instance of exact matching.

In this section we will try to develop fast algorithms for this seemingly simple problem. First, let's explore the following question:

Given a pattern (query), does it exist in the text (reference)?

Example:

```
text position     1  i            n = 17
Text              ATTCACTATTCGGCTAT
Pattern           GCAT
pattern position  1..m = 4
```

Before reading the following, try to come up with an algorithm to solve this problem. How 'expensive' is this algorithm?

As a quick aside – in Computer Science we generally compute the 'cost' of an algorithm in terms of two main commodities: time and memory needed to execute the algorithm. To simplify calculations at this point, assume that each object (letter, integer, etc.) uses up one memory location and that the only 'expensive' operation is the comparison of two characters. Thus, the run time of the algorithm would be expressed as the number of comparison operations. Also, as convention throughout the class, assume the text has length n and the pattern length m.

3.1 Naïve Approach

The most simple algorithm for exactly matching a string to another one works as follows. Start with the first character of the text and the first character of the pattern and compare each character individually. If a mismatch occurs shift the pattern one character and repeat the procedure. Terminate when either a match is found, or not enough characters remain in the text.

Here is the algorithm in pseudo-code:

```
for i=1,n - m + 1
     for j=1,m
          if P[j] != T[i+j-1]
               goto NOMATCH
      end for
     report match found @ position i
     NOMATCH
end for
```

Now, let us try to analyze this algorithm. First, its memory usage. Clearly we are only storing the pattern and the text (n + m memory locations) and a few additional variables (i, j, n, m), or a total of n + m + 4 memory locations. Since the memory usage is proportional to the total size of the inputs we say that our algorithm has 'linear memory usage'.

How about the run time? A quick back of the envelope calculation tells us that we repeat the comparison operation P[j] != T[i + j -1] m times in the inner for loop, which itself is repeated n – m + 1 times in the outer for loop, or a total of (n – m + 1) * m = nm – m^2 + m operations.

If we just want to coarsely compare the performance of different algorithms, we do not really need to go into this much detail, rather it is sufficient to focus on the largest term of the equation above and it suffices to say that our algorithm has a runtime 'order of' nm, also written as O(nm). This kind of reasoning is called 'asymptotic analysis' and focuses on just the behavior of the algorithm as the size of the inputs grows towards infinity. A full description of this approach is beyond the scope of this class and I encourage you to look up additional information if these terms are unfamiliar to you.

Question: *Does the algorithm above always perform nm – m^2 + m operations?*

The answer depends on whether we stop as soon as we find a match, or we continue until we have found all the matches of the pattern within the text. In the latter case we will have to do all the comparisons to make sure we do not miss any matches. In the former, as soon as the first match is found the algorithm stops, possibly using a lot fewer operations.

Question: *Assume we just want to find the first match, do you ever need nm – m^2 + m operations? If yes, when?*

The worst case scenario occurs when the pattern 'almost' matches the text, i.e., we have to compare characters all the way to the end of the pattern before figuring out the pattern doesn't match. See the example below.

```
1  i            n
AAAAAAAAAAAAAAAAA
AAAAT (m comparisons)
 AAAAT (m comparisons)
  AAAAT (m comparisons)
   ... (repeat for n-m+1 characters)
```

OK, so is the runtime the best we can do? Is it good enough for our typical applications? Let's imagine we want to match a 1000 bp sequence to the 3 billion bp human genome. The total runtime would be on the order of 3 10^{12} operations. On a 3 GHz processor (and assuming each comparison takes exactly one CPU cycle), the calculation would take 1,000 seconds, or about 16 minutes. Given that a typical sequencing experiment generates tens to hundreds of millions of sequences, this runtime is clearly too slow for even the simplest applications.

But how can we improve on this algorithm? The key idea is to use information we've already figured out when making earlier comparisons. If we can somehow reuse our work we might be able to save time. To illustrate this idea we will start with a simple algorithm, the Z algorithm, developed by Dan Gusfield (Gusfield 1997) as an educational tool to help explain the basics of other string matching algorithms.

3.2 Z Algorithm

For a text T, let us define a simple function Z that captures the internal structure of the this text. Specifically, for every position $i \in (0, n]$ in T, we define Z[i] to be the length of the longest prefix of T[i..n] that matches exactly a prefix of T[0..n]. We also define Z[0] = 0. Here is an example:

Text:	A	T	T	C	A	C	T	A	T	T	C	G	G	C	T	A	T
Z[i]	0	0	0	0	1	0	0	4	0	0	0	0	0	0	0	2	0

Question: *Can Z values help in aligning a pattern to a text?*

To use Z values as a tool for exact matching, we can construct the following string S = P$T, where P and T are the pattern and text, respectively, and $ is a character that does not match any other character in either pattern or text. Let us assume for the moment that we have a black box function that computes the Z values for string S.

Question: *Can the Z values computed for the string S defined above help us find matches between P and T?*

The answer is yes – it suffices to look at the Z values corresponding to characters in T. Any Z-value equal to the length of P indicates the pattern matches. Specifically, for any i > m, if Z[i] == m, pattern P matches text T starting at position i. (**Aside:** can Z[i] be greater than m?)

Question: *What is the runtime and memory usage of this algorithm?*

First, the memory – we clearly need to store the Z values in addition to the pattern and text, for a total of $2(n + m) + 1$ memory locations. Even though we are using more memory than the naïve algorithm, the memory consumption is still linear in the size of the inputs. There might be additional memory required by the algorithm that computes the Z values, but we'll get to that in a moment.

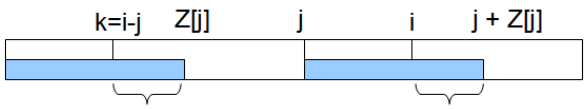
In terms of runtime, again omitting the work done to compute the Z values, we simply need to examine each position in the Z array to find all matches between P and T, or to decide that a match doesn't exist, i.e., the runtime is simply O(n) (we only need to examine the Z values corresponding to positions in the text). Here we see a much bigger improvement over the earlier naïve algorithm.

To sum up, if we have available an approach for computing Z values, we can perform exact matching much faster than the naïve algorithm, without using up (too much) more memory.

Of course, this result is predicated on the assumption that the Z values can be computed efficiently.

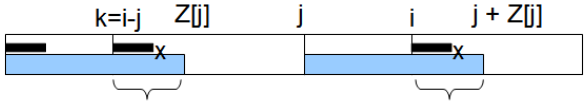
Question: *Can Z-values be computed in linear time and linear space?*

Note that one could come up with a naïve algorithm for computing the Z values using essentially the same approach used by the naïve exact matching algorithm. Clearly this approach would also yield a quadratic time algorithm (**Aside:** write down this algorithm and figure out its complexity in time and memory).



A much faster algorithm can be developed if we cleverly reuse the work done earlier. The intuition is that when computing Z[i] we can use the Z[j] values for j < i. Let us look at the figure below, where we assume that there exists a position j < i such that the Z 'box' at position j extends past i, i.e., j + Z[j] > i (see the figure).

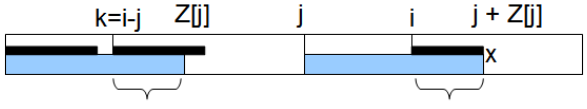
I.



The section of the Z 'box' of j that extends past i is identical to the same region at the beginning of the string (bracket in the figure), information that we can now use to compute the Z[i] value.

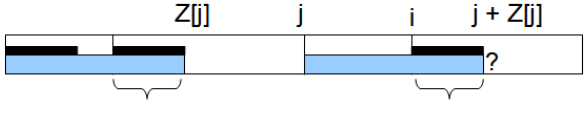
We can distinguish several cases:

II.



I. the Z 'box' for value k = i – j does not extend past the blue region (Z[j] box), k+Z[k] < Z[j]. In that case we know that Z[i] = Z[k] – the Z[i] box cannot extend past that point as the following character (marked with an x) is different from the corresponding character in the prefix of the string.

III.



II. The Z 'box' for value k extends past the blue region (k + Z[k] > Z[j]). In this case Z[i] = j + Z[j] – i, i.e., the Z[i] box ends at the end of the Z[j] box. The reasoning is that the character marked with an x in the figure is different from the character at the end of the Z[j] box in the prefix of the string (otherwise Z[j] could have been extended), and thus the Z[i] box cannot be extended any further.

III. Finally, in the remaining case, where Z[k] + k = Z[j], we know that Z[i] >= j + Z[j] – i. The exact value cannot be known without checking the additional characters (starting with the one marked ? in the figure).

If you check carefully the three cases above, you can see that in cases I and II we can directly assign the Z[i] value without doing any additional comparisons, thus all the work is done in case III. We will show below that this approach allows us to dramatically speed up the algorithm.

But first, what j should we use? For any i, we will select the value j < i such that j + Z[j] > i and this value is maximal, i.e., the corresponding Z 'box' extends the farthest into the string. The intuition is that this way we are making maximal use of the information already computed. What happens, however, if no Z[j] box extends past i? In that case, we simply revert to the naïve algorithm, i.e. we compare T[i] to T[1], T[i + 1] to T[2], and so on and stop when we find a mismatch, appropriately updating the Z value.

Z algorithm:

```
Z[0] = 0
maxZ = 0
j = 0
for (i = 1; i < n; i++)
  if (maxZ < i)  // must do the hard work
    l = 0
    Z[i] = 0
    while (T[i + l] == T[l])
      Z[i]++
      l++
    end while
    maxZ = i + Z[i]
    j = i
  else
    k = i – j
    if (Z[k] + i < maxZ) // case 1
      Z[i] = Z[k]
    else if (Z[k] + i > maxZ) // case 2
      Z[i] = maxZ – i
    else // case 3 Z[k] + i = maxZ
      Z[i] = maxZ – i
      l = Z[i]
      while (T[i + l] == T[l])
        Z[i]++
        l++
      end while
      maxZ = i + Z[i]
      j = i
    end if
  end if
end for
```

Question: *Check the algorithm above and fix the 'off by one' errors (i.e., should k be i – j or i – j + 1?)*

Now, why is this algorithm efficient? It should easy to see that other than a few additional variables we do not add any more memory, thus the memory usage remains linear. But how about the comparisons? They only occur within the two while loops, and only involve characters that have never been compared before, thus the total runtime cannot exceed the number of characters in the text, i.e., the runtime is linear.

Question: *Is it important that we only focus on the j whose Z[j] extends the furthest in the text? Argue/prove why this is important for ensuring either correctness or efficiency. (Hint: draw a situation where you are using a sub-optimal j, i.e., there exists another j you could use that extends further).*

Exercises

1. Implement the Z algorithm in your favorite programming language.

3.3 KMP – Knuth, Morris, Pratt Algorithm

As mentioned earlier, the Z algorithm was developed by Dan Gusfield as a way to teach string algorithms. The ideas we just presented should help guide you through the approach used by the Knuth Morris Pratt (KMP) string matching algorithm.

To develop the basic intuition about this algorithm, let us start with the naïve string matching algorithm described at the beginning of this chapter. The matching process compares the pattern to the text until either a match or a mismatch is found. In case of mismatch, the algorithm shifts the pattern position by one and restarts all the comparisons. Can we, however, reuse some of the information we learned while matching the pattern to the text. Let us look at the example below.

```
1 i                 n
XYABCXABCXADCDAFEA
  ABCXABCDE
      ABCXABCDE
```

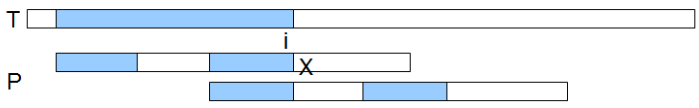
The naïve algorithm starts by matching all the characters highlighted in green, then mismatches on character D in the pattern, aligned to character X in the text. The intuition behind the KMP algorithm is that we can now simply shift the pattern over until its prefix matches a previously matched section of the text, as highlighted on the third row of the example. Thus, we can potentially save a lot of computation (comparisons) by not trying to match the pattern earlier.

Let us now formalize this approach better. We will define a new function, sp[i], which is defined for every position in the pattern. For every $i \in [1, m]$ sp[i] is the length of the longest non-trivial suffix of P[1..i] that matches exactly a prefix of the pattern.

Definition: SP[i] longest suffix of P from 1.. i that matches prefix of pattern.

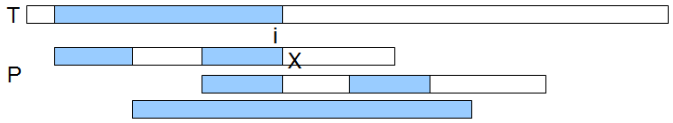
pattern:	A	T	T	C	A	C	T	A	T	T	C	G	G	C	T	A	T
sp[i]	0	0	0	0	1	0	0	1	2	3	4	0	0	0	0	1	2

As in the case of the Z algorithm we will assume for now that the sp[i] values can be computed efficiently. Can we use these values to speed up alignment? The basic idea is to shift the pattern i – sp[i] positions (rather than 1) once we mismatch at position i + 1, such that the prefix of the pattern is aligned to the corresponding matching suffix (see figure below). The matching continues from where it stopped (position in the text aligned to the X in the figure) as we already know the shaded region matches the text.



Question: *Are we missing any matches when shifting so far?*

To convince ourselves (i.e. prove) that the shifts are correct, i.e., we are not skipping any possible matches, we proceed with a proof by contradiction. Assume that there is another match between the original position of the pattern and the new one (which would be missed if we just jump along). The figure below shows this situation.



Assume the completely shaded pattern matches the text fully. As a corollary, the prefix of the pattern matches within the text in the same region where the pattern orginally matched (region before the X at the first position of the pattern), region that also matches the suffix of P[1..i] represented by the sp[i] value. This is clearly a contradiction with our initial assumption that sp[i] is the maximal such value. QED

Question: *Is the KMP algorithm efficient?*

Let us concentrate on the number of comparisons we make. It should be obvious that once a character in the text was matched with a character in the pattern, we never examine that specific character again. At the same time, the pattern can mismatch the same character in the text (that is aligned to the X in the figure) multiple times. Notice, however, that every time we hit a mismatch, we shift the pattern by at least one position (sp[i] < i), thus the number of shifts (and therefore comparisons) is bounded by the total length of the text. To sum it up, during our algorithm we encounter at most n correct matches, and we shift the pattern at most n times, for a total run time of 2n, i.e., the KMP algorithm is efficient. The memory usage is also good – we only use an additional m memory locations to store the sp[i] values.

The only bit remaining is computing the sp[i] values in linear time. The most simple approach is to use the Z algorithm, as there should be obvious similarities between the two concepts. Before you attempt to write such an algorithm, let us make a small change to the KMP algorithm. Let us define a new sp array, sp', where sp'[i] is the length of the longest suffix of P[1..i] that matches a prefix of P, **and** P[sp'[i] + 1] != P[i + 1] (the character after the matching suffix-prefix is different).

You can quickly check that running the KMP algorithm using the sp' values is equally correct and efficient. The new values are, however, easier to compute using the Z values.

Note: The sp' values highlight an interesting aspect – we are essentially being smarter about predicting what character is being aligned next (or more precisely what character isn't aligned next). An obvious extension of the sp' values would give us even more advance information. Since it's so obvious, we'll discuss it in class.

3.3.1 Exercises

1. Write out the algorithm for computing sp' values using the Z values.
2. Explain why the sp' values are easier to compute using Z values than the original sp values.
3. Implement, in your favorite language, the algorithms described above.
4. The genomes of many bacteria are circular. In order to represent the data in a linear form, genome assembly programs pick a random location in the genome to break the circle. Thus, it is possible that running the same program multiple times we would get different answers, corresponding to different circular rotations of the same string. Using the Z values described in class, describe a simple algorithm that identifies whether two DNA strings are circular rotations of each other. For example, string ACATTCAT is a circular rotation of string TTCATACA. (Hint: this algorithm is a simple extension of

the algorithm used to perform string matching based on Z values).

5. Write out the Z-values for the following string:

```
                        1 1 1 1 1 1 1 1 1 1 2 2 2 1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
A G A G A G A C C A G C A G A G A G A G A C G T
```

Assume you are trying to compute Z[14] - please explain how you compute this value using the efficient algorithm described in class. Specifically:

a) what are the values for j and Z[j]?
b) which earlier value of Z do you need to make this computation?
c) do you need to perform any additional character comparisons?

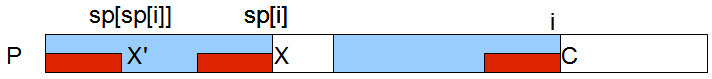
6. Give an example showing that k, the number of occurrences in T of patterns in set P, can grow faster than O(n + m). Be sure you account for the input size n. Try to make the growth as large as possible.
7. Define one possible relationship between sp(i) and sp(i – 1).
   Can you define a similar relationship between Z[i] and Z[i – 1]?

3.4 The original KMP computation of sp values

The Z algorithm and its use in the computation of the KMP sp' values has served its pedagogical role and has hopefully provided you with a better understanding of the way we reason about string matching. Below we will describe the original KMP algorithm (specifically the computation of the sp values). While we can clearly achieve the same results using the Z algorithm, the KMP approach will prove useful as we explore other aspects of string matching, such as the ability to simultaneously match multiple strings.

As you should already expect by now, the KMP algorithm achieves efficiency by reusing the results of earlier computations. This general optimization technique is called **memoization** and will pop up repeatedly in this class, most prominently when we discuss dynamic programming.

Thus, let us assume that we are trying to compute the value sp[i + 1] and that we already know the values sp[j] for all j <= i. Also, let us denote by c the character at position i + 1 in the pattern (c = P[i + 1]), and by x the character at position sp[i] + 1 (x = P[sp[i] + 1), i.e., the character that follows the prefix of P that matched the suffix ending at i. If the two characters match (x == c) we can simply extend the matching prefix/suffix pair by one character to obtain sp[i+1] = sp[i] + 1. What happens, however, if x != c? In this case sp[i] provides us with no additional information, but we might be able to use sp[sp[i]], the suffix of P[1..sp[i]] that matches a prefix of P. Here we will compare c with x', the character following this new prefix (see figure below). If they match, sp[i+1] = sp[sp[i]] + 1. If they don't, we simply continue trying further sp[sp[...[sp[i]]...]] values until we either find a match, or we 'bottom out' indicating no such prefix exists. In the latter case we simply set sp[i + 1] = 1 if c = P[1], or sp[i + 1] = 0, otherwise.



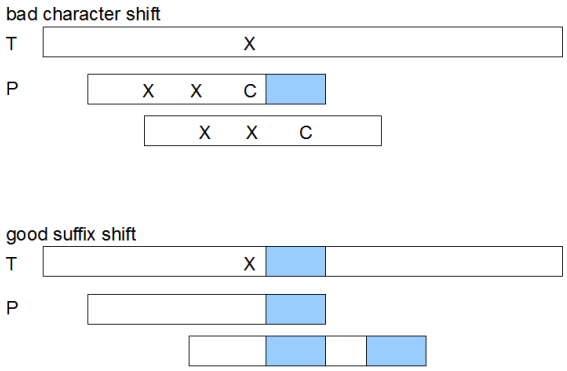
The correctness of this algorithm should be fairly obvious. Its efficiency, however, is unclear. At every position in the patter we might have to recurse through many sp[j] values before being able to compute the value of sp[i+1]. For now we will omit a proof of this property, which can be found in (Gusfield 1997) pages 50-51(Gusfield 1997). The intuition behind the proof, however, is that we can 'charge' every recurrence to a previously matched character in the pattern. Thus, while any particular computation may 'stall' for a number of steps, the total number of recurrences will not exceed the length of the pattern, yielding an overall linear run time.

3.5 Boyer-Moore algorithm

Another algorithm with a similar performance to KMP is the Boyer-Moore algorithm, though the 'philosophy' of this algorithm is quite different. The key idea behind Boyer-Moore is that our naïve algorithm made an arbitrary choice in how it compares the pattern to the text. Specifically, we shift the pattern to the right after every mismatch, and when we compare the pattern to the text, we start comparing from the left. We could, however, have as easily started the comparisons from the right and the algorithm would not be changed at all. Starting to match from the right, however, gives us a window into the future – based on how the pattern matched the text, now we have some information about the text that might be useful as we move forward.

The Boyer-Moore algorithm tries to use this information using two simple rules:

1. **The bad character rule** – if P[i] mismatches character x in T, shift pattern P until x aligns to the identical letter in P (of course we want to shift by the smallest amount such that this property holds, otherwise we might miss matches).
2. **The good suffix rule –** this rule is very similar in spirit with that of the KMP algorithm: shift pattern P until we find another occurrence of the suffix of P that already matched the text in our earlier attempt (see picture below).



The intuition behind the 'bad character' rule is very simple – there's no point shifting the pattern to a position where we know the aligned characters do not match. Similarly, the 'good suffix' rule ensures that at least part of the pattern matches the text at the new position. If neither rule applies, we can simply shift the pattern past the beginning of the match in T, specifically to a point where a prefix of P matches a suffix of P.

Clearly we need to preprocess the pattern in order to make these operations possible. For the 'bad character' rule we will need to record the position of all letters in the pattern. More specifically, we will construct an array R[x, i] which stores the rightmost position less than i where character x occurs. This information can be computed in linear time (**Question:** *how? outline the algorithm*) and can help us implement the 'bad character' rule.

For the 'good suffix' rule we will need two new arrays: L'[i] and l'[i]. The first stores, for each position i, the largest value < m such that P[i..n] matches a suffix of P[1..L[i]], and P[i -1] differs from the character preceding the corresponding suffix of P[1..L[i]]. You can see here a similarity with the definition of sp' in KMP. The l' array focuses on just the suffix-prefix matches, l'[i] is the length of the longest suffix of P[i..n] that matches a prefix of P. Both of these arrays can be easily computed using the Z algorithm (Exercise for you to figure out).

When executing the Boyer-Moore algorithm, we always choose the shift that advances the pattern the most.

Example:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6

pattern:	A	T	T	C	A	C	T	A	T	T	C	G	G	C	T	A	T
L'[i]	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	1	14
l'[i]	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	0

Question: *Does Boyer-Moore good suffix rule miss any matches?*

Proof is similar to that for KMP

Question: *Is the Boyer-Moore algorithm efficient?*

The answer is yes but the proof is quite complex. See (Gusfield 1997) pages 39-47.

An important feature of the bad character rule is that now the efficiency of the algorithm is directly tied to the size of the alphabet – the memory required to store the R array is dependent on alphabet size.

3.5.1 Exercises

1. Does the bad character rule add anything? In other words, are there situations where the bad character rule allows the algorithm to be faster than if just the good prefix rule was used?
2. The Boyer Moore (BM) algorithm uses a "bad character" rule in addition to the suffix rule for determining how far to shift the pattern across the text after a mis-match. In short, BM compares the pattern against the text, character by character, starting at the end of the pattern. Once a mismatch occurs—character x in the pattern is aligned to character y in the text—the BM "bad character rule" shifts the pattern to the first occurrence of character y in the pattern. In addition, BM uses a "good suffix rule", shifting the pattern, after a mis-match, to the point where the suffix of the pattern that has been compared against the text matches again within the pattern. In contrast, the KMP algorithm, compares the pattern from left to right, and uses a "good prefix" rule, shifting the pattern until a prefix of the pattern matches the suffix of the already-compared region of the pattern.

   It is possible to add a "bad character" rule to KMP in addition to the "good prefix" rule. Evaluate how efficient this rule is and explain why it is more effective in the Boyer-Moore algorithm.

3.6 Periodicity of strings

Here is just a quick introduction to several useful properties of strings, specifically related to periodicity. For now we won't make much use of these concepts but arguments based on periodicity can form the basis of elegant proofs in the realm of strings, such as the proof that the Boyer Moore algorithm runs in linear time. First some definitions.

Definition 1. A string S is periodic if S = BBBBB..B for some shorter string B. The length of B is the *period* of the string.

Definition 2. A string S is suffix semiperiodic if S = BBBB..Bα for some prefix α of B

Definition 3. A string S is prefix semiperiodic if S = αBBBB..B for some suffix α of B

It may not be immediately obvious but any string that is prefix semiperiodic is also suffix semiperiodic with the same period.

It is also possible to show that if a string is semiperiodic with two different periods of lengths p_1 and p_2 it is also semiperiodic with the greatest common denominator of the two periods GCD(p_1, p_2).

Ok, so what does periodicity have to do with sequence matching? Imagine that a pattern of length m matches at two overlapping positions in the text, t_1 and t_2, and that these positions are spaced by less than half the

length of the pattern ($|t_1 - t_2| < \left\lfloor \frac{m}{2} \right\rfloor$). In that case we can show that the pattern must be semiperiodic with period $|t_1 - t_2|$. This property is actually useful in the proof that Boyer Moore good suffix rule leads to a linear time algorithm. The general idea of the proof is that either the algorithm shifts the pattern a lot or parts of the pattern are periodic – property that allows us to bound the amount of work done by the algorithm. If you are curious, the full proof is described in (Gusfield 1997) pages 39-47.

3.6.1 Exercises

1. Prove that a prefix semiperiodic string is also suffix semiperiodic with the same period.
2. Prove that a string semiperiodic with two different periods is also semiperiodic with their greatest common denominator.
3. Prove that a pattern matching a text at two locations spaced by less than the length of the pattern is semiperiodic with the period equal to the spacing between the matching locations.

3.7 Dealing with multiple patterns – Aho-Corasick algorithm

So far we have managed to solve the exact matching problem efficiently when we only need to search one pattern against a text. What happens, however, if we want to match multiple patterns against a same text? Can we do better than matching each character to the text separately? This situation occurs quite commonly, e.g., when mapping all the reads generated in a sequencing experiment against a genome.

Let us assume we have z patterns, each of length m, and we want to match them against a text of length n. The trivial algorithm leads to a runtime of O(z(n + m)), which can be prohibitive for large values of z (e.g., millions of sequences matched to a genome).

A first intuition behind the Aho-Corasick algorithm is the idea that we should exploit similarities between the patterns as much as possible, the same way we've exploited self-similarity within the patterns in KMP and Boyer-Moore. The datastructure we'll use is called a *keyword tree* – just a fancy word to say we'll construct a tree structure that merges together patterns which have common prefixes (see below for patterns abcq, abcr, bcde, cd). Each edge in this tree is labeled with one single character, and every path from the root to a leaf spells the sequence of one of the patterns.

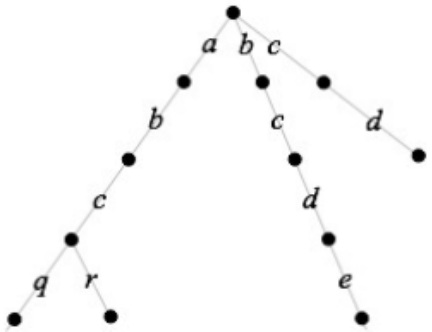


Figure 2: Keyword tree

This structure is just a (slightly) compressed way to store the patterns together. Having the shared prefixes along the same path in the tree should also allow us to simultaneously match (up to a point) multiple patterns, though in the worst case this ability alone does not allow us to improve on the naïve search of each pattern against the text.

Ok, so how would the matching process work with a keyword tree? We simply align the root of the tree to some position in the text, then follow the path indicated by the corresponding letters in the text until we either

reach a leaf (i.e., one of the patterns matched) or find a mismatch. In the later case we simply shift the position of the root in the text and restart the alignment. This is clearly a worse algorithm than running KMP on each string independently, as I've just described a tree-based version of the quadratic time naïve matching algorithm.

The key insight behind speeding up the algorithm is that we can try to implement the KMP preprocessing approach on all the patterns simultaneously. To better understand this idea, think of simply following one of the paths in tree until we hit a mismatch. For one of the patterns P along that path we already know how to compute the sp[i] values, and thus can shift the pattern (and implicitly the tree) to the position indicated by the corresponding sp value. Would this shift, however, be correct? By the definition of the sp values we know that as far as pattern P is concerned, shifting by the amount indicated by the sp value will not miss any other matches of P. What we cannot guarantee is that we do not miss matches of another pattern Q in the set. What we are, thus, looking for, is a way to extend the sp values to the entire set of the patterns. Specifically, we will try to find the longest prefix of some pattern (longest path starting at the root) that matches a suffix of the path we have already matched. We will encode this information as links between nodes in the tree.

More formally, we will define a collection of *failure links* that, for every node N in the tree, connect N to another node M which has the following properties: (i) the path from root to M spells a suffix of the path from root to N; and (ii) M is the deepest such node (the longest suffix-prefix match in the tree). If no such node exists we simply link N to the root of the tree.

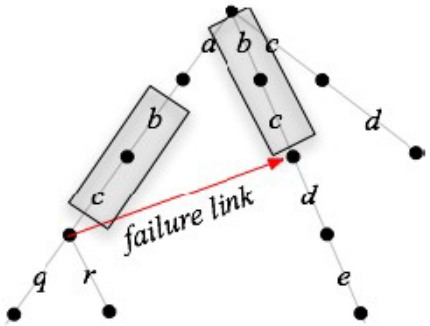


Figure 3: Failure links

So how can we use this information to align the tree to the text? As in the naïve algorithm, we start with the root of the tree aligned to the beginning of the text and match the text and a path in the tree with each other while advancing a pointer in both of them. When we encounter a mismatch, we simply follow the failure link in the tree, and restart the matching process from the new location, without moving the pointer in the text, the same way we have done in the KMP algorithm. It might help to work through a few examples but you should be able to see that we never need to 'rewind' within the text as we'll always be able to guarantee that the path from the root to the current node in the tree already matches the text (due to the definition of the failure links).

The runtime of this algorithm (ignoring the time needed to compute the failure links) is simply O(n + m) (rather than O(n + zm)), following the same type of argument we used in the construction of the KMP sp values. Clearly every time we match a character in the text we never examine that character again. We could, however, have many mismatches at any specific position in the text. The number of such mismatches can however be bounded by the total matches made so far, as follows. Every time we have a match we go deeper into the tree. When we find a mismatch we follow a failure link which, by definition, pushes us higher up in the tree. As we cannot go higher than the root of the tree, a mismatch will not cost us more than the amount of matches we've already 'banked'.

Can the failure links, however, also be efficiently computed? The answer is yes – we can simply use essentially

the same algorithm used to construct the KMP sp values. Specifically, we will process the tree one level at a time, and assume that all failure links have been computed up to a specific level in the tree. Assume we are at a node N whose failure link (N-M) has already been computed and want to now compute the failure link for this node's child. We simply examine the character on the edge connecting N to its child and compare it to the labels on the edges coming out of M (the target of the failure link). If one of them matches, we simply link the child of N to the corresponding child of M. Otherwise, we continue this process by following the failure link out of M, and so on. Just as for KMP we can prove that this algorithm is linear time (O(zm) due to the size of the pattern tree).

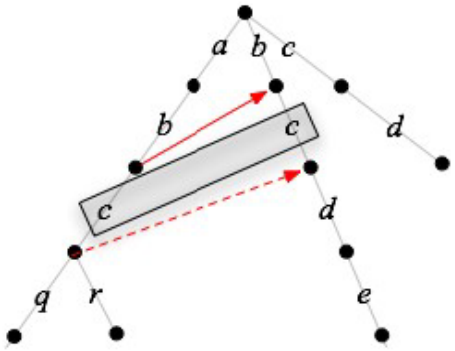


Figure 4: Computation of failure links.

One situation we have glossed over is that of a pattern that is a proper substring of another one. In the tree we used as an example in this class, see, for example, pattern 'cd' which is also found as a substring in pattern 'bcde'. Following the algorithm outlined above blindly we might miss the match of the smaller pattern. For example, assume that we matched the string 'bcd' and mismatched on character 'e'. The failure link would bring us to the leaf labeled 'cd', yet our algorithm would not report a match. Of course, a simple solution is to report a match anytime we reach a leaf, whether by matching a character or by following a failure link.

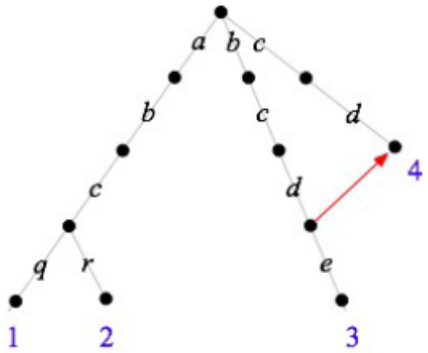


Figure 5: Pattern matching within another pattern.

This simple solution, however, can be very expensive, as we would have to follow the failure links from every node just to make sure we haven't missed any matches. To understand that, imagine that pattern 3 in the figure above matches the text. Simply following the pattern to the end would make us miss the fact that pattern 4 also matches. To find that match we would have to follow the highlighted failure link even though we didn't mismatch. Such link transitions cannot be amortized against earlier matches and can, thus, lead to inefficient runtimes.

To address this performance issue, we will store a new set of links – the *output links* – which connect a node to all the leaves that should be reported once we reach that node.

Question: *Can the output links be computed efficiently?*

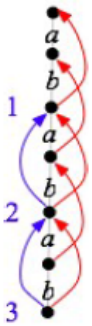


Figure 6: Output links (blue) and failure links (red) for patterns ab, abab, ababab.

You should be able to answer that question by slightly modifying the algorithm for computing failure links.

Now, why do output links help speed up the algorithm? At first glance, we still spend time following links to all the patterns that match. But that's exactly the point – we only follow output links if we output a match, so the work we do is justified. The resulting runtime is O(n + k) where k is the number of matches found during the duration of the execution, or O(n + mz + k) overall if we include preprocessing time. This situation highlights another interesting parameter in evaluating the performance of algorithms – the dependency of the runtime on the size of the output. This is also another knob we can tune as we try to find the best tradeoff between efficiency and utility.

3.8 Matching with wildcards

So far we have assumed that all characters in the pattern must all match. We can also envisage a situation where we want to allow mismatches at specific locations in the pattern. For example, we might want to match the following DNA string, ACCATGNTAGGNCGANNTAGGC, where the character N can match any of the four nucleotides. The trivial solution to this problem involves generating all possible such strings (there are 64 of them) then reporting a match when any one of them matches. Of course we can use the Aho-Corasick algorithm for this process, however the size of the pattern set grows exponentially with the number of 'don't care' symbols we include.

A better approach involves simply breaking the pattern into a collection of subpatterns comprising all strings between the N characters (in our case ACCATG, TAGG, CGA, TAGGC). We then use the Aho-Corasick algorithm to find all locations in the text where these patterns match. Matches of the whole original pattern will be easily recognizable from the spacing of the individual matches. For example, the individual patterns may match at positions 12,19, 24, 29 in the example above.

3.9 Special strings

When playing with strings, either for fun, or to test and execute algorithms, it is always useful to have some way of creating artificial strings with specific properties. Here are two special strings that have specific properties that make them useful when reasoning about strings (e.g., to find worst case scenarios) or when trying to trip up the code you wrote.

3.9.1 Fibonacci strings

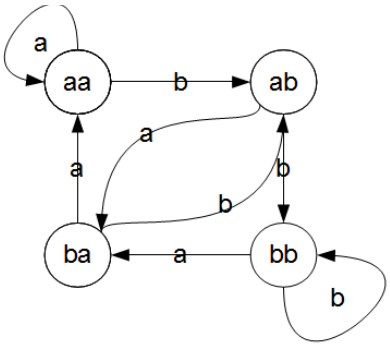
As you can imagine, they are strings inspired by the recursively define Fibonacci series. Specifically, let us denote by F_n the Fibonacci number of order n. By definition $F_n = F_{n-1} + F_{n-2}$ for all $n >= 3$. $F_1 = 1$, $F_2 = 1$. Similarly, we can define $FS_n = FS_{n-1}\ FS_{n-2}$ to be the Fibonacci string of order n, where the addition operation is replaced by the string concatenation operation. Also, $FS_1 = b$, $FS_2 = a$. It should be easy to see that the Fibonacci strings have lengths equal to the Fibonacci numbers of the same order. Here are some examples:

FS_1 = b
FS_2 = a
FS_3 = ab
FS_4 = aba
FS_5 = abaab

...

3.9.2 De Bruijn strings

De Bruijn strings are a type of 'comprehensive' strings. Specifically, for a given alphabet Σ and a length k, a de Bruijn string of order k contains as substrings all strings of length k over Σ. To see that such strings exist, are compact, and can be constructed we can build a simple graph (called a de Bruijn graph – more on that later in the course). This graph contains as nodes all strings over Σ of length k-1, and its edges connect nodes that overlap in a suffix-prefix fashion by exactly k-2 characters. Specifically, an edge exists for every pair of nodes A, B, where the corresponding strings have the form xα, αy where α is a string of length k-2 and x and y are characters in Σ.



It is easy to see that the graph is Eulerian (each node has equal in-degree and out-degree) and thus we can traverse it in such a way that every edge is seen exactly once. This traversal (which is not unique – rather one can have an exponential number of such traversals) spells out the de Bruijn string of order k.

3.10 Automata

So far we have developed specific algorithms to handle the different string matching problems. Here we will take a short side-trip into the world of automata, a formalism that is intrinsically tied to string matching.

In this context we define an automaton as a construct (to be formalized a bit more below) that recognizes a specific language. The language can itself be formally defined as some subset of the total collection of strings that can be constructed over a specific alphabet.

The automaton can be defined as a collection of *states* connected by *arcs* or *transitions*. Some states are special: there is an *initial state*, where the execution of the automaton starts, and one or more *terminal states*, which indicate that the automaton has recognized a specific string from the language. The transitions are represented as triplets (s, t, c) of a *source state* s, *target state* t, and character recognized c. Essentially, we have described a directed graph, having as nodes the states, edges as transitions, and the edges are labeled with the character recognized by the specific edge.

An automaton operates as follows. Assume a text is fed to the automaton. The automaton starts in the initial state, then transitions to different states by matching the character seen in the input with the label on one of the edges leaving the current state. A string is *recognized* or *accepted* by the automaton if it represents the label of a path from the initial state to one of the terminal states.

As you can see in the example below, a simple automaton can recognize strings that are quite complex.

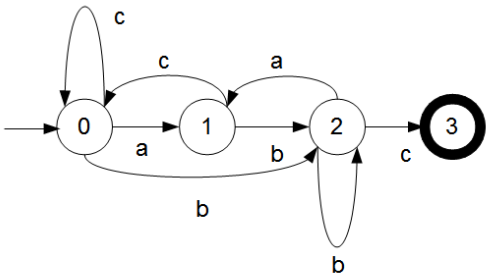


Figure 7: Automaton that recognizes all strings that end in bc

It is now easy to see that 'recognizing' a string is essentially the same as finding a match of that string within a text (the string input to the automaton). Thus, for a particular string we could simply construct an automaton that recognizes it. The automata are, however, a lot more powerful than that – they can recognize languages: sets of strings with certain properties. As an example, the graph we showed above that generates de Bruijn strings can be converted into an automaton that recognizes de Bruijn strings by simply making the starting state a terminal state as well (this automaton would also trivially recognize the empty string).

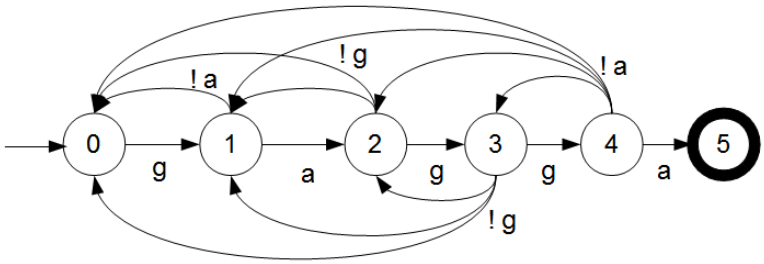
It is important to note that there are many other definitions of automata beyond the one given here, and automata theory is the basis for much of what we know as computer science. Also, the automata's ability to recognize languages is a central part of parsing, whether we are focused on programming languages (the YACC program essentially converts a representation of a grammar into C code that implements the automaton that recognizes words from the grammar), or natural language.

Let us try to build a simple automaton that can match the pattern GAGGA.

(to fill in during class)

Similarly, we can easily see now that the Aho-Corasick keyword tree is simply an automaton that recognizes the patterns stored in the tree. Of course, the output (or success) links do not quite fit within the formalism stated above as they are state transitions that do not consume any characters from the input (though that can be easily fixed by having them accept the empty string ε). More importantly, however, following the success link transitions should bring us back to the state we started from – something that is not easy to model cleanly.

The automata we have describe above are all *deterministic* – for any state, input character combination we know exactly which is the next state we will explore. This property can be relaxed in *non-deterministic* automata where the execution could follow multiple equivalent paths. These are primarily useful as a theoretical construct – e.g., the pattern matching automaton described above could be more simply constructed as a non-deterministic automaton as shown below.



In this automaton, multiple edges have the same label (! a – any character that is not a), and the automaton

'magically' figures out to which state it should proceed. While it is clearly more simple to build such an automaton, it is not immediately obvious how to execute one. It can be proven, however, that non-deterministic automata can be converted into deterministic automata. Also, there are probabilistic interpretations of non-deterministic automata that we will discuss later in the course.

As I've hinted above, there has been substantial research conducted in automata theory which we can possibly leverage for string matching applications. Building an appropriate automaton for a specific matching task can be difficult, however once built these automata can be quite efficient (and can be easily implemented in hardware). This is yet another design choice you may have to face as you develop your string matching algorithms.

3.11 Introduction to suffix trees

Now let's get back to the Aho-Corasick algorithm. In its design we needed the 'output' or 'success' links so that we can output all the pattern matches even when they occur as a substring of another pattern. We encounter a similar situation if we relax a bit our matching problem. Specifically, we try to find not necessarily a full match of a pattern against a text, but the longest substring of the pattern that matches a substring of the text. For example, for P = ATTCGCTTAGCCTA and T = GGAGCTTAGAACT the answer is GCTTAG. Can any of the algorithms described earlier solve this *longest common substring problem*?

It may not be immediately obvious, but none of the earlier algorithms can solve it efficiently. Key to their correctness and efficiency was the shift function, yet this function only guarantees that we will not miss a complete match of the pattern within the text, not that we will not miss a longest substring match between pattern and text. Also, all the algorithms focus on matches that occur at the beginning (KMP, Aho-Corasick) or the end (Boyer-Moore) of the pattern, while we want to be able to find any substring.

As before, let's ignore efficiency for now and focus on a naïve solution to the problem. This naïve algorithm is very simple:

```
Construct all substrings of P
Use Aho-Corasick to match all these sub-patterns to T
```

It is not hard to see, however, that the algorithm is inefficient. We have $O(m^2)$ different patterns, each of maximal length O(m), thus yielding an $O(n + m^3 + k)$ runtime (as before k is the number of matches we may find). Can we possibly do better?

3.11.1 The suffix tree structure

Let us try to improve this algorithm. First, the algorithm is fairly inefficient because of the large number of patterns we have to store in the tree. To reduce this number we'll just store O(m) patterns, specifically, all the suffixes of P. Even though we don't store all possible substrings, we may have some hope to solve the longest common substring problem as any substring match will be the prefix of some suffix, i.e., we'll run into it as we match the suffix along the text.

Thus, we have our first instance of a suffix tree (or trie), as shown below for the pattern ABC.

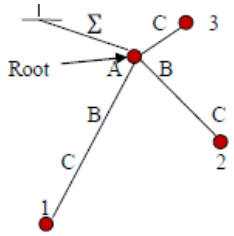


Figure 8: Suffix trie for the string ABC

Note that we have labeled the leaves with the starting position of each suffix (1 is the longest suffix, etc.) and also added the empy suffix Σ. We have also omitted the internal nodes along each path.

This new structure saves a bit of space, reducing the memory requirement from $O(m^3)$ to $O(m^2)$. While the number of leaves is O(m), even if we don't explicitly store the nodes along the simple paths in the tree we still need to store $O(m^2)$ characters. Or do we? A simple 'trick' can help us further lower the space. Specifically, each edge within the tree corresponds to a substring of the pattern, and can, therefore, be represented by its coordinates (start and end) within the pattern. As a result, each edge needs to only store constant information, and the total size of the tree becomes O(m).

The more mathematically pedantic amongst us may object at this point as the edge labels are not necessarily constant information, nor are the labels on the leaves of the tree. Rather, this information requires O(log m) bits of storage, leading to the overall size of the suffix tree to be O(m log m). Nonetheless this is still better than the $O(m^2)$ we had before. Also, for simplicity, we will ignore this log m term as for virtually all practical strings, coordinates within the string can be represented in a single computer word (especially on the now ubiquitous 64-bit architectures). As you write practical code, however, you should remember this omission (which we'll make for many other algorithms in this class) and make sure the underlying assumption holds for your specific application.

3.11.2 Adding the failure links

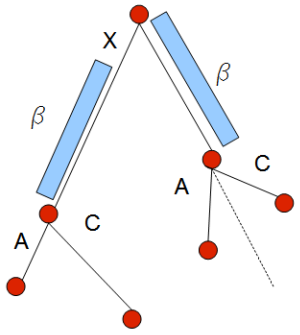
OK, so far we have built a new type of datastructure which only takes up linear space and stores all the suffixes of a given string. We are still missing a bit that will make this datastructure useful. Specifically, remember the failure links in Aho-Corasick. Can we represent them as well without blowing up the space? Just as a refresher, the failure link out of node N with label (string spelled by path from root to N) $\alpha\beta$ connected N to a node M whose label is β, and β is the longest such string in the tree (the longest suffix-prefix match between two patterns). What do these links look like in a suffix tree?

It's not hard to see that for any leaf in the suffix tree with label cβ (where c is a character), there exists another leaf with label β. This property stems from the fact that the suffix tree stores all suffixes of the original string. Also, β is the longest such string possible, i.e., the link connecting the two leaves satisfies the property of a failure link in the Aho-Corasick algorithm.

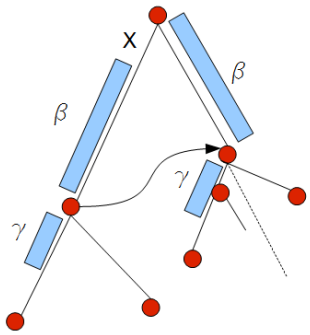
Question: *Does the same property hold for internal nodes of the suffix tree?*

Specifically, take a particular internal node of the tree – is the suffix link out of that node well defined, i.e., will it connect to another node in the tree or just along some path? Like above, assume the path for the node is xβ. Also, by definition the node is the boundary of a fork in the tree and has at least two children. To match the figure below let us assume the labels of the edges start with A and C. Like for the leaves we can convince ourselves that there exists at least some path in the tree with prefix β, thus the end of that path is the target for the suffix link out of the chosen node. To convince ourselves that the end of the suffix link is a node in the tree we can observe that there are at least two suffixes in the tree with prefixes xβA and xβB, and thus there must

also exist two suffixes that are one character shorter: βA and βB. These share a common prefix β, but differ at the next character and therefore the prefix β must be followed by a fork.



Just as in the case of Aho-Corasick, storing an additional suffix link for each node of the tree does not blow up the space and the whole datastructure remains O(m). Note, however, that we cheated a bit – in the suffix tree we hide all internal nodes of the tree with the exception of branches. Do we need to store failure links (called *suffix links* here) for all the internal nodes as well? If we do, the space would blow up to quadratic. See the situation in the figure below where we are looking at a position in the tree with prefix xβγ, an that this postion occurs along an edge (rather that at an internal node).



Do we need to store a suffix link from this position? Following the same argument as above there must exist somewhere else in the tree a position with prefix βγ. We can implicitly find this position without explicitly storing a link by simply following the suffix link of the parent node, then skipping down length(γ) characters starting from this new node. It is important to note that we may have to make a decision at internal nodes found along the string γ in order to follow the correct path, thus following these implicit links may not be a constant time operation.

Just to make this part a bit more clear, here's a pseudo-code description on how we would follow an implicit suffix link:

```
k – number of characters we have progressed along edge e
n – parent node
sn – target of suffix link from n

select edge se out of sn that starts with the same character as edge e
I:
if length(se) > k simply progress k characters along se
otherwise progress to node at end of se
follow the edge starting with the character at position length(se) along edge e
repeat from I: replacing k with k – length(se) until finding the target of the suffix link
```

Note that all operations are constant time (skip a certain number of characters, compare a character with the first character of an edge) and the cost of this operation is only proportional to the number of nodes we encounter along path γ. We will show later that this number will not affect the runtime of algorithms using the suffix tree.

3.11.3 Separating out the suffixes

Note that as described above our algorithm can end a suffix at an internal node (see suffix A in string GATTACCA – figure to follow) and can even end a suffix along an edge. To avoid such situations and greatly simplify the representation we simply add the special character $ at the end of the string. This character doesn't match any other character and, thus, forces all the suffixes to end at a separate leaf. From now on we will just focus on this variant of suffix trees.

3.11.4 Longest common substring computation with suffix trees

So far we have managed to develop a variant of the Aho-Corasick tree that efficiently stores all suffixes of a string. Let's see how we can use it to solve the longest common substring problem. We simply run the Aho-Corasick algorithm as before matching characters in the text to a path in the tree. While we do so we also record the deepest location in the tree encountered and the node immediately below this position (possibly a leaf). Upon mismatches we follow the suffix links using either the links themselves (if we get stuck at a node) or the implicit suffix link algorithm described above. We proceed until the end of the text, then report the longest match found so far.

Question: *How do we report the matches?*

First, it is important to figure out that using the tree is somewhat different than matching a pattern against a text. Specifically, we are not truly shifting the pattern along the text, rather we position the root of the tree at a particular position in the text. We, thus, implicitly align at that position in the text some prefixes of suffixes of the pattern at the same time. The figure below may clarify things a bit.

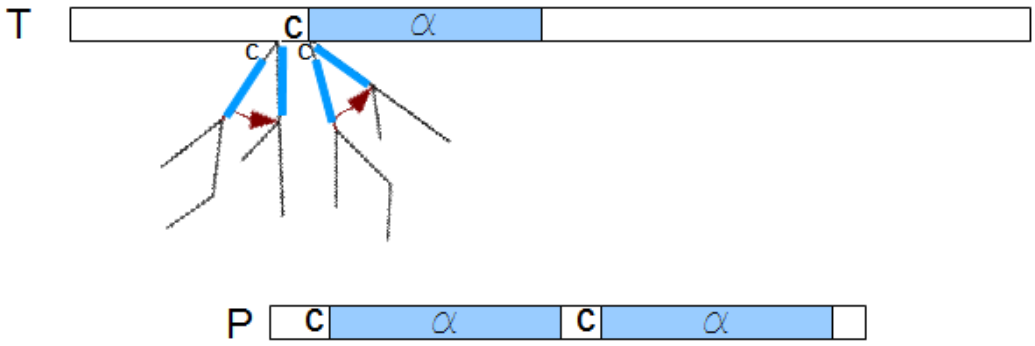


Figure 9: Suffix tree 'aligned' to two different positions in the text after following a suffix link.

Here we show a suffix tree that aligns at two different positions in the text. At the first position, the string cα in the text occurs as a path in the tree, i.e., it is the prefix of some suffix in the pattern P. Note that since there are more than one leaf below the point where the match ends, the string cα must occur more than once in the pattern (once per leaf below that point), as seen in the figure. When following the suffix link, the root of the tree gets implicitly moved one position in the text, essentially skipping over character c. It should be now clear that at any stage in the algorithm, the pattern is not aligned to a particular position in the text, rather all matching substrings are simultaneously aligned to the same position in the text.

Thus, the algorithm for computing the longest common substring of a text proceeds as follows.

```
d = 0 # deepest (in terms of characters) position in the tree found so far
dn = root # node below the deepest position

I:
while tree path matches text
  proceed deeper in the tree

# now we mismatched
update d and dn if the current depth deeper than d
follow implicit suffix link
repeat from I: until end of text is reached.
```

Question: *Is this algorithm efficient (linear time)?*

The key is limiting the number of operations done when following the implicit suffix links. To do so we will keep track of the depth within the tree that we have reached at a particular time.

Observation: *When following a suffix link the tree depth can decrease by at most 1.*

Proving this observation is fairly straightforward. First, note that the depth of a particular node N in the tree is simply the number of nodes encountered on the path from the root to N. By the construction of the suffix tree, each of these nodes has a suffix link, and all these links must point to distinct nodes along the path from root to s(N). Assume, for example, that the path to n is $c\beta$, and hence the path to s(N) is β. Any node M on the path from root to N would have path $c\gamma$, where γ is a prefix of β, and therefore s(M), with path γ, must occur on the path from root to s(N). Thus, the node depth of s(N) is at least as large as that of N, with one exception – it is possible that one of the nodes above n has a suffix link to the root, i.e., the depth of s(N) may be one smaller than the depth of N.

This observation now allows us to bound the total amount of work performed by the algorithm when chasing implicit suffix links. Specifically, we have shown in Aho Corasick that the total number of failure links followed during the algorithm is bounded by the length of the text. Now we'll use the same 'banking' argument to bound the amount of nodes traversed each time we follow a suffix link. By the observation above, the node depth can at most decrease by one every time we follow a suffix link. The depth can also never exceed m (length of longest suffix) or decrease below 0. As a result, the total number of nodes visited has to be proportional to the number of suffix links followed during the algorithm as each increase in node depth is 'amortized' against one of the decreases in node depth, and we can only decrease the node depth by at most one per suffix link followed.

3.11.5 Matching statistics

Let us define a generalization of the longest common substring problem. For every position i in a text T we'll define ms(i) to be the longest substring of T starting at i that is also a substring (somewhere) of a pattern P. It can be easily seen that computing the ms(i) values is a by-product of the algorithm for finding the longest common substring. Specifically, in ms(i) we store the depth (in terms of characters) reached during the current execution. When following a suffix link we simply increment i and set ms(i+1) = ms(i) – 1, then proceed with the algorithm as before.

3.12 Linear time construction of suffix trees

We will now describe Esko Ukkonen's algorithm for constructing suffix trees in linear time (Ukkonen 1995).

Note that the suffix tree structure was developed earlier by Weiner and subsequent work by McCreight and Ukkonen simplified the construction approach.

The following description can be quite confusing. I suggest you also use online resources to better understand what's going on. The following link http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english provides a nice description and also a number of other pointers [Thanks Max].

The basic idea of the approach is that we will iteratively build the tree on prefixes of the string. For example, for string BANANAS, we will build the tree for string B, then for BA, BAN, and so on. We'll denote by T_i the suffix tree built on the ith prefix of the string S (T_1 corresponds to B in our example, T_3 is BAN, etc.) At each stage in the process we will add one character to all the suffixes stored in the earlier trees, as well as add the final suffix for the just added character. Note that we will ignore the $ character for now, and allow suffixes to end along a path in the tree.

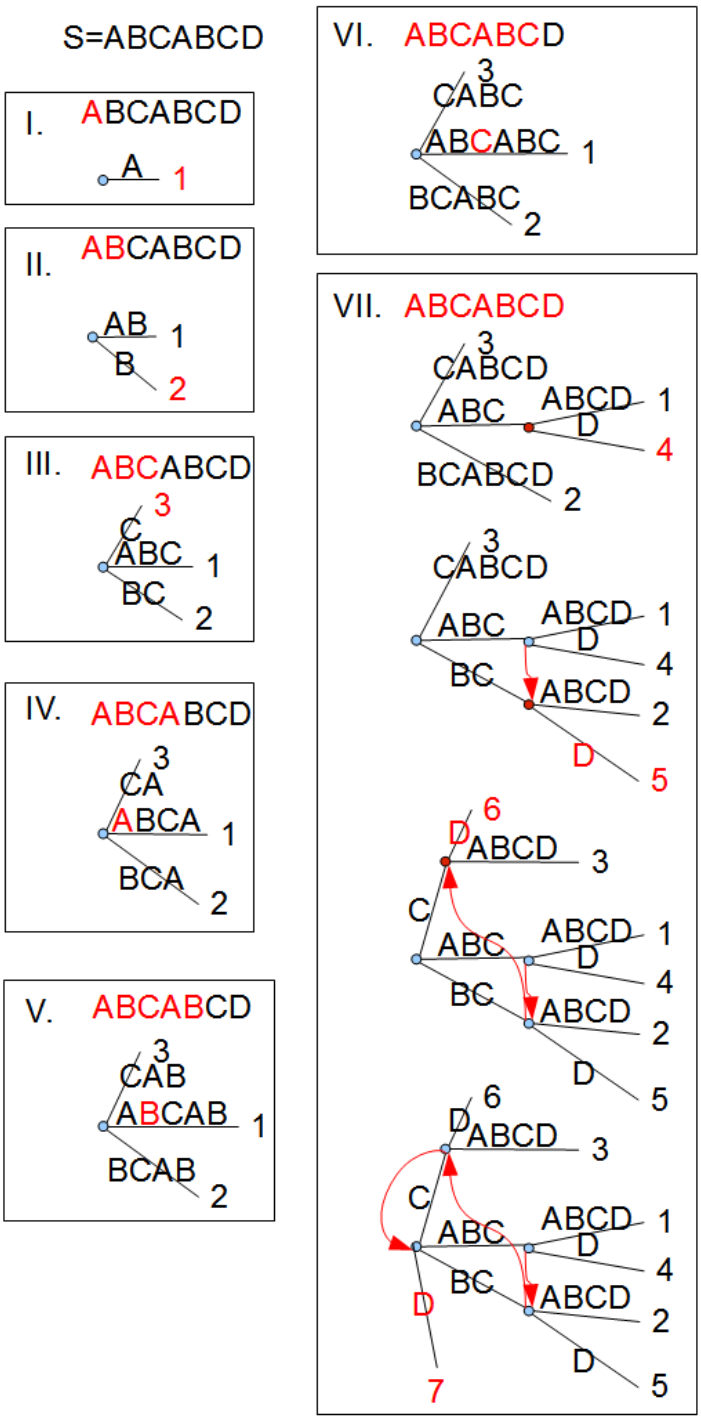
Clearly this algorithm appears to be quite expensive. We process n different prefixes, and at each prefix i we have to update i different suffixes, which leads to an $O(n^2)$ algorithm. If we take into account the time needed to 'find' each of the suffixes we may end up with an $O(n^3)$ solution, far from what we are trying to achieve.

A couple observations can help us speed up this algorithm dramatically. For the following, assume we are processing suffix α of tree T_i and try to extend it with character S[i+1].

1. **Once a leaf, always a leaf.** Once the algorithm has created a leaf, that leaf will never disappear. Furthermore, the string labeling the edge leading to this leaf will always end at the end of the current string (position i in the original string for tree T_i) As a corollary, at each stage in the algorithm we do not need to update the leaves that are already created. Instead, once we create a leaf we can simply label the end of the edge with a special 'end of string' character (remember, the edge is simply a pair of numbers corresponding to the coordinates of the edge in the original string). Once the tree is constructed we will spend O(n) time to replace these numbers with n – the end of the whole string. Also note that there are O(n) leaves in the tree and we only spend computation when creating each leaf, thus the total time spent on leaves in our algorithm is overall O(n).

2. **If string αS[i+1] already exists in the tree, all subsequent suffixes also exist.** This observation is fairly easy to prove. If αS[i+1] existed in tree T_i, then clearly all other suffixes shorter than it would also be in T_i by the definition of this tree. As a result, when the our algorithm reaches this situation, we can simply stop and proceed with the next prefix of S, i.e., tree T_{i+1} is complete and we can proceed to T_{i+2}. As a corollary, we can also bound the number of times we encounter this situation (a suffix of T_{i+1} is already in T_i) to O(n) as the situation only occurs once per iteration.

3. **If a node is created by the algorithm, the node to which the suffix link will point either exists in the tree, or will be created immediately afterward.** To see why this is true, assume we are processing suffix α which is not a leaf (i.e., α either ends in the middle of an edge, or is a node with two or more children), yet the character(s) following α is/are not S[i+1]. If α ends at a node, the suffix link must already exist since T*i* is a correct suffix tree. If α ends in the middle of an edge, we must now split the edge and create a new node. Now our algorithm will try to add character S[i+1] to the next suffix, β such that $\alpha = c\,\beta$. If β ends at a node, we simply add a suffix link from the newly created node to this node. If β ends in the middle of an edge, the character following β is not S[i+1], i.e., a new node will have to be created, and this node will be the target for the suffix link from the previously created node. Thus, suffix links will be correctly created. Furthermore, the process of following from suffix to suffix can use the same approach we used to find the longest common substring. Following the same logic, over the total execution of the algorithm, this process also only used O(n) computation.



Putting all these observation together we get an O(n) algorithm for constructing suffix trees.

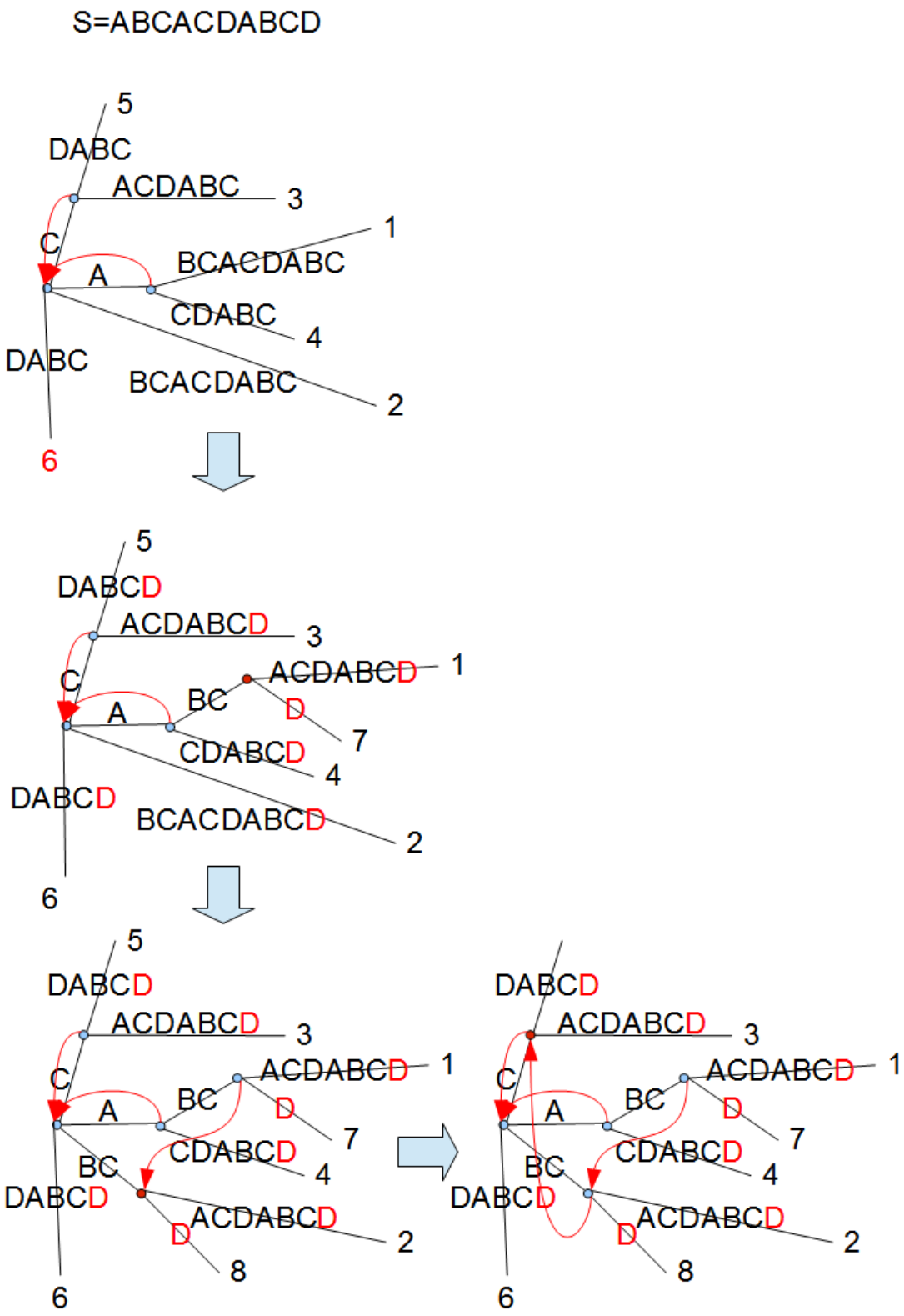
Here is an example of how the algorithm works.

In this example, we build the suffix tree for string ABCABCD. At step I we simply build the tree for the first prefix, comprising a single suffix (A), and create a first leaf in the tree. By the observation above, this leaf will not need to be updated again, rather the string will be filled in automatically. In step II, we add another suffix (B), the suffix AB having been automatically added by the leaf rule, and create a new leaf. Same deal for step III. In step IV, like in the earlier steps, we implicitly add suffixes ABCA, BCA, and CA represented at the corresponding leaves. We just need to add suffix A (the 4th suffix). As in the previous iteration we were implicitly left at leaf 3, we progress to the root of the tree and try to find a path labeled A. This path already exists in the tree and we terminate this round. The current pointer is now placed at the end of this path (character A). Note that no new leaves or nodes have been created. At step V, we again know that suffixes ABCAB, BCAB, CAB are already in the tree at the corresponding leaves. We now try to add suffixes AB and B, starting from the last position we visited in the tree (character A highlighted in red in step IV). We find that suffix AB is already in the tree and terminate this round. Note that we did not need to check if suffix B is also found in the tree as we already know this fact to be true. The pointer in the tree has now progressed to the end of suffix 4, the B character highlighted in red. Step VI is the same as step V, progressing another position down the same edge. Finally, we reach step VII where we try to add the final character to all the suffixes not ending in a leaf. We start from the position where we last ended (the red C character in panel VI), the end of suffix 4 (we started

with this suffix in steps V, VI, and VII). As we do not find suffix ABCD in the tree we create a new node and a leaf for suffix 4, then progress to search for suffix 5. We do so by using the skip-count trick we developed earlier in the context of search. Specifically we find the parent of the node (the root in this case), and progress (blindly) down the path from the root that starts with B for two characters (length of ABC minus the first character). At this position we again try to add character D to the tree, and since it is not found, we create a new node and leaf 5. The newly created node becomes the target of the suffix link from the previously created node. We then follow a similar approach (move to the root, follow path labeled C one character down) to insert the 6ths suffix (CD), and add a suffix link for the previously created node. Finally, we insert the 7th suffix (D) in the tree, and create a suffix link from the previous node to the root.

Throughout this approach we see that at each stage we either created a leaf, or terminated the execution of the algorithm as soon as we found that a suffix existed in the tree.

Note that when creating the suffix links, the node to which the link will point to may already exist in the tree (see below), however we are still guaranteed that we will visit that node immediately after creating the new node. The example below follows the insertion of character D in the suffix tree constructed for the string ABCACDABC. In the last panel, the red node already existed in the tree when we try to fill in the suffix link for the newly created node (red node in the second to last panel).



3.13 Applications of suffix trees

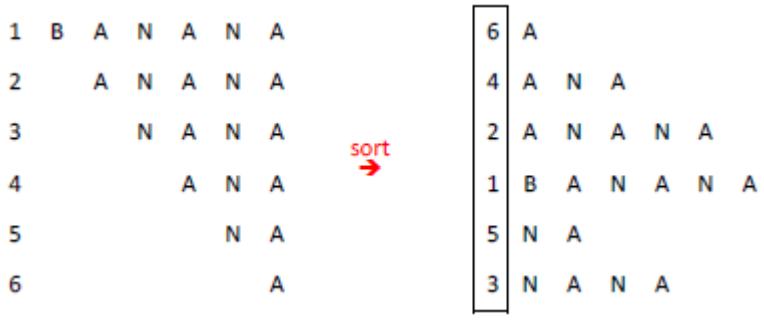
- Searching a pattern P against a text T in time O(m) (m – size of pattern, ignoring preprocessing time)
- Finding genomic repeats (sections of a genome that are exactly repeated)

(to be added)

3.14 Suffix arrays

Suffix trees appear, at first glance, to solve a wide range of exact matching problems efficiently, in both time and space consumption. The constants associated with the O(n) space consumption, however, can be quite large as the tree can be viewed as a collection of O(n) pointers, each of which can use up to 4-8 bytes. In practice, the best implementations of suffix trees use about 15 bytes per base (Kurtz 1999), compared to 2 bits/base needed to store the actual sequence. Can we store all the suffixes of a string more efficiently, and still allow nice searches?

A simple answer is the suffix array datastructure of Manber and Myers (Manber and Myers 1993). In it's most basic form, we can store all the suffixes of a string in a sorted list. We already know how to search sorted lists in O(log n) time, thus the problem may be easily solved.



Note, however, that in its basic form, this datastructure is quadratic in size as it must store all the characters in all the suffixes. Just like in the case of suffix trees we can overcome this limitation by simply storing the position of the suffix in the string (one number suffices, the starting position). The resulting structure is much more compact – O(n log n), or O(n) if we assume each position can be stored in a memory word (which is pretty much guaranteed on 64-bit machines for typical datasets).

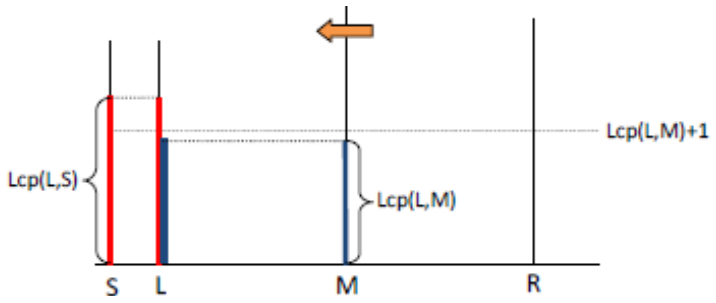
How long does it take, however, to search this datastructure? At first glance, the simple binary search algorithm would yield an O(log n) runtime. Binary search, however, assumes that each comparison is performed in a unit of time. Comparing strings requires a runtime proportional to the length of the string, i.e., in the worst case the binary search may require O(n log n) time. This is much worse than for the suffix tree.

To fix the runtime, we make a simple observation: if two suffixes are next to each other in the sorted suffix array, these suffixes likely share a common prefix. Thus, while performing the binary search we may be able to reuse some of the comparison information.

Let us formalize this idea a bit better. We will construct a 'longest common prefix' (LCP) array that stores, for every pair of suffixes the length of the longest common prefix of those suffixes. In other words, LCP[i,j] contains the largest number of characters that match exactly at the beginning of suffixes i and j of text T, or T[i .. i+LCP[i,j]] = T[j .. j+LCP[i,j]]. Let us see how the LCP array helps find a match in a suffix array for text T.

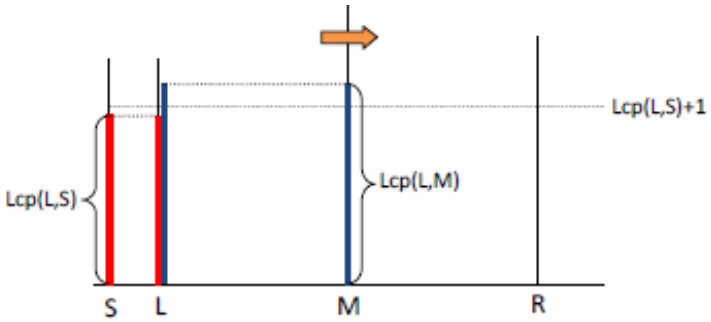
We will follow the execution of the binary search algorithm as it's trying to find if a string S matches the prefix of some suffix in T. We will pick up the algorithm in the middle of its execution. Just as a brief reminder, binary search operates by maintaining the left (L) and right (R) boundaries of the range in an array where a particular value may be found, starting with L = 0, R = n, where n is the size of the array. At each stage, the algorithm interrogates the position in the middle of this range (M = (L+R)/2) and recurses on the left [L, M]or right half [M, R] of the search interval. We will, thus, assume that we are at particular positions L, M, R in the suffix array. We will also assume that we have, sofar computed the values LCP[S, L], the longest common prefix of the query string S and the suffix at the left boundary of the current search interval. We will now compare this value with the precomputed LCP[L, M] value (remember, all LCP[i,j] values are precomputed for the suffix array). We encounter three possible situations:

1. LCP[L, S] > LCP[L, M]



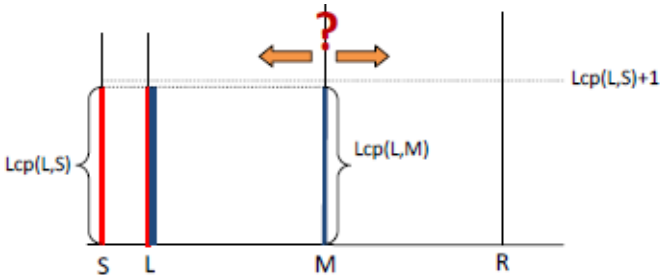
Without comparing any characters, we can now directly recurse on the left side of the search interval as we know that string S is lexicographically smaller than suffix M. Otherwise, the longest common prefix of L and M would have been longer. Another way to see this situation is to realize that S[LCP[L,M] + 1] = L[LCP[L, M] +1] < M[LCP[L,M] + 1] (the character following the LCP[L, M] prefix in S is the same in L but must be smaller than the corresponding character in M, otherwise the LCP property would be invalidated).

2. LCP[L, S] < LCP[L, M]



Using a similar argument, we now know that we have to recurse on the right side of the search interval. Here we know that S cannot occur between L and M since the all suffixes in that range share the prefix LCP[L, M]. By exclusion, S can only occur to the right of M (if S matches a suffix, this suffix must occur between L and R by induction as we start with a search interval spanning the entire suffix array).

3. LCP[L,S] == LCP[L, M]



In this case we do not have sufficient information to decide whether we should recurse on the left or right side of the array. To figure out where S may fit we need to compare the characters of S beyond LCP[L, S] = LCP[L, M] to the corresponding characters of M.

To understand why this algorithm is efficient, it suffices to notice that in cases 1 and 2 we never compare characters, rather make a decision based solely on the values of the LCP array. In case 3, we only compare characters we have not matched before, limiting the number of successful comparisons to O(m) (m = length(S)). Just as in the case of KMP, we may mismatch a same character of S multiple times, however each time we do we recurse in the algorithm. The total runtime is, thus, O(m + log(n)).

Note, however, that we have cheated a bit by ignoring the runtime necessary to construct the suffix array and corresponding LCP array.

Observation: *The suffix array of string S can be constructed in O(n) time from the suffix tree of S.*

(to be shown in class)

Ok, we cheat a bit, but at least we now have a linear time algorithm for constructing the suffix array. The LCP array appears to be more complicated. We defined this array for every pair i,j of suffixes in S, i.e., the array must contain $O(n^2)$ values and would, thus, require at least $O(n^2)$ runtime to compute. To reduce this time, we notice that we do not need to store all LCP values. Rather, we only use the LCP[L,M] or LCP[M,R] values where L, M, R are a triplet of values encountered in the execution of the binary search algorithm. It is not difficult to prove that not all pairs of coordinates will be examined. To see why, you can represent the execution of the binary search as a binary search tree. Each node in the tree corresponds to a different (L, M, R) triplet encountered during the binary search process. It is, thus, easy to see that the number of such triplets is linear (the tree has O(n) leaves, and therefore O(n – 1) internal nodes). It thus suffices to store only a linear number of LCP values.

While we solved our storage problem, we still need to compute these values in linear time.

Observation: *The LCP array can be computed in linear time using the suffix tree of string S.*

Observation: *For any triplet (L, M, R), LCP(L, R) = min(LCP(L, M), LCP(M, R))*

(to be shown in class)

The new data structure, including the suffix array itself (O(n) space) and the LCP array (O(n) as well), requires linear space, just like the suffix trees (of course a multiplicative O(log(n)) term is added if we do not assume all indices can be represented in constant space). The constants are much lower than for suffix trees, requiring just 4 bytes per base in the most basic implementations (Manber and Myers 1993, Abouelhoda, Kurtz et al. 2004). Note, however, that we have taken a hit in terms of runtime as the matching algorithm requires O(m + log(n)) time, as opposed to O(m) in the case of suffix trees.

3.14.1 Constructing suffix arrays without relying on suffix trees

The initial algorithm described by Manber and Myers for constructing suffix arrays did not rely on suffix trees. Instead, this algorithm used an elegant version of radix sort that we will sketch here. For the full details please refer to the suffix array paper (Manber and Myers 1993).

Our goal is to sort in lexicographic order all the suffixes of a string. While sorting takes O(n log(n)) time, we are dealing with strings and also have to account for the time necessary to compare two strings, leading to an $O(n^2 \log(n))$ algorithm if implemented naïvely. Note that instead of actually sorting the suffixes, it suffices to assign each suffix an index indicating its position in the sorted list. We will build this index array I throughout the execution of the algorithm.

At a first stage we simply perform a radix sort operation using the first character in each suffix. More precisely, we assign the array I with values 0, 1, 2, or 3 depending on whether the first character in each suffix starts with A, C, G, or T, respectively. We have, thus, implicitly broken up the set of suffixes into a collection of buckets (corresponding to the distinct values of the I array) such that all suffixes in each bucket share the same first character. At the same time we have also filled the first two bits of the final values of the I array. We will now continue to further break up the initial buckets while filling in more information in the I array. At each stage we will double the number of bits known for each value in the I array. Within each bucket we will also double the size of the prefix shared by all suffixes in the bucket. To do so we rely on a simple observation. Let us assume that at stage k we have broken up the array into buckets such that all suffixes assigned to a same bucket share a prefix of length l. The order in which these suffixes occur in the final suffix array can be determined by examining the bucket membership of suffixes i + l. To clarify this point, assume we are sorting the suffixes of the string GATTACA. After the first stage, the suffixes will be grouped into buckets:

0 – A, ACA, ATTACA

1 – CA
2 – GATTACA
3 – TACA, TTACA

The order of the three suffixes in bucket 0 is simply determined by the second character in each of these suffixes, which is implicitly encoded in the bucket membership of the subsequent suffixes. For example, we know ACA occurs before ATTACA because suffix CA is in an earlier bucket than suffix TTACA.

Thus, at each stage k, we can identify the bucket structure by simply concatenating to the I value of each suffix i, the I value from the previous round for the suffix $i + 2^{k-1}$ (assuming we start with stage 0). Here is the algorithm exemplified on the GATTACA example.

Stage 0:

I	Suffix
2	GATTACA
0	ATTACA
3	TTACA
3	TACA
0	ACA
1	CA
0	A

Stage 1: (note that now all suffixes with the same I value share two characters)

I	Suffix
20	GATTACA
03	ATTACA
33	TTACA
30	TACA
01	ACA
10	CA
00	A

Stage 2:

I	Suffix
2033	GATTACA
0330	ATTACA
3301	TTACA
3010	TACA
0100	ACA
1000	CA
0000	A

Stage 3:

I	Suffix
20330100	GATTACA
03301000	ATTACA
33010000	TTACA
30100000	TACA

01000000 ACA
10000000 CA
00000000 A

Thus, after just 3 rounds we have managed to sort all the suffixes (though in this case their order did not change after round 1). In general it is easy to see that we will have to execute O(log(n)) rounds, and at each round we only spend constant time for each suffix (look up I value for suffix $i + 2^{k-1}$ and append to current I value), for a total runtime of O(n log(n)). While this runtime is slower than the O(n) time for constructing a suffix tree, the entire procedure requires very little extra memory (just the I array). With a bit of extra work we can also fill in the LCP array at the same time as computing the I array.

Note that the way we describe the algorithm above, the number of bits added to the I array is proportional to the length of the string, still yielding an $O(n^2)$ memory footprint. In practice (though harder to explain), the algorithm simply maintains the boundaries of the various buckets within the suffix array and sorts this array in place, not requiring additional memory.

Important note: The suffix array, as described, seems to be a big improvement over the suffix tree: for a slight penalty in search runtime we can achieve a much lower footprint. Note, however, that suffix arrays cannot easily solve the 'longest common superstring' problem due to the absence of the suffix link information. Such information can be, however, added to the suffix arrays in a structure called an Enhanced Suffix Array (Abouelhoda, Kurtz et al. 2004).

3.15 Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) is a datastructure originally developed in the context of data compression (Burrows and Wheeler 1994). We will illustrate this datastructure with a simple example for the string BANANA. Just like in the case of suffix trees, we'll add character $ at the end of the string. We will also assume that this character is lexicographically less than any of the other characters in the alphabet.

The following description is primarily for the purpose of explanation and is not the way the BWT is computed in practice.

The first step in the procedure is to create a table that contains all circular rotations of the original string:
BANANA$
ANANA$B
NANA$BA
ANA$BAN
NA$BANA
A$BANAN
$BANANA

In a second step, we lexicographically sort this array:
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

Finally, we retain the final column of the array:

$BANANA**A**
A$BANA**N**
ANA$BA**N**
BANANA$ => ANANA$**B => ANNB$AA**
BANANA**$**
NA$BANA**A**
NANA$B**A**

Now you can get a bit of intuition about why the transform can be used for compression: the final string appears a lot more compressible (strings of a same character can be easily compressed with simple techniques such as run-length encoding).

Also note that while the construction approach described above appears to be quite convoluted, there is a very natural connection between the Burrows-Wheeler transform and suffix arrays. Specifically, if you remove all characters after the $ in the sorted table you have:

$
A$
ANA$
ANANA$
BANANA$
NA$
NANA$

or exactly the suffix array of the string. The Burrows-Wheeler transform itself is the string of characters preceding each of the suffixes. This observation also points out to a way for computing the Burrows-Wheeler transform without needing to build the $O(n^2)$ sized table.

Coming back to the Burrows-Wheeler transform, we have so far presented a simple approach for taking a string, scrambling its letters, and obtaining a new string that is more compressible. This approach would be completely useless if we didn't have a way to unscramble the string, a task that is by no means obvious. Let us examine the sorted BWT table more closely.

**$** BANANA**A**^{1}
A^{1}$BANAN**N**^{1}
A^{2}NA$BAN**N**^{2}
A^{3}NANA$**B**
B ANANA**$**
N^{1}A$BANA**A**^{2}
N^{2}ANA$B**A**^{3}

The last column is the Burrows-Wheeler transform. The first column is rather un-interesting – it is simply a sorted list of all the characters in the string. Let us number the occurrences of each character in this column (shown with superscripts above). A careful examination of the table shows that the same characters occur in the same order in the last column. This property allows us to undo the transform and reconstruct the original string from just the transformed string. In other words, we can reverse the compression and can thus get a useful compression algorithm (incidentally, the bzip2 unix utility uses this algorithm).

To prove this 'last-first' property it suffices to look at the order in which the characters occur in the last column. Let us focus on just the As (a similar argument can be made for any other string of a same character). Note that A^1 is the first character in suffix 6, A^2 is the first character in suffix 4, and A^3 is the first character in suffix 2. The order in which these characters occur in the last column is determined by the order of the subsequent suffixes in the table: suffixes 7, 5, and 3, respectively. Now, shifting our attention to the first column, the same

three characters occur in the same order because the corresponding strings all start with an A followed by the same three suffixes (7, 5, and 3). In other words, the two orderings are the same in both columns.

Before we describe the decompression algorithm, note that we need to have available to us the first and the last column of the Burrows-Wheeler table. The last one is the Burrows-Wheeler transform itself, while the first one can be easily computed in O(n log n) time by sorting the individual characters in the text. Now, the algorithm simply proceeds by starting with the first column in the first row (starting with $), thus identifying the last character in the original string. The last character in the same row (A) is the second to last character in the string. To continue this algorithm we need to identify the same character in the first column, operation that is now easy based on the earlier observation – we simply need to find the first A in the first column, proceed to the last column (now finding an N, the third to last character), and so on.

3.16 The FM-index and the compressed suffix array – memory efficient exact matching

The Burrows-Wheeler transform has so far been useful for compression – can we, however use it for matching? Clearly, the suffix array is implicit in the Burrows-Wheeler transform. What additional information do we need to perform matching with the BWT? Note that performing matching with the BWT has the potential to give us a very efficient algorithm – the BWT is truly O(n) in space and doesn't even incur the potential log(n) overhead for representing the suffix coordinates.

To sketch the search algorithm, let us focus on the BANANA example and try to find strings NAN and CAN within this text. We will proceed in the same fashion as we did before, moving from the end of the pattern being matched. When 'unrolling' the transformed string we knew where to start the process, specifically at the $ character. For pattern search, instead, we will simply find all the suffixes that start with N (the last character in the pattern):

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

We will then examine the last character of these suffixes and retain just those whose last character matches the second to last character of our string. In this case both rows we identified end with the same character (A) which also matches the string. To proceed we must identify the location in the first column where these characters occur (the second and third As respectively):

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BAN**A**
NANA$B**A**

Again, we check the last character in the corresponding rows and determine that the string NAN exists in our text (one of the rows ends in N) while the string CAN does not (none of the last characters are C).

In essence the procedure we follow is very similar to a binary search procedure – the search range is always represented in the first column as a contiguous range of characters, all matching the current position reached

in the pattern. At each stage we update this range by selecting from the last column the characters that match the next position in the pattern.

Note that at any stage in the process we need to find the current character in the pattern within the last column of the Burrows-Wheeler table, a process that may require O(log n) time, leading to an O(m log n) time runtime for our algorithm. To accelerate this process we can record a simple type of information – for each row we will record the number of each letter preceeding that row (see below):

	ABN
$BANANA	1 0 0
A$BANAN	1 0 1
ANA$BAN	1 0 2
ANANA$B	1 1 2
BANANA$	1 1 2
NA$BANA	2 1 2
NANA$BA	3 1 2

This simple information allows us to identify the right range in the first column by simply looking up the current character in the pattern at the boundaries of the current range. For example, when looking for A after having matched N (the first stage described above), we notice that the values of the 'previous' array for A are 2 and 3, indicating that the new search range corresponds to the second and third As in the first column. Similarly, after having matched AN, we find that only one N character occurs within the search range and that it is the second N in the array. This simple approach results in a runtime of O(m) as we no longer have to perform binary search to find the characters in the array.

Question: *The algorithm described above can easily find whether a match occurs as well as count the number of matches. How can we determine the location of the matches in the text?*

The simple answer (to be discussed in more detail in class) is that we can unroll the text from the location where a match is found in order to identify its position in the text, thereby incurring an additional cost of O(n). This cost can be overcome by simply storing the coordinate of each suffix as well (in addition to the table of character counts), allowing us to simply look up the position of the match as soon as we determined that a match occurred.

3.16.1 The FM-index – balancing memory and speed

As hinted above, the basic use of the Burrows-Wheeler transform in matching can be slow, requiring O(m log(n) + n) time to find (using binary search) and identify the location of matches. Speeding up the process requires the use of additional memory which essentially transforms the datastructure into a suffix array (of size O(n log(n)), thereby offsetting the main memory advantage of the Burrows-Wheeler transform. Can we find a balance between these two extremes?

The basic idea is to only record the additional information (character counts, suffix array information) for specific rows in the Burrows-Wheeler table. Assume we store this information for every *b*th row. Our algorithm can simply look up the necessary information at bucket boundaries, but needs to spend O(b) time to compute the information within a particular bucket (bounded by the rows where we recorded the information). The total memory requirement is thus reduced to O(n log(n) / b), while the runtime is slightly increased to O(m b + b) for matching a pattern of length m. By varying the size of the bucket b we can, thus, choose a different space versus speed tradeoff. This basic approach is the key idea behind the FM-index (Ferragina and Manzini 2000), a search datastructure built upon the Burrows – Wheeler transform.

In the FM-index, the bucket size was chosen to be b = log(n) in order to minimize storage (which becomes O(n)) without affecting runtime too much (O(m log(n) + log(n)). An FM-index representation of the whole

human genome can occupy less than 3 Gbp (less than 1 byte per base) as opposed to about 12 Gbp for a suffix array or 60 Gbp for a suffix tree.

Also, the FM-index introduces an additional 'trick' that can reduce both memory and runtime at the same time. As mentioned earlier, the Burrows-Wheeler string is inherently more compressible. The FM-index uses this property to compress the string within each bucket, thereby further reducing the space requirements. Interestingly, depending on the exact compression technique used, operations such as finding the number of characters matching the pattern within a bucket can be performed more efficiently as well. For example, by using run-length encoding the character counts can be computed directly within the compressed string. Since the runtime depends on bucket size, the computation can, thus, be sped up by compressing the buckets.

Note: In the original FM-index paper they used a combination of 'move to front' compression and run-length encoding, combination that the authors argue is close to the information theoretic lower bound for compression. The FM-index is, thus, a nice example of a compressed datastructure that allows searches without requiring one to uncompress the data – imagine being able to search inside a .gz file. This line of research is likely to become 'hot' as the computational biology community is trying to deal with the large amounts of data becoming available.

Note: One of the most 'famous' applications of the FM-index is the sequence aligner Bowtie (Langmead, Trapnell et al. 2009), a tool developed by Ben Langmead during his graduate studies in the Computer Science Department at the University of Maryland, College Park.

3.17 Exercises

1. Remember that a suffix tree is a compressed representation of all suffixes in a string, such that each suffix is represented by a different leaf in the tree. The least common ancestor of two nodes in a tree is the lowest node shared by the paths from the two nodes to the root. Assume n is the least common ancestor of leaves i and j in a suffix tree for string S.

   a) What does this node represent?
   b) Describe an algorithm that will compute the Z values for S, using the suffix tree.

2. Construct the suffix tree for the string: ATATCCATCAT$ Please label each leaf with the corresponding suffix sequence. Also indicate the suffix links.

3. Write the pseudo-code for a recursive function that prints the suffixes (just the labels on the leaves) from a suffix tree in lexicographically sorted order.
4. Describe how you would use a suffix tree to compute the sp(i) values for the KMP algorithm. How would the algorithm change if you tried to compute the sp'(i) values
5. What is the string corresponding to the Burrows-Wheeler Transform shown below? Only the first and last columns from the BWT table are shown.
   actgcag$ta
6. Assume we are trying to find whether a longer string L can be imperfectly matched to one or more locations of G. Many heuristic algorithms start by searching for exact matches of short substrings of L – task for which one can use the algorithm described in point a). Assume, however, that instead of consecutive characters we want to use a spaced-seed approach to find potential matches between L and G. This problem becomes a variant of matching with don't care symbols – i.e. the string S is composed of a mixture of letters that must exactly match corresponding letters in G, and the don't care symbol * that can match any character of G. An example is: ACA*G**GA.

   Can the Burrow-Wheeler search algorithm be modified to allow don't care symbols?

7. We showed how to use a suffix tree for a small string P to compute matching statistics $ms_T(i)$ for each position i in the long text string T, i.e. the length of the longest substring of T starting at i that matches some substring of P. Suppose we also want to compute matching statistics $ms_P(j)$ for each position j in P. The number $ms_P(j)$ is defined as the length of the longest substring starting at position j in P that matches some substring in T. Show how to find all the matching statistics for both T and P in $O(|T|)$ time, using only a suffix tree for P.
8. For any pair of strings, we can compute the length of the longest prefix common to the pair in time linear in their total length. Suppose we are given k strings of total length n and want to compute the minimum length of all the pairwise longest common prefixes over all of the $\binom{k}{2}$ pairs of strings, that is, the smallest length of the pairwise pairs. The obvious way to solve this is to solve the longest common prefix problem for each of the pairs in $O(k^2+kn)$ time.

   a) Show how to solve this problem in O(n) time, independent of k, using a suffix tree.
   b) How would you solve the converse problem of finding the maximum length among all pairwise longest common prefixes?
9. Given two strings (s1 and s2), design an efficient algorithm that identifies the shortest substring of s1 that is **not** a substring of s2.

4 Inexact alignment

So far we have only discussed situations where the two strings we align to each other match exactly (with the exception of the alignment with wildcards discussed under the Aho-Corasick algorithm). What if we want to allow for more extensive differences between the two strings? The differences we may allow are simple 'typos' such as characters inserted or deleted in one of the strings, and also allowing characters to be aligned to mismatching characters. How can we define a good alignment, and how can we compute it?

4.1 The edit distance and its computation with dynamic programming

In biological data we encounter a number of 'errors' that require us to find less than perfect alignments between two sequences. Assume for example that we are searching for string ABBA. We might encounter the following situations. The characters aligned to each other indicate how we might 'edit' one string to make it look like the other

ABBA vs ABA (insertion in ABBA, deletion from ABA)
ABBA
AB-A

ABBA vs ABCA (substitution)
ABBA
ABCA

ABBA vs ACA (insertion/deletion and substitution)
ABBA
A-CA

ABB-A
A—CA

Essentially we can view the problem of inexactly aligning one sequence to another as an editing operation – changing one string by inserting, deleting, or substituting characters until it perfectly matches another one. As it is clear from the above, there are many ways we could perform such edits. Clearly finding an algorithm that reports all such possible alignments is impractical, both computationally and in terms of the interpretability of the results. For simplicity, we will focus for now on the task of finding just the 'best' match between two sequences defined in terms of the number of edits that need to be performed.

Aside: Often times we (as computer scientists) like to focus on well defined optimization problems even though we do not fully understand what objective function we need to optimize in order to match a specific biological 'meaning' (which itself is not too well defined a concept). An emerging and promising area of research is the development of approaches that enumerate some sub-optimal solutions as well, allowing additional biological information to be brought in to identify the most biologically-relevant solution even when computationally modeling such biological information is impractical.

Coming back to the inexact alignment problem, we'll define the 'edit distance' between two strings to be the minimum number of insertions, deletions, and substitutions necessary to convert one of the strings into the other. This distance is also called the 'Levenshtein distance' based on the scientist that studied it in the 1960s (i.e., before we could sequence DNA). To compute this distance we will rely on a programming paradigm called 'dynamic programming'. Specifically, let us assume that we have two sequences S1 and S2 and that we

have managed to align to each other the ith and jth prefix of the two strings respectively. In other words, we assume that we know E[i, j] = edit distance between S1[1..i] and S2[1..j]. We will now attempt to find the edit distance for longer prefixes of S1 and S2, and proceeding by induction, we will eventually obtain the edit distance between S1 and S2.

We can distinguish two situations:

1. S1[i + 1] == S2[j + 1] (the characters after the already aligned prefixes)
In this case, we can set E[i + 1, j + 1] = E[i, j] – we simply extend the previous alignment for free (no edit necessary)

2. S1[i + 1] != S2[j + 1]

In this case it's not immediately clear how to align longer sequences and we can distinguish three more situations:

2a. E[i + 1, j + 1] = E[i, j] + 1 (simply 'pay' for a mismatch)

2b. E[i + 1, j+1] = E[i, j + 1] + 1 (S1[i+1] is deleted from S1 or inserted into S2)

2c. E[i+1, j + 1] = E[i + 1, j] + 1 (S2[j + 1] is deleted from S2 or inserted into S1)

We choose among these 4 options by picking the one that leads to the smallest score (min(1, 2a, 2b, 2c)) as we try to minimize the number of edits.

It is easy to see that we can represent the E values as a two dimensional array of size n=length(S1) x m = length(S2), and that the 4 cases outlined (which we will call 'recurrence equations') above allow us to fill in this array starting with just its boundaries. This brings up the question of what these initial conditions should be. Specifically, can we compute E[0,j] and/or E[i,0]? Note that these represent the alignment of the prefixes of one string against an empty string. Under our definition of edit distance the alignment score we can set E[i, 0] = i, i.e., all the characters have been deleted.

Based on the description above you can see that the score of the best alignment between S1 and S2 is stored at location E[n, m] in the table. But what is the exact pattern of edits implicit in this score? Note that each of the recurrence equations corresponds to a specific edit, and when computing the value E[i + 1, j + 1] we implicitly select one of the edits (the edit operation that minimizes the value). We can use this information to work back from the best score and reconstruct one of the alignments corresponding to that score. Specifically, this **backtracking** operation starts by asking for E[n, m] whether its value was derived from equation 1, 2a, 2b, or 2c, corresponding to a match, mismatch, deletion from S1, or deletion from S2, respectively. We then repeat this process from the relevant prior location in the matrix (E[n-1, m-1] for cases 1 and 2a, E[n-1, m] for case 2b, and E[n, m-1] for case 2c).

[example to be added]

Above we assumed all edits have equal cost. The recurrence equations can, however, be modified to allow an arbitrary cost for each operation as follows. Assume we are given $f : \Sigma \cup \{'-'\} \rightarrow \Sigma \cup \{'-'\}$ a function that assigns a cost to the alignment of any two characters in the two strings, or of a character to the '-' symbol implying a deletion operation. The equations become:

1. E[i + 1, j + 1] = E[i, j] + f(S1[i + 1], S2[j + 1]) # match

2a. E[i + 1, j + 1] = E[i, j] + f(S1[i + 1], S2[j + 1]) #mismatch

2b. E[i + 1, j+1] = E[i, j + 1] + f(S1[i + 1], -) # deletion from S1

2c. E[i + 1, j+1] = E[i + 1, j] + f(-, S2[j + 1]) # deletion from S2

The value for E[i + 1, j + 1] will be selected as the maximum (or minimum) of the four values listed above

depending on whether the function f is a penalty (e.g., f(A, A) = 0, f(A, -) = 1, ...) or a reward (f(A, A) = 1, f(A, -) = -1).

Note: A nice animation of this algorithm is available at: http://statgen.ncsu.edu/slse/animations/module1.html

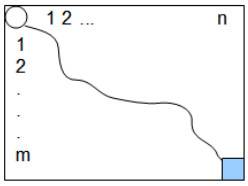
4.1.1 Local versus global alignment

The algorithm described above requires that the two strings match fully, i.e. it constructs a **global** alignment. We can easily modify this algorithm to relax this requirement. For starters, let us try to find the best prefix-prefix alignment between the two strings, i.e. we will identify the prefixes of S1 and S2 that, when aligned, have the best edit distance score from among all possible such prefixes. It should be easily seen that we already have the answer. The value at each location in the dynamic programming table is exactly the score of aligning some prefix of one string to a prefix of the other one (by definition), thus solving the prefix alignment problem simply relies on finding the best score in the entire table (rather than the score E[n,m]).

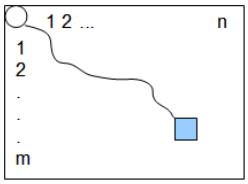
How about converting this algorithm to find the inexact version of the longest common substring problem, i.e., find the substrings of each of the strings that align to each other with the best score? The basic intuition is that we want to allow the path through the dynamic programming table to begin elsewhere than the top left corner (see figure below). Note, however, that the way the initial values are set up, we are penalized for omitting characters from the beginning of one or the other string, penalty that we can eliminate by simply converting the first row and column of this table to 0s. The problem is still not fully solved as we would still pay a penalty for any mismatches or indels within the table, penalty that we can eliminate by simply adding '0' as an option in the dynamic programming recurrence.

To summarize, we can solve the **local alignment** problem simply by filling the initial row and column of the dynamic programming table with 0s, and by using the following recurrences:

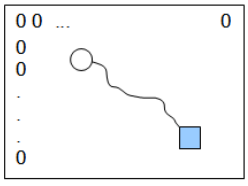
$$E[i+1,j+1]=max\begin{Bmatrix}0\\E[i,j]+f(S1[i+1],S2[j+1])\\E[i,j+1]+f(S1[i+1],'-')\\E[i+1,j]+f('-',S2[j+1])\end{Bmatrix}$$



Global alignment



Prefix alignment



Local alignment

4.1.2 Affine gap penalties

As described above, the edits are completely context independent. In some cases it is important to construct more complex context-dependent rules. We exemplify this idea in the context of gaps (or indels). In the

original formulation of the dynamic programming approach the cost of the following alignments is the same:

```
ABBBA
A--BA
A-B-A
```

They both incur the cost of two deletions. In some cases, however, we may want to penalize the non-adjacent gaps more (the second alignment), as the biological 'cost' of inserting a string of gaps is lower than that of inserting gaps at different places in a sequence. A good example is the case of the eukaryotic gene structure - in mRNA the gene is a contiguous sequence, while in DNA the gene is a collection of exons separated by introns. The latter get 'spliced' out during transcription. An alignment of the RNA sequence to the DNA should, thus, not be penalized for the length of the gaps corresponding to the 'missing' introns.

The dynamic programming algorithm can be modified to take into account more complex gap scores, either fully generically – e.g., a function $g(n_{gaps})$ which gets incorporated in the score depending on the number n_{gaps} of adjacent gaps in the alignment.

Incorporating such information blows up the runtime to $O(n^3)$ as we will see later. A less general gap penalty, but which can be computed more efficiently, is the affine gap penalty, where the score of a string of n_{gaps} can be represented as $g_{open} + n_{gaps}\, g_{extend}$, where g_{open} and g_{extend} are penalties for, respectively, opening and extending a string of gaps.

Historical note: Alignment started with proteins (which we could 'sequence') in the 1970s. Originally the alignments were global because known proteins were globally related. Once new proteins were discovered that had local similarities but no global ones scientists started developing approaches for finding such local alignments. The approach that eventually got adopted as standard is the Smith-Waterman algorithm (the local alignment algorithm described above).

4.2 Sequence alignment statistics

The sequence alignment problem can be viewed as the interplay of three main considerations:

1. The definition of what is a good alignment (generally modeling some biological meaning)
2. Algorithm to compute a good alignment
3. Statistical arguments to determine whether an alignment can be explained by chance

The three considerations are in tension: the most biologically relevant score may be intractable to compute or may entail intractable statistics, while tractable definitions may not be closely related to biology.

Above we described algorithms for finding an inexact match between two strings and assumed we had some arbitrary scoring function that determined the score of aligning or mis-aligning characters to each other. In this section we delve a bit more in the statistical meaning behind those scores.

Throughout the following we will focus primarily on local alignment as the statistics are well defined and easier to reason about. Such a statistical framework does not exist for global alignment.

We will focus on two main questions: (i) how to define the alignment scores; (ii) how to estimate the probability that a particular alignment can be explained by chance.

The latter question is particularly relevant in the context of database searches. Assume we have a new biological sequence (for now we'll assume we are looking at protein sequences) and try to find a similar sequence in a database. Assume our alignment algorithm finds an alignment of score S between this sequence and another sequence in the database. How do we know that the alignment we found is 'real', i.e., it is not due simply to chance. This question falls into the category of statistical hypothesis testing where we are asking whether a particular observation belongs in a particular distribution. The null hypothesis assumes that the

observation is indeed derived from the distribution, and statistical tests can tell us whether this null hypothesis is invalidated. In the context of alignment, the null hypothesis is that our sequence found a match simply by chance. Rejecting the null hypothesis would mean that the match found is 'surprising', i.e., the matching sequence is probably related to our query.

4.2.1 Significance of an alignment

Before we proceed it is important to realize that we try to find a scoring function that has some biological meaning, in order to insure that our alignments are biologically correct. By necessity, to make the statistics tractable, we will make some simplifying assumptions which we will relax later. For now we will assume that the alignments have no gaps.

The scores are represented by a 20x20 matrix that contains the score of aligning two amino-acids to each other. These scores are defined statistically - they are deviations from chance that two amino-acids are aligned to each other. We need a definition of background probabilities and then try to find a string of letters that are aligned in a 'surprising' way.

We will describe how these scores are computed later. For now, we will focus on the properties we might want the scores to have. It may not be immediately obvious from the description of the dynamic programming algorithm, but the type of alignment we find depends strongly on the scoring function. With a small change we can, for example, change the dynamic programming algorithm to only find exact matches (an overkill given that we already know how to do it more efficiently). We will require the following two properties from a 'good' scoring system:

- The expected score is negative. This property is useful for the math that will follow, but also ensures that we can find real alignments among the noise. If scores are largely positive, the alignment algorithm would generally produce very long alignments even when aligning random sequences, as simply adding more characters to the alignment will increase the alignment score.
- At least some of the scores are positive. This property is useful for insuring we will find at least some alignments. If all scores are negative, the Smith-Waterman algorithm would always select the null alignment of score 0.

Any scoring system satisfying these properties can be written as $s_{i,j} = \left(\ln \frac{q_{i,j}}{p_i p_j} \right) / \lambda = \log \frac{q_{i,j}}{p_i p_j}$. This property can be proven as shown in the lecture slides. The values $q_{i,j}$ are called the **target frequencies** and we will discuss them later. The λ value is simply a positive scaling factor that will not affect the actual alignment algorithm but it is useful for defining the statistics. The value of this parameter can be computed analytically as described in the slides.

We will now focus on the question of aligning two random sequences. To perform the statistical hypothesis test described above, we need to find, for each score S, the expected number of independent local alignments that are greater than or equal to this score E(S, m, n). The product of the lengths of the two strings N = m x n is the size of the **search space**.

Simply focusing on intuition we can see that E(S, m, n) ~ mn (if you double the space you double the number of alignments expected). How does E(S, m, n) vary with S? Again intuition indicates that the expectation of the number of random alignments must decrease with increasing S (the higher the score, the fewer such alignments), and that it will decrease exponentially. The intuition is based on a simple coin-flipping experiment where we try to count the number of heads at the beginning of a series of flips. The likelihood of a string of heads decreases exponentially with the length of the string.

Thus, we can expect that $E(S, m, n) \sim e^{aS}$ where a is a constant.

One can prove mathematically that $E(S,m,n)=K\,mn\,e^{-\lambda S}$ for a constant K, however this proof is very complex. Note that this expectation can be easily interpreted, even if the whole math underlying it is complex. Also, note that this formula is asymptotic – it approaches this value for large values of m and n. For small values of m and n additional corrections are necessary.

The expectation (or E-value) is just one part of the equation – the statistical hypothesis test relies on a p-value – the probability that an alignment with score higher than S exists (remember, E is simply the number of such alignments). Using Poisson statistics it can be shown that $p=1-e^{-E}$. For small values of E, p ~ E.

Also note that the formulas above are parametrized by parameters K and λ, parameters specific to the scoring function which must be computed in order to compute the statistics. To simplify the equations we can simply fold the parameters into the score, obtaining a new score $S'=(\lambda S-\ln k)/\ln 2$. Now E(S, m, n) can be written as $E(S',m,n)=\frac{mn}{2^{S'}}$, with S' represented in bits (also called the **bit score**). This score can be easily interpreted, as increasing the score by one bit results in a halving of the number of random alignments.

The highest score follows an extreme value distribution, a property that will be useful in dealing with gaps. As we already mentioned, we focused on a simplified version of the alignment problem where we allow no gaps. The dynamic programming algorithm we described earlier can tolerate gaps, and we can clearly add a gap scoring function to that alignment. It turns out that this situation can also be modeled statistically in a similar way to the description above, however this fact is only evident empirically and has not yet been proven analytically.

We need to make sure gaps are penalized enough otherwise we can always get a good alignment by simply stringing together high scoring letter pairs with long stretches of gaps. There is a transition point in the gap score beyond which the gapped alignments behave just like a local alignment. The transition point can be found experimentally, but there's no mathematical proof that allows one to calculate it.

For a large enough penalty we have the same general statistics but we do not know how to calculate the values of λ and K. Instead we can estimate these parameters using the following approach. We run multiple gapped alignments (many different random sequences), find the highest score, then fit an extreme value distribution to the observed values. From the fitted distribution we can now calculate λ and K.

All our theory so far assumes a model of proteins as random strings of letters. Proteins, however, have sections that do not behave like the random model in part due to the actual structure of the protein. In globular proteins the random model is pretty good. For more 'unusual' proteins the statistics will not work, e.g., the expected scores in certain regions are positive. Some such examples are proteins with long 'tails' comprised of a small repeated patterns. Database searches try to screen out such regions because the statistics don't work, and in some sense the alignment idea itself does not make sense in these regions anyway.

4.2.2 Alignment scores

As pointed out above, any 'good' alignment score can be written as a log odds: $s_{i,j}=\log\frac{q_{i,j}}{p_i p_j}$. This is true for any score, whether or not it captures a deeper biological meaning. In this section we will focus on how to tie more closely the scores to actual biological information. We will do so by defining the target frequencies $q_{i,j}$.

To get a better intuition of what we are looking for, note that the distribution/histogram of alignment scores for random alignments has an exponential distribution – the higher the score, the fewer random alignments with that score. For a search space $N=mn$, we expect at most one random score to exceed $\log N$. This property holds for all possible scoring systems. We will try to find a scoring system that maximizes the scores for true non-

random alignments, i.e., a score that best separates true matches from random matches. We will use a score that relies on the frequency at which amino-acids align to each other in 'true' alignments, specifically we will set the target frequencies $q_{i,j}$ to be the observed frequency with which amino-acid i is aligned to amino-acid j in a true alignment. Note that this definition is intuitively correct – if the frequency of aligning the amino-acids is larger than expected by chance ($q_{i,j} > p_i p_j$), the corresponding score $s_{i,j}$ will be positive.

Note that the idea of 'aligned amino-acids in true alignments' is not a coherent concept. The alignments and amino-acid pairing frequencies depend on the evolutionary distance between the two proteins. Thus, the scoring system must be adapted to a specific degree of evolutionary divergence.

Deriving scoring matrices from alignments which themselves are constructed with the use of scoring matrices entails a certain level of circularity. This circularity was broken by Margaret Dayhoff – she focused on proteins that are only 15% diverged. Due to the high similarity, the alignment can be computed correctly irrespective of the scoring function. She then modeled evolutionary change by modeling the frequency with which amino-acids could mutated into each others. From these evolutionary changes she could extrapolate out to other evolutionary distances, leadin to the PAM model (Point Accepted Mutations). A point mutation is a mutation at a single position. An accepted point mutation, is a mutation that survived evolution. The basic unit, starting point for the PAM model, is 1 PAM – evolutionary distance such that on average there is only one mutation per 100 sites.

The best PAM matrix, that consistent with the evolutionary distances most interesting to biologists at that time, was PAM 250 (proteins ~20% identical) which became widely used.

Heninkoff and Heninkoff sought to avoid the extrapolation to large PAM distances (certain phenomena, such as slowly evolving amino-acids, cannot be accurately modeled from short-term data) by looking directly at certain levels of evolutionary divergence to avoid being biased by slow/fast evolving rates. Again, they ran into the circularity issue. To overcome this issue they relied on large multiple alignments of sequences at a certain level of divergence, and identified blocks of good alignments from which substitution rates could be estimated. This approach led to the creation of the BLOSUM matrices. BLOSUM-62 is the most widely used and is roughly equivalent the PAM-180.

One question that arises is whether we can estimate the average substitution score, i.e., the contribution to the alignment score of a single aligned letter. From this score we can estimate the score of an alignment based on its length. For this purpose, we will introduce the concept of **relative entropy.** Suppose two sequences are diverged by 100 PAMs and we compare them with matrix PAM-100 – the average score can be expressed as:

$H = \sum_{i,j} q_{i,j} s_{i,j} = \sum_{i,j} q_{i,j} \log \frac{q_{i,j}}{p_i p_j}$ This is the relative entropy. If using $\log_2$, the entropy is expressed bits per position. Note that this is the average score of *true* alignments and it can be shown to be positive. Replacing $q_{i,j}$ with $p_i p_j$ we get the score for *random* alignments, score which is negative.

We can plot H for different scoring functions (PAM distances, see slides). The further the sequences diverge the less information we get from the alignment. If we want our alignment to appear above statistical noise, we can estimate the number of bits necessary (30-40 bits, depending on database size). We can simply divide the relative entropy by this number to find out the minimum alignment length needed before we can separate noise from signal.

Note that in general we do not know what divergence level to look for. What if we use the wrong matrix? We can define the efficiency of a matrix at a particular evolutionary distance (e.g., PAM 150 at 100 PAMs distance). The efficiency is the proportion of the score you obtain with wrong matrix compared to the score that you would get with the correct matrix (see slides for example).

The concept of efficiency allows us to understand why we use the BLOSUM-62 matrix. Assume, for example,

that we have sequences that are highly similar. The corresponding alignment would carry a lot of information irrespective of scoring function, thus the true alignment will be easily distinguished from noise. Conversely, poor alignments will almost always be missed, irrespective of scoring function. The precise use of the scoring function matters only for alignments that have just enough information to be differentiated from noise (~30-40 bits). This region, the 'twilight zone' of alignments, falls exactly in the BLOSUM-62 (or PAM-180) range for typical alignment lengths biologists are most interested in (~100 or so amino-acids), thus BLOSUM-62 is the matrix that leads to the least loss of information for such alignments.

All that we have discussed above can be extended to DNA alignments. We can come up with an evolutionary model for DNA (just like Dayhoff's model) and use it to construct PAM matrices for matches/mismatches, allowing us to perform alignments the same way as for proteins.

Note that we can compare DNA sequences at the DNA level or at the protein level (the translated search – simply translate the DNA into an amino-acid sequence and compare the amino-acid sequences). Which approach is better? We can estimate the information in DNA-level and protein-level alignments. At close evolutionary distances (highly similar sequences), we DNA alignments carry more information. This can be easily seen from the fact that an exact alignment of two codons (3 base-pairs) contains 6 bits of information as opposed to just about 4 for a pair of amino-acids. At longer evolutionary distances (> 50 PAMs), however, we get more information in the protein sequence.

Note that the observations above apply to naïve alignments of DNA sequences and more complex alignment strategies can recover more information. There are, however, computational tradeoffs to be taken into account. For example, translated DNA searches multiply the runtime by a factor of 6 (three reading frames on the 'forward' strand, and three on the 'reverse' strand). Similarly, 'threading' – an alignment for protein sequences that takes into account structural information in addition to sequence information – is more accurate biologically, but is NP-hard, i.e., finding the optimal alignment requires sifting through an exponential number of possible alignments.

Notes:

Many of the ideas described above appeared in (Karlin and Altschul 1990) and (Altschul 1991). The BLOSUM matrices first described in (Henikoff and Henikoff 1993) turn out to have been incorrectly computed in practice, yet their original implementation is more effective at sorting out through the 'twilight zone' than the correctly computed BLOSUM-62 matrix (Styczynski, Jensen et al. 2008).

4.3 Advanced topics in sequence alignment

4.3.1 Context-sensitive alignment (handling complex gap scores)

The approaches we presented earlier assumed that all gaps are equal weight, i.e., the alignments shown below are equivalent:

```
ABBBA
A--BA
A-B-A
```

As we already pointed out above, these alignments are not necessarily biologically equivalent: a string of consecutive gaps may occur in real sequences more frequently than multiple disjoint gap sections (in other words, nucleotide insertions and deletions frequently involve more than a single base). To model such phenomena, we can add to the dynamic programming score a gap-specific term, specifically a function $g(n_{gap})$ which assigns a string of n_{gap} gaps a single score. In our original formulation this score is simply $g(n_{gap}) = n_{gap} * g$ where g is the cost of a single insertion or deletion.

Question: *Can we compute the more general score?*

Question: *Can we do so efficiently?*

To address these questions we will modify the dynamic programming recurrences as follows. Note that we've renamed the table of scores V (rather than E as we did above) in keeping with historical usage.

$$V[i+1,j+1]=max\begin{Bmatrix}0\\V[i,j]+f(S1[i+1],S2[j+1])\\max_{1\le k\le i}V[i-k+1,j+1]+g(k)\\max_{1\le k\le j}V[i+1,j-k+1]+g(k)\end{Bmatrix}$$

First, let us convince ourselves that this approach yields a correct algorithm. Note that we have replaced the recurrences corresponding to the gap insertions in one or the other string with an exploration of all possible gap lengths that could be inserted at that location. Thus, the new algorithm will automatically pick the appropriate gap length to maximize the score according to the generic gap penalty g.

In terms of performance, however, note that the new recurrences require us to perform O(n) and O(m) work, respectively, at each cell in the O(nm) dynamic programming table, leading to a runtime of O(nm(n+m)) or $O(n^3)$. Thus, we can incorporate more biology in our algorithm, yet we pay a major penalty in terms of runtime. Can we find a reasonable trade-off between these two requirements?

It turns out that for certain forms of the gap function we can more efficiently compute the alignment score. In particular, we will focus on affine gap penalties of the form $g(n_{gap})=g_{open}+n_{gap}g_{extend}$ where g_{open} and g_{extend} are the penalties of creating a gap, and adding another gap character to an already existing gap string, respectively. Clearly, $g_{open} > g_{extend}$, otherwise it would always make sense to create new gaps rather than extend existing ones. This formulation of the gap penalty will allow us to 'remember' the information computed in earlier stages of the algorithm, and thus avoid the additional penalty in runtime. Specifically, we will record three new dynamic programming table, E, F, G, in addition to V. In table E we will retain the best alignment scores for alignments that end with a gap in S1, in F are the best alignments which end with a gap in S2, G contains the best alignment scores for alignments that do not end in a gap, while V, as before, is the best alignment score overall:

$$E[i+1,j+1]=max\begin{Bmatrix}0\\V[i+1,j]+g_{open}+g_{extend}\\E[i+1,j]+g_{extend}\end{Bmatrix}$$

$$F[i+1,j+1]=max\begin{Bmatrix}0\\V[i,j+1]+g_{open}+g_{extend}\\F[i,j+1]+g_{extend}\end{Bmatrix}$$

$$G[i+1,j+1]=max\begin{Bmatrix}0\\V[i,j]+f(S1[i+1],S2[j+1])\end{Bmatrix}$$

$$V[i+1,j+1]=max(E[i+1,j+1],F[i+1,j+1],G[i+1,j+1])$$

Note that these tables can be computed in constant time per operation, or O(nm) overall. Of course, the total space requirement has tripled (the V table can be maintained implicitly), but that is a small price to pay for shaving a linear multiplicative factor from the runtime.

The approach outlined above is due to Gotoh, and the affine gap penalties are the standard definition of gap scores in the community as they offer the best tradeoff between efficient computability and biological plausability.

Exercises:

1. Using the ssearch program(part of the FASTA package), investigate multiple setting for the gap scores

and their effect on the alignments created.

4.3.2 Sequence alignment in linear space

The overall runtime of O(nm) for inexact sequence alignment is not easy to overcome (though we will try to do so soon). Can we, however, reduce the overall memory usage? For computing the scores, it is easy to see that we do not need to store the entire dynamic programming table, rather it is sufficient to remember just two rows (or columns, or diagonals) in order to find out the best score in the table, as all the values necessary for computing a cell are found on just two rows of the table (the current row, and the preceding one). If we want to also find the actual alignment, however, we need to record backtrack pointers from each cell and thus store the entire O(nm) table.

To gain some intuition on why we might be able to do better than O(nm) space, note that the actual best alignment itself can be stored in linear space – we simply store one out of the many possible paths through the table connecting the two corners (assuming global alignment). The problem is that we do not know what this path will be until the whole table is filled out, hence the need to store O(nm) pointers. The key idea is that we can, in linear space, figure out parts of the optimal path which allows us to eventually reconstruct the whole path.

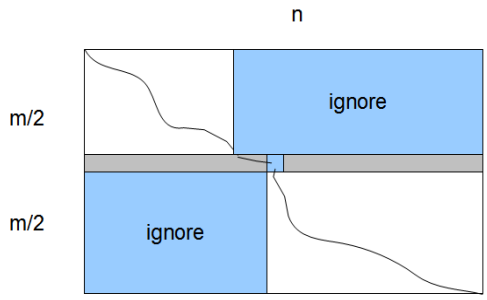
Specifically, the algorithm breaks up the dynamic programming table into two, by focusing separately on the alignments of the first and second halves of one of the strings (see below). For the first half, the alignment proceeds as usual, filling the first half of the dynamic programming table up to (and including) the middle row. For the second half, we fill the table backwards, starting from the bottom, effectively matching the reverse of the second half of one string to the other. It is easy to see that the optimal alignment should be the same for both the forward and reverse versions of the problem. Again, we fill the dynamic programming table up to (and including) the middle row. Note that the alignment algorithm we use only computes the dynamic programming score (does not store backtrack pointers) and it can, thus, be executed in linear space.

Also note, that the middle row is computed by both the forward and the backward dynamic programming approaches. We will set the value of this middle row to be the sum of the values obtained by the two executions. We argue that the optimal path through the dynamic programming table crosses this middle row exactly through the cell with the maximum value, as this location maximizes the sum of the dynamic programming scores of the two halves of the string. At this point, we can directly fill in part of the optimal alignment by following the path this alignment takes along the middle row up to the point where it diverges into the top or bottom halves of the table.

So far we have, thus, identified a small section of the optimal path (which, as pointed above, can be stored in linear space) while using only linear space. We, then, repeat this process by recursing on the two halves of the dynamic programming table, at each step reconstructing another small segment of the path, until we eventually reconstruct the whole path. Note that at each stage the recursion only needs to focus on the white sections of the dynamic programming table as we know that the optimal path will not cross through the blue parts.

While it is clear that this algorithm uses linear space, it is not at all obvious that it is efficient. To compute its runtime, note that at the first stage the runtime is exactly the same as that for the traditional dynamic programming approach: $O(n\ m/2 + n\ m/2) = O(nm)$.

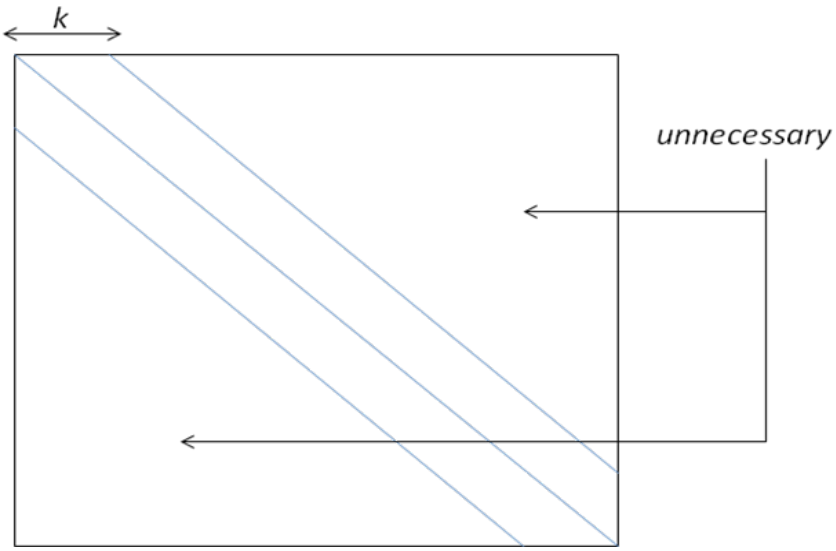
As we recurse, however, at each stage we halve the runtime as we ignore half of the dynamic programming table, leading to the recursion: $O(nm + nm/2 + nm/4 + ...) <= O(2\ nm)$ [inequality due to the properties of exponential series]. Thus, we can achieve a linear space algorithm without a significant penalty in terms of runtime.



4.3.3 Alignment with bounded error

All the inexact alignment algorithms we have described so far require quadratic run time, irrespective of how well sequences match each other. Can we do better for sequences that match well? For example, in the absence of gaps global alignment simply requires filling the main diagonal of the table, leading to a linear algorithm. To explore the relationship between alignment runtime and the amount of error, let us denote by k the numbers of errors we encounter in an alignment. Assume for now that we know *a priori* what k is, or more realistically, we have set an upper bound on its value (e.g., we are only interested in alignments with fewer than k=10 edits).

In this case, we can easily convince ourselves that certain parts of the dynamic programming table do not need to be filled in. Specifically, reaching a cell that is more than k diagonals away from the main diagonal of the table requires us to incur more than k mismatches. As a corollary, no alignment with fewer than k mismatches makes use of those cells and they can be safely ignored (see below)



As a result, our algorithm only has to fill in O(2 k min(n,m)) cells, which leads to essentially linear time execution for k << min(n,m).

Note that is certainly possible that the best alignment found in the 'trimmed' table contains more than k differences, thus our algorithm must check this fact before reporting the alignment.

Question: *If the best alignment has more than k mismatches, can we still report it as the best alignment of the two strings?*

Note that if we do not know the value of k *a priori* we can find it using binary search. Specifically, we begin with a guess for k and find the best alignment. If this alignment has > k edits, we repeat the process with k'=2k, and so on, until we find an alignment with fewer edits than the current value of k. We can easily see

that this algorithm will always find the best alignment in the most parsimonious way (k is never more than twice as large as necessary).

What is the cost of this algorithm, however? Let us denote by k_M, the maximal value of k encountered in the algorithm, i.e., the best alignment has less than k_M edits (generally between $k_M/2$ and k_M edits). Clearly, the maximum memory used by the algorithm is $O(k_M \min(n,m))$. We can enumerate the steps that took us to this point and estimate their computational complexity. Specifically, the runtime can be written as:

$O(k_M + k_M/2 + k_M/4 + ... + k)$ where k is our initial guess. Otherwise written we have the runtime $O(k_M (1 + 1/2 + ...)) = O(2k_M)$ (this is derived from simple properties of exponential series) In other words, despite the possibly large number of iterations of the 'guessing' algorithm we only double the runtime compared to an algorithm that would have guessed the correct value of k the first time (actually our algorithm may be 4-times as expensive as the best algorithm as the correct value of k can be roughly equal to $k_M/2$).

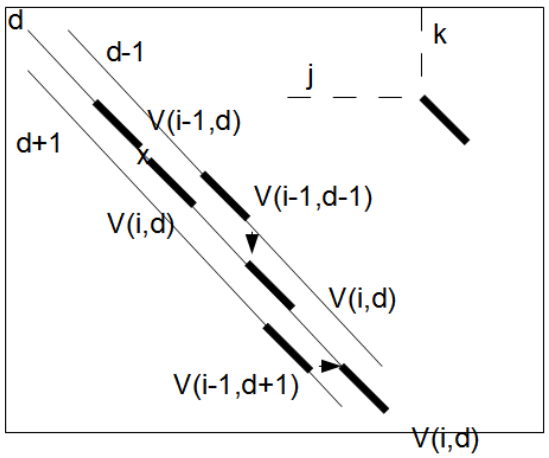
4.3.4 Landau-Vishkin algorithm

Another approach for computing an alignment of two strings in time (and space) proportional to the actual distance between the strings was proposed by Landau and Vishkin (Landau and Vishkin 1986). This approach focuses on the different diagonals within the dynamic programming table – a perfect match represents simply a stretch of one of these diagonals (shown in thick lines in the figure below). The basic idea behind the algorithm is to try to find the furthest along each diagonal we can go with a certain number of edits. For this purpose, we will record values V(i, d), representing the rightmost end of an alignment that contains i mismatches and ends on diagonal d.

Note that by diagonal we mean all the cells in the dynamic programming table that have the same difference between the coordinates in the two strings: d = k – j, where k is the coordinate in S1 and j is the coordinate in S2 (see figure below). To further clarify the following better, V(i, d) will refer to the coordinate in S1 of the end of the alignment. Note that from the definition of d and V(i,d) we can immediately compute the coordinates of the cell ending this alignment to be V(i,d) in S1 (by definition), and V(i, d) – d in S2.

It should be fairly easy to see that V(i,d) can be computed from nearby values as follows:

$$V(i,d) = max\begin{Bmatrix} V(i-1,d)+LCP(V(i-1,d), V(i-1,d)-d) (mismatch\ along\ d) \\ V(i-1,d-1)+LCP(V(i-1,d-1), V(i-1,d-1)-d) (deletion) \\ V(i-1,d+1)+LCP(V(i-1,d+1)+1, V(i-1,d+1)-d-1) (insertion) \end{Bmatrix}$$



In the equations above, LCP(k, j) represents the longest common prefix of the suffixes k and j of S1 and S2, respectively.

It should be fairly obvious now that computing the V(i, d) values is sufficient for finding the optimal alignment. Specifically, we are looking for the smallest i value that leads to one of the V(i, d) values to reach the bottom right corner of the dynamic programming table. If we fill in the V(i, d) values in increasing order of i, we stop once we reach the correct value i = k, and the total number of values filled in is O(nk). As a result, the memory usage is automatically 'tuned' to the divergence between the two strings.

The runtime depends on how quickly we can compute the LCP values. One approach relies on generalized suffix trees – suffix trees constructed from the suffixes of two or more strings (it is easy to see that Ukkonen's algorithm can easily construct such trees). If we construct a suffix tree from strings S1 and S2 the LCP(i, j) for suffixes i in S1 and j in S2 is simply the string depth of the lowest common ancestor of the leaves corresponding to the two suffixes. An algorithm due to Vishkin can compute this information in constant time (more specifically, this algorithm can find the lowest common ancestor of two nodes in any tree), leading to an overall O(kn) runtime for our algorithm.

4.3.5 Parallel alignment

Today's computers have multiple processors, and can also perform fine-grain parallelism (e.g., perform arithmetic operations on multiple processor registers at the same time). In fact, parallelism is key to much of the speed-ups we discussed above, as we often assume that operations on a single computer word take a constant amount of time, i.e., all bit-level operations are done in parallel.

A full description of parallel algorithms is beyond the scope of this class, however it is important to note two main issues:

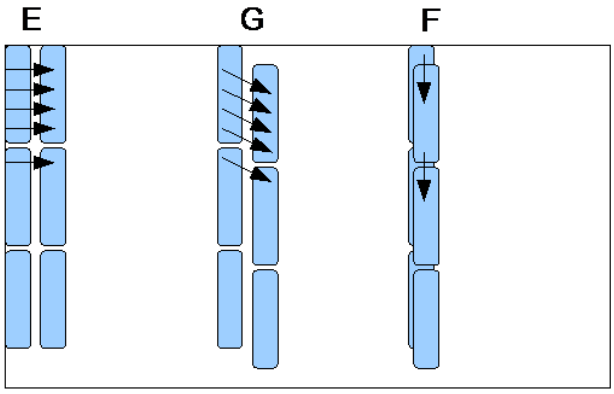
1. Any parallel algorithm can be executed in the same amount of time (defined as runtime * number of processors) on a serial computer. This should be obvious as we can have the same processor pretend to be each of the parallel processors in turn, without needing to account for any additional computational time. As a result, parallelism cannot 'buy' us more than a speed-up proportional to the number of processors used.
2. The speed-up (ratio of the serial runtime to the parallel runtime) is limited by the amount of serial time spent coordinating between the p processors used. For a serial runtime of R, the runtime in parallel on p processors, R(p) >= R/p, with equality only when there is no communication.

In other words, parallel algorithms are not a universal panacea, and can only reduce the runtime by a (possibly large) constant determined by the number of processors used. This constant improvement can, however, be quite important in practice, e.g., it could represent the difference between running a program for one month versus one day.

The dynamic programming alignment algorithm provides nice opportunities for parallelism as the data dependencies are quite limited. Specifically, the three dynamic programming tables for affine gap processing, E, F, and G, each rely on information from the adjacent column, same column, and shifted adjacent column, respectively, as seen in the figure below. Entire group of cells can, thus, be computed in a single operation, assuming the computer being used allows such parallel operations (most modern processors do), also called SIMD instructions (single instruction multiple data).

Of course things are not quite as easy as outlined here, rather implementing an efficient parallel algorithm requires a number of 'tricks' (see (Farrar 2007, Rognes 2011) for a more extensive description). First, note that when computing the table G, we must identify, for each character in the query S1[i], the score S(S1[i], S2[j]) of aligning it to the corresponding character S2[j] in the reference. At first glance, this lookup cannot be performed in parallel. Given that the alphabet is limited in size (20 letters for amino-acids, 4 for DNA), we can, however record a table of size O(m $|\Sigma|$) which records the substitution scores at each position in the query for all possible choices of characters in the reference. This table can be precomputed at the beginning of the

algorithm and a single lookup suffices to identify in constant time an entire block of scores.



A second issue we encounter relates to the idiosyncrasies of the parallel architecture available to us. For example, the size of the values stored in the dynamic programming table affects the number of operations we can do in parallel. For example, if our processor can perform in parallel integer operations across 128 bits, we can process 16 cells at the same time as long as the dynamic programming score can be represented in 8 bits. If 16 bits are needed, the level of parallelism drops to 8.

Thirdly, we can run into more issues due to the fact that our algorithm must choose the large value from among three possible values. Such decision branches can affect the ability of the compiler to generate code that gets executed efficiently, offsetting some of the speedup that we might be able to achieve.

4.3.6 Exercises

1. (fitting alignment) Suppose we have sequences $v = v_1, \ldots, v_n$ and $w = w_1, \ldots, w_m$, where v is longer than w. We wish to find a substring of v which best matches all of w. Global alignment won't work because it will try to align all of v. Local alignment won't work because it may not align all of w. The problem (called the fitting problem) can be formulated as the problem of finding a substring v' of v that maximizes the score of alignment s(v', w) among all substrings of v. Give an algorithm which computes the optimal fitting alignment in O(nm) time. Describe both how to compute the score of this alignment and how to compute the actual alignment.

Note: You don't need to spell out all the details of the algorithm. If you use the traditional dynamic programming approach, it is sufficient to specify the initial conditions (what values are written in the first row/column of the matrix), where will the score for this best alignment be located in the matrix, and whether you use the global alignment recurrence in the algorithm (max value between left, diagonal, and above), or the local alignment recurrence that allows the alignment to restart at any location by taking the maximum of four values: 0 and the three values mentioned above.

2. Describe how you would modify the dynamic programming alignment algorithm to compute overlaps between pairs of sequences provided as input to a genome assembler. By overlap we mean finding a suffix of S1, S1[k..n], and a prefix of S2, S2[1..l], which align best (in terms of the dynamic programming score) among all possible suffix/prefix combinations. Please specifically address the following points:

a. What are the initial conditions in the dynamic programming table?
b. Where will the answer be found?
c. What recurrence relations will you use: local alignment or global alignment (no need to actually write them out)?

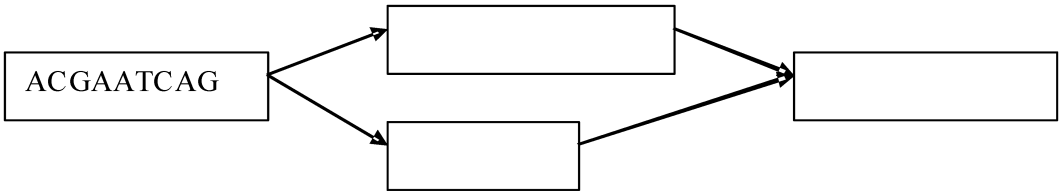
Note: Assume all overlaps are 'proper', i.e. none of the sequences is contained within another one. Also assume we only care about overlaps between "forward" versions of the sequences (no need for reverse complements).

3. Bacterial gene finders have difficulty identifying the correct start codon of a gene though the stop codon is always correctly determined. Assume you are given a collection of protein sequences determined computationally from predicted genes and want to align these sequences to each other, accounting for possibly incorrect predictions of gene starts. Specifically, you want the ends of the genes to be aligned to each other, however you are willing to ignore mis-alignments at the beginning of the sequences. How would you modify the dynamic programming alignment algorithm described in class to compute such alignments. Specifically:

a) What are the values stored in the first row and column of the dynamic programming table?
b) What are the recurrence equations?
c) Where will the answer be found in the dynamic programming table?
d) What is the stopping condition for the backtracking procedure?

4. The partial order alignment (POA) technique describes a multiple sequence alignment as a directed acyclic graph where the nodes represent sections of DNA shared by two or more sequences and the edges of the graph connect sequence segments that occur in the same sequence. For example, in the figure below, the DAG represents the alignment of sequences ACGAATCAGGGTACAGGCTAGTTCGACCT and ACGAATCAGACGACCGTTCGACCT. The classical dynamic programming alignment algorithm can be extended to allow the alignment of such a DAG against the sequence of a genome. Describe the details of such an approach: setup of the dynamic programming table, initial conditions, and the recurrence equations.

Hint: The optimal alignment will follow one of the paths in the DAG.



5. Homopolymers – stretches of DNA in which a single base is repeated (e.g. AAA...) - complicate genome sequencing and frequently lead to sequencing errors. The most common error leads to an incorrect number of bases to be "read" by the sequencing machine. Assume you want to account for this fact when you align DNA sequences to each other, specifically, you want to allow gaps in the alignment when necessary to align homopolymers of differing lengths. In all other cases the gap penalties will follow the standard edit-distance approach. Describe how you would modify the "traditional" local sequence alignment algorithm (without affine gaps) to allow the correct alignment of homopolymer regions. For simplicity, assume the only homopolymers allowed are repetitions of nucleotide A. **(Hint: think affine gaps)**.

6. We described how the Aho-Corasick algorithm can be used to compute an exact alignment with "don't care" symbols. Describe an algorithm that can perform inexact alignments using "don't care" symbols. (Note: a "don't" care symbol '*' is a character that can be matched to any other character without a penalty). Will your algorithm allow a '*' to be aligned to a gap?

7. Assume we want to compute high-quality **ungapped** local alignments of two strings. Do you still need a dynamic programming table? Describe an algorithm that finds the maximum scoring ungapped local

alignment.

4.4 Sequence alignment heuristics

The algorithms described above are the most efficient approaches for inexact alignment **in the general case**. In other words, if our goal is to find the best scoring alignment between two strings, irrespective of how good this alignment is, we cannot beat the quadratic runtime of the basic algorithm. For most applications this runtime is simply prohibitive – imagine, for example, aligning several million 100bp sequenced reads to the entire 3 billion base-pairs in the human genome, or aligning one entire 3 Mbp bacterial genome against a collection of other 10,000 such genomes. To efficiently perform such alignment tasks we must relax our goals, by focusing, for example, on just the highest quality alignments, or even give up on finding the best scoring alignment. In this section we will discuss some such heuristic approaches. Note, however, that such heuristics must be used with care and only when the biological application is compatible with the relaxed assumptions of the heuristic approach. Thus, before you propose a 'smart' new heuristic that outperforms other alignment approaches try to fully understand the needs of the biologist users.

4.4.1 Exclusion methods

Exclusion methods attempt to restrict the search space to just regions where good alignments are likely to be found. Assume, for example, that we are interested in finding an alignment with fewer than k edits of a pattern P of length m against a text of length n. Baeza-Yates and Perleberg observed that if we break P into k+1 equally-sized blocks of size m/(k+1), at least one of these blocks contains no edits, i.e., the block matches perfectly against the text. As any good alignment satisfies this property it suffices to focus on the parts of the text that contain exact matches with the pattern of length at least m/(k+1). For each of those locations we can find the optimal alignment in $O(m^2)$ rather than O(mn).

To find the locations where the pattern has an exact substring match to the text one can use any of the exact string matching algorithms we discussed earlier in this class. Specifically:

- The pattern can be split into k+1 equally-sized blocks, which are then searched against the text using Aho-Corasick.
- The longest common subsequence matching between the pattern and the text can be found (as well as all matching substrings longer than m/(k+1)) using a suffix tree constructed on the text.
- An index of all strings of length m/(k+1) (also called k-mers) can be constructed from the text allowing us to quickly find all k-mers that are found in both the pattern and the text.

The latter option warrants further discussion. This type of index is called an **inverted index** as it maps content (k-mers) to their location in the text. For each k-mer it lists all indices in the text where that k-mer is found. Such indices can be quite efficient as they can be organized as a hash table with effectively constant access time per k-mer. The efficiency, however, depends on the size of the k-mer (the shorter the k-mer the more efficiently the index can be stored), and the level of repetitiveness of the genome (the more times a k-mer is found, the less efficient is the index).

Case study: Blast

The Blast program is one of the most widely used and heavily cited programs in bioinformatics. It is 'simply' a database search program that searches individual sequences against a large database. We have already discussed how Blast uses statistical arguments to distinguish 'real' matches from random alignments. That discussion was predicated on having an alignment score between a query string and the database. To identify putative alignments and their scores, Blast constructs a k-mer index for the query, i.e. it stores all k-mers of a certain length (k=3 for proteins, k=11 for DNA by default) and their location in the query. For each k-mer Blast

also records the 1-off neighbors of the k-mer (e.g., in addition to ACA it would also store TCA, GCA, CCA, as well as ACT, ..., etc.). The algorithm then attempts to find exact k-mer matches between the database and the query, and performs a local dynamic programming alignment around each of the matches found to identify high scoring pairs (HSPs), which are then evaluated statistically.

Note that Blast is not guaranteed to find the best match between a query and the database – assume, for example that it compares two versions of a gene's DNA sequence, and these versions differ in the third position of their codons while still encoding the same protein. In these sequences one will not find any exact match longer than 2 bp and the biologically correct alignment will be missed by blast (which uses k=11 for DNA sequences), even though it tries to account for mismatches in its search. At the same time, using a smaller value of k, e.g., k=5 (after accounting for one mismatch we should be able to find matching k-mers in this case) might lead to too many exact matches requiring a prohibitive amount of time to compute all alignments and find the best one.

This trade-off between efficiency and sensitivity (ability to find the correct alignment) is key to most heuristic choices in alignment.

4.4.2 Spaced seeds

Let us explore a bit more the example given above, where we try to align two versions of a same gene, e.g:

```
g1:   AATCAAGGUAAATTA
g2:   AACCAGGGAAAGTTG
prot: N  Q  G  K  L
```

Clearly no 11-mer matches exactly or even with one mismatch between the two versions of the gene and this alignment would be missed. We can, however find the alignment using a special type of seed called a **spaced seed,** where we allow certain positions to mismatch. Specifically, we will represent the seed as a string of 1s and 0s, and require that the characters aligned to a 1 match each other. We will compare two seeds of length (also called width) 11: 11111111111 (the original 11-mer) and 11011011011 (an 11-mer that allows 'wobble' basepairs). Let us see how they do in the example above:

```
g1:   AATCAAGGUAAATTA
      11111111111
g2:   AACCAGGGAAAGTTG
prot: N  Q  G  K  L
g1:   AATCAAGGUAAATTA
      11011011011
g2:   AACCAGGGAAAGTTG
```

Note that the contiguous seed mismatches (positions marked with red) but the spaced seed (containing 0) can accommodate the mismatches and find the correct alignment. In other words, the spaced seed appears to be better – it has the same width yet finds a match missed by the contiguous seed, and also only requires us to look at 8 characters (the weight of the seed) instead of 11.

We can formalize this property by defining the **sensitivity of a seed** to be the probability that at least one seed matches between two aligned strings. The sensitivity is specific to the similarity between the strings, defined by the fraction of identities in the alignment. Note that we can think of the alignment as simply a series of 1s and 0s, where 1s indicate that the corresponding characters match perfectly (we ignore gaps here). If the rate of identities is p, we expect 1s to occur in the alignment with probability p.

We can get a bit of an intuition on why spaced seeds are better by observing that we can compute the expected number of seed matches between the two strings as follows. Let L be the length of the alignment, and W be the

width of the seed, and p the identity rate (66% in the case above – 1 in 3 letters mismatch). There are L-W possible placements of the seed in the alignment. For the exact seed, the probability that all 11 positions match identical characters in both strings is simply $p^{11}=(0.66)^{11} = 0.0103$. For the inexact seed we only need 8 positions to match, leading to $p^8 = (0.66)^8 = 0.0360$. Given that L-W is the same for both seeds, the inexact seed yields more expected hits in the same alignment range and thus has a higher likelihood of finding the correct alignment.

To formally compute the sensitivity of a seed, we encode possible alignments as strings of 0s and 1s representing the position of the exact matches and mismatches. We define the sensitivity of a seed as the probability that the seed matches an alignment, i.e., that the seed (represented as a string of 0s and 1s) occurs as a substring within the corresponding string representing the alignment. To compute this probability we will rely on dynamic programming, as follows.

First, we define f(i, b) to be the probability that the seed matches a prefix of length i of the alignment that ends in the binary string b (of length equal to the width of the seed). The probability of the seed matching an alignment of length L is simply $sensitivity(seed) = \sum_{all\, b} f(L, b) * p(b)$ where p(b) is the probability of each string of length b. The latter can be computed as described above from the actual string b and the probability of match p.

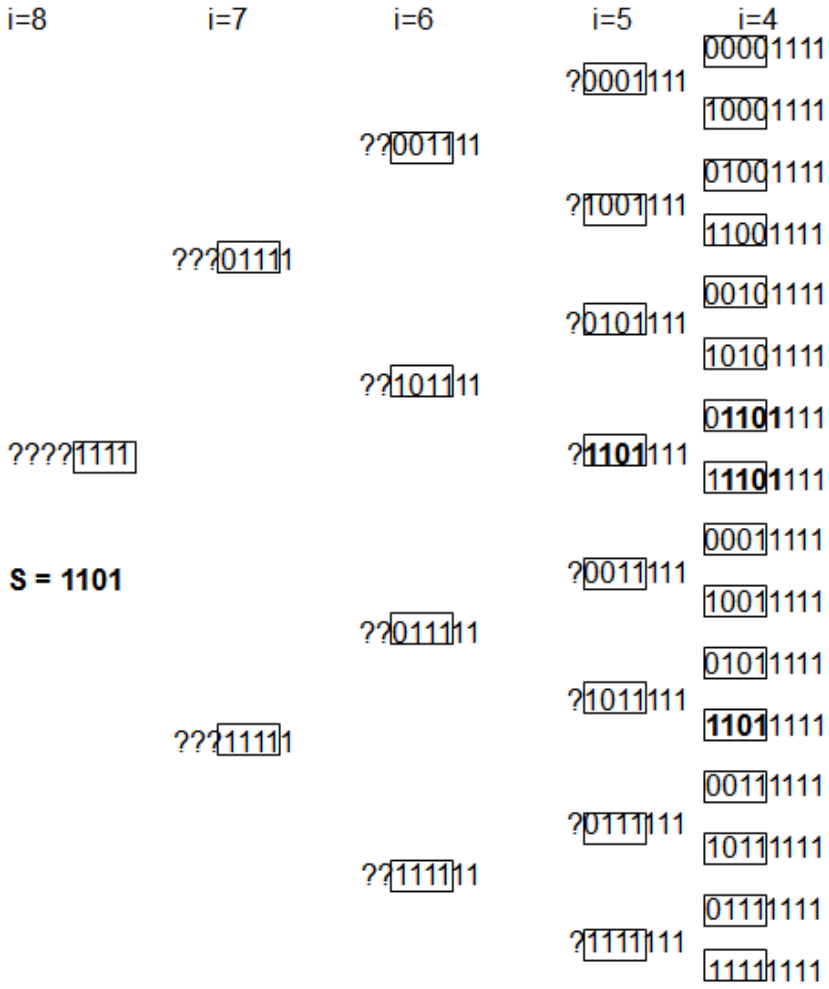
In other words, the dynamic programming table f(i,b) focuses on some prefix of possible alignments, and checks whether the seed matches the suffix of this prefix. If S = b, f(i,b) = 1 as clearly the seed matches all alignments that end in b. If S <> b, we need simply use the recurrence equation:

f(i,b) = f(i-1, 0b') (1-p) + f(i-1, 1b') p

where b' is the string b without its last bit. Essentially we are shifting along the possible alignment strings and checking again if we have a match with the seed. The values for f(W, b) are trivially set to 0 if S <> b and 1 if S = b.

Note that the dynamic programming approach outlined above essentially enumerates all possible alignments of length L and computes the proportion of these alignments that contain our seed. The runtime of this algorithm is exponential in the size of the alignment $O(2^W 2^{L-W})$ (the first term is the enumeration of all words of length W, while the second one enumerates all the extension of these words to the total length of the alignment).

An example is shown below for the seed 1101 and alignments of length 8 ending in an exact match of 4 characters (1111). In bold are the locations where the seed if found to match the alignment. Assuming p = 0.5 (equal likelihood of matches/mismatches) f(L, 1111) = 3/16 (three of the 16 different equally likely alignments contain the seed).



Additional intuition:

Note that any specific spaced seed is unlikely to be fully sensitive (any good alignment contains a matching seed), with the exception of trivial corner cases (e.g., exact matches must contain other exact matches). As a result, algorithms that rely on spaced seeds use multiple such seeds in order to increase the chances of finding an alignment. What is important in this case is that the seeds are independent such that together they cover a large fraction of the space of possible alignments. Examining the example above should provide you with the intuition that independent seeds do not overlap each other well, i.e., as we shift the b box during the execution of the algorithm, the boxes that match a particular seed occur in different parts of the tree. While computing the sensitivity of a seed is highly expensive (as we have seen above), this simple observation can lead to effective and efficient algorithms for constructing seed sets – see Ilie and Ilie (Ilie and Ilie 2009).

Historical note:

The concept of spaced seeds is closely related to the concept of **locality-sensitive hashing**. In general terms, a locality-sensitive hashing function is 'compatible' with some definition of distance between objects. The closer the objects are (e.g., the more similar two sequences are), the more likely it is that the hash function will put these objects in a same bucket. One can easily see that selecting a set of positions from a sequence (the 1s in the example above) and mapping the sequence to the bucket corresponding to the base-pairs such identified is a hashing function, and that the more similar two sequences are, the more likely it is that they will agree over some subset of their positions. Such ideas were initially introduced by Buhler before the concept of spaced

seeds was developed, in part by Li and Ma.

4.4.3 Inexact seeds

A concept similar to spaced seeds was introduced by Ghodsi and Pop (Ghodsi and Pop 2009) and also independently by ?. The main limitation of spaced seeds is that they do not tolerate insertions and deletions, thus their sensitivity is inherently limited to finding alignments where most differences are due to substitutions. As we have seen so far, handling insertions and deletions can be quite expensive, while exact matches can miss too many alignments, or lead to too many options that need to be explored. A trade-off between these two extremes can be obtained based on the following simple lemma:

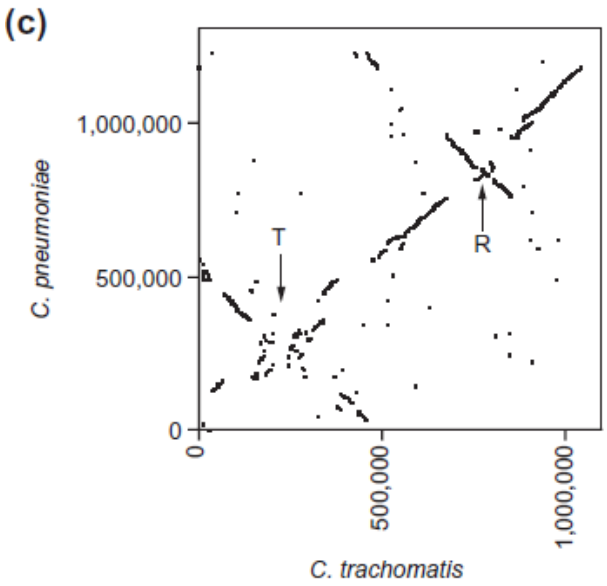
Lemma: *For any l < L, an alignment of length L with k edits must contain a sub-alignment of length l with at most* $\lfloor \frac{kL}{l} \rfloor$ *mismatches.*

At the extreme, this lemma is equivalent with the Baeza-Yates observation that any alignment with k edits contains an exact match of length l = L/(k+1). The lemma is easy to prove by observing that a path through the dynamic programming algorithm representing alignment L cannot always exceed $\lfloor \frac{kL}{l} \rfloor$ mismatches in all of its subpaths of lengths l.

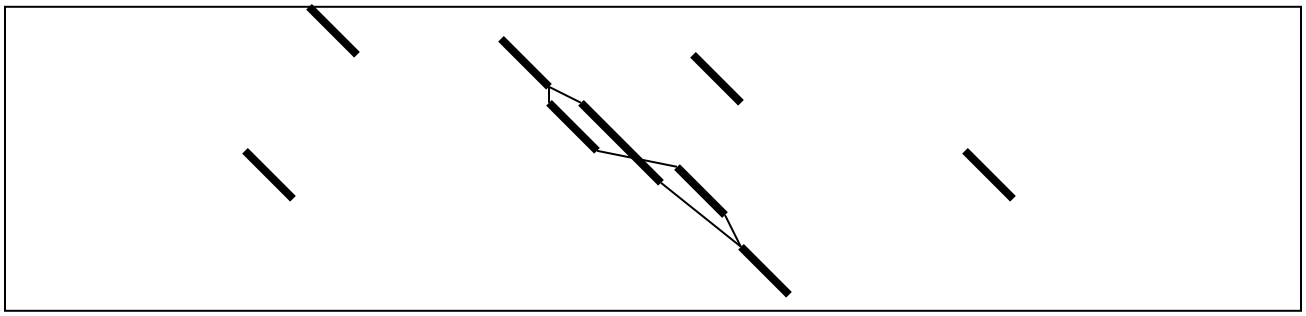
4.4.4 Whole-genome alignments: chaining algorithms

Most of what we presented above assumes that the strings being aligned are fairly similar to each other (for biological reasons). This is generally true for short strings (e.g., proteins or genes) but is no longer a valid assumption when aligning long DNA segments, such as whole genomes. A global alignment of such long sequences is no longer biologically relevant due to possible large-scale changes between even closely related genomes, such as insertions/deletions of large segments of DNA (e.g., antibiotic resistance cassettes), or large-scale rearrangements (e.g., large segments that are duplicated or inverted or swapped). To meaningfully align such large segments we simply focus on constructing a collection of high-quality local alignments from which we can infer the relationship between the sequences being aligned. For example, the figure below (from (Eisen, Heidelberg et al. 2000)) shows an alignment of two related genomes from the genus *Chlamydia* which highlight large-scale inversions occurred in these genomes since they diverged. In this case, the inversions occur around the origin and terminus of replication due to 'errors' in the mechanism used by bacteria to replicate their DNA.

Note that there are many possible local alignments and we'd like to identify an 'optimal' subset that best explain how the genomes relate to each other. A similar issue also occurs, for example, when we use a filtering approach to find a set of exact matches and wish to further explore sections of the alignment space where these matches indicate a good alignment may exist.



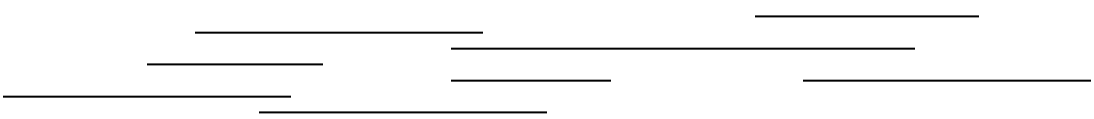
The main question here is how we define the "goodness" of a set of local alignments. To gather a bit of intuition, see the picture below, representing a dynamic programming table. The thick lines are good local alignments (be it exact or inexact). Just by visual inspection, it appears that there is a good alignment in the middle of the table – there simply is a large density of local alignments around the same diagonal. There are different ways of 'completing' these local alignments into global alignments (the thin lines). Which one of them is better?



To formalize, we will define the 'goodness' of a chain of alignments to be simply the sum of the scores of the chained alignments, as long as the alignments are 'compatible'. By 'compatible' we mean that the alignments are not allowed to overlap. We can also add a score term (or penalty) related to the gap between the alignment (chaining across large gaps in the alignment could be penalized, for example), however we'll leave the details of that implementation as a detail.

One dimensional chaining

To understand how we can find the best chain, we'll first focus on a 1-dimensional version of this problem. Specifically, assume you have a set of intervals on a line (see below) and you want to find the set of non-overlapping intervals that has maximum weight, where for each interval i we define w(i) to be its weight.



The algorithm for computing this maximum weight 'cover' (not to be confused with the traditional set theory max cover problem) follows the dynamic programming paradigm.

First, we define V[i] to be the weight of the maximum weight set of non-overlapping intervals ending with interval i (we assume the intervals are ordered from 1 to n by their left most coordinate). A simple (and inefficient) dynamic programming approach would iterate over all i values in increasing order. For each i this algorithm sets $V[i]=max_{j<i,\ interval\ j\ ends\ before\ interval\ i}(V[j]+w(i))$. In other words, at each iteration we check all prior intervals which do not overlap the current one and choose the one with maximum score. It is easy to see that this algorithm runs in $O(n^2)$.

A more efficient algorithm can be obtained by observing that we do not need to check all prior intervals, just the one with the maximum score that ended before the current interval started. Specifically, the algorithm will record, as above, the score V[i], as well as a value MAX representing the maximum V[j] over all intervals j that ended before the current position. The ends of all intervals are first sorted, keeping track of whether each coordinate represents a left or a right end of an interval. Then the algorithm proceeds by filling in the V[i] and MAX values as follows:

```
foreach interval end e from left to right (in increasing order of coordinates)
    if (e ==l(i)) # if e is the left end of interval i
        V[i] = MAX + w(i)
    end if
    if (e == r(i)) # e is the right end of interval i
         if (V[i] > MAX)
            MAX = V[i]
        end if
    end if
end foreach
```

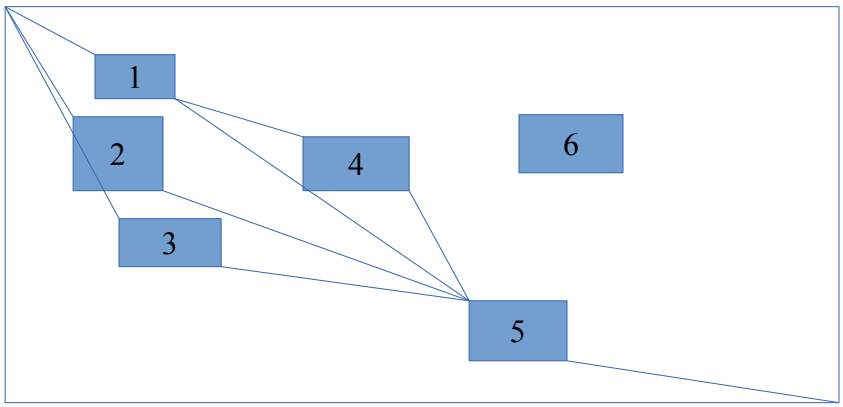
At the end of the algorithm, the value MAX will contain the weight of the maximum weight set of non-overlapping intervals. The actual set of intervals can be found through backtracking by recording a bit more information.

The correctness of this algorithm is easy to see – MAX is always set to the weight of the maximum chain of intervals that has been completed (the rightmost position of its rightmost interval was seen), as MAX is only set once we encounter the right end of an interval. In addition, when setting V[i], the current interval is implicitly compatible (non-overlapping) with the interval embodied by MAX as the latter's rightmost end occurred before the left end of the current interval (V[i] is only set at the left end of an interval).

The runtime of the algorithm is dominated by sorting the interval ends, yielding a runtime of O(n log n).

Two dimensional chaining

The first dynamic programming algorithm described above can be easily extended to two dimensions. Like in the 1D case we can visit the intervals (which are now rectangles bounding the 'good' diagonals) in order of their x- or y-coordinates, and check all compatible intervals to extend a current chain. Like before, V[i] is the weight of the maximum weight chain that ends with rectangle i. In 2-D, the concept of compatibility is a bit more complicated – the upper-left corner of the current rectangle must occur to the right and below the lower right corner of some previous rectangle in order to be able to chain them together.



In the figure above, for example, interval 4 is only compatible with interval 1, while interval 5 is compatible with all prior intervals. Note that we could also solve this problem by simply building a graph comprising all the compatible interval links, then find a maximum weight path in this graph, again with the same $O(n^2)$ runtime.

Converting the more efficient algorithm to the 2-D setting is a bit more complicated. Let us assume that we will proceed along the x axis. We will follow a same procedure as before, visiting every interval end in order to update the value of the chain ending with the current interval (V[i]). At each position, however, we will have to take into account more than just one maximum value, tracking the vertical placement of the intervals.

Specifically, instead of a single value MAX, we will maintain a vector MAX that contains the tuple (b[i], V[i]) for every interval i in this vector, where b[i] is the bottom coordinate of the interval. We will maintain this vector sorted by b[i].

While visiting the intervals along the x axis, as before, we distinguish two situation:

1. If the current position is the left end of an interval i

We find the position of t[i] (the top coordinate of the interval) within MAX, and identify the lowest interval j with b[j] > t[i]. By definition, intervals i and j are compatible, therefore we can set V[i] = V[j] + w(i). Note that this step requires O(log n) time as we need to perform a binary search to find the value of j.

If no such j exists, we can simply set V[i] = w(i).

2. If the current position is the right end of an interval i

Now we must update the vector MAX. We first search for b[i] within MAX, and let j be the lowest interval with b[j] >= b[i].

If V[i] < V[j] we terminate this round.

Otherwise, we insert (b[i], V[i]) into the MAX vector. We also remove from MAX all intervals k with b[k] < b[i] and V[k] < V[i]. In other words, we make sure that all intervals following i in MAX have V values larger than that of i.

Note that this algorithm can be executed in O(n log n), partly due to the initial sorting operation, and partly due to the fact that at each of the n iterations we perform a O(log n) binary search.

The correctness is a bit harder to prove than before. First, note that the MAX vector is always sorted by both b (by construction) and V (due to the pruning procedure in step 2 above. Thus, both the pruning and the insertion can be done in O(log n) time through binary search. Second, observe that maintaining the MAX vector sorted by V ensures that at each left end we will always pick the best compatible interval to extend. Specifically, in the figure above, when we reach interval 5 MAX could contain all four prior intervals, sorted by their bottom coordinate (1, 2, 4, 3). To compute V[5] there is no reason to check all intervals – is suffices to pick interval 3 as it has the highest V value and is compatible with interval 5. Assume, however, that interval 4 had

a higher V value than 3. In that case interval 3 would never be chosen by any subsequent interval as all intervals compatible with 3 are also compatible with 4 which has a higher value (at the point in time when both 3 and 4 are in the MAX vector). Thus, interval 3 can be safely removed from the MAX array, preserving the property that this array is sorted by both b and V.

Note that it is not sufficient to just store the maximum prior interval and we truly need the entire MAX array. Assume that, as described before, interval 3 has the highest value. When reaching interval 6 we cannot use this value when computing V[6] as the two intervals are not compatible, and we must use an earlier 'suboptimal' V value which may, however, eventually lead to the maximum weight chain.

Note: In the description above we use 'upper' and 'lower' to match the picture and, thus, assume the coordinate (0, 0) is in the bottom left corner of the matrix. In a traditional alignment approach the (0, 0) coordinate is in the top left corner and the math would have to be modified accordingly.

Summary

To summarize, when aligning long genomes we start by finding good local alignments, then chain these into either a global alignment (as described above), or simply longer local alignments (using a local version of the algorithm described above). Once the alignments are chained, we can fill in the gaps between them using the traditional global alignment approach.

Historical note

The program MUMmer developed by Art Delcher (former scientist at the CBCB) introduced the idea that the local alignment 'guides' should be Maximal Unique Matches (MUMs) (Delcher, Kasif et al. 1999) – longest matching segments that are unique in both of the strings being aligned. Adam Phillippy (former UMD grad) applied the chaining algorithm to find inexact matches around the MUMs found by MUMmer (Delcher, Phillippy et al. 2002). He implemented the $O(n^2)$ version since it is easier to implement and debug and the improved performance of the O(n logn) algorithm was not necessary in typical applications. The MUMmer package was originally developed when Art worked at Celera Genomics. Open-source suffix tree code from Stefan Kurtz allowed the package to be released open source (Kurtz, Phillippy et al. 2004).

4.4.5 Exercises

1. For a sequence of n integers, the longest increasing subsequence problem (LIS) represents finding the longest subsequence in which all integers are listed in increasing order. For example, for string:

5,3,4,9,6,2,1,8,7,10

one possible answer is:

3,4,6,8,10

a. Describe a simple dynamic programming algorithm to compute the longest increasing subsequence in $O(n^2)$.
b. Can you see any similarity between the LIS problem and the chaining algorithms described in the course? Indicate whether the 1-D or 2-D chaining algorithms is most similar to the LIS problem, and sketch a mechanism for solving the LIS problem in O(n log n), assuming you already have code that can compute the optimal solution to the chaining problem.

2. You are trying to find the placement of sequences of length 100 along the human genome (3 Gbp), allowing for at most 4 mis-matches, however you do not want to use Smith-Waterman directly. Instead you will use an exact matching algorithm to find good candidate matches then extend these with Smith-Waterman. What is the length of the exact match seeds you will use in order to guarantee you that no correct alignments are missed?

3. You are given two strings of integers, e.g.:

S1= 1 7 5 3 2 8 9
S2= 14 4 1 3 6 5 4

Assume you partition these strings into the same number of sections. Sum up the numbers present in each section of each string S, and replace the string with a new (possibly shorter) string S' representing the sums of its parts. For example, if S1 is partitioned: | 1 7 5 | 3 2 | 8 | 9 |, you would replace it with the string: S1' = 13 5 8 9. Compare the strings S1' and S2' by summing up the squares of the differences between corresponding numbers (remember, the prime strings have the same number of integers because the original strings were partitioned into the same number of sections).

The goal is to describe a dynamic programming algorithm that computes the optimal way of partitioning the strings, i.e. the approach that minimized the sum of the squared differences between the primed strings.

For example, for the strings shown above a possible solution is:

S1= | 1 7 5 | 3 | 2 8 | 9 |
S2= | 14 | 4 | 1 3 6 | 5 4 |
S1'= 13 3 10 9
S2'= 14 4 10 9

Diff= 1 1 0 0
Dist= 12 + 12 = 2

Additional constraints: each string must be split into two or more sections, and none of the sections can be empty.

Hint: Try to think in terms of the spaces between numbers, instead of the numbers themselves, i.e. possible locations of a "cut" defining a subsection of a string. Try to determine how to pair up cut-points between the two strings.

5 Multiple sequence alignment

In biology we are frequently faced with the problem of aligning multiple sequences together, e.g. we have managed to extract the sequence of a gene of interest from multiple organisms, and want to find out how these different versions of the sequence relate to each other.

5.1 Introduction to multiple alignment

Specifically, we want to create a matrix where each sequence is represented as a row, and each column represents the alignment of characters from the original sequences, also including gap characters as necessary. Note: we generally require that at least one character in each column is not a gap.

```
Seq1  ACC-AGTGA-CT
Seq2  -CGTAGCGA-CT
Seq3  ACC-AGTGT-CT
Seq4  ACGTAGTCATCT
```

Unlike the pairwise sequence alignment problem, the multiple alignment problem is a bit harder to formalize. A commonly used theoretical framework states the problem as follows:

Multiple Alignment Problem. Given a set of n sequences s_1 .. s_n, find the multiple alignment M that minimizes the sum of the pairwise distances between these sequences: $\sum_{s_i, s_j \in S} d_M(s_i, s_j)$

$d_M(s_i, s_j)$ is the distance between sequences s_i and s_j as implied by the multiple alignment M, i.e., the sum of the distances between the characters of the two sequences that are aligned to each other in the multiple alignment. Generally, alignments between gap characters are assigned score 0. The score of alignments between individual nucleotides or amino-acids depends on the specific context: in protein alignments the scores can be derived from the BLOSUM matrix scores, while in phylogenetic analysis of nucleotide sequences, the alignment scores are determined by an appropriately defined evolutionary model.

Note that $d_M(s_i, s_j)$ is generally larger than $D(s_i,s_j)$ - the optimal pairwise distance between two sequences (as defined by the pairwise alignment problem discussed earlier in the class).

The Multiple Alignment Problem can be "easily" solved through an extension of the dynamic programming algorithm for pairwise sequence alignment. The intuition is to extend the V(i,j) values from the 2-sequence context, to V(i, j, k) for 3 sequences, etc. The resulting algorithm runs in $O(L^n)$ where L is the length of the aligned sequences, and n is the number of sequences. Clearly, this algorithm is impractical, though some run-time improvements can be obtained by carefully pruning the dynamic programming table.

In general, the Multiple Alignment Problem is NP-hard, leading to the need for heuristic approximation algorithms.

5.1.1 Star Alignment

One such algorithm is called **Star Alignment** – deriving its name from the fact that the multiple alignment is constructed by aligning all the sequences to a same "center" sequence. The alignment process is called "progressive alignment" and proceeds as follows:

1. Construct alignment of sequence 1 with the center sequence (using standard pairwise alignment)
2. Construct alignment of sequence 2 with center sequence (using standard pairwise alignment). Any gaps inserted within the center are propagated to the already constructed alignment with sequence 1.
3. repeat 2 until all sequences have been aligned

Note: the implementation can be a bit tricky as you need to avoid having spurious gaps inserted in the alignment – e.g., if two sequences both lead to the insertion of a gap at the same location in the center sequence, the alignment algorithm should ensure a single gap is inserted (rather than two adjacent gaps).

Choice of the center sequence. For reasons that will become apparent below, the center sequence s_c is chosen to be the sequence that minimizes the equation

$$\sum_{i \neq c} D(s_i, s_c)$$

over all possible choices of s_c.

Theorem: If the distance between sequences satisfies the triangle inequality, the star alignment has a score at most 2 * OPT, where OPT is the score of the optimal multiple alignment.

Proof:

Let S* be the "sum of pairs" score of the star alignment.

$$S^{star} = \sum_{i \neq j} d_{star}(s_i, s_j) \leq \sum_{i \neq j} (D(s_i, s_c) + D(s_j, s_c)) = 2(k-1) \sum_i D(s_i, s_c)$$

where k is the number of sequences and d_{star} is the distance implied by the star alignment. The first inequality is due to the triangle inequality – each pairwise comparison between two sequences can be bounded by the longer 'path' passing through the chosen center sequence. The second equality requires a bit of counting – there are k (k – 1) pairs of distinct sequences, and for each of them we compute $D(s_i, s_c)$ twice.

Let OPT be the score of the optimal alignment.

$$OPT = \sum_{i \neq j} d_{OPT}(s_i, s_j) = \sum_j \sum_i d_{OPT}(s_i, s_j) \geq \sum_j \sum_i D(s_i, s_j) \geq k \sum_i D(s_i, s_c)$$

The first equality simply transforms the pairwise sums into a collection of star alignments with a different sequence (j) as the center. The inequality is due to the fact that the pairwise distance between two sequences in the optimal alignment is at most as good as the optimal pairwise alignment between the sequences. The final equality is due to the fact that the star with the center sc obtains the best score from among all possible choices of center.

Combining the two equations above we get $\frac{S^{star}}{OPT} \leq 2 - \frac{2}{k} < 2$, i.e., the star alignment is at most twice as bad as the optimal alignment. We can also say that the star alignment achieves a 2-factor approximation.

5.1.2 Steiner and consensus string

Let us spend a bit more time on the alignment problem, at the conceptual level (not much math following here). Above we came up with an alignment approach that picked a special string, the center, to guide the alignment. This center string is one from among the strings being aligned. What if we could pick another string such that the score of the star alignment for this string is optimized? This string (denoted T in the following) is called the **Steiner string**. For this string, $\sum_i D(s_i, T)$ is minimized over all possible choices of T. In some sense you can get the intuition that an alignment built around T would have a better score than that built around another choice of center. Also note that this string T can at a very broad level be viewed as a way to summarize the alignment (though more precise ways of summarization will be described below). This string at some level embodies the commonality between the different strings being aligned.

Another way of summarizing an alignment is to construct a **consensus string**. Specifically, for each column in the alignment we pick the letter that minimizes the distance from all the other letters in the alignment (essentially a Steiner nucleotide or amino-acids). The resulting string is the consensus string (see below)

```
Seq1 ACC-AGTGA-CT
Seq2 -CGTAGCGA-CT
Seq3 ACC-AGTGT-CT
Seq4 ACGTAGTCATCT
Cons ACCTAGTGA-CT
```

Note that for large enough alignments of similar enough sequences, the consensus string simply comprises the most abundant letter in each column (the sum of distances contains many zeroes).

It is not immediately obvious but there is a connection between the consensus string and the Steiner string – specifically, the consensus string of the optimal alignment is actually the optimal Steiner string. Needless to say, finding this string is NP hard.

Nonetheless, the consensus sequence of a multiple alignment (not necessarily the optimal one) is commonly used in practice as a way to summarize the alignment, e.g., in the context of progressive alignment (sequences are iteratively aligned to the consensus sequence constructed so far).

5.1.3 Tree-guided alignment

Neither the optimization problem implied in the definition of the multiple alignment problem, nor the approximation provided by the star alignment have a biological interpretation. As a corollary, the resulting multiple alignments may not capture interesting biological phenomena. A more "biologically meaningful" approach to multiple alignment relies on the construction of a guide tree that, intuitively, captures the evolutionary relationship between the sequences being aligned. The guide tree stores the sequences at its leaves, and the progressive alignment approach described in the context of the star alignment can be modified to allow sequences to be merged together in a bottom-up fashion into a single multiple alignment.

Virtually all "popular" multiple alignment programs rely on this paradigm. Two heavily used programs are ClustalW (Thompson, Higgins et al. 1994) and Muscle (Edgar 2004).

5.1.4 Multiple alignment of genomes

The problem: so far we've discussed global multiple alignment – the sequences have to be aligned end-to-end. This approach simply doesn't scale to whole genomes. Also, whole genomes have rearrangements, events that are not captured in the typical edit-distance alignment algorithms.

Example: the Mauve system (Darling 2004) – http://asap.ahabs.wisc.edu/mauve/

Basic idea:
1. Find a collection of MUMs shared by 2 or more genomes (multi-MUMs).
Can think of a variety of ways of doing this, but in principle the basic idea is to first find exact k-mer matches across multiple genomes (making sure none is a repeat in any of the genomes) then extend these matches along groups of genomes as long as the corresponding sequences agree.

2. Find locally collinear blocks (LCB) among the MUMs – these are sets of MUMs that occur in the same order in all genomes. Note: the LCB algorithm requires all MUMs to occur in all genomes so it's only applied to those MUMs.

The idea of the LCB algorithm is to repeatedly sort the MUMs based on their occurrence in each genome, and mark breakpoints in this sorting – places when the MUMs sorted by genome i occur out of order in genome j.

3. A global guide-tree is constructed from all genomes, using the weight of the shared MUMs as a similarity/distance measure. This tree is then used to align the regions between the MUMs using ClustalW.

5.1.5 Exercises

1. The optimal Steiner multiple alignment for a set of k strings is the alignment that minimizes $E(S^*) = \sum_{S_i} D(S^*, S_i)$ over all possible choices of the Steiner string S*. Show that, if the scoring function D satisfies the triangle inequality, there exists a string $\overline{S}$ in the original set of strings such that $E(\overline{S}) / E(S^*) < 2$.

2. We showed that the sum-of-pairs score of the optimal star alignment is at most twice the score of the optimal multiple alignment of a set of sequences. Can you generalize this result? Specifically, the star is a special type of tree. Assume that instead of a star you could construct a tree containing the sequences being aligned at both the leaves and internal nodes. The score of such a tree is, just as in the star case, the sum of the optimal pairwise alignment scores for sequences adjacent in the tree. Assume you had an algorithm that could compute the optimal tree alignment (sum of distances is minimized). Can you prove anything about the sum-of-pairs score of the induced alignment? (Note: assume distances satisfy triangle inequality)

5.2 Profile alignments and hidden Markov models

Above we described the idea that the consensus string (or Steiner string) of a multiple alignment can be viewed as a 'summary' of a multiple alignment. Such a summary may come in handy, for example, when comparing a new protein sequence against a family of proteins aligned to each other in an alignment. For example, one could compare a newly discovered protein against a multiple alignment of myoglobins in order to determine whether the new protein is a myoglobin as well, and how it relates to the known sequences. Another possible use for such a summary is a better progressive alignment (see above under section 5.1.1) which avoids mistakes by taking into account more of the alignment than just its consensus sequence.

5.2.1 Profile alignments

The simplest approach for summarizing multiple alignments is the **alignment profile** approach proposed in (Gribskov, McLachlan et al. 1987). The idea is to construct a new type of consensus sequence that captures more than just the best character for each column. Instead we simply record all characters found within that column as well as their proportion, as shown below.

```
Seq1  A      C      C      -      A      G      T      G      A      -      C      T
Seq2  -      C      G      T      A      G      C      G      A      -      C      T
Seq3  A      C      C      -      A      G      T      G      T      -      C      T
Seq4  A      C      G      T      A      G      T      C      A      T      C      T
Prof  A(.75) C(1)   C(.5)  T(.5)  A(1)   G(1)   T(.75)G(.75)A(.75)-(.75) C(1)   T(1)
      -(.25)        G(.5)  -(.5)                C(.25)C(.25)T(.25)T(.25)
```

While the profile appears quite intuitive, we still have the problem of aligning sequences to it. Specifically, what is the score of aligning a particular sequence to a profile? We can define this score to be the expected alignment score between the character being aligned and the characters in the profile. More formally, assume we are aligning character S[i] to profile P[j], where P[j] is defined as a list of k pairs (p_l[j], C_l[j]) corresponding to the probability p_l[j] of observing character C_l[j] in column j of the multiple alignment. The score V[i,j] of aligning character S[i] to profile column P[j] is simply $V[i, j] = \sum_{1 \le l \le k} p_l[j] f(S[i], C_l[j])$, where the function f is the scoring function for aligning two characters, e.g., the value found in the BLOSUM matrix. Using this definition of score it is easy to see that one can simply use the traditional dynamic programming to align a sequence to a profile.

5.2.2 Hidden Markov models

The alignment of a sequence against an existing multiple alignment can be viewed simply as a machine learning/classification problem. The multiple alignment is some data-structure linking together the positive examples we have seen so far (e.g., various instances of myoglobin proteins), and the alignment of a sequence against this multiple alignment is equivalent to trying to decide whether a new example falls in the same category. For this purpose we can use formalisms developed in the context of machine learning, specifically Hidden Markov Models. Their use was originally proposed by Sean Eddy (Eddy 1995, Eddy 1998).

The basic idea is to convert a multiple alignment into a probabilistic generative model which produces sequences with similar properties as those contained in the alignment. The alignment of a sequence against such a model corresponds to computing the likelihood that the particular sequence would have been generated by the alignment.

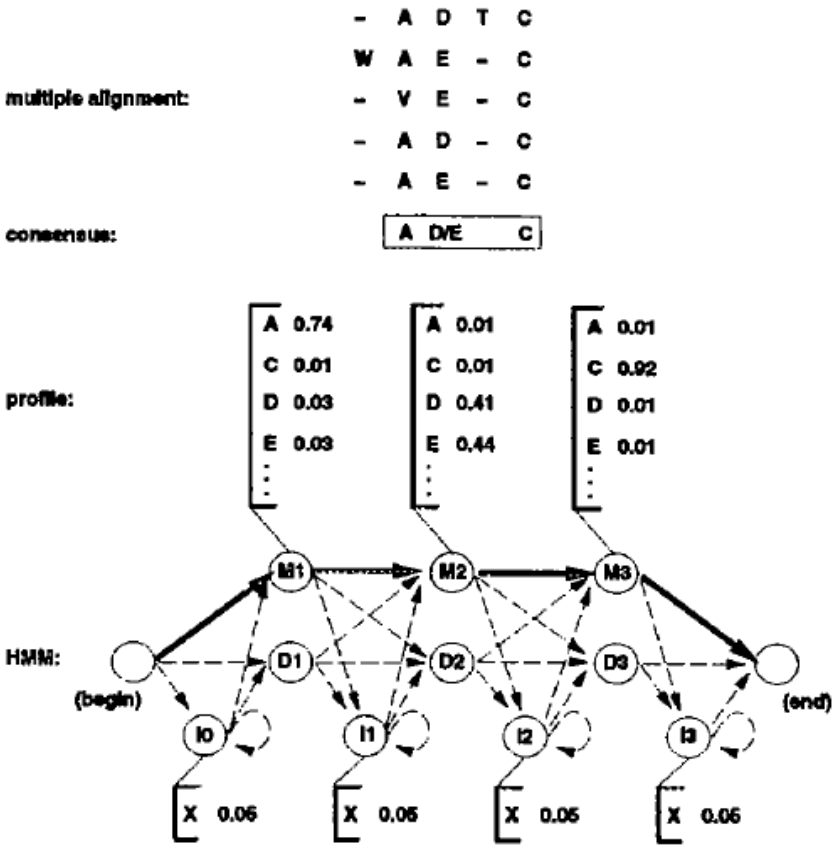


Figure 10: Example of multiple alignment Markov Model (from (Eddy 1995)*)*

To make things more concrete, the model comprises a series of states (like we earlier described in the context of automata) and transitions between these states. At a broad level we can view each state to correspond to a particular column in the alignment (the M states in the figure above). Each state also has associated with it an emission probability, indicating the probability a particular character is emitted by the model in that state. The most basic form of such a model simply links together the states corresponding to each column, and the emission probabilities are set to the profile described above. One slight change is necessary – the unobserved characters must be given some very low probability, otherwise the model would not be able to align new characters against that specific column. To allow insertions and deletions, additional states can be added (I and D above) which model an insertion of one or more characters between two columns, or the deletion of a column from the alignment. Insertion states also have emission probabilities, generally assuming all characters may be inserted with equal probability. Deletion states do not emit any characters. In addition, all transitions

have an associated transition probability.

To compute the probability that a particular sequence was emitted by a given model, we proceed with yet another version of dynamic programming, the **Viterbi algorithm**. As in the case of sequence alignment, we will try to compute V[i, j] – the probability that the prefix i of sequence S was generated by the model up to state j. We can see that $V[i,j] = ep_j(S[i]) max_{all\ pred} V[i-1, pred]\, tp(pred \rightarrow j)$ where ep_j is the emission probability in state j, tp is the transition probability between two states and *pred* is one of the states preceding j. In other words, the probability we reach state j with character S[i] is the product (logical AND) of the probablity S[i] would be emitted in state j and the probability that we would reach state j from some prior state *pred*. Obviously to maximize this score we choose the best such prior state.

5.2.3 Pairwise sequence alignment in a probabilistic setting

Earlier we discussed pairwise sequence alignment in terms of either minimizing edits or maximizing an alignment score, while in the HMM alignment we think of alignment as maximizing a probability. These two scenarios are not in fact that different. If you remember, the alignment scores are computed as a log odds $\frac{\log p_{AB}}{\log p_A \log p_B}$. Summing up these scores is equivalent to multiplying the corresponding probabilities – i.e., maximizing the score is equivalent to maximizing the probability of the alignment. The difference from the HMM case is that in HMMs the probabilities are computed from a specific alignment, while with BLOSUM the probabilities are derived from a broader set of sequences and are thus more general.

Note that we could formulate the pairwise alignment problem as an HMM alignment simply by constructing an HMM from a single sequence, with the same structure as the profile HMM described above, however with probabilities inferred from the BLOSUM matrix (with one sequence we don't really have the statistical power to estimate these probabilities). This formulation leads to essentially the same solution for the alignment problem as the traditional dynamic programming approach, but provides additional flexibility in defining the structure of possible alignments. As an example, affine gap penalties can be easily incorporated by creating two states instead of one for deletions and/or insertions, corresponding to gap openings and gap closing events.

5.3 Local multiple alignment/motif finding

Before discussing local multiple alignment we would like to highlight the strong dependency of the results on the assumptions of the scoring function used. There are a number of scoring approaches that have been developed in the context of multiple sequence alignment which include sum-of-pair scores (Murata, Richardson et al. 1985, Bacon and Anderson 1986), tree scores (the scores take into account an underlying phylogenetic tree connecting the sequences) (Sankoff 1975), entropy scores (Schneider, Stormo et al. 1986), or generalizations of log-odds scores to multiple alignments (Ye, Yu et al. 2010) (to be discussed further below). The tree scores are perhaps the more biologically relevant as they capture the underlying evolutionary relatedness of the sequences, however sum-of-pair scores are widely used due to their conceptual simplicity.

In the following we will ignore gap scores but would like to note that if used gap scores need to be considered within the same mathematical formalism/context as the substitution scores used (e.g., tree-based gap scores for tree-based substitution scores) (Altschul 1989).

Coming back to the local multiple alignment problem, we are specifically trying to find short segments within the sequences being aligned that can be aligned to each other into one or more high scoring multiple alignments.

We need to define an appropriate measure of alignment quality. The measure should ideally satisfy several conflicting requirements: (i) reflect known amino-acid relationships; (ii) make no constraints on alignment

width; (iii) handle missing or duplicate patterns; (iv) allow gaps; (v) be independent of the order of sequences; (vi) be computable by a deterministic algorithm; (vii) be computable in linear time. No known algorithm that satisfies all or even a sufficiently large subset of these criteria. The choice of constraints that are optimized depends on the specific requirements of the underlying problem. In the following we will describe several approaches for local multiple alignment and highlight the respective tradeoffs involved in these approaches.

First, some notation:

N – number of sequences
L – average seqence length
A – alphabet size
W – width of pattern/alignment

5.3.1 Consensus word method

The idea behind the consensus word method is to attempt to find some word in one of the sequences that is similar to words in some or all of the sequences being aligned. Essentially we are building a star alignment around a chosen word.

For each word of width W in one sequence we compute the neighborhood of B adjacent words (may differ by 1, 2, etc. letters), with an associated score/penalty. We then find within each sequence words that belong to this neighborhood. The approach proceeds until one word is found that has near neighbors in most of the sequences.

The runtime is $O(NLB + A^W)$, and space $O(A^W)$.

The consensus word method is efficient (linear time) for a fixed word size, however it places significant restrictions on pattern length and the words that may align to the chosen consensus word (words outside the chosen neighborhood are discounted). These restrictions limit its use in protein sequences.

Some papers describing this approach are: (Queen, Wegman et al. 1982, Waterman, Arratia et al. 1984, Galas, Eggert et al. 1985, Staden 1989)

5.3.2 Template method

The template method extends the consensus word approach by defining a set of templates of size B, (e.g., V*C**D – where * is a wild card). These templates are iteratively searched through the set of sequences and words that best match the template are retained.

This approach has the same advantages and disadvantages as the consensus word approach.

Some papers describing this approach are: (Sobel and Martinez 1986, Posfai, Bhagwat et al. 1989, Smith, Annau et al. 1990, Leung, Blaisdell et al. 1991)

5.3.3 Progressive alignment

The progressive alignment approach is similar to progressive alignment heuristics used in global multiple alignment. Specifically, the approach starts by finding local alignments in the first two sequences. Then additional sequences are added iteratively each time attempting to match to only the highest-scoring local alignments from earlier iterations.

The runtime is O(NLBW).

This approach can be viewed as a generalization of the template method where the template is refined throughout the iterations. The approach places no constraints on pattern width and allows arbitrary scoring

functions. At the same time, however, the approach is very sensitive to sequence order and cannot guarantee an optimal solution.

Some papers describing this approach are: (Bacon and Anderson 1986, Stormo and Hartzell 1989, Hertz, Hartzell et al. 1990)

5.3.4 Pairwise consistency methods

Pairwise consistency methods extend the iterative approach by focusing on all pairwise alignments of sequences, alignments which can be seen as defining two-dimensional diagonals ina multi-dimensional search space. Sets of consistent diagonals are then used to define multi-dimensional diagonals along which a multiple alignment is computed.

The runtime is $O(N^2L^2 + f(H))$ where H is the score threshold used to define a good scoring diagonal.

This approach has many advantages – does not restrict pattern width, is independent of sequence order, can use arbitrary scores, and can find the optimal alignment. Its efficiency is, however, poor, especially for low alignment quality thresholds.

The approach is implemented in the MACAW program (Schuler, Altschul et al. 1991).

Note that the efficiency of the approach is dependent on the number of dimensions. While there are roughly $N\ L^{N-1}$ diagonals that need to be considered, $\binom{N}{2}$ 2-dimensional diagonals must be consistent in order to define an N-dimensional diagonal. As the number of dimensions increases the number of valid choices rapidly decreases.

Another paper on this approach is (Vingron and Argos 1991).

5.3.5 Gibbs sampling

Gibbs sampling (Geman and Geman 1984) falls into the general class of optimization approaches for high-dimensional spaces. The structure of the solution space has a huge impact on the efficiency of the optimization approach. For example, in spaces where the optimization function is smooth and has a single extreme value the solution can be easily found through analytical method. In spaces with no structure (multiple random minima/maxima) the optimization problem is intractable. Most of the practical problems focus on spaces where the optimization function has multiple, but not random, local optima

In our case we are focusing on N sequences of fixed width W and try to find the highest scoring ungapped alignment combining these sequences. Note that as described earlier, from any given multiple alignment we can construct a profile which can then be used to evaluate the 'fit' of a sequence with this multiple alignment. The Gibbs sampling approach essentially alternates between defining the profile and optimizing the alignment of sequences to this profile (Lawrence, Altschul et al. 1993).

Specifically, the process starts by picking a random multiple alignment of words of width W from the sequences. Then, an iterative process starts by removing one sequence from the alignment and constructing a profile from the remaining sequences. The resulting profile is then used to find words in the removed sequence that could be aligned with a good score to the profile. One of these is selected at random with a probability proportional to its fit to the profile, and the process (pull out a sequence and replace it with a better fit to the remaining profile) is repeated until convergence.

Why does this approach work? As we 'fit' new sequences to the profile, the corresponding scoring matrix gets more and more biased towards patterns matching the already aligned sequences, and the corresponding scores get improved.

This simple approach is quite powerful but has several limitations. First, there is no guarantee that the approach will find the optimum and it's generally useful to restart the process from different random configurations. Also, the solution can focus on a sub-optimal shifted solution – it is easy to converge to a solution very close to the optimal, essentially overlapping the optimal pattern. A possible solution is adding a new type of 'move' in the algorithm that shifts the entire pattern by a few positions in all sequences.

Note that we presented the algorithm with a fixed width – we can interleave the general iterative approach with new 'moves' that increase or decrease the width of the alignment in order to capture more of the information. Such approaches should be used carefully as it's easy to overtrain by focusing on very specific (generally short widths).

Finally, a major limitation of Gibbs sampling approaches arises in situations where there are many nearly identical versions of sequences – the approach relies on the fact that removing one sequence changes the profile sufficiently to allow a better fit of another word from that sequence. When multiple sequences are close to identical, removing one of them will have no impact on the alignment profile, ultimately biasing the final alignment to sub-optimal patterns in these highly similar sequences.

5.4 Scoring systems for multiple alignments

As described above, an important question to consider is the scoring function used to define the quality of a multiple alignment. Specifically, how do we score the alignment of an amino-acid letter to a column of letters. The traditional approach, as described above, is to simply average the scores of pairwise alignments (sum-of-pairs scores). This approach, however is unable to capture intrinsic properties of the multiple alignment. Imagine, for example, a position in a protein that only contains leucines in all variants of the protein. We would ideally want to strongly discourage alignments without a leucine in this position, yet averaging pairwise alignment scores do not have this effect.

A solution to this problem is to generalize the log odds scores from pairwise to multiple alignments:

$s_i = \log \frac{q_i}{p_i}$ where q_i is the probability of observing amino-acid i in that column. We can estimate the probabilities q_i as simple fractions, but we do not know how to deal with probabilities for situations we haven't seen. For example, the amino acid tryptophan has a background probability of 1/100, so it will not be observed in most alignment columns unless many sequences are examined.

We can view this problem in a Bayesian context. We want to put a prior on the space of all possible multinomials (the mixture of the distribution of the frequencies of amino-acids that add to 1), which we modify based on observations to find the posterior probability distribution.

We can view the multinomial as a generalized tetrahedron in a multi-dimensional space (20 dimensions for amino-acids), such that each point on the tetrahedron represents a particular combination of amino-acid frequencies, all adding to 1 (the tetrahedron has effective dimension 19 as the last amino-acid's frequency is defined by that of the others).

Mathematically, the prior is a Dirichlet distribution $p(\vec{x}) = Z \prod_i x_i^{\alpha_i - 1}$ where $Z = \Gamma(\sum \alpha_i) / \prod \Gamma(\alpha_i)$ chosen such that $p(\vec{x})$ integrates to 1.

The simplest way to think about this distribution is to imagine a hill on the multi-dimensional tetrahedron. The distribution can be parametrized by the center of mass $\vec{p} = \vec{\alpha} / \sum \alpha_i$ and by $\alpha = \sum \alpha_i$. Intuitively, the vector $\vec{p}$ defines the center of mass of the distribution and α defines how 'peaked' the distribution is, i.e., how much of the distribution mass is concentrated around the center of mass.

As we observe events (specific amino acids), we can modify our prior estimation (generally assigning all

amino-acids an equal representation) by shifting the center of mass and increasing the height of the corresponding 'hill', thereby deriving the posterior Dirichlet distribution. increase the height of the 'hill' to define the posterior distribution.

A single Dirichlet distribution fails to capture our prior knowledge concerning relationships between the amino-acids (distinct regions of the multinomial space are known to be favored by proteins – not all alignment columns are the same). To incorporate additional information, such as alignment positions that like to be positively charged, hydrophobic, etc., we can use a mixture of Dirichlet distributions – the Dirichlet mixture - a richer description of the prior. Specifically, this mixture consists in a collection of 'hills' each summarizing a specific type of multiple alignment column. Each component has 21 parameters (19 are the position of center, 1 peak, and the amount of the total probability mass assigned to it). Total number of parameters of the Dirichlet mixture is thus M(L+1) – 1 for an M-component mixture. This formalism was introduced by (Brown, Hughey et al. 1993).

As in the case of pairwise alignment scores, we cannot estimate the priors for the Dirichled mixture from first principles. We, thus, have to derive these priors from actual experimental data. As before, we invert the problem, starting with putatively correct multiple alignments, each column in these alignments providing information about possible components of this mixture. Fitting a maximum-likelihood Dirichled mixture from the data is a difficult optimization problem which can be solved using typical approaches such as Gibbs sampling (Geman and Geman 1984), the EM algorithm (Dempster, Laird et al. 1977, Brown, Hughey et al. 1993), Metropolis search (Metropolis, Rosenbluth et al. 1953), etc.

Here we will describe a Gibbs sampling approach. The approach starts with N multiple alignment columns, and attempts to find M components (for now we assume we know the number of components but we will relax this assumption later). The alignment columns are assigned to individual components at random, and this information is used to estimate the individual Dirichlet distributions. We then proceed by randomly picking one of the columns and re-assigning it to a distribution within which it fits better. Just as described earlier for motif finding, this process is repeated until convergence.

Simple approaches for choosing the appropriate number of components (e.g., picking the set of components that maximizes the fit with the data) are prone to overfitting. Instead one can use the minimum description length principle (Grünwald 2007) – select the least complex model that can explain the data. Essentially we define the description length of the data S given a model M – DL(S|M) - as the log probability of S implied by the maximum likelihood embodied in the model M. The complexity of the model COMP(M) is the log of the effective number of the independent theories embodied in the model. The best model minimizes DL(S|M) + COMP(M). Essentially, we penalize simultaneously the complexity of the model and the fit of the data to the model. This approach is a formalization of Occam's razor.

The specific approach used to simultaneously pick the optimal number of components and fit the parameters of these components is based on a Dirichlet process – a process that essentially provides a prior distribution on the weights of the components, assuming a possibly infinite set of components. This process is related to the Chinese restaurant process – customers arrive at a restaurant with an infinite number of tables, each of which can sit an infinite number of people, and can sit at any table they choose. The probability that a customer chooses a particular table is proportional to the number of people already sitting at the table, though the customers can also choose to sit at an empty table with a probability determined by a parameter γ. The tables can be seen as the individual components of the Dirichlet mixture, and it is easy to see that this approach does not restrict the number of components but encourages the number of components to stay low (it is more likely to sit at an occupied table than to pick an entirely new one).

Gibbs sampling can be modified to operate without a prior assumption on the number of components, by allowing a column to start a new component with some low probability. Also, components can disappear if all the columns have been removed.

This approach was recently used in (Nguyen, Boyd-Graber et al. 2013) and revealed that the actual landscape of protein multiple alignment scores is best explained by a probability 'ridge' rather than a collection of individual 'peaks'. The Dirichlet mixtures estimated from the data comprise large numbers of components.

5.5 Exercises

1. Assume you are given a set of K sequence of length L. You are also given a special C or Java function:

Alignment AlignSeqs(Gapped_sequence S1, Gapped_sequence S2);

a) Write out the pseudo-code for constructing the multiple alignment of these sequences, using the AlignSeqs function.

b) Do you need to extend the AlignSeqs function to accept other types of inputs? Please provide the definition (parameter and return types) for any additional functions you might need and briefly outline their functionality.

c) What is the running time of your algorithm?

2. The optimal Steiner multiple alignment for a set of k strings is the alignment that minimizes $E(S^*) = \sum_{S_i} D(S^*, S_i)$ over all possible choices of the Steiner string S^*. Show that, if the scoring function D satisfies the triangle inequality, there exists a string $\bar{S}$ in the original set of strings such that $E(\bar{S}) / E(S^*) < 2$.

3. We showed that the sum-of-pairs score of the optimal star alignment is at most twice the score of the optimal multiple alignment of a set of sequences. Can you generalize this result? Specifically, the star is a special type of tree. Assume that instead of a star you could construct a tree containing the sequences being aligned at both the leaves and internal nodes. The score of such a tree is, just as in the star case, the sum of the optimal pairwise alignment scores for sequences adjacent in the tree. Assume you had an algorithm that could compute the optimal tree alignment (sum of distances is minimized). Can you prove anything about the sum-of-pairs score of the induced alignment? (Note: assume distances satisfy triangle inequality)

6 RNA structure and structural alignment

RNA molecules are usually single-stranded (though double-stranded RNA also exists in nature) and often fold into a three-dimensional structure driven by the pairing of complementary nucleotides. The exact shape is dependent on many biophysical factors, however a reasonable approximation can be obtained using fairly simple assumptions.

6.1 RNA structure prediction (RNA folding)

The simplest definition of the RNA folding problem involves pairing up complementary nucleotides (A-U, C-G) in a way that maximizes the number of "contacts" - correctly paired nucleotides. This problem can be easily solved if we assume all pairings are properly nested, i.e. the structure can be expressed as a series of parantheses:

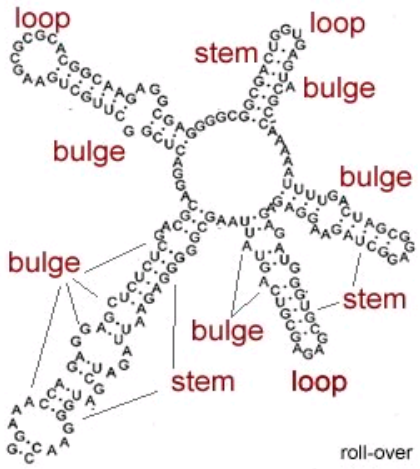


Figure 11: RNA fold structure (from: http://virtuallaboratory.colorado.edu/Biofundamentals/lectureNotes/Topic3-2_NucleicAcids.htm)

```
>Sample RNA
AAAAAAAAAAAAAGGGGGGGUUUUUUUUUUUUUUCCCCCCCCCCCCCCCCC
.............(((((((.............)))))))..........
```

In many cases this assumption is not true in real life, resulting in what are called pseudo-knots - pairings of nucleotides that do not obey the "proper parenthesis" rules (see figure below). Figuring such situations out is a lot harder to do computationally and will be discussed under probabilistic folding methods below.

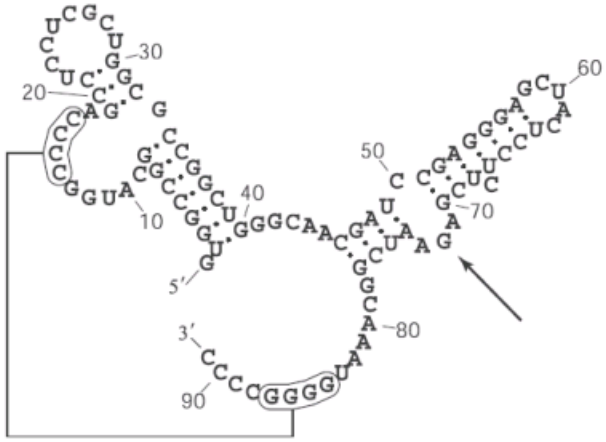


Figure 12: Pseudo-knot - the link indicates nucleotides that are physically "connected" in the 3D conformation.

6.2 Nussinov's algorithm – Dynamic Programming folding approach

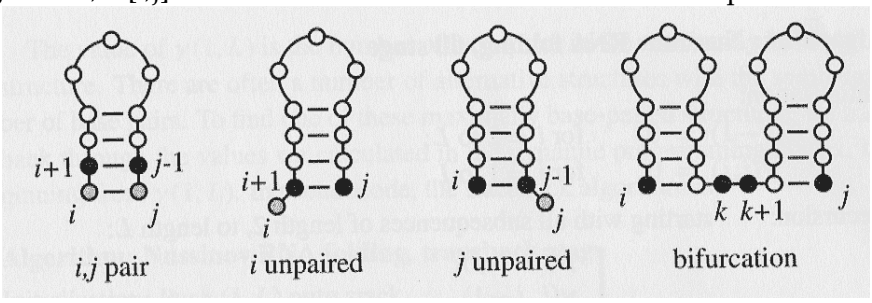
This algorithm is a variant of the CYK algorithm used to parse Chomsky Normal Form grammars.

The basic idea is to focus on the simpler problem of folding a short stretch of RNA – i.e. the dynamic programming recurrence will build upon value V[i, j] - the number of contacts in the best fold of substring S[i..j].

We can distinguish four different cases:

1. V[i,j] = V[i+1, j-1] + 1 - if nucleotides S[i] and S[j] match
2. V[i,j] = V[i+1,j] - nucleotide S[i] is unpaired
3. V[i,j] = V[i, j-1] - nucleotide S[j] is unpaired
4. V[i,j] = $\max_{i<k<j}$ (V[i,k] + V[k+1,j]) - fold of S[i..j] is built from two subfolds of S[i..k], S[k+1,j]

At each step in the algorithm, V[i,j] will be set to the maximum of the 4 cases specified above (see figure below)



As initial values, the DP matrix can be filled in with 0s for V[i,i] and V[i, i+1] as adjacent nucleotides cannot be paired up. Also, if we expect that loops cannot have fewer than k nucleotides, we could set V[i, i+j] to 0 as well for all j < k.

Once the initialization is complete, the DP matrix can be filled in, diagonal by diagonal (for each diagonal the difference between i and j is constant) starting from the main diagonal (i-j = 0) as all the information needed will be available in previous diagonals. V[1,n] will eventually contain the maximum number of contacts achievable.

If we want to find out all the pairings, the algorithm will need to backtrack from V[1,n] using a similar approach as that described for sequence alignment. There is one catch, however, the backtrack pointers form a tree structure due to the bifurcation operation (case 4 above) – and the backtrack needs to recurse on either side of the bifurcation.

6.3 Zucker's algorithm – Extension to actual energy functions

Nussinov's algorithm simply assumes that each nucleotide pairing is equally good, and also that there are no penalties for unpaired bases – both assumptions that are incorrect in the real world. To account for these factors Michael Zucker extended Nussinov's algorithm to incorporate energy terms (see http://www.bioinfo.rpi.edu/zukerm/rna/energy/ for a list of the structures being evaluated). The goal here is to find the fold that minimizes the free energy of the resulting structure – a more biophysically reasonable objective function than # of base pairings.

A full description of Zucker's algorithm (implemented in the program mfold) is beyond the scope of this document. I just want to mention two features. First, the algorithm allows the computation of sub-optimal folds by performing several rounds of (imperfect) backtracking with the goal of identifying structures that have similar energies but differ significantly. The assumption is that due to a variety of reasons (e.g. energy terms cannot be correctly computed and differ from actual energies in the real system) the actual structure might not be the one that minimizes the energy, rather is one of the many structures with low energy.

A second feature of Zucker's algorithm is that it allows the computation of "stacking" energies – taking into account the fact that the energy of a stem is not simply the sum of the energies of the individual base-pairings. To account for length-specific stem energies the algorithm uses a similar approach as that used in computing affine gap penalties in sequence alignment.

6.4 Probabilistic folding

The folding approaches described above assume a fair amount of prior knowledge – which bases can be paired, what are the energy terms for certain simple structures, etc. Assuming we had a large number of sequences that we know fold the same way (e.g. because they do the same things), could we learn the fold itself?

A simple approach starts by taking all the sequences and constructing a multiple sequence alignment. Within this alignment (assuming it's well built – which is generally not the case given that multiple alignment is NP-hard) we can assume that each column represents a same position within the fold common to all the sequences. We can observe that if two nucleotides are paired up in the common fold, then these nucleotides need to be complementary in all the sequences, i.e. the corresponding columns in the alignment are correlated. A way to measure this correlation is through the concept of mutual information – how much information can we learn about column j given that we know the content of column i.

Mathematically, the mutual information between columns i and j can be expressed as:

$$M(i,j)=\sum_{x_i,x_j} f_{x_i x_j} \log_2 \frac{f_{x_i x_j}}{f_{x_i} f_{x_j}}$$

where x_i and x_j are the corresponding nucleotides in the two columns within the aligned sequences, and the f terms are the frequencies of a specific pairing or nucleotide, respectively, in the whole set of sequences. M(i,j) varies between 0 (no correlation) and 2 (full correlation). By computing M(i,j) for all pairs of columns in the alignment we can identify those positions that are likely paired up in the final structure. If we assume the structure is properly nested we can use a variant of Nussinov's algorithm to maximize the mutual information

between paired-up positions. In the non-nested case (pseudo-knots) the problem is likely NP-hard.

6.5 Structure alignment

When aligning sequences it is often the case that at least one of the sequences being aligned has some known structure. An example are 16S ribosomal RNA databases – the structure of the ribosomal RNA is known, now we want to use this information to guide the alignment of a new sequence against the database.

Jiang et al. (Jiang, Lin et al. 2002) describe several variants of this problem depending on whether the structure is nested or not, and on whether the structure is known for one or both of the sequences. In brief, the (NOSTRUCTURE, NOSTRUCTURE) alignment is simply the traditional sequence alignment algorithm. (NOSTRUCTURE, NESTED) can be solved with a variant of Nussinov's algorithm, and all other variants ((NESTED, NESTED), (NON-NESTED, NOSTRUCTURE), (NON-NESTED, NESTED), (NON-NESTED, NON-NESTED)) are NP-hard.

Note that formalizing the structure-sequence alignment can be challenging – Jiang et al. create a set of scores for the various ways in which pairings are broken up during alignment.

An alternative approach is a probabilistic alignment framework similar to the profile HMMs described earlier in the class. A good example is the Infernal package from the Eddy lab (Nawrocki, Kolbe et al. 2009) which relies on stochastic context free grammars to take into account the pairing of columns in a multiple sequence alignment then extends the grammars in a similar way HMMs extend the standard profile alignment.

6.6 Exercises

1. Bacterial transcription terminators are characterized by a stem-loop secondary structure. Below is the recurrence for a simple adaptation of Nussinov's algorithm to the problem of finding stem-loops in a string of DNA. Note that we are not trying to find if the whole string of DNA folds into a stem-loop, but to find a section of the DNA that likely folds into a stem-loop structure. In addition, we wish to guarantee that the loops are at least 4 nucleotides in length. In the equation below, F[i,j] represents the score of folding the substring S[i..j] from the original string of DNA.

$$F[i,j] = max_{i,j}\begin{Bmatrix} F[i+1,j-1]+1 \; iff \; S[i], S[j] \; are \; complementary \\ F[i+1,j] - penalty(S[j] \; unpaired) \\ F[i,j-1] - penalty(S[i] \; unpaired) \end{Bmatrix}$$

Please answer the following questions:

a. What are the initial conditions for this algorithm?

b. Where will the answer be located?

c. Why did we have to subtract a penalty in case of unpaired sequences (note Nussinov's algorithm does not have such a penalty).

2. Nussinov's RNA folding algorithm could be extended to allow the incorporation of "stacking" energies, i.e. the incorporation of a score term S[k] representing the score of k adjacent base-pairings within a stem. Assume S[k] != k. Provide the details of a modified Nussinov algorithm that can take these stacking energies into account.

Please be thorough: describe the dynamic programming table(s) you need, what the values stored in these tables represent, as well as the recurrence equations that allow you to fill in the table.

3. 1. Given the following sequence representing an RNA molecule
ATTAGCTGCCAACACTCAGCTAA
draw a possible fold of the sequence into a hairpin structure.

7 Sequence clustering and phylogeny

7.1 Introduction

Data clustering is one of the key tools used in various incarnations of data-mining - trying to make sense of large datasets. It is, thus, natural to ask whether clustering approaches can enhance our understanding of biological sequences. A common application of clustering is the analysis of 16S rRNA sequences extracted from an environment. In theory, each distinct rRNA sequence should represent a single organism, however, due to sequencing errors and recent evolutionary changes a single organism is represented by many closely related RNA sequences. Thus, by clustering the set of sequences we might infer how many organisms are present in our sample.

Clustering approaches can be classified into three main categories:

- Agglomerative – algorithm starts with each sequence as an independent cluster, then iteratively joins together sequences to create larger clusters
- Divisive – algorithm starts with all the sequences in one cluster then iteratively breaks up the cluster into a collection of smaller clusters.
- Partitioning/partition methods – e.g. k-means – algorithm starts with a pre-defined set of clusters then refines the cluster membership to improve cluster quality.

Various approaches use different criteria for when the clustering should stop, usually based on some concept of cluster quality. The basic goal is to create clusters such that all sequences within a cluster are similar to each other(clusters are compact), and sequences contained in different clusters are dissimilar to each other (clusters are well separated). There are a variety of mathematical measures that capture one or both of these criteria. Also, the quality of a cluster can be assessed either in an unsupervised way (clustering algorithm and validation algorithm have access to the same information) and in a supervised/assisted way (validation algorithm has access to additional information that can be used to evaluate cluster quality). A good introduction to these topics is presented in (Handl, Knowles et al. 2005).

Note that clustering algorithms need to determine explicitly or implicitly the distance between the objects being clustered. In the context of sequence data, such distances are defined by measures of sequence similarity such as edit distance. In the general case one must compute all pairwise distances between sequences – a process that is computationally prohibitive for large datasets. Various heuristics can be used, in specific situations, to reduce the runtime either by using simple measures of sequence similarity, or avoiding to compute all pairwise distances. For example, the triangle inequality can be used to bound the distance between sequences without actually computing it.

Some good introductions to clustering are: (Fasulo 1999, Jain, Murty et al. 1999).

7.2 Partitioning methods

The prototypical partitioning clustering approach is the k-means algorithm (see e.g., (Kanungo, Mount et al. 2002)). This approach is very similar in spirit to the Gibbs sampling approach. The goal is to cluster N data points (in our case we can assume they are sequences) in to K different cluster, where K is known *a priori*. The clustering will attempt to optimize the quality of the clusters, usually defined as the squared distance between the clustered sequences and the corresponding cluster centers. K-means clustering starts by randomly assigning sequences to K clusters, then computing the corresponding cluster center. Then the process iteratively reassigns sequences to the nearest cluster center, recomputes the centers, etc., until convergence.

An approach used to speed up the k-means approach, by avoiding the computation of exact distances is

canopy clustering (McCallum, Nigam et al. 2000). This approach relies on approximate distances to construct a series of 'canopies' that cover the objects being clustered. A more precise clustering algorithm is then performed within each canopy. Specifically, a data point p is selected and all other points within a distance threshold T1 from p are added to p's canopy. Among these, a second threshold T2 < T1 determines the points closest to p which are then 'frozen' and no longer processed. The process proceeds by selecting another point not found within p's canopy, until all points are assigned to a canopy. Note that the different canopies can overlap. Also, the number of canopies found can provide a suggestion for the choice of K when running k-means to refine the clustering.

7.3 Hierarchical clustering

Hierarchical clustering is a variant of agglomerative clustering that continues the agglomeration process until only one cluster remains. Essentially, through hierarchical clustering you construct a tree, having the sequences as leaves, that captures the relationship between the sequences at various levels. Clusters can be constructed by cutting the tree at particular points (imagine slicing a head of broccoli with one slice - each floret will be an individual cluster) - i.e. for a given node, all its descendents are part of the same cluster.

The basic hierarchical clustering algorithm proceeds as follows, in a greedy manner:

1. Compute all pair-wise distances between the sequences
2. Merge together the sequences that are closest (most similar) to each other
3. Compute the distance between the newly created cluster and all other sequences/clusters
4. Repeat from 2.

Different hierarchical clustering methods differ in the way they define the distance between already computed clusters, or between clusters and individual sequences. Thus we have:

- Nearest neighbor (single linkage) - distance between clusters i and j is equal to the **smallest** distance among all pairs of sequences, the first from cluster i and the second from cluster j.
- Furthest neighbor (complete linkage) - distance between clusters i and j is equal to the **largest** distance among all pairs of sequences, the first from cluster i and the second from cluster j.
- Average neighbor (average linkage, UPGMA) - distance between clusters i and j is equal to the **average** distance among all pairs of sequences, the first from cluster i and the second from cluster j.
- Ward's clustering – the clusters being merged are the ones that will construct a cluster with the lowest variance in within-cluster distances.

7.3.1 Semi-supervised hierarchical clustering (VI-cut)

In general the goal of clustering is to uncover certain relationships between a set of previously unknown sequences. In many cases, however, these relationships are known, or can be inferred. In the 16S rRNA example above, we might already know the organisms associated with some of the sequences. This information could be used to validate the clusters constructed through one of the clustering methods. A good metric for comparing the "true" clustering to the computed clustering is the Variation of Information metric: VI(C1,C2) = H(C1) + H(C2) - 2 I(C1,C2), where H is the entropy, and I is the mutual information between these clusterings. VI(C1,C2) = 0 if and only if C1 and C2 are the same clustering.

Navlakha et al. (Navlakha, White et al. 2010) showed that the VI distance has a nice decomposition property (it is easy to compute the VI distance between two clusterings given the distances between subsets of these clusterings) which allows the construction of a dynamic programming algorithm for finding the "perfect" cut-set in a hierarchical clustering tree – i.e. the set of cuts in the tree that maximizes the concordance between the

resulting clusters and previously known clustering of a subset of the sequences. This approach can be considered as semi-supervised as it relies on prior information about some of the objects being clustered.

7.3.2 Exercises

1. Given the following distance matrix, construct a UPGMA tree of the 6 sequences (A-F) shown below.
   a) Write out the formula for computing the distance between clusters (BC) and (AD).
   b) Fill in the intermediate distance matrices (provided below)
   c) Show the topology of the tree.
   d) How would answer to a) change if you were performing farthest neighbor clustering?
   e) How would answer to a) change if you were performing nearest neighbor clustering?

	A	B	C	D	E	F
A	0	2	5	7	8	8
B	2	0	5	7	8	8
C	5	5	0	6	7	7
D	7	7	6	0	5	5
E	8	8	7	5	0	2
F	8	8	7	5	2	0

2. Above we mentioned that furthest-neighbor clustering tends to create tighter clusters than nearest-neighbor clustering. Please explain this intuition. Also, provide an example (a collection of sequences/points and corresponding distances) demonstrating this effect.

7.4 Phylogenetic trees

The hierarchical clustering approaches described above can apply to any type of data, as long as a distance can be defined between the objects being clustered. One biological application of clustering attempts to infer the evolutionary history that gave rise to a set of sequences observed today. In this version, the distance between sequences attempts to capture the evolutionary relatedness of the sequences, or the time (usually measured as number of mutations) that separates the sequences from each other.

Thus, the goal is to reconstruct a tree that best captures the "heritage" of a set of sequences – the root of the tree being the most recent common ancestor of all the sequences. In the context of protein sequences, the BLOSUM and PAM matrices discussed during sequence alignment provide a reasonable approximation of the evolutionary distance. For nucleotide sequences, several models of evolution apply that attempt to estimate the likelihood a certain base will mutate into another base during a given evolutionary time. The evolutionary time separating two sequences can be estimated from the number of identities in the alignment between the sequences.

7.4.1 Neighbor-joining

Simple hierarchical clustering approaches have been applied in the context of phylogenetic tree reconstruction, specifically, the UPGMA (average linkage) clustering algorithm. One limitation of this algorithm is that it does not allow for uneven mutation rates along different branches of a tree. A solution is proposed by the neighbor-joining algorithm of Saitou and Nei (Saitou and Nei 1987). The basic idea is to correct the distance between two

sequences/clusters by a factor that depends on the distance between each sequence and the rest of the sequences. Part of the intuition is that two sequences should be clustered together if they are "isolated" (far from) the rest of the sequences, even if they are fairly apart from each other.

$NJD(i,j) = d(i,j) - \frac{1}{n-2}\left(\sum_{k \neq i,j} d(i,k) + d(j,k)\right)$ where d(i,j) is the distance between sequences i and j (out of n sequences).

The two sequences with the lowest neighbor-joining distance are clustered together. Whenever two sequences are clustered together, a new node (cluster) *m* is created and it's distance to all other sequences/clusters is computed as follows:

$$d(m,k) = \frac{1}{2}\left(d(i,k) + d(j,k) - d(i,j)\right) \forall\, k \neq i,j$$

$$d(i,m) = \frac{1}{2}\left(d(i,j) + \frac{1}{n-2}\left(\sum_{k \neq i,j} d(i,k) - \sum_{k \neq i,j} d(j,k)\right)\right)$$
$$d(j,m) = \frac{1}{2}\left(d(i,j) + \frac{1}{n-2}\left(\sum_{k \neq i,j} d(j,k) - \sum_{k \neq i,j} d(i,k)\right)\right)$$

Then the process is repeated until all sequences belong to a single cluster (the root of the tree).

The neighbor-joining algorithm can be shown to produce the optimal phylogenetic tree – in terms of reconstructing the true evolutionary history of a set of sequences – if the pairwise distances between sequences are additive, i.e., $d(i, j) = d(i, k) + d(k, j)$ for any sequences *i, j, k*. This property should hold for true evolutionary distances however is unlikely to hold for distances computed heuristically, e.g. through inexact sequence alignment.

7.4.2 Computing parsimony/likelihood scores

Complementary to distance based methods, which assume that we can accurately estimate the evolutionary distance between sequences without using any information about their heritage, are approaches that define global measures of the quality of a phylogenetic tree.

The two major classes of such methods are *parsimony* (best tree is the one that minimizes the number of mutations during evolution) and *likelihood* (best tree is the one that optimizes the "fit" with a predefined stochastic model of evolution). A key component of these methods is an algorithm for computing the "goodness" of a tree, given a set of sequences stored at its leaves. For simplicity, I will only focus on the parsimony approach, however the algorithm can be easily adapted to compute the likelihood of a tree.

Also, I will focus on a special case where each sequence is comprised of a single nucleotide – for longer sequences the parsimony score (to be defined) is simply the sum of the parsimony scores of each individual character. Note that in phylogeny we generally try to work with "traits" (in our case nucleotides) that can be inferred to be related to each other. In the context of sequence analysis a trait is represented by one column in the multiple alignment of the sequences being aligned, as we can infer that all the letters in that column have been derived, through evolution, from a same ancestral nucleotide. If this assumption is incorrect, so will the conclusions of the analysis – hence the "obsession" in the field for manually curating multiple alignments. Since the multiple alignment problem is NP-hard, the alignments found in practice by heuristic approaches frequently contain errors.

Problem definition: Given a phylogenetic tree (assume binary for now but not an essential restriction) having

the leaves labeled with letters from a given alphabet (ACGT for nucleotides), determine a labeling of the internal nodes of the tree (i.e. ancestral sequences) that minimizes the sum, over all the edges in the tree, of the number of differences between adjacent letters.

In other words, once you label all the nodes of a tree, you can label all the edges with a 1 if the ends of the edge have different labels, or 0 if they have the same label. We are looking for the assignment of labels in the tree that minimizes the number of 1s (in the picture below the parsimony score is 2).

The computation of the parsimony score, and the assignment of ancestral sequences, can be easily done with dynamic programming. Specifically, we'll define *V[n, c]* for all nodes *n* in the tree, and all characters *c* in the alphabet, to be the minimum parsimony score for the subtree rooted at *n*, given the root is labeled *c*.

V[n,c] can clearly be easily computed for the leaves by setting *V[leaf, leaf_label] = 0*, and *V[leaf, not_leaf_label] = -∞*.

Also, once the *V* values have been computed for the children of a node *n*, the *V* values for *n* can be easily computed with the following recurrence:

$$V[n,c] = min_{i \in \Sigma}(V[l,i] + \{1, if\ i \neq c\}) + min_{i \in \Sigma}(V[r,i] + \{1, if\ i \neq c\})$$

where *l* and *r* are the left and right children of *n*, respectively.

Using this recurrence, the tree can be filled in using a post-order traversal.

The parsimony score is the minimum value in *V[root,]*. The ancestral states can be easily assigned through backtracking.

The algorithm described in this section is due to David Sankoff and is commonly known as Sankoff's algorithm (Sankoff 1975).

7.4.3 Finding the best tree

While Sankoff's algorithm provides a way to evaluate (in polynomial time) the quality of a given tree, finding the tree that maximizes the quality is NP-hard. Most algorithms for constructing phylogenetic trees rely on heuristic search through the space of possible trees. A discussion of these algorithms is beyond the scope of this class.

7.4.4 Models of evolution

The calculation of a maximum likelihood phylogenetic tree implies an underlying model of how evolution works. Such models are generally represented by the probability that one amino-acid or nucleotide changes into another one over some given evolutionary period: $P(A_i | A_j, t)$ – the probability that letter A_j was changed in to letter A_i after time t. The time parameter could be either actual time, or some other measure of the evolutionary changes across an edge in the tree. The corresponding probabilities represent the substitution matrix across a particular evolutionary distance – S(t). Note the similarity with the alignment substitution matrices defined earlier.

The matrix S(t) can be inferred from certain assumptions about the rate of evolution. A simple model for DNA sequences is the Jukes-Cantor model which assumes that all nucleotides undergo evolution at the same rate α. Specifically, the rate of nucleotide substitutions is defined by the matrix:

$$R = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} -3\alpha & \alpha & \alpha & \alpha \\ \alpha & -3\alpha & \alpha & \alpha \\ \alpha & \alpha & -3\alpha & \alpha \\ \alpha & \alpha & \alpha & -3\alpha \end{pmatrix} \end{array}$$

The substitution matrix can be inferred from the substitution rates by focusing on a very short evolutionary time ε. Over this period we can assume $S(\epsilon) = I + R\epsilon$ where I is the identity matrix (1s on the main diagonal and 0s elsewhere). To derive S(t) for an arbitrary time t, we note that the matrix R defines the rate of change in S(t), i.e, S'(t) = S(t)R, leading to the differential equations:

$$\frac{dr}{dt} = -3\alpha r + 3\alpha s$$

$$\frac{ds}{dt} = -\alpha s + \alpha r$$

where r and s are the diagonal and off-diagonal elements of matrix S(t). Solving these equations leads to:

$$r(t) = \frac{1}{4}(1 + 3e^{-4\alpha t})$$

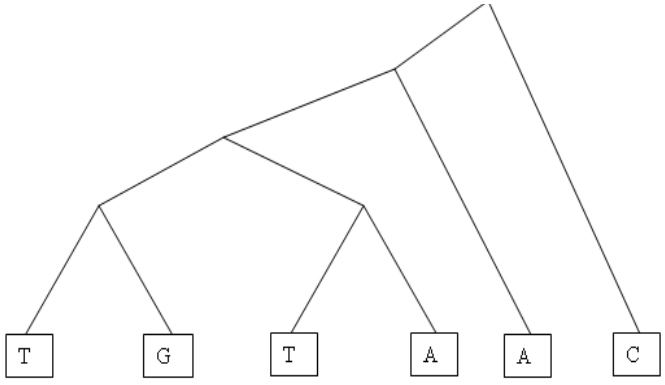
$$s(t) = \frac{1}{4}(1 - e^{-4\alpha t})$$

Note that the equations imply that after infinite evolution, the substitution matrix converges to uniform probabilities, i.e., all nucleotides are equally likely.

More complex models have been defined to capture the lack of symmetry between the different nucleotide substitutions – the Kimura model assumes that purine-purine substitutions (transitions) are more likely than purine-pyrimidine substitutions (transversions).

7.4.5 Exercises

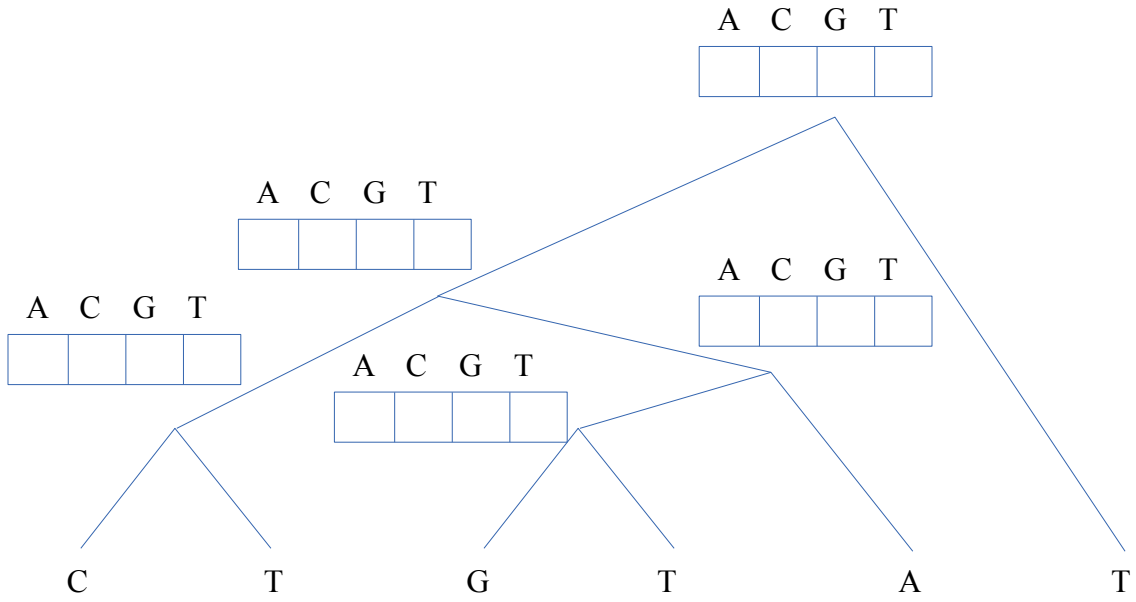
1. In the phylogenetic tree shown below, calculate the parsimony score (total number of mutations in the tree) and indicate what the ancestor sequence was at each internal node. Hint: Sankoff's dynamic programming algorithm for computing the parsimony score computes, for each internal node in the tree, the best parsimony score assuming each nucleotide is placed, in turn, at this node.



2. Using Sankoff's algorithm, calculate the **maximum** score for the phylogenetic tree shown below, as well as the characters at each internal node. Note that the scoring matrix is not symmetric - the score of evolving A into T is different from the score of evolving T into A. In the scoring matrix, assume that the row represents the parent and the column the child in the tree. For example, if a node is assigned A and it's child C, the score along the branch is 3. If the parent is C and the child is A, the score is only 2. At each internal node in the tree please provide both the four scores corresponding to each individual nucleotide being present at that node, and the inferred ancestral sequence at that node (by circling that letter at the node). If multiple ancestral sequences are possible at each node, please pick one of them

(e.g. the answer A/T is not acceptable)

	A	C	G	T
A	5	3	-1	0
C	2	5	0	-2
G	0	1	5	2
T	-1	-1	2	5



7.5 Greedy clustering

Most of the clustering approaches described above require the computation of all pairwise distances between a set of sequences, i.e. the running time is proportional to n^2, where n is the number of sequences. Given the large size of datasets commonly seen in current research (millions to billions of sequences), such algorithms are impractical.

An alternative is provided by greedy clustering algorithms, exemplified by CD-hit (Li and Godzik 2006). The basic algorithm is as follows:

1. sort the sequences in decreasing order of their lengths
2. pick a sequence not assigned to a cluster – this sequence becomes the center of a new cluster
3. compare all unassigned sequences to already computed cluster centers, first using a quick k-mer distance approach, then checking the promising alignments with a full Smith-Waterman algorithm.
4. repeat from 2.

The reason sequences are first sorted by length is to avoid incorrect clusterings. Since the only comparisons are between a sequence and the cluster center, the distance between two non-center sequences is implicitly bounded by triangle inequality. If the clustered sequences can be longer than the center, this property no longer holds, as shown below.

ACAGGTACTCA-
ACACGTACTCAG
ACAG-TACTCAT

Figure 13: Incorrect clustering when cluster center is short. The two sequences differ from the cluster center by one edit, however they differ from each other by 3 edits contradicting the triangle inequality. The contradiction is due to the last mismatch which is not captured in the distance between each sequence and the cluster center.

Note that only sequences within a prescribed radius (in terms of similarity) from the cluster center are added to a cluster – the goal of the algorithm is to create clusters that are fairly tight.

This basic algorithm can be implemented in various flavors:

- the sequences could be indexed making it easy to find all the sequences that match a given cluster center (approach used in DNAclust (Ghodsi, Liu et al. 2011))
- the clusters could be indexed, making it easy to find the appropriate cluster center for each sequence
- the algorithm can be sped up by relaxing the criteria that define the distance from the center of a cluster (e.g. the clustering tool UClust (Edgar 2010) does not implement the full dynamic programming algorithm).

The choice of flavor depends on the specific problem being addressed and has an important impact on the runtime. Assume, for example, that the number of actual clusters in the data is K and the number of sequences is N. The original CD-hit algorithm requires a runtime of O(N f(K)) where f(K) represents the time needed to search one sequence against the database of cluster centers (in a trivial implementation f(K) = K as each center is searched independently of the others). DNAclust requires a runtime of O(K f(N)) as the centers are searched against the rest of the sequences. For highly similar sequences, DNAclust has an advantage as the index constructed from the sequences can be quite efficient. For more dissimilar sequences, CD-hit has an advantage.

An interesting phenomenon occurs in the case of sequences that cannot be well clustered. In this case we fall into the realm of the 'curse of dimensionality' – in high-dimensional spaces all points are relatively close to each other. As a result clustering becomes prohibitively expensive, requiring $O(N^2)$ time. To see this, imagine a collection of 501bp sequences that we want to cluster at a 1% divergence rate. Assume we only look for mismatches – i.e., two sequences are clustered together if they differ by at most 5 mismatches. For every given sequence s, we can find $3 \cdot 5^3 \cdot \binom{500}{5} \approx 95 \cdot 10^{12}$ different sequences that differ from s in exactly 5 locations within the first 500 bp and have a mismatch in the 501st position. All of these sequences need to be fully checked before deciding whether they should be clustered together with s.

8 Genome assembly

(for more information see: http://www.cbcb.umd.edu/research/assembly_primer.shtml, (Pop 2009, Nagarajan and Pop 2013)

8.1 Introduction

Sequencing technologies can only "read" short fragments from a genome. Reconstructing the entire sequence of the genome, thus, requires that these fragments be joined together in a jigsaw-puzzle-like process. Note that, in order for the reconstruction to even be possible, the individual sequences must be sampled from random locations in the genome. Also, enough sequences must be sampled to ensure that the individual sequences overlap, i.e. enough information is available to decide which sequences should be joined together. The process through which the sequences are generated is called "shotgun sequencing", and involves the random shearing (through a physical process) into small fragments of a collection of copies of the genome of interest.

8.2 Is assembly possible? Lander Waterman statistics

Note that it is not even clear that the assembly of a genome from small pieces should be possible. Given that the process through which the sequences are generated is random, it is possible that certain parts of the genome will remain uncovered unless an impractical amount of sequences are generated.

To assess the theoretical feasibility of the assembly of shotgun sequencing data, Eric Lander and Mike Waterman developed a statistical analysis based on Poisson statistics (Lander and Waterman 1988). Briefly, if some events occur uniformly at random (e.g. the start of a sequencing fragment along a genome can be assumed to be chosen uniformly at random), the number of events occurring within a given time interval is represented by a Poisson distribution.

Given an average "arrival rate" λ (# of events occurring within a given interval of time), the probability that exactly n events occur within the same interval is expressed by the formula:

$$f(n,\lambda)=\frac{\lambda^n e^{-\lambda}}{n!}$$

In the context of sequencing we are interested in finding intervals that contain no events (n=0) - these would represent gaps in the coverage of the genome by sequences.

The Lander-Waterman statistics estimate the number of gaps in coverage (conversely the number of contiguous DNA segments) that can be expected given the following set of parameters:

- G – genome length
- n – number of sequences
- L – length of sequences
- c = nL/G – depth of coverage (number of times genome over-sampled by the set of sequences)
- t – the amount by which two sequences need to overlap in order to computationally detect this overlap
- σ = (L-t)/L

Among other numbers, the L-W statistics provide estimates for the expected number of contigs: $ne^{-c\sigma}$

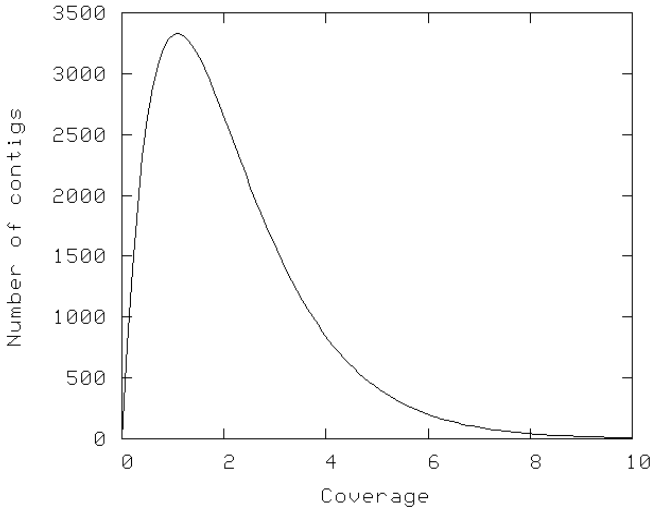


Figure 14: Lander Waterman - expected number of contigs given coverage.

As can be seen from the figure above, the expected number of contigs rapidly decreases once coverage exceeds about 8-10-fold, i.e. after over-sampling a genome by about 10 times, the assembly should be theoretically possible.

8.3 Shortest common superstring and greedy algorithm

A simple formulation of the assembly problem as an optimization problem phrases the problem as the Shortest Common Superstring Problem: find the shortest string that contains all the sequences as substrings. In other words, find the most parsimonious "explanation" for the set of sequences. A fair amount of work went into this problem in the 80s-90s – the problem was shown to be NP-hard, and a series of approximation algorithms were developed that approach an approximation factor of 2 (the reconstructed string is at most twice as long as the optimal) (Tarhio and Ukkonen 1988).

A simple greedy algorithm can be proven to yield a 4-approximation, however it is conjectured that this algorithm is actually 2-optimal, given that no example has yet been found where the greedy assembly algorithm has generated a worse approximation.

The greedy assembly algorithm proceeds as follows:

1. compare all sequences in a pairwise fashion to identify sequences that overlap each other.
2. pick the sequences that overlap each other the best and merge them
3. repeat step 2 until no more sequences can be merged, or the remaining overlaps conflict with existing contigs.

While this algorithm is only an approximation, it has been extremely successful in practice – most early genome assemblers (phrap, TIGR Assembler, CAP) relied on this simple greedy heuristic, and were successful in reconstructing a range of genomes.

8.4 Graph formulations for assembly

The parsimony definition of assembly implicit in the SCS problem can be easily seen not to be relevant in a

biological setting, primarily due to the presence of repeated DNA sequences (repeats) within the genomes of most organisms. These redundant sequences would be collapsed into a single "unit" by any algorithm that attempts to solve the SCS problem.

Instead other optimization criteria have been proposed that attempt to capture the biological nature of the problem. Myers proposed, for example, that we should phrase the assembly problem as the task of reconstructing a layout of the sequences that is consistent (in terms of Kolmogorov statistics) with the characteristics of the random process that generated the sequences (Myers 1995). Unfortunately this formulation is hard to translate into a practical algorithm, though it is important to keep in mind especially in the context of assembly validation.

Instead, most modern assemblers formulate the assembly as a graph traversal problem.

8.4.1 Overlap-layout-consensus/string graph

A first formulation creates a graph that represents each sequence as a separate node, and creates an edge between any two nodes whose corresponding sequences overlap. In this formulation, we want to find a traversal of the graph that uses all the sequences, each exactly once (you cannot use a same sequence in multiple places in the genome), i.e. we are looking for a Hamiltonian path – a well known NP-hard problem. This formulation dates to the early 80s in the work of Esko Ukkonen and colleagues (Peltola, Soderlund et al. 1984).

Recently, Myers has shown that through a few simplifications, including the removal of transitive edges, the problem can be rephrased as a Chinese Postman problem (find the shortest tour that traverses each edge in a graph at least once) - a problem that can be solved in polynomial time (Myers 2005). The transformed graph is termed the *assembly string graph,* and the paper describing this concept is listed on the course website.

8.4.2 DeBruijn graph

An alternative formulation for the assembly problem arose form early explorations of a sequencing technology called "sequencing by hybridization". In this approach, one could find out if a given k-mer occurred in the genome being sequenced, i.e. the assembly problem could be phrased as:

*Given the collection of **all** k-mers (strings of length k) in a genome, reconstruct the genome's sequence.*

In some sense, this problem is equivalent to a shotgun process that generates reads of length exactly k, and which guarantees that exactly one read starts at each position in the genome (perfect coverage).

Pavel Pevzner formulated this problem in the context of de Bruijn graphs (Pevzner 2001): a de Bruijn graph of order k is a graph that contains all strings of length k-1 from a given alphabet as nodes, and contains an edge between two nodes if the corresponding strings overlap by exactly k-2 letters. De Bruijn graphs have been discussed earlier in Section 3.9.2 as the automata that generate de Bruijn sequences. In other words, each k-mer in the genome is represented by two nodes in the graph connected by an edge. In the context of assembly we are looking at the subgraph of the complete deBruijn graph that contains just the k-mers present in the genome (as inferred from the set of reads).

As we are trying to find an assembly that contains all the k-mers of the genome, we are looking for a path through the graph that visits every edge at least once (edges visited multiple times are repeats), i.e. a Chinese Postman path, problem which can be solved in linear time. Note that if we can infer from experimental data (e.g. from the number of sequences containing a particular k-mer) the number of times we need to visit each k-mer, the problem becomes NP-hard – we are looking for a Chinese Postman path of a given length. Finding a longest or fixed-length path in a graph is generally NP-hard.

8.4.3 Complexity results

The formulation of assembly as a variant of the Chinese Postman problem implies that the assembly problem can be solved in polynomial time. This is only partly true – a solution to the assembly problem can be found in polynomial time, however there are an exponential number of possible solutions, i.e. the problem is underconstrained (Kingsford, Schatz et al. 2010). Adding additional constraints, such as adding information about the multiplicity of repeats, generally results in an NP-hard variant of the problem.

Just how many possible Eulerian tours are in a graph (assuming an Eulerian graph, e.g. as generated from the Chinese Postman path by duplicating certain edges)? The answer can be fairly easily computed for directed graphs (though it's intractable in undirected graphs). The reasoning is based on a simple algorithm for finding an Eulerian tour:

1. Pick an arbitrary node in the graph n
2. Construct a directed spanning tree having n as a root, such that all the edges are pointing upwards (the tree is a collection of directed paths connecting the leaves with the root)
3. Start traversing the graph in a greedy fashion, starting from n – after entering a node, leave the node using an edge that does not belong to the spanning tree, if possible. In other words, whenever a node is encountered on the traversal of the graph, the unique spanning tree edge leaving this node (linking the node to its parent) is traversed last.

This algorithm completes only after finding an Eulerian tour – the requirement that the tree edge is the last visited ensures that there is always an "out", thus the algorithm cannot get stuck before visiting all the edges in the graph.

Thus, the number of possible tours can be computed as a function of two terms: (i) the number of spanning trees in the graph; and (ii) the different possible traversals of the graph once the spanning tree is fixed. The latter is simply the product of all different permutations of the edges exiting each node in the graph (minus the spanning tree edge), i.e.:

$$N_{Eulerian\,tours} = N_{spanning\,trees} \prod_{v \in V(G)} (outdeg(v) - 1)!$$

8.4.3.1 Counting spanning trees

[to be added]

8.4.4 Theory vs. practice

As shown above, the assembly problem can be shown to be NP-hard under a number of formulations. However, these theoretical results contradict the empirical observation that even simple heuristic algorithms can generate correct assemblies in most situations. I.e. the instances encountered in practice are not necessarily complex.

Niranjan Nagarajan and myself tried to more precisely determine the boundaries of this disagreement between theory and practice, specifically we parametrized the assembly complexity as a function of the relationship between read length (L), overlap length (o), and the length of the repeats (R) found in the genome. Below is a quick summary of the results. For full details see the paper (linked from the syllabus site).

Case 1: R < o - repeats do not confuse the assembly, any algorithm will find the unique correct solution.

Case 2: R > 2L-o - the assembly problem is equivalent to the Chinese Postman problem - a solution (not necessarily the correct one) can be found in polynomial time. Also, in this case a solution to the Shortest Common Superstring problem is not a solution to the assembly problem.

Case 3: o <= R <= 2L-o - the solution to the assembly problem is the same as the solution to the SCS problem. Assembly is NP-hard.

Note, that as a corollary, if sufficient coverage is generated to ensure that o = L-1, the third case disappears, i.e. the assembly is either trivial, or underconstrained, but not NP-hard.

8.5 Genome assembly validation

8.5.1 Introduction

A common thread in scientific computation is the fact that often the scientific answer that we seek is simply defined by the output of the program used to compute the answer. Given the fact that the results often represent previously unknown information, there is no external way to validate these results. In other words, any bugs in the software are virtually undetectable.

There are two recent examples that provide a compelling demonstration of this fact:

- A paper in the journal Science had to be retracted once the authors realized that the results reported in the paper had been incorrectly computed due to a "+1/-1" error in their code. The mistake was only detected a few years later when another research group failed to reproduce the results (in this case the scientific result was the 3D structure of a previously uncharacterized protein)
- When the records of climate researchers came under scrutiny due to an email leak, scientists and skeptics found out that the software that computes the climate models, and has predicted the effects of global warming, is full of bugs and poorly written code.

Thus, a major challenge in bioinformatics and other scientific fields is to develop approaches for ensuring that the software is correct even in the absence of external means for validation. Below I describe several ways in which validation can be achieved in the context of genome assembly.

8.5.2 Intrinsic measures of assembly quality

As mentioned in the assembly lecture, one of the early definitions of the goal of assembly is to determine a layout of the sequences that is consistent with the random process that generated these sequences in the first place. While this goal is difficult to achieve during the assembly process, the divergence between the output of the assembler and the characteristics of the random shotgun process can be used to assess whether the assembly is correct.

Note that, in general, we do not fully know the characteristics of the sequencing process, however useful information can still be achieved by assuming the process to be fully random (in practice many biases cause deviations from randomness).

8.5.3 Coverage measures

If we assume that the shotgun process is truly random, we can expect the depth of coverage (# of reads spanning a particular location in the genome) to not vary much along the genome (other than random fluctuations from the expected value). Regions where too many reads pile up can be inferred to represent collapsed repeats.

There are several ways in which overly-deep regions can be determined. A simple k-mer based measure can be determined as follows:

- for each k-mer count its number of occurrences in the set of reads. Let this value be $n_r(k)$. In unique regions this value should roughly equal the coverage c, while in repeats with R copies, the value will be

approximately cR

- for each k-mer count its number of occurrences in the consensus of the contigs produced by the assembler. Let this value be $n_A(k)$. In unique regions this value will be 1, while in repeats it will be R.
- for each k-mer compute $n_r(k)/n_A(k)$ - if the assembly is correct (number of copies of a repeat in the genome matches the number of copies in the assembled contigs) the ratio will be roughly c. If, however, certain repeats are collapsed, the ratio will be correspondingly higher - allowing us to detect regions where a collapse might have occurred.

A more principled solution is to statistically evaluate whether the pile-up of reads is truly "surprising" given our expectation of the random process generating the sequences. Such an approach is implemented as the A-statistic used in Celera Assembler. Specifically, assume that the expected arrival rate (number of DNA sequences divided by length of genome) is α, then the probability that k sequences start every ρ base-pairs within a unique region is:

$$p(\rho, k) = \frac{(\rho\alpha)^k}{k!} e^{-\rho\alpha}$$

while in a two-copy collapsed repeat we have:

$$p_2(\rho, k) = \frac{(2\rho\alpha)^k}{k!} e^{-2\rho\alpha}$$

The log ratio of these two values represents the A-statistic:

$$Astat = \log_2\left(\frac{p(\rho, k)}{p_2(\rho, k)}\right) = \log_2\left(\frac{\frac{(\rho\alpha)^k}{k!} e^{-\rho\alpha}}{\frac{(2\rho\alpha)^k}{k!} e^{-2\rho\alpha}}\right) = \log_2(e)\rho\alpha - k$$

Negative values indicate that more "arrivals" occur within a region than expected, i.e. the region represents a collapsed repeat.

It is important to note that the background arrival rate α needs to be estimated from the assembly data, as generally the genome size is not known before running the assembly. This creates somewhat of a chicken and egg problem - information obtained from the assembly is used to evaluate the assembly. If errors are rare, this process should, however, allow the detection of outliers (misassemblies), as the misassemblies cannot significantly affect the statistics.

8.5.4 Mate-pair consistency

In addition to sequences, a typical sequencing experiment also provides mate-pair information - information specifying that pairs of sequences are separated by a given (approximate) distance, and that they have a specific orientation (usually the two sequences are in opposite orientations - originate from different DNA strands). In a correct assembly, the constraints imposed by this information should be preserved (modulo experimental error), i.e. the paired sequences must be oriented and spaced as inferred from the parameters of the sequencing project. Any deviations from this ideal indicate the presence of a potential misassembly.

Several approaches can be used to detect disagreements between the experimental data and the information found in the assembly. First, one can easily find groups of mate-pairs that are "broken" because their corresponding endpoints are incorrectly oriented with respect to each other. These indicate the presence of rearrangements or inversions in the assembly with respect to the correct structure of the genome.

In order to identify regions in the assembly where the distance between mates is different from the

experimentally derived values, we can phrase the problem as a statistical hypothesis test:

- First compute the "correct" mate-pair spacing by averaging the lengths of mate-pairs found within the assembly. As described under coverage statistics, this approach may be confused by misassemblies but should generally work if there are few misassemblies. Also, it is usually necessary to "second-guess" the biologists that generated the data given that the methods for measuring mate-pair sizes are highly inaccurate. The result of this step are the values L_{true}, and SD_{true}, the mean length and standard deviation of the "correct" mate-pairs.
- Second, use a sliding window through the assembly to identify the mate-pairs spanning each particular location in the assembly. Compute $L_{assembly}$ and $SD_{assembly}$ from the local mate-pair neighborhood
- Compare L_{true} and $L_{assembly}$ using a statistical test, e.g. the t-test: $\frac{|L_{assembly} - L_{true}|}{SD_{true}}$

The t-test provides an estimate of the statistical confidence in the disagreement between the two length estimates. The sign of the difference in length indicates whether the assembled region has been collapsed or expanded with respect to the correct assembly of the genome. This simple test is also called the C/E statistic (Zimin et al.).

8.5.5 Detecting "missing" data

It is not uncommon for an assembler to only use part of the data provided as input, i.e. not all the sequence reads end up being used in the final assembly. Valuable validation information can be obtained by mapping the unassembled sequences onto the assembly, in order to explain why they were not used by the assembler. Several cases are possible:

1. Unassembled reads to not match the assembly - this indicates the reads are likely contaminant sequences.
2. Unassembled reads match the assembly perfectly - possibly bug in the assembler but that does not affect the quality of the reconstruction (though it can affect coverage estimates)
3. Unassembled reads match the assembly imperfectly, as shown in the figure below - this indicates that a tandem (2-copy) repeat has been collapsed into a single copy. Sequences spanning the joint between the two copies cannot fit in the resulting assembly and get tossed out.

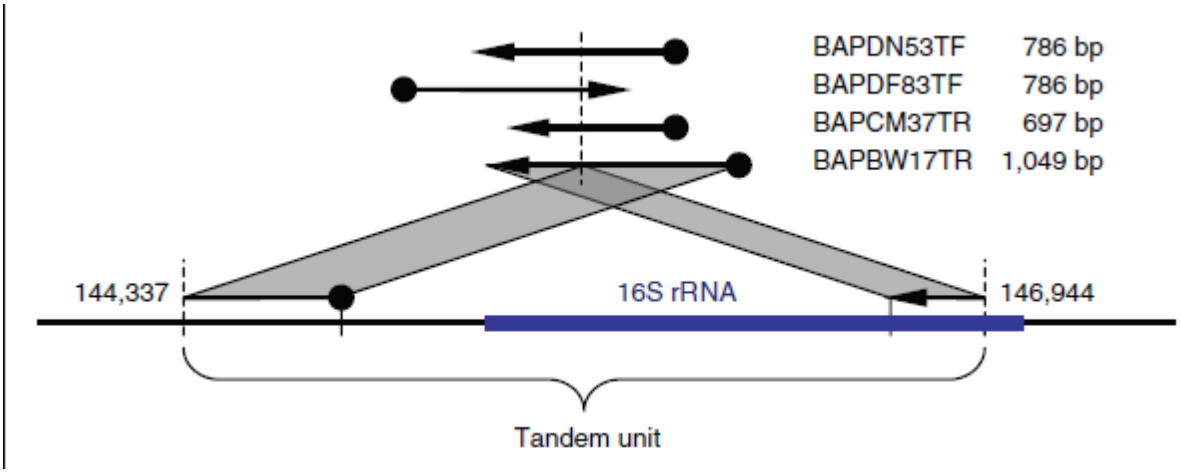


Illustration 1: Alignment of reads to the assembly in a region where a tandem repeat has been collapsed.

8.5.6 "Validation" as a tool for detecting genomic variation

The tools for assembly validation described above are not only valuable for assessing the quality of a genome assembly. Rather, there is increasing interest in understanding how the genome structure varies within a population. E.g. in the human population there is increasing evidence that the genomes of different people, and even within a same person, differ by more than just simple mutations: entire segments of DNA are duplicated, deleted, or rearranged between the genomes. Also, certain phenotypic traits (e.g. hair/skin color) might be caused by differing copy numbers for certain genes.

In the context of genomic variation, we want to map sequences/mate-pairs generated from a person's DNA, to a reference sequence. Discrepancies between the two datasets indicate places where there are structural differences between the two genomes.

8.6 Exercises

1. The output of a genome assembler consists of detailed information regarding the placement of each read inside the genome. Specifically, for each read we know the coordinates of its beginning and end inside the genome (e.g. read i is represented by the pair (b_i, e_i)). Due to the randomness of the sequencing process, some locations in the genome are covered very deeply while others are covered sparsely. Describe an algorithm that computes, for a given assembly, the highest depth of coverage, as well as one of the locations (coordinates along the genome) where this occurs. The running time of this algorithm should not exceed O(n log n), where n is the number of reads in the assembly.

9 Miscellaneous

9.1 Optical maps

9.1.1 Introduction

Optical mapping is a technology that allows us to experimentally determine the relative position of certain landmarks (usually restriction sites - places where a specific restriction enzyme can cut the DNA - usually recognized by a specific 6-8 bp sequence) along a stretch of DNA. Unlike sequencing, optical maps provide long range information - the fragments being mapped are usually on the order of several 100s of kbp - however the information provided is sparse. Thus optical mapping is useful as a complement to sequencing. Also, it can be used as a cheaper way (than sequencing) of getting information about the global structure of a genome.

Computationally an optical map is just an ordered list of sizes, together with estimates of the error in the size estimates, representing the list of gaps between adjacent restriction sites.

9.1.2 Mapping algorithm

I'll focus here on just the problem of aligning an experimentally determined optical map to an *in silico* optical map constructed, e.g. from the output of an assembler.

Formally, the experimental map is represented as the array:

$$emap = \{(o_k, s_k), k = 1, n\}$$

where o_k and s_k are the size and standard deviation for fragment k.

The *in silico* map is:

$$ismap = \{e_k, k = 1, m\}$$

where e_k is the size of the corresponding fragment

Aligning the two maps can be performed pretty easily using a dynamic programming algorithm similar to the sequence alignment algorithm.

Specifically, V[i,j] is the score of aligning the first i fragments from the experimental map to the first j fragments from the *in silico* map.

The recurrence equation is:

$V[i,j] = \min_{k<i, l<j} \{ V[k,l] + score(k..i, l..j)\}$

where score(k..i, l..j) is a score of how well the set of fragments between k..i, and l..j, match each other. This score can be defined as a combination of a X^2 score and a penalty for missed sites:

$$score(k..i, l..j) = \frac{\left(\sum_{s=k}^{i} o_s - \sum_{t=l}^{j} e_t\right)^2}{\sum_{s=k}^{i} s_s^2} + C(i - k + l - j)$$

where C is a constant that can be used to tune the contribution of the two components.

9.1.3 Interesting research directions

- Can optical maps be used to guide genome assembly?
- How do you efficiently align maps to an already sequenced genome to identify structural variants?
- How do you efficiently align two maps to each other to identify structural differences?

10 References

Abouelhoda, M. I., S. Kurtz and E. Ohlebusch (2004). "Replacing suffix trees with enhanced suffix arrays." Journal of Discrete Algorithms **2**(1): 53-86.

Altschul, S. F. (1989). "Gap costs for multiple sequence alignment." J Theor Biol **138**(3): 297-309.

Altschul, S. F. (1991). "Amino acid substitution matrices from an information theoretic perspective." J Mol Biol **219**(3): 555-565.

Bacon, D. J. and W. F. Anderson (1986). "Multiple sequence alignment." J Mol Biol **191**(2): 153-161.

Brown, M., R. Hughey, A. Krogh, I. S. Mian, K. Sjölander and D. Haussler (1993). Using Dirichlet mixture priors to derive hidden Markov models for protein families. Ismb.

Burrows, M. and D. J. Wheeler (1994). A block-sorting lossless data compression algorithm, Digital Equipment Corporation.

Darling, A. C. E. (2004). "Mauve: Multiple Alignment of Conserved Genomic Sequence With Rearrangements." Genome Research **14**(7): 1394-1403.

Delcher, A. L., S. Kasif, R. D. Fleischmann, J. Peterson, O. White and S. L. Salzberg (1999). "Alignment of whole genomes." Nucleic Acids Res **27**(11): 2369-2376.

Delcher, A. L., A. Phillippy, J. Carlton and S. L. Salzberg (2002). "Fast algorithms for large-scale genome alignment and comparison." Nucleic Acids Res **30**(11): 2478-2483.

Dempster, A. P., N. M. Laird and D. B. Rubin (1977). "Maximum likelihood from incomplete data via the EM algorithm." Journal of the Royal Statistical Society. Series B (Methodological): 1-38.

Eddy, S. R. (1995). "Multiple alignment using hidden Markov models." Proceedings / ... International Conference on Intelligent Systems for Molecular Biology ; ISMB. International Conference on Intelligent Systems for Molecular Biology **3**: 114-120.

Eddy, S. R. (1998). "Profile hidden Markov models." Bioinformatics **14**(9): 755-763.

Edgar, R. C. (2004). "MUSCLE: multiple sequence alignment with high accuracy and high throughput." Nucleic Acids Res **32**(5): 1792-1797.

Edgar, R. C. (2010). "Search and clustering orders of magnitude faster than BLAST." Bioinformatics **26**(19): 2460-2461.

Eisen, J. A., J. F. Heidelberg, O. White and S. L. Salzberg (2000). "Evidence for Symmetric Chromosomal Inversions Around the Replication Origin in Bacteria." Genome Biology **1**(6).

Farrar, M. (2007). "Striped Smith-Waterman speeds database searches six times over other SIMD implementations." Bioinformatics **23**(2): 156-161.

Fasulo, D. (1999) "An analysis of recent work on clustering algorithms."

Ferragina, P. and G. Manzini (2000). Opportunistic data structures with applications. 41st Annual Symposium on Foundations of Computer Science.

Galas, D. J., M. Eggert and M. S. Waterman (1985). "Rigorous pattern-recognition methods for DNA sequences: Analysis of promoter sequences from Escherichia coli." Journal of molecular biology **186**(1): 117-128.

Geman, S. and D. Geman (1984). "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images." IEEE Trans Pattern Anal Mach Intell **6**(6): 721-741.

Ghodsi, M., B. Liu and M. Pop (2011). "DNACLUST: accurate and efficient clustering of phylogenetic marker

genes." BMC Bioinformatics **12**: 271.

Ghodsi, M. and M. Pop (2009). Inexact Local Alignment Search over Suffix Arrays. IEEE International Conference on Bioinformatics and Biomedicine (BIBM). X.-W. Chen and S. Kim. Washington, DC, IEEE**:** 83-87.

Gribskov, M., A. D. McLachlan and D. Eisenberg (1987). "Profile analysis: Detection of distantly related proteins." Proc. Natl. Acad. Sci. USA **84**: 4355-4358.

Grünwald, P. D. (2007). The minimum description length principle, MIT press.

Gusfield, D. (1997). Algorithms on strings, trees, and sequences, The press syndicate of the university of Cambridge.

Handl, J., J. Knowles and D. B. Kell (2005). "Computational cluster validation in post-genomic data analysis." Bioinformatics **21**(15): 3201-3212.

Henikoff, S. and J. G. Henikoff (1993). "Performance evaluation of amino acid substitution matrices." Proteins **17**(1): 49-61.

Hertz, G. Z., G. W. Hartzell, 3rd and G. D. Stormo (1990). "Identification of consensus patterns in unaligned DNA sequences known to be functionally related." Comput Appl Biosci **6**(2): 81-92.

Ilie, L. and S. Ilie (2009). "Fast computation of neighbor seeds." Bioinformatics **25**(6): 822-823.

Jain, A. K., M. N. Murty and P. J. Flynn (1999). "Data clustering: a review." ACM computing surveys (CSUR) **31**(3): 264-323.

Jiang, T., G. Lin, B. Ma and K. Zhang (2002). "A general edit distance between RNA structures." J Comput Biol **9**(2): 371-388.

Kanungo, T., D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman and A. Y. Wu (2002). "An efficient k-means clustering algorithm: analysis and implementation." Pattern Analysis and Machine Intelligence, IEEE Transactions on **24**(7): 881-892.

Karlin, S. and S. F. Altschul (1990). "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes." Proc Natl Acad Sci U S A **87**(6): 2264-2268.

Kingsford, C., M. C. Schatz and M. Pop (2010). "Assembly complexity of prokaryotic genomes using short reads." BMC Bioinformatics **11**: 21.

Kurtz, S. (1999). "Reducing the space requirement of suffix trees." Softw. Pract. Exper. **29**(13): 1149-1171.

Kurtz, S., A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu and S. L. Salzberg (2004). "Versatile and open software for comparing large genomes." Genome Biol **5**(2): R12.

Landau, G. M. and U. Vishkin (1986). Introducing efficient parallelism into approximate string matching and a new serial algorithm. Proceedings of the eighteenth annual ACM symposium on Theory of computing. Berkeley, California, USA, ACM**:** 220-230.

Lander, E. S. and M. S. Waterman (1988). "Genomic mapping by fingerprinting random clones: a mathematical analysis." Genomics **2**(3): 231-239.

Langmead, B., C. Trapnell, M. Pop and S. L. Salzberg (2009). "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome." Genome Biol **10**(3): R25.

Lawrence, C. E., S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald and J. C. Wootton (1993). "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment." Science **262**(5131): 208-214.

Leung, M. Y., B. E. Blaisdell, C. Burge and S. Karlin (1991). "An efficient algorithm for identifying matches with errors in multiple long molecular sequences." J Mol Biol **221**(4): 1367-1378.

Li, W. and A. Godzik (2006). "Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences." Bioinformatics **22**(13): 1658-1659.

Manber, U. and E. W. Myers (1993). "Suffix arrays: A New Method for On-Line String Searches." SIAM Journal of Computing **22**: 935-948.

McCallum, A., K. Nigam and L. H. Ungar (2000). Efficient clustering of high-dimensional data sets with application to reference matching. Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller (1953). "Equation of state calculations by fast computing machines." The journal of chemical physics **21**: 1087.

Murata, M., J. S. Richardson and J. L. Sussman (1985). "Simultaneous comparison of three protein sequences." Proc Natl Acad Sci U S A **82**(10): 3073-3077.

Myers, E. W. (1995). "Toward Simplifying and Accurately Formulating Fragment Assembly." J. Comp. Bio **2**(2): 275-290.

Myers, E. W. (2005). "The fragment assembly string graph." Bioinformatics **21 Suppl 2**: ii79-ii85.

Nagarajan, N. and M. Pop (2013). "Sequence assembly demystified." Nat Rev Genet **14**(3): 157-167.

Navlakha, S., J. White, N. Nagarajan, M. Pop and C. Kingsford (2010). "Finding biologically accurate clusterings in hierarchical tree decompositions using the variation of information." J Comput Biol **17**(3): 503-516.

Nawrocki, E. P., D. L. Kolbe and S. R. Eddy (2009). "Infernal 1.0: inference of RNA alignments." Bioinformatics **25**(10): 1335-1337.

Nguyen, V. A., J. Boyd-Graber and S. F. Altschul (2013). "Dirichlet mixtures, the Dirichlet process, and the structure of protein space." J Comput Biol **20**(1): 1-18.

Peltola, H., H. Soderlund and E. Ukkonen (1984). "SEQAID: a DNA sequence assembling program based on a mathematical model." Nucleic Acids Res **12**(1): 307-321.

Pevzner, P. A. (2001). "An Eulerian path approach to DNA fragment assembly." Proceedings of the National Academy of Sciences **98**(17): 9748-9753.

Pop, M. (2009). "Genome assembly reborn: recent computational challenges." Brief Bioinform **10**(4): 354-366.

Posfai, J., A. S. Bhagwat, G. Posfai and R. J. Roberts (1989). "Predictive motifs derived from cytosine methyltransferases." Nucleic Acids Res **17**(7): 2421-2435.

Queen, C., M. N. Wegman and L. J. Korn (1982). "Improvements to a program for DNA analysis: a procedure to find homologies among many sequences." Nucleic Acids Res **10**(1): 449-456.

Rognes, T. (2011). "Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation." BMC Bioinformatics **12**: 221.

Saitou, N. and M. Nei (1987). "The neighbor-joining method: a new method for reconstructing phylogenetic trees." Mol Biol Evol **4**(4): 406-425.

Sankoff, D. (1975). "Minimal mutation trees of sequences." SIAM Journal on Applied Mathematics **28**(1): 35-42.

Schneider, T. D., G. D. Stormo, L. Gold and A. Ehrenfeucht (1986). "Information content of binding sites on nucleotide sequences." Journal of molecular biology **188**(3): 415-431.

Schuler, G. D., S. F. Altschul and D. J. Lipman (1991). "A workbench for multiple alignment construction and analysis." Proteins **9**(3): 180-190.

Smith, H. O., T. M. Annau and S. Chandrasegaran (1990). "Finding sequence motifs in groups of functionally related proteins." Proc Natl Acad Sci U S A **87**(2): 826-830.

Sobel, E. and H. M. Martinez (1986). "A multiple sequence alignment program." Nucleic Acids Res **14**(1): 363-374.

Staden, R. (1989). "Methods for discovering novel motifs in nucleic acid sequences." Comput Appl Biosci **5**(4): 293-298.

Stormo, G. D. and G. W. Hartzell, 3rd (1989). "Identifying protein-binding sites from unaligned DNA fragments." Proc Natl Acad Sci U S A **86**(4): 1183-1187.

Styczynski, M. P., K. L. Jensen, I. Rigoutsos and G. Stephanopoulos (2008). "BLOSUM62 miscalculations improve search performance." Nat Biotechnol **26**(3): 274-275.

Tarhio, J. and E. Ukkonen (1988). "A greedy approximation algorithm for constructing shortest common superstrings." Theoretical Computer Science **57**(1): 131-145.

Thompson, J. D., D. G. Higgins and T. J. Gibson (1994). "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice." Nucleic Acids Res **22**(22): 4673-4680.

Ukkonen, E. (1995). "On-line construction of suffix-trees." Algorithmica **14**: 249-260.

Vingron, M. and P. Argos (1991). "Motif recognition and alignment for many sequences by comparison of dot-matrices." J Mol Biol **218**(1): 33-43.

Waterman, M., R. Arratia and D. Galas (1984). "Pattern recognition in several sequences: consensus and alignment." Bulletin of mathematical biology **46**(4): 515-527.

Ye, X., Y.-K. Yu and S. F. Altschul (2010). "Compositional adjustment of Dirichlet mixture priors." Journal of Computational Biology **17**(12): 1607-1620.