CSCI 5454, CU Boulder | **Christopher Aicher & Ryan Hand**
Sequence Alignment Lecture | **April 2, 2013**

1 Introduction to Sequence Alignment

The natural occurrence of *strings* in our daily lives motivates several applications where discovering the similarities have great significance and utility. The definition of a string is different based on the field of study and context of the experiment we are conducting. There are many prevalent problems that can be solved by discovering underlying structure in strings, both continuous and discrete. For example, if we take a user's search input and attempt to find a legitimate match for a misspelled word, a string in this case is the ASCII character map. Another example is speech recognition, where a continuous sound-wave string is mapped to a word with meaning. The area of looking for underlying structure and patterns in strings incorporates many problems including our topic *sequence alignment.*

Multiple sequence alignment refers to finding and 'optimal alignment of k strings, through a series of operations, transformation such that they are all the same. For example, a popular application of sequence alignment in biology is comparing the required number of operations to align nucleotides in DNA/RNA sequences or the amino acids in protein chains.

Sequence alignment is also very flexible: different edit-operations lead to different alignments corresponding to which particular type of problem we are trying to solve. For example, Global alignment compares over the entire length of the strings, whereas, Local alignment compares segments of the strings and is not worried about making the entire sequence align.

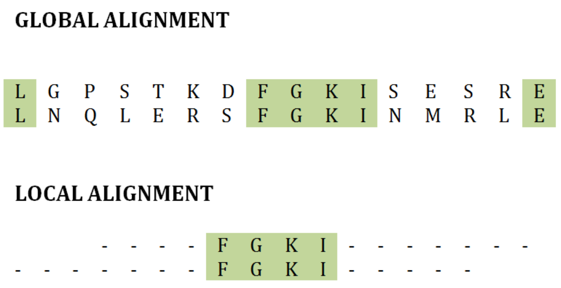
GLOBAL ALIGNMENT
L G P S T K D F G K I S E S R E
L N Q L E R S F G K I N M R L E
LOCAL ALIGNMENT
- - - - F G K I - - - - - - -
- - - - - - - F G K I - - - - -

Figure 1: Global and Local Alignment

Here, we will only consider global alignments. A special case of multiple sequence alignment is *pairwise sequence alignment*, for the comparison or transformation of $k = 2$ strings. We will show that this problem can be solved efficiently using dynamic programming algorithms; however, multiple sequence alignment is trickier to deal with. But before all that, we will first define what we mean by 'optimal alignment.

1.1 Mathematical Formulation

We will now formulate sequence alignment as an abstract question: Suppose we are given k sequences or strings $(s_1, \ldots, s_k)$ with letters from an alphabet Σ and a set of allowed *edit-operations* $E \in \mathcal{E}$. Each operation E modifies the structure of a string and has an associated cost $c(E) \geq 0$. We define an *alignment* of the k sequences as a collection of edit operations that transforms all k sequences into some ancestral sequence s^*. Our goal in sequence alignment is to find an alignment that minimizing the total cost of operations.

Similar to our discussion of phylogenetic trees, we are not particularly interested in the ancestral sequence s^*, but rather in minizing the total cost of the edit-operations. In particular, the optimal alignment and s^* is not necessarily unique.

2 Pairwise Sequence Alignment

In the case of pairwise sequence alignment, we call the total cost of the edit-operations the *edit distance* between the two strings. When each edit-operations has unit cost, the edit distance is the number of 'edits' required to transform one string (the source) into another (the target).

2.1 Edit-Operations

The classic allowed edit-operations are substitution, insertion-deletion, and transposition [7].

- *Substitution* is the operation that allows a letter to be substituted by another letter at the same position in the string. Substitution is also known as the Mutation or Replacement operator. We denote the substitution operator as *Sub*.
- *Insertion* and *Deletion* are operations that change the length of the string. Intuitively, Deletion removes a letter and Insertion adds a letter. In pairwise sequence alignment, deletion in one string is equivalent to insertion in the other therefore we denote this operation as *InDel*. For notational convenience, we denote inserted letters with a special gap character $-$, while understanding that $-$ corresponds to either a deletion or an insertion of a specific letter. An example of InDel operations is when an extra keystroke is recorded when typing or when DNA/RNA skips over a nucleotide in replication.
- *Transposition* is an operation that swaps the position of two sequential letters in a string. We dentote the transposition operator with *Swap*. Transpositions commonly occur in typing and spell check. For our discussion, we restrict ourselves to only tranposing sequential letters and allowing letters to be transposed only once. Generalizations to longer distance swaps and permutations exist, but are more complicated and require restrictions.

2.2 Cost/Distance Functions

The other area of flexibility in our abstract definition is how we assign costs $c(\cdot)$ to each edit-operation.

Common unit cost functions are described below:

- Hamming Distance only allows substitutions. This is the metric we used in our discussion of phylogenetic trees, but isn't particularly interesting for sequence alignment as there is no insertion/deletion 'alignment' involved.
- Longest Common Subsequence 'distance' only allow insertion/deletions.
- Levenshtein Distance allows for substitutions and insertion/deletions. It is primary metric in sequence alignment and is commonly referred to as 'Edit-Distance'.
- Damerau-Levenshtein Distance allows for substitutions, insertion/deletion, and transpositions. It is a generalization of Levenshtein Distance.

Therefore by changing the cost function we are able to solve a wide variety of questions.

2.2.1 Example: Their vs. There

To fix ideas, consider the following pairwise sequence alignment of `THEIR` and `THERE`.

The straightforward approach is to substitute the last two characters,

```
THEIR
|||ss
THERE
```

This alignment costs two substitution operations (Hamming Distance).

Another approach is to insert and delete characters so that the `R`'s line up,

```
THEIR-
|||d|i
THE-RE
```

This alignment costs two *InDel* operations (Longest Common Subsequence).

To give an example of a poor alignment consider inserting and deleting the entire word,

```
THEIR-----
dddddiiiii
-----THERE
```

This alignment costs ten *InDel* operations.

2.2.2 Extensions

To more accurately capture desired behavior, we can modify how we assign costs for each operation. In the `THEIR` versus `THERE` example, it is initially unclear which of the three alignments described is correct.

Two common extensions are in measuring the cost of substitution and insertion/deletion operators.

For substitution, it may be desirable to penalize different substitutions differently. For example substituting the letter 'a' for 's' may cost less than 'a' for 'p' based on the location of the letters on a QWERTY keyboard. To implement this, we store the cost of each specific substitution in a $|\Sigma|$-by-$|\Sigma|$ *substitution matrix* that is symmetric and has a 0 diagonal. For example, the substitution matrix

```
   A  B  C  D
A  0  1  9  9
B  1  0  9  9
C  9  9  0  1
D  9  9  1  0
```

penalizes substituions from `A` to `B` less than to `C` or `D`.

For insertion/deletion operators, it may be desirable to penalize consecutive gaps differently. For example, multiple insertions/deletions in a row may be penalizes more or less than the same number of individually seperated gaps. We capture this by assigning a *gap-penalty function* $g(\cdot)$ that assigns a cost to consecutive gaps. For example, in our currently model, each insert/delete costs the same amount $c(InDel)$, therefore $g(x) = x * c(InDel)$. However we could use a quadratic penalty $g(x) = O(x^2)$ or a constant penalty $g(x) = O(1)$ if we so desire, penalizing or preferring long gaps respectively. Unfortunately, it much easier to deal with concave/subadditive gap penalties $g(x + y) \leq g(x) + g(y)$ for implementation purposes. A popular gap penalty in biology takes the form $g(x) = a + b \log x$. For more on this topic see [8].

Of course with any extension, the additional complexity requires additional care in the implementation.

3 Dynamic Programming Algorithm

Recall that Dynamic Programming algorithms are based on exploiting some intrinsic structure of the problem, such that the full solution can be found by piecing together solutions to subproblems. This relationship between solivng the full problem and solving the subproblems defines a recursive function that can build the full solution from subproblem solutions.

Now this recursive relationship alone does not give us a polynomial running time; we need (i) the subproblems to be ordered, (ii) the space of subproblems to be polynomial in size and (iii) the recursive function to be polynomial in time. With these requirements, memorizing subproblem solutions or iteratively building up subproblem solutions will result in a polynomial time algorithm.

Requirement (i) is necessary for us to be able to iteratively build up the subproblem solutions. If the subproblem space was not ordered, then there might be a circular loop where solving problem A requires solving problem B and solving problem B requires solving problem A. The ordered space of subproblems can be visualized as a directed acyclic graph (DAG) where each vertex is a subproblem and an edge from subproblem A to subproblem B implies the solution to B requires a solution to A. If there is a cycle, then we will not be able to build up solutions from the base cases.

Requirements (ii) & (iii) are necessary for the running time. If it takes polynomial time to merge solutions and we merge solutions a polynomial number of times, then our overall running time is still polynomial.

For more details, see Chapter 15 of CLRS [2].

3.1 Ordered Subproblems

In sequence alignment, the substructure we will exploit the additivity of cost in a sequence of edit operations. Any alignment of strings of length i, j is equal to the cost the last edit-operation and the cost of the subalignment composed of the other operations. Therefore working backwards, if our last edit operation was a substitution, then our subalignment must align the substrings of length $i-1, j-1$. Similarly, if our last edit operation was an insert/delete then our subalignment must align the substrings of length $i-1, j$ or $i, j-1$. This will define the recursive relation below.

Let $OPT(i,j)$ be the minimum cost of aligning the substrings $s_1[1 \dots i]$ and $s_2[1 \dots j]$. We define the base case, $S(0,0) = 0$.

We can update $S(i,j)$ using the following recursive relation,

$$OPT(i,j) = \min \begin{cases} OPT(i-2, j-2) + c(Swap) \\ OPT(i-1, j-1) + c(Sub) \\ OPT(i-1, j) + c(InDel) \\ OPT(i, j-1) + c(InDel) \end{cases}, \qquad (*)$$

where the cost of a swap $c(Swap)$ is the combination of transposition and substitution if necessary and $c(Sub) = 0$ if $s_1[i] = s_2[j]$.

Therefore we find the optimal alignment of two strings of lengths n_1, n_2 by first solving the subproblem of finding the optimal alignment between the first i, j letters of the two strings.

One useful way of visualizing the space of subproblems is as a matrix. Since $S(i,j)$ is defined for $0 \le i \le n_1, 0 \le j \le n_2$, the space of subproblems forms a matrix. The connections from previous subproblems to the current problem is described by the DAG in Figure 2.

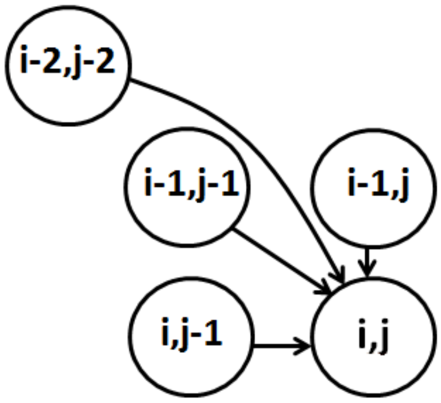

Figure 2: Visualization of the recursion as a DAG.

3.1.1 Example: Exponential vs. Polynomial

To fix ideas consider the following example of aligning `EXPONENTIAL` and `POLYNOMIAL` taken from [3].

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Figure 3: Subproblem Alignment Costs of Exponential vs. Polynomial. Figure from [3].

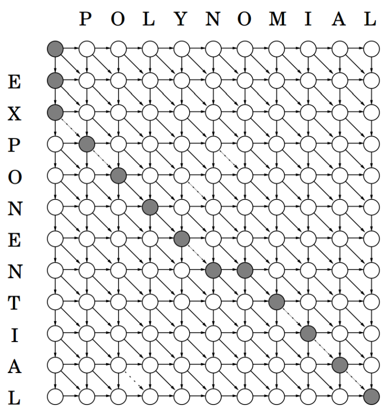

Figure 4: Subproblem Alignment Path of Exponential vs. Polynomial. Figure from [3].

The matrix of subproblem solutions is shown in Figure 3. For example, the minimum alignment cost of `EXP` and `POLY` is calculated as

$$OPT(3,4) = \min\{OPT(1,2)+3, OPT(2,3)+1, OPT(2,4)+1, OPT(3,3)+1\} = \min\{2+3, 3+1, 4+1, 3+1\} = 4 \ .$$

Now the matrix only gives the minimum alignment cost. To find an optimal alignment, we keep track of the previous optimal subalignments that are used: the path through the DAG. An example path is shown in Figure 4.

3.2 Proof of Correctness

Because our solutions are built recursively, the standard technique for proving the correctness of a dynamic programming is induction. Therefore, we will prove the correctness of any sequence alignment algorithm using the recursion relation (*) with induction.

Claim. *Any alignment of strings s_1, s_2 satisfying the recursion relation (*) is a minimial cost alignment.*

Proof. Base Case: The cost of aligning nothing is zero, $OPT(0,0) = 0$. Inductive Step: Assume we've calculated the minimial alignment for $OPT(k,l)$ for $k < i, l < j$. Then there are only 4 possible previous subalignments based on the last operator(s),

- Transposition + Substitution: We swap $s_1[i-1]$ with $s_1[i]$ and then substitute them for $s_2[j]$ and $s_2[j-1]$ respectively. These three edits cost $c(Swap)$. The rest of the alignment cost is from aligning $s_1[1 \ldots i-2]$ and $s_2[1 \ldots j-2]$, whose minimum value is $OPT(i-2, j-2)$. Therefore the minimum cost ending with a swap is $OPT(i-2, j-2) + c(Swap)$.

- Substitution: We substitute $s_1[i]$ for $s_2[j]$. This costs $c(Sub)$. The rest of the alignment cost is from aligning $s_1[1 \ldots i-1]$ and $s_2[1 \ldots j-1]$. Therefore the minimum cost ending with a substitution is $OPT(i-1, j-1) + c(Sub)$.

- Delete/Insert s_1/s_2: We add a gap a gap character after $s_2[j]$ to match $s_1[i]$. This costs $c(InDel)$. The rest of the alignment cost is from aligning $s_1[1 \ldots i-1]$ and $s_2[1 \ldots j]$. Therefore the minimum cost ending with a substitution is $OPT(i-1, j) + c(Sub)$

- Insert/Delete s_1/s_2: Similarly, we add a gap a gap character after $s_1[i]$ to match $s_1[j]$. This costs $c(InDel)$. The rest of the alignment cost is from aligning $s_1[1 \ldots i]$ and $s_2[1 \ldots j-1]$. Therefore the minimum cost ending with a substitution is $OPT(i, j-1) + c(Sub)$

Taking the mininum over the possible operations gives the recursive relation (*). Since these are the only possible paths to aligning substrings (i,j), the recursion gives the minimal cost for aligning (i,j). □

4 Algorithm Implementation

An iterative algorithm that builds up OPT is described in Algorithm 1. This algorithm is a generalization of the original dynamic programming algorithm introduced by Needleman and Wunsch in 1970.

Algorithm 1 Iterative Sequence Alignment

```
 1: input Strings $s_1[1 \dots n], s_2[1 \dots m]$.
 2: Initialize an $n+1$-by-$m+1$ matrix $OPT$        // Stores the minimal subalignment cost
 3: Initialize an $n+1$-by-$m+1$ matrix $prev$       // Stores the subalignment path
 4: Set $OPT[0,0] = 0$
 5: Set $prev[0,0] = NULL$
 6: for $i = 0$ to $n$ do
 7:     for $j = 0$ to $m$ do
 8:         Update $OPT[i,j]$ using (*)              // Solve the subproblems
 9:         Update $prev[i,j]$ with arg min of (*)
10:     end for
11: end for
12: return $OPT[n,m]$ and path built from $prev[n,m]$
```

The space requirement and running time of this algorithm is $O(nm) = O(n^2)$. Clearly the space requirement is $O(nm)$ since, we store each subproblem as a cell in our $O(n)$-by-$O(m)$ matrix OPT and $prev$. For the running time, initializing the matrices takes $O(nm)$ time and since each iteration of the for loop (lines 8-9) takes constant $O(1)$ time with $O(nm)$ iterations, the for loop takes $O(nm)$ time. Therefore the overall running time is $O(nm)$.

4.1 Algorithm Extensions

4.1.1 Space Efficient Modification

Algorithm 1 is the natural implementation of our recursion, but as computer scientists, we wish to know if we can do better than $O(n^2)$ space and time. A useful observation is that each row of the OPT matrix only depends on the previous two rows. If we recycle rows, then we can calculate $OPT[n,m]$ using $O(n)$ space. Pseudocode for this 'Space Efficient' Algorithm is described below

Algorithm 2 Space Efficient Alignment

```
1: input Strings $s_1[1 \dots n], s_2[1 \dots m]$.
2: Initialize an 3-by-$m$ matrix $OPT$                // Stores the minimal subalignment cost
3: Set $OPT[0,0] = 0$
4: for $i = 0$ to $n$ do
5:     for $j = 0$ to $m$ do
6:         Update $OPT[i \mod 3, j]$ using (*)        // Solve the subproblems
7:     end for
8: end for
9: return $OPT[n \mod 3, m]$
```

Unfortunately calculating $OPT[n,m]$ recycling rows, costs us the ability to keep track of the path through the DAG. Therefore, we are able to calculate the minimal alignment cost, but no longer know how to construct a minimal alignment.

To construct a minimal alignment using the space efficient algorithm, we need to be clever. Luckily, Hirschberg in 1975, developed a divide and conquer algorithm to do this. We sketch out some of the details below, but refer the interested reader to [6]

Notice that if we chop the OPT matrix in half along a column $m/2$, the optimal alignment will have to pass through the minimal element q of the column to get from $OPT[0,0]$ to $OPT[n,m]$. To calculate the values along the column using the space efficient algorithm for the first half and add the score of aligning the second half in reverse. Therefore we can find an element on the optimal path $OPT[q, m/2]$ that an optimal alignment passes through in $O(n^2)$ time and $O(n)$ space. We then run divide-and-conquer on the smaller problems of aligning $[0, 0] \to [q, m/2]$ and $[q, m/2] \to [n, m]$. The running time is governed by the equation $T(n, m) = O(nm) + T(q, m/2) + T(n - q, m/2)$ and is solved with $T(n, m) = knm$ for some k. Therefore we obtain both the alignment and its cost using $O(n^2)$ time and $O(n)$ space.

4.1.2 Implementing Gap Penalties

See the lecture. The short version is that instead of updating $OPT[i, j]$ using (*), we must now add links from all $OPT[l, j] : l < i$ and $OPT[i, k] : k < j$ for possible consecutive insert/delete gaps. This increases the running time to $O(n^3)$, but again, if we are clever and with some restriction on g we can reduce this running time. For more on this topic see [8].

5 Multiple Sequence Alignment

Now we will return to the general problem of aligning k strings and describe why it poses a more difficult challenge.

5.1 Why Aligning k strings is harder

Recall that our Dynamic Programming approach to pairwise sequence alignment required us to find all optimal alignments of substrings $s_1[1 \dots i], s_1[1 \dots j]$. However the equivalent formulation for k strings is to find the optimal alignments of substrings $s_1[1 \dots i_1], \dots, s_k[1 \dots i_k]$. This corresponds to a k-D array instead of a 2D matrix. Therefore in multiple sequence alignment, the space of subproblems is exponential in k, $O(n^k)$ space and therefore the straightforward approach has an exponential running time. This isn't too bad for $k = 3$ or 4, but as the number of sequences gets larger we quickly run into space and time issues.

An alternative idea is to some how exploit our ability to solve pairwise sequence alignment to more intelligently find the optimal multiple sequence alignment. Unfortunately, as the following example motivates, pairwise sequence alignment does not easily solve our problems.

5.1.1 Example: Their vs. There vs. They're

We will compare the pairwise alignments of `THEIR`, `THERE`, and `THEY'RE` under the Levenshtein distance to their multiple sequence alignment.

The optimal pairwise alignment of `THEIR` and `THERE` is one of the following,

```
THEIR    THEIR-
|||ss    |||d|i
THERE    THE-RE
```

The optimal pairwise alignment of `THEIR` and `THEY'RE` is one of the following,

```
THEI-R-    THE-IR-
|||sd|d    |||ds|d
THEY'RE    THEY'RE
```

However the optimal multiple sequence alignment of `THEIR`, `THERE`, and `THEY'RE` is,

```
THEI-R-   THE-IR-
THE--RE   THE--RE
THEY'RE   THEY'RE
```

Which isn't intuitively obvious from the pairwise sequence alignments (at least for us).

5.2 Heuristic Methods

However exploiting pairwise sequence alignment is the start of many heuristic techniques for multiple sequence alignment. For example, one approach is to create a hierachical guide tree to merge pairwise sequences iteratively, using all pairwise sequence alignment costs to decide which sequences to merge. The approach is very similar to how we built the phylogentic tree structure. Other alternative methods that have been proposed are Hidden Markov Models, Genetic Algorithms, and iterative Divide and Conquer. This is an active area of research and we will not go into the details here. For those interested [4, 1] is a survey of methods that can serve as a good jumping off point.

References

[1] Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, October 1988.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2001.

[3] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Virkumar Vazirani. Algorithms. 2006.

[4] Robert C Edgar and Serafim Batzoglou. Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368–373, June 2006.

[5] Heikki Hyyr. Explaining and extending the bit-parallel approximate string matching algorithm of myers. Technical report, Citeseer, 2001.

[6] Jon Kleinberg and Eva Tardos. *Algorithm design.* Pearson Education India, 2006.

[7] Joseph B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, April 1983.

[8] Webb Miller and Eugene W. Myers. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology*, 50(2):97–120, 1988.

[9] B. J. Oommen and R. K. S. Loke. Pattern recognition of strings with substitutions, insertions, deletions and generalized transpositions. *Pattern Recognition*, 30(5):789800, 1997.

[10] Knut Reinert, Jens Stoye, and Torsten Will. An iterative method for faster sum-of-pairs multiple sequence alignment. *Bioinformatics*, 16(9):808–814, September 2000.

[11] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, January 1994.