Computational Molecular Biology

Last updated: November 8, 2023

Hello! These lecture notes were originally created during the 2018 iteration of CS 181 (Computational Molecular Biology) taught at Brown University by Sorin Istrail. The original notes were provided by Shivam Nadimpalli, and future updates have been made by course TAs including Elliot Youth, Daniel Ben-Isvy, Sam Maffa, Iris Huang, and Hannah Beakley. Please email any comments or typos to cs1810tas@lists.brown.edu.

Contents

1 Sequence Alignment

(Sept. 12)

In this chapter of the course, we will take a look at those algorithms that revolutionized the field of computational biology, and laid down robust foundations for its future.

We will first look at the problem of global alignment, after which we will consider the statistical foundations of this topic, and then finally we'll study local and gap alignment. The algorithmic technique of dynamic programming will play a key role throughout. Let's get started.

1.1 Global Alignment

Consider two DNA sequences $X =$ `ATTACG` and $Y =$ `ATATCG`. What's the optimal way to *align* these two sequences?

We haven't formally defined what any of this means, but going off our intuition, we would guess that the fewer mismatches an alignment has, the better it is. It's easy to see that the best alignment for these two sequences, then, is:

```
ATTA-CG
|||||||
A-TATCG
```

The problem of *global alignment* of two sequences is that of finding an alignment for every residue in each sequence.

Some foreshadowing: we will soon be looking at *local* alignment, in which we look for the best alignment of any subsequences of the given sequences. The motivation behind this, as well as how it differs from global alignment, will be made clear then.

1.1.1 Formal setup

However, what if a `(T, -)` alignment is really bad for us for some reason? Or if we want to minimize the number of `(A, C)` alignments? We can do so by penalizing such undesirable alignments, and this can be done by choosing an appropriate scoring scheme.

Given two strings over a finite alphabet Σ, a *scoring scheme* is a function $\delta : \Sigma \times \Sigma \to \mathbb{R}$. In the case of DNA sequences, $\Sigma = \{\texttt{A}, \texttt{T}, \texttt{G}, \texttt{C}\}$, and in the case of protein sequences Σ would be the set of 20 amino acids.

It's usually convenient to represent a scoring scheme in the form of a table. For example:

	A	T	C	G
A	1	0	0	0
T	0	1	0	0
C	0	0	1	0
G	0	0	0	1

Figure 1. The Unitary or Diagonal Scoring Scheme.

We also introduce a related concept: the *gap penalty* is a function $\delta : \Sigma \to \mathbb{R}$ which tells you the cost for aligning a character with a gap (-). Gaps are also called *indels* by biologists, which stands for "insertion/deletion mutations".

Abuse of Notation: We will use δ to refer to the scoring scheme and the gap penalty.

Finally, suppose we have an alignment $x_1 \dots x_n$ and $y_1 \dots y_n$ where $x_i, y_i \in \Sigma \cup \{\texttt{-}\}$, then the *score* of the alignment is given by $\Sigma_{i=1}^{m} \delta(x_i, y_i)$.

For example, under the unitary scoring scheme (Figure 1) with 0 gap penalty, the alignment of X and Y given in §1.1:

```
A   T   T   A   -   C   G
|   |   |   |   |   |   |
A   -   T   A   T   C   G
-----------------------------
1 + 0 + 1 + 1 + 0 + 1 + 1  =  5
```

has a score of 5.

If we change the gap penalty to -1, then the score of the same alignment changes to 3. From this, it's clear that you can get different scores for the same alignment under different scoring schemes, and that the optimal alignment of two sequences depends on the scoring scheme being used.

Finally, we can formally state the problem of global alignment:

> The optimal global alignment of two sequences under a chosen scoring scheme and gap penalty is the alignment of the two sequences with maximum score.

Now, how would one find this optimal global alignment? One way is to enumerate all possible alignments of the given sequences, and pick the one with the best score. But we'll see in the homework that this process is computationally inefficient. The principle of *dynamic programming*, however, comes to our rescue, as we will see in §1.1.3.

1.1.2 The edit graph

Let Σ be a finite alphabet, and let $X = x_1 \dots x_m$ and $Y = y_1 \dots y_n$ be strings over Σ. To X and Y, we associate a directed graph termed the *edit graph* of X and Y.

The vertices of the graph are the vertices of the rectangular grid of size $(m+1) \times (n+1)$. By $v(i, j)$ we will mean the vertex in the grid corresponding to the $(i+1)^{\text{th}}$ row and the $(j+1)^{\text{th}}$ column.

Between the vertices, we have three kinds of edges:

1. A *gap-in-Y* edge: for $1 \le i \le m, 1 \le j \le n$, we have an edge $v(i-1, j) \to v(i, j)$. This edge corresponds to $(x_i, \texttt{-})$ in the alignment.
2. A *gap-in-X* edge: for $1 \le i \le m, 1 \le j \le n$, we have an edge $v(i, j-1) \to v(i, j)$. This edge corresponds to $(\texttt{-}, y_i)$ in the alignment.
3. An *alignment* edge: for $1 \le i \le m, 1 \le j \le n$, we have an edge $v(i-1, j-1) \to v(i, j)$. This edge corresponds to (x_i, y_j) in the alignment.

What exactly does this graph tell us? This will become clear through the following example:

Example 1.1.1. Suppose we want to construct an edit graph for two sequences `AC` and `AGC`. We set up a grid of dimensions 3×4 as shown in the picture below:

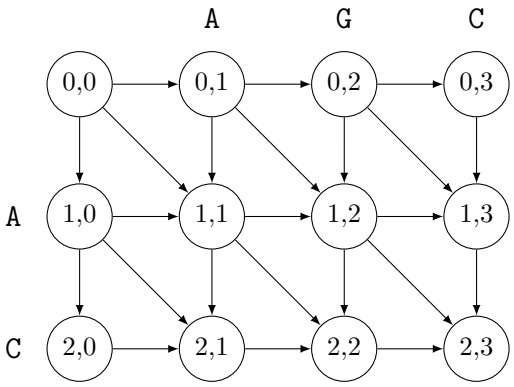


Figure 2. Edit Graph for `AC` and `AGC`.

Let's try looking at a path in this graph. Say we have the following sequence of vertices:

$$v(0,0) \to v(1,1) \to v(1,2) \to v(2,3)$$

Note that it can be interpreted as the following alignment:

```
A-C
AGC
```

Similarly, suppose we're given an alignment:

```
AC--
A-GC
```

We can construct a directed path in the graph:

$$v(0,0) \to v(1,1) \to v(2,1) \to v(2,2) \to v(2,3)$$

We come to the crucial insight which will allow us to develop an algorithm for global alignment:

> Given two sequences X and Y, there is a 1-to-1 correspondence between directed paths from $v(0,0)$ to $v(m,n)$ in their edit graph and their alignments.

We will explore this idea further in the next section.

1.1.3 The Needleman-Wunsch algorithm

(Sept. 19)

Recall that any path through the edit graph represents a sequence of matches, mismatches, and indels that uniquely represents one possible alignment of the two sequences. It's worth noting that in grid corresponding to the edit graph, we index from 0.

The fundamental concept of Needleman and Wunsch was that to calculate the optimal alignment score, one would only need to calculate and enumerate all the ways that an aligned pair can be added to a shorter segment to produce an alignment of an additional position in the sequences. If one always chooses the highest possible score for the extension, the highest score accumulated in the end is the global optimal score. And the path taken during the summation of the optimal score is the optimal alignment.

1.1.4 The main algorithm

Suppose we have two sequences X, Y over Σ of lengths m and n respectively. By X_i we will represent the i^{th} character of X.

We initialize a table of size S of dimensions $(m+1) \times (n+1)$. By $S_{i,j}$ we will represent the element in the $(i,j)^{\text{th}}$ cell of our table S.

function GLOBAL ALIGNMENT($X \in \Sigma^m, Y \in \Sigma^n$)
  $S_{0,0} \leftarrow 0$
  for $i \in \{1, 2, \ldots, m\}$ **do**
    $S_{i,0} \leftarrow S_{i-1,0} + \delta(X_i, \text{-})$
  end for
  for $j \in \{1, 2, \ldots, n\}$ **do**
    $S_{0,j} \leftarrow S_{0,j-1} + \delta(\text{-}, Y_j)$
    for $i \in \{1, 2, \ldots, m\}$ **do**

$$S_{i,j} \leftarrow \max \begin{cases} S_{i-1,j-1} + \delta(X_i, Y_j) \\ S_{i-1,j} + \delta(X_i, \text{-}) \\ S_{i,j-1} + \delta(\text{-}, Y_j) \end{cases}$$

    end for
  end for
  return $S_{m,n}$
end function

1.1.5 How to obtain the optimal alignment from this table?

Obviously for many applications, we don't just want to know what the score of the optimal alignment of two sequences is, but also what the alignment that achieves this score is.

There are several strategies for obtaining the optimal alignment:

- One solution is to fill out the table in the algorithm above, and then starting at the bottom right cell of the matrix, and recalculate the max that lead to the cell's value. This time, rather than just taking the max, record the cell from which the max originated (resolving ties arbitrarily: optimal alignments aren't unique in general).

  In other words, we determine whether the term of the recurrence originating from an X-gap, a Y-gap, or a match/mismatch was optimal, and backtrace to the corresponding position.

- An alternative (and computationally more efficient) solution is to place backpointers as we're computing a cell entry, indicating which cell it came from. And then we can follow these backpointers from $S_{m,n}$ to $S_{0,0}$ to obtain the optimal alignment.

1.1.6 Complexity of Needleman-Wunsch

We first look at the time complexity of the above algorithm. Indeed, we have:

- Line 2 requires 1 operation.
- Line 4 requires 3 operations, and is repeated m times.
- Line 7 requires 3 operations, and is repeated n times.
- Line 9 requires 9 operations, and is repeated mn times.
- Line 12 is a single operation.

Thus, Needleman-Wunsch runs in $\mathcal{O}(1 + 3m + 3n + 9mn + 1) = \mathcal{O}(mn)$ time. If $m \sim n$, then the runtime is $\mathcal{O}(n^2)$, and so this is a *quadratic* algorithm.

Note that it also requires $\mathcal{O}(mn)$ space, which is the size of our table.[1]

1.1.7 BLOSUM Matrices

As stated in the introduction in §1.2, we want to account for the probability of finding two different amino acids together during sequence alignment using empirical data. In order to do so, we use the technique of *substitution matrices* to assign match and mismatch scores more meaningfully.

The most popular type of substitution matrices are the BLOSUM (BLOcks SUbstitution Matrices) matrices introduced by Henikoff and Henikoff (1992), so named because they are created from data in the BLOCKS database. Instead of a formal construction, we'll look at a concrete example.

Let us align strings over the alphabet $\Sigma = \{\text{A, B, C }\}$. A *block* is just a fixed region in a given set of aligned sequences. For example, suppose that from some species whose proteins encoded over Σ, we obtain six sequences of the same protein that is 4 residues long from 6 different organisms. This gives us a block as follows:

Organism 1	B	A	B	A
Organism 2	A	A	A	C
Organism 3	A	A	C	C
Organism 4	A	A	B	A
Organism 5	A	A	C	C
Organism 6	A	A	B	C

Figure 3. Sample block.

We want to figure out what evolutionary information the block above encodes, so as to improve on our scoring scheme for sequence alignment.

Now, in our block above, we have a total of $6 \times 4 = 24$ residues. Of these 24 residues, we have 14 As, 4 Bs, and 6 Cs.

[1] However, it is possible to reduce the space usage to $\mathcal{O}(m + n)$ using Hirschberg's divide-and-conquer algorithm.

As we have 6 letters per column, there's $\binom{6}{2} = 15$ ways of picking a pair of letters in each column. As there are 4 columns, we have $15 \times 4 = 60$ possible ways of picking a pair of letters at the same position in the strings (i.e. in the same column) from the above table.

The table below lists the frequencies of these possible alignment pairs that we observe in the table above:

Aligned Pair	**Observed Frequency**
A to A	26/60
A to B	8/60
A to C	10/60
B to B	3/60
B to C	6/60
C to C	7/60

Here's an example of how the entries in the table above were computed: note that the only "B to B" alignments in the table above are in the third index in our sequences, and we have three such instances. One is between Organism 1 and Organism 4, the second is between Organism 1 and Organism 6, and the third one is between Organism 4 and Organism 6.

Now that we've gotten some observed data, let's see how it lines up with our expected data. As we have A occurring in our sequences with probability 14/24, B with probability 4/24, and C with probability 6/24 (see 2 paragraphs before table above), we should have an "A to A" alignment with probability $14/24 \times 14/24$.

Similarly, we must have an "A to B" alignment with probability $2 \times 14/24 \times 4/24$ and so on. We can thus obtain expected frequencies.

We'll use the log-likelihood ratio f of each possible aligned pair in order to compare these two frequencies. This value is defined as:

$$f = 2 \times \log_2 \left(\frac{\text{Observed frequency}}{\text{Expected frequency}} \right)$$

Let's put everything together:

Aligned Pair	**Observed frequency**	**Expected Frequency**	f
A to A	26/60	196/576	0.70
A to B	8/60	112/576	-1.09
A to C	10/60	168/576	-1.61
B to B	3/60	16/576	1.70
B to C	6/60	48/576	0.53
C to C	7/60	36/576	1.80

Figure 4. BLOSUM matrix scores.

How does this help us in our alignment algorithm in any way? Indeed, we can use a scoring scheme taking the value of f in the table above, and this is a statistically-justified scoring scheme.

For example, if the observed frequency of a substitution between A and B is less than the expected frequency, then $\delta(\text{A}, \text{B}) < 0$ as seen above. Namely, we penalize for an A-B alignment, and we can do so because from our observed data, this alignment doesn't really happen that frequently.

(Sept. 21)

1.2 Local Alignment

Recall that in global alignment, we want to find an alignment for *every* residue in all the sequences given. What if, however, we're given two sequences that we know to be very dissimilar, but which contain some regions of similarity?

Motivating Example 1. Here's a silly example. Say we have two sequences `TTCCCGGGAA` and `AAAAAAACCCGGGTTTTTTT`, and say we penalize mismatches with -2, gaps with -1, and alignments with $+1$. The optimal global alignment, then, is:

```
AAAAAAA__CCCGGG__TTTTTTT
_______TTCCCGGGAA_______
```

But what if we just want to find the region of similarity, namely `CCCGGG` in both sequences?

Motivating Example 2. Here's some more motivation rooted in biology: suppose we had 2 distantly related genomes, and we wanted to identify genes that were highly conserved between the two. In this case, we can't expect the strings to align well globally, but if we can find two regions that are highly conserved between the strings, then we can expect these regions to align well.

This is the problem of local alignment, which we will define formally in the next section.

1.2.1 Formal setup

Given a sequence $X = x_1, \ldots, x_m$, a *subsequence* of X is any string of the form $x_i x_{i+1} \ldots x_j$ for some $1 \leq i, j \leq m$. Note that we do not preclude $i = j$.

Let's define the problem of local alignment.

> Given X and Y over some finite alphabet Σ, and we want to find two subsequences α and β of X and Y respectively, such that the global alignment score between any pair of subsequences of X and Y is maximized by α and β.

A quick sanity check: the problem defined above allows us to identify regions of two strings that align well, even when the remainder of the strings aligns poorly. This does indeed match up with what we wanted to do, as described in our motivating examples at the start of this section.

Naively aligning all possible pairs of subsequences is prohibitively expensive from a computational viewpoint. In the next section, we'll see how to make slight changes to our algorithm for global alignment to get an efficient algorithm for local alignment.

1.2.2 The Smith-Waterman Algorithm

Given two sequences X and Y of length m and n respectively, we set up the edit graph and dynamic programming table S for X and Y just as we did for global alignment. We also carry over the notation used in that section.

Here's the algorithm for local alignment:

1: **function** LOCAL ALIGNMENT($X \in \Sigma^m, Y \in \Sigma^n$)
2: $S_{0,0} \leftarrow 0$
3: **for** $i \in \{0, 1, 2, \ldots, m\}$ **do**
4: $S_{i,0} \leftarrow 0$
5: **end for**
6: **for** $j \in \{1, 2, \ldots, n\}$ **do**
7: $S_{0,j} \leftarrow 0$
8: **end for**
9: **for** $j \in \{1, 2, \ldots, n\}$ **do**
10: **for** $i \in \{1, 2, \ldots, m\}$ **do**
11: $$S_{i,j} \leftarrow \max \begin{cases} 0 \\ S_{i-1,j-1} + \delta(X_i, Y_j) \\ S_{i-1,j} + \delta(X_i, -) \\ S_{i,j-1} + \delta(-, Y_j) \end{cases}$$
12: **end for**
13: **end for**
14: **return** $\max\{S_{i,j} \mid 0 \leq i \leq m, 0 \leq j \leq n\}$
15: **end function**

How does backtracking work in this case? It's the same as in the global case, except we start backtracking from $\max\{S_{i,j} \mid 0 \leq i \leq m, 0 \leq j \leq n\}$, i.e. the cell in the table with the maximum value (as opposed to from $S_{m,n}$ as in the case of global alignment), and we stop as soon as we encounter a cell with entry 0.

1.2.3 How's this different from global alignment?

Let S be the table for our alignment algorithm as above. In the case of global alignment, a table entry $S_{i,j}$ corresponded to the optimal alignment of the i^{th}-prefix of X (denoted by $X[i]$) and the j^{th}-prefix of Y (denoted by $Y[j]$).

What does an arbitrary table element correspond to in the case of local alignment? If you think about it, $S_{i,j}$ corresponds to the optimal global alignment between all the suffixes of all the prefixes of $X[i]$ and $Y[j]$.

Finally, in the case of global alignment, adding a constant value to all entries of the scoring matrix does not alter the optimal alignment (this only changes the scores of the alignments, but the *relative* scores remain the same). This, however, isn't the case with local alignment. Note that there are no negative numbers in the local alignment table.

1.2.4 Complexity of Smith-Waterman

In the algorithm above, lines 3 and 6 require a single operation each, the first being executed $m + 1$ times and the second n times. In line 10, we maximize over 5 cases, requiring an additional comparison relative to the global alignment for a total of 10 operations, and we execute this line nm times. In line 13, we maximize over nm items. Let's say each item requires a load and a comparison operation, for a total of $2nm$ operations.

The total number of operations is then $m + n + 12nm + 1$ operations, which again is in $\mathcal{O}(nm)$.

(Sept. 26)

1.3 Alignment with Gaps

We'll next look at a alignment with a special emphasis on gaps.

1.3.1 Some biological motivation

When a protein is synthesized, the DNA is *transcribed* from DNA to RNA. RNA is a similar macromolecule to DNA: the sugar deoxyribose is replaced by ribose, and the Thymimne base (`T`) is replaced by Uracil (`U`).

As we will see in the HW, DNA consists of large chunks of non-coding DNA called *introns*. These introns, once the DNA is transcribed into RNA, are removed via a process called *splicing*. The remaining sequences, called *exons*, are *translated* into protein.

Now, biologists can usually obtain this RNA from cells, but unfortunately it's already undergone splicing, i.e. the non-coding regions have been removed. Using this RNA, we can obtain DNA complementary to it, which is called cDNA.

In order to figure out what proteins parts of DNA are coding for, we want to find the location of this cDNA in the genome. Namely, we wish to align this cDNA within the genome. But this isn't as easy as it appears on first sight...

1.3.2 Gap Penalties

Recall that we wanted to align the cDNA of a protein transcript to the nuclear DNA from which it was transcribed. The difficulty arises in that large sections of nuclear DNA are excised via splicing before we get the mature RNA, so we will need to be tolerant of large gaps in our alignment. However, we don't want there to be too many of these gaps, and we don't want them to be too small.

In this case, we have a somewhat different notion of *gap*. A small gap and a big gap are in some sense equivalent, as we don't care how *long* the introns are, rather we want to work under the assumption that there aren't "too many" of them.

One way to approach this situation is to pick an appropriate scheme to penalize gaps during sequence alignment. We do so by means of *gap functions*.

Formally, a *gap function* is a map $w(l) : \mathbb{Z} \to \mathbb{R}$. Here, l is an integer representing the length of a gap, and $w(l)$ is the amount by which we penalize a gap of length l. A few different gap functions of varying degrees of usage are presented in the table below:

Gap Function	$w(l)$
Linear	τl
Affine	$\gamma + \tau l$
Logarithmic	$\gamma + \tau \log(l)$
Quadratic	$\gamma + \tau l^2$

You can think of γ as the "opening penalty", and τ as the "extension penalty".

Note that the linear gap function is the usual gap penalty that we used in the earlier sections. Relative to the linear gap function, affine and logarithmic gap penalties favor long gaps, whereas the quadratic gap penalty allows for short gaps but penalizes longer gaps. Make sure your intuition for these functions matches their mathematical form.

(Sept. 28)

1.3.3 Global alignment with gap function

In this subsection, we'll look at how adding a gap function to our scoring scheme alters our global alignment algorithm.

Adding a gap function requires us to use 4 matrices to keep track of the optimal alignment—think about why this is the case. We set them up as follows:

$$V_{i,j} = \max(E_{i,j}, F_{i,j}, G_{i,j})$$

$$G_{i,j} = V_{i-1,j-1} + \delta(x_i, y_j)$$

$$E_{i,j} = \max_{k \in \{0,1,\ldots,i-1\}} V_{i,k} - w(j-k)$$

$$F_{i,j} = \max_{l \in \{0,1,\ldots,j-1\}} V_{l,j} - w(i-l)$$

Here $V_{i,j}$ gives the gapped alignment cost of prefixes of lengths i, j. V is defined in terms of G, E, and F. $G_{i,j}$ gives the optimal gapped alignment score of prefixes of x, y of lengths i, j, conditional on the fact that the ith and jth characters are *aligned*. E and F on the other hand give optimal alignments of prefixes conditional on the last character containing a gap: $E_{i,j}$ is conditional on the fact that the terminal position in the alignment contains a *gap* in sequence x, and $F_{i,j}$ requires that the alignment ends on a gap in sequence y.

With these definitions, clearly $V_{i,j}$ is correct: note that $E_{i,j}$, $F_{i,j}$, and $G_{i,j}$ give the optimal gapped alignment cost under three mutually exclusive conditions, and $V_{i,j}$ gives the optimal amongst these. $G_{i,j}$ requires that the final character pair are aligned, so its score is identical to the *aligned* (no gap) case in linear gapped alignment. $E_{i,j}$ represents the case where x has a gap. We consider all possible alignments that end in an x gap: such an alignment can start with an arbitrary amount of the y string consumed. The maximum over k represents the maximum score over alignments of the prefix of length k of y and length i of x, and then the remainder of x is filled with gaps in the alignment. Similarly, we may flip x and y to make the same argument for $F_{i,j}$.

1.3.4 Complexity

Using this recurrence, we can handle *any* gap penalty function. However, we pay an asymptotically significant cost in terms of time, and a constant factor cost in terms of memory, to use this technique. Because the maxima in E and F are over a non-constant number of cells, the time complexity increases. Each such maximum needs to examine no more than $\max(m,n)$ cells of V, and there are $2mn$ cells in E and F. G on the other hand only needs to examine a constant number of cells, so evaluating it is efficient, requiring $\mathcal{O}(mn)$ time, and by the same reasoning, evaluating V also requires $\mathcal{O}(mn)$ time. The total time to evaluate every cell in each of these four matrices is thus $\mathcal{O}(mn) + \mathcal{O}(mn) + \mathcal{O}(mn(m+n)) + \mathcal{O}(mn(m+n)) = \mathcal{O}(mn(m+n))$, asymptotically[2].

[2]Note that $\mathcal{O}(\max(m,n)) = \mathcal{O}(m+n)$: to see this, consider the following: $\lim_{m,n} \to \infty \frac{max(m,n)}{m+n} \leq \frac{\max(m,n)}{2\max(m,n)} = \frac{1}{2}$.

Note also that the memory requirement increases. Rather than requiring $(m+1)(n+1)$ matrix cells, we now require $4(m+1)(n+1)$ matrix cells. However, this change is only by a constant factor, so there is no asymptotic difference.

1.3.5 Global Alignment with Affine Gap Alignment

In the previous section, we saw that general gapped alignment is asymptotically inferior to linear gapped alignment. Here we derive the recurrence relationship for the special case of affine gap penalties and show that it is asymptotically equivalent to linear gapped alignment, again with a constant factor memory increase.

Let the following be so:

- $\alpha =$ match score.
- $\beta =$ match penalty.
- $\gamma =$ gap opening penalty.
- $\tau =$ gap extension penalty.

Under these hypothesis, note that the affine gap penalty is given by $w(l) = \gamma + \tau l$. Now, we take the following recurrence relationships.

$$V_{i,j} = \max(E_{i,j}, F_{i,j}, G_{i,j})$$

$$G_{i,j} = \begin{cases} V_{i-1,j-1} + \alpha & x_i = y_j \\ V_{i-1,j-1} - \beta & x_i \neq y_j \end{cases}$$

$$E_{i,j} = \max(E_{i,j-1} - \tau, V_{i,j-1} - \gamma - \tau)$$

$$F_{i,j} = \max(F_{i-1,j} - \tau, V_{i-1,j} - \gamma - \tau)$$

Here V, G, E, and F all have the same interpretations as above.

Note: Here, we wrote V, G, E, and F in terms of match/mismatch/gap penalties, instead of using a scoring function δ and gap penalty function $w(l)$. You should convince yourself that these formulae may be equivalently expressed in terms of δ and $w(\cdot)$, rather than α, β, γ, and τ.

Therefore, in order to prove this algorithm correct, all we need do is show that $V_{i,j}$ here is equivalent to $V_{i,j}$ in the general gapped alignment recurrence. Note that V and G are defined equivalently, but both are dependent on E and F, thus we need only show that V and F are equivalent, and it follows that G, and thus V as well, are also equivalent.

The only thing we changed was that we fixed $w(l) = -\gamma - \tau$. This algorithm does not work for *general* gapped alignment, so we will need to rely on this change in order to prove that it works for *affine* gapped alignment.

Intuitively, E and F make sense, as gaps cost γ to open, and given that we are opening a gap, the previous pair did not have an equivalent gap, thus we take the score in $V_{i,j-1}$, subtract cost γ to open a gap, and subtract τ to extend this gap to length 1. If, on the other hand, there is a highly scoring alignment that *did* have an equivalent gap in the previous alignment, then we may take $E_{i,j-1} - \tau$ over $V_{i,j-1} - \gamma - \tau$. Here the cost is only τ, as the gap has already been opened. Note that this may occur even if $E_{i,j-1} < V_{i,j-1}$: the gap open penalty γ is used to control this behavior.

The above intuition could be turned into a formal proof, but we would require some technical details and essentially have to redo all the work from the general gapped alignment proof. I don't give a formal proof here, but I give an alternative simple argument that could easily be turned into one. Instead of directly showing correctness, we need only show that $E_{i,j}$ and $F_{i,j}$ are equivalent between the two cases. An informal proof follows:

Sketch of proof. Let E' denote the E from the general recurrence, and E denote the E from the affine gap penalty recurrence. Also, we proceed without loss of generality, noting that the following applies to F as well by symmetry.[3].

The crux of the argument is the following simple observation: $E_{i,j-1} - \tau \geq E_{i,j-2} - 2\tau$ for any i, j. This follows by way of contradiction; if this were not true, $E_{i,j-1} = \max(V_{i-1,j-2}, E_{i,j-2} - \tau)$ would be violated, as this would imply that $E_{i,j-1} < E_{i,j-2} - \tau$. This fact implies a much stronger looking property of E: namely that $E_{i,j-1} \geq E_{i,k} - (\tau((j-1)-k))$ for any $k \leq j-1$. To see this, simply note that this is repeated application of the original observation: each time k increases, Ei, k decreases by no more than τ.

Now, consider $E'_{i,j} = \max_{k \in \{0,1,\ldots,i-1\}}(Vi, k - w(j-k))$. Using the scoring methodology of E, this is equivalent to $\max(V_{i,j-1} - \gamma - \tau, E_{i,j-1} - \tau, E_{i,j-2} - 2\tau, E_{i,j-3} - 2\tau, \ldots, E_{i,0} - j\tau)$. Using the stronger property above, we see that any term after the second term is no greater than the second term, so we may remove them from the consideration in the maximum. This only leaves the first two terms, namely, we have that $E'_{i,j} = \max(V_{i,j-1} - \gamma - \tau, E_{i,j-1} - \tau) = E_{i,j}$. □

You should try to formally understand the above argument on its own, but it may help to understand it from a less formal intuition based perspective as well. The key observation translates to the following: "The optimally scoring alignment that ends in a double gap on string y over prefixes of x and y of lengths i and $j-1$ and $j-2$ scores at least as well as the optimally scoring alignment that ends in a triple gap on string y over prefixes of i and $j-2$." The stronger form of this is basically that "extending a gap of any length has score no more than extending the optimal sequence ending in a gap by 1." We then translate E' to use the same scoring mechanism as E, and then using this property, show that the maximum is equivalent to a maximum over two values: $V_{i,j-1} - \gamma - \tau$ and $E_{i,j-1} - \tau$, or in other words, the cost of *creating a new gap* and *extending an existing gap.*

[3]In mathematics, we need to be *very* careful with symmetry, as it is far too easy to claim symmetry incorrectly to simplify a proof. As this is an informal proof, I do exactly this; such an argument doesn't hold up under scrutiny unless we are very careful. In this alignment problem, we have a very strong notion of symmetry when δ is a symmetrical matrix (*i.e.* $\delta_{a,b} = \delta_{b,a}$), in that Global Alignment Score(x, y, δ) = Global Alignment Score(y, x, δ). Even without this property, we have a weaker notion of symmetry that is sufficient for this argument, as we never directly refer to δ. The basic notion is that E and F correspond to one another, with one operating over V and the other over V^T where we flip the strings x and y. This is an obscenely long footnote!

Now that we see the algorithm *works*, the natural next question to ask is how much time it takes? Have we beaten the $\mathcal{O}(mn(m+n))$ cost of general gap alignment? Hopefully by now you are getting the hang of asymptotic analysis (if not, a great exercise would be to repeat the analysis for the general gapped alignment for this case), so I will skip a few steps. Each of the 4 matrices has $(m+1)(n+1)$ cells, and each cell requires constant time to calculate. Therefore, we may conclude that the algorithm requires only $\mathcal{O}(mn)$ time.

1.3.6 Addendum

We mentioned several families of gap penalty in the earlier lecture. We saw that the general gapped alignment algorithm is asymptotically slower than linear gapped alignment (Needleman-Wunsch), but we also saw that the special case of affine gapped alignment is asymptotically equivalent. We may wonder what other types of gapped alignments may be efficiently computed. Miller and Meyers showed that for arbitrary concave[4] weighting functions (including logarithmic), for two sequences each no longer than n the optimal gapped alignment may be computed in $\mathcal{O}(n^2 \log(n))$ time.

We can also (relatively) efficiently compute quadratic gap penalty alignments by a simple argument. Suppose $\alpha = \beta = \gamma = \tau = 1$. Then, the trivial ungapped alignment has score $-n$. Similarly, if every character aligns, the trivial ungapped alignment has score n, and furthermore, no alignment of any pair of substrings of x and y can have score in excess of n. Now, we can argue that the optimal quadratic gapped alignment does not contain any gaps of length $\sqrt{3n}$ or greater.

By way of contradiction, without loss of generality, assume x can be split into x_a, x_b, x_c such that the concatenation $x_a \circ x_b \circ x_c = x$, and that x_b has length l_{xb} at least of $\sqrt{3n}$. Assume that the optimal alignment of x and y has a gap in x throughout x_b. Therefore, the optimal alignment score is given by $AS(x_a, y_a) - w(l_{xb}) + AS(x_c, y_b)$, for some y_a, y_b such that $y = y_a \circ y_b$.

Now, by the above bound $AS(x_a, y_a)$ and $AS(x_c, y_b)$ each have score no more than n, and $w(l_{xb})$ has cost $\gamma + \tau l_{xb}^2 = 1 + l_{xb}^2 < 1 + 3n$, therefore $AS(x_a, y_a) - w(l_{xb}) + AS(x_c, y_b) < 2n - 3n = -n$. As above, $AS(x, y) \geq -n$, thus we may conclude that no alignment with a gap greater than $\sqrt{3n}$ is optimal.

Asymptotically, for nonzero $\alpha, \beta, \gamma, \tau$, similar results hold, and these constants become unimportant. In general, we can conclude that no gaps of $\mathcal{O}(\sqrt{n})$ exist in the optimal alignment, so when evaluating the maxima in E and F, we only need to consider gaps of length $\mathcal{O}(\sqrt{n})$, yielding the complexity $\mathcal{O}(mn\sqrt{n})$.

1.4 Statistics and Alignment

In previous lectures, we have mentioned how the values in the similarity matrix can affect the behavior of local alignment, causing it to behave like global alignment *on average* when its expected value is negative. In this section, we will discuss some other results that arise from specific parameter choices when using local alignment.

First, we will examine some cases where certain parameter sets can transform local alignment into other well-known problems.

[4] A *concave function* is any f such that $\forall x \leq y \leq z$, $\frac{f(x)+f(z)}{2} \leq f(y)$. In other words, for any y in the domain of f, the value $f(y)$ lies on or above the line connecting some distinct $(x, f(x))$ and $(z, f(z))$.

Let's compare two random sequences of length n.

1. $$\begin{cases} \text{match} & = 1 \\ \text{mismatch penalty} & = -\infty \\ \text{gap penalty} & = 0 \end{cases}$$

   Notice that these parameters will cause local alignment avoid pairing mismatched characters. As a result, it finds an ordered sequence of exact matches between strings, and it will add gaps freely as needed. This is the **longest common subsequence** between the strings.

   On average, these alignments will have score linear in the length of the sequences. That is, the score will be on the order of n.

2. $$\begin{cases} \text{match} & = 1 \\ \text{gap penalty} & = -\infty \\ \text{mismatch} & = -\infty \end{cases}$$

   Notice that in this case, local alignment will avoid creating gaps and pairing mismatched characters. So it will locate only contiguous, or connected, regions of matches. This is the **longest common substring** between the sequences.

   On average, the score of the alignment is proportional to the logarithm of the length of the sequence. That is, the score will be on the order of $\log(n)$

   This is because the probability of random sequences matching exactly decays exponentially as the sequences grow longer. For example, the probability of aligning two random DNA bases is $\frac{1}{4} = (P(AA) + P(CC) + P(GG) + P(TT))$. The probability of matching two random length-2 DNA strings will be $\frac{1}{16} = (\frac{1}{4} \cdot \frac{1}{4})$. The probability for length-3 strings is $\frac{1}{64} = (\frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{4})$, and so forth. Since the alignment score is equal to the length of continuous matches, as the lengths of the sequences increase, there are more substrings of all lengths from 1 to $n + 1$, each of which has an exact match probability which increases logarithmically, so the score will grow logarithmically as well.

We also know the following general results about the relation between local alignment behavior and global alignment behavior:

1. For a particular set of parameters, if the expected score of global alignment on two random sequences is *positive*, then the local alignment score with the same parameters grows **linearly** with the lengths of the sequences.
2. For a particular set of parameters, if the expected score of global alignment on two random sequences is *negative*, then the local alignment score with the same parameters grows **logarithmically** with the lengths of the sequences.
3. For a particular set of parameters, if the expected score of global alignment on two random sequences is 0, then local alignment is at its **phase transition**. This means that the behavior of local alignment score as it relates to sequence length falls at the boundary between the logarithmic growth and linear growth regimes. This behavior

can be unpredictable, and even slight parameter changes may cause local alignment adopt linear or logarithmic behaviors.

The expected score of sequence alignment is very difficult to compute exactly, but we can estimate its value by calculating a large sample of alignments and averaging their scores.

The proofs of each of these results is beyond the scope of this course, but hopefully your intuition about alignment algorithms agrees with these conclusions!

(Oct. 5+10)

1.5 Connections with Graph Theory

As we have discussed in previous lectures, identifying the maximum similarity score is equivalent to finding the *highest-weight* or *longest* path in the edit graph. In general, finding the *longest* simple path through a graph is NP-hard, but in a *directed acyclic graph*, the problem is equivalently difficult to finding the *shortest* path.

What is the P versus NP problem?

Some questions are easy (from a computational point of view) to *solve* quickly (i.e. in polynomial time), whereas some questions have solutions that are easy to *verify*. For example, solving a sudoku question is significantly harder than verifying whether an instance of a filled-in sudoku puzzle is correct right or wrong.

The P versus NP question asks whether every problem whose solution can be quickly verified (technically, verified in polynomial time) can also be solved quickly (technically, in polynomial time). You can read more about this here.

To see this, note that a DAG contains *no* cycles: once a path reaches a node, there is no way to return to this node. For this reason, we have no issues with infinite weight paths and negative cycles. Then, we can simply negate the weight of each edge, and the shortest path through the negated graph is the longest path in the unnegated graph. Figure 3 below illustrates this concept:

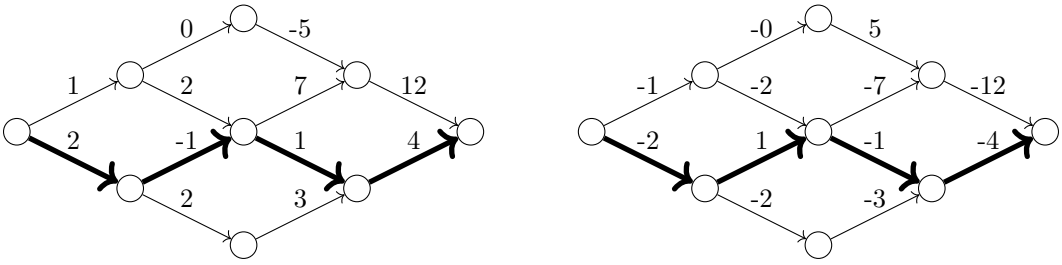


Figure 4. Graphs G (left) and G' (right) obtained by negating weights. The shortest path in G corresponds to the heaviest path in G'.

Dijkstra's Algorithm (here's the Wikipedia page) is guaranteed to find the shortest path through any graph that *does not* contain any negative-weight edges. This is not directly applicable when we have negative similarity scores, but we can still use this algorithm to solve global alignment problems with the following trick:

Suppose we wish to identify the optimal global alignment for any strings x, y with scoring scheme δ. We may choose some δ' with a constant c such that:

$$\delta'(a,b) \doteq \begin{cases} a \neq \text{-} \wedge b \neq \text{-} & : \quad \delta(a,b) + 2c \\ a = \text{-} \vee b = \text{-} & : \quad \delta(a,b) + c \end{cases}$$

$$\delta'(a,b) = \begin{cases} a, b \in \Sigma & : \quad \delta(a,b) + 2c \\ a \text{ or } b \text{ is a gap} & : \quad \delta(a,b) + c \end{cases}$$

$$\delta'(a,b) = \begin{cases} \delta(a,b) + 2c & a, b \in \Sigma \\ \delta(a,b) + c & \text{either } a \text{ or } b \text{ is a - (gap)} \end{cases}$$

We may select c such that each value in the range of δ is negative, and then the negated edit graph contains only positive edges, allowing us to use Dijkstra's Algorithm to solve for the longest path.

To see that the alignments will be the same, consider that, for strings of lengths m and n, supposing without loss of generality that $m \leq n$, there are exactly a aligned characters and b gaps, such that $a \in \{0, 1, 2, \ldots, m\}$, and $b = n - m + 2(m - a)$. Any alignment produced using δ' will have the score of the alignment produced using δ, plus an additional $2ca$ for aligned characters and cb for gap characters.

Substituting these values in, we see that $2ca + c(n - m + 2(m - a)) = 2ca + cn - cm + 2cn - 2ca = c(n + m)$. This value does not depend on a and b (only on the lengths of x and y, thus we may conclude that the score of any alignment produced using δ differs from that produced by δ' differs only by a constant amount.

In a nutshell, Djikstra's algorithm allows us to find the optimal global alignment.

But what about local alignment? Note that we *cannot* use the trick above to ensure a graph with nonnegative edge weights from the edit graph. We can, however, use the Bellman-Ford algorithm, which runs in $\mathcal{O}(nm(n + m))$ time on a graph of *diameter* in $\mathcal{O}(n + m)$. Note that this time complexity is inferior both to that of Dijkstra's algorithm (which runs in $\mathcal{O}(nm \log(nm))$ time), and our previous dynamic programming solution for local alignment.

Fortunately, we can leverage the fact that we have a DAG to obtain a more efficient algorithm. In a DAG, we can use *topological sorting*, where we sort the vertices of the graph according to the partial ordering $a < b \Leftrightarrow b$ is reachable from a. The existence of this partial ordering is a property of the DAG: if the edit graph contained cycles, such an ordering *would not* exist.

In the context of alignment, this is very easily interpretable: any node $S_{i,j} < S_{k,l}$ implies that $(i, j) \neq (k, l), k \geq i$, and $l \geq j$.

To topologically sort a graph, we require $\mathcal{O}(V + E)$ time, and to identify the shortest path in a topologically sorted graph, we also require $\mathcal{O}(V + E)$ time. In a two-dimensional

edit graph, there are $\mathcal{O}(nm)$ vertices, and each vertex is associated with no more than 3 edges, thus there are $\mathcal{O}(nm)$ edges as well. We may therefore conclude that identifying the shortest path in a DAG via topological sorting is possible in $\mathcal{O}(nm)$ time: exactly the same time complexity as by our dynamic programming algorithm!

So, we saw that Djikstra's doesn't work for local alignment, but Bellman-Ford does. This is slower than our earlier solution, so we can instead use topological sorting, giving us an algorithm that runs (asymptotically) just as fast as our earlier solution.

You must be wondering, why exactly did we do all this?

We had algorithms for computing the global, local, general gapped, and affine gapped alignments of any pair of sequences, and we also had some idea (from Homework 1) of how to generalize these algorithms to operate over more than two input sequences. However, by examining the problem from a graph theoretic perspective, we can also convert the problem into a DAG, and then apply an existing algorithm (shortest path via Topological Sort) to solve the problem.

Furthermore, since the edit graphs for global, local, and affine gapped alignment all contain $\mathcal{O}(nm)$ edges and vertices, we can obtain the same time complexity using this technique as with our dynamic programming solutions.

If we approach the problem in this manner, our solutions are *theoretically simpler* and our implementations (generally) require less code. On the theoretical side, we only need to prove that our algorithm produces the correct edit graph and that the longest path through the edit graph is equivalent to the maximum similarity alignment. As for implementation, we need only write a program that translates the recurrence relationships into edit graphs: the work of initializing and filling the dynamic programming matrix is subsumed by the longest path algorithm. Backtracing too becomes simpler: we need only translate the longest path into an alignment.

With this, we finish our chapter on sequence alignment, and move onto combinatorial pattern matching.

2 Combinatorial Pattern Matching

Over the years the expression combinatorial pattern matching has become a synonym for the field of theoretical computer science concerned with the study of combinatorial algorithms on strings and related structures.

The term *combinatorial* emphasizes that these are algorithms based on mathematical properties and a deep understanding of the individual problems, in contrast to statistical or machine learning approaches, where general frameworks are often applied to model and solve the problems.

Work in this field began in the 1960s with the study of how to efficiently find all occurrences of a pattern string p in a text t. We, too, take this as our starting point...

But before looking at the highlight of this section, the Knuth–Morris–Pratt algorithm for string matching, we'll get a taste of the core computational model used by the algorithm, namely *deterministic finite automata* (DFAs for short).

2.1 Finite Automata and Regular Expressions

Finite automata are the most basic models of computation. What can such a basic model of computation do? Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.

Before studying finite automata, let's set up some terminology:

Definition. An *alphabet* is a finite set of symbols or letters. A *word* or *string* over an alphabet Σ is a finite sequence of symbols from Σ.

Definition. The empty string (the string with no symbols) is denoted by ϵ.

The empty string ϵ is to strings what 0 is to the integers. This'll become clear in a second:

Definition. Given two strings $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$, the *concatenation* of x and y is denoted by xy and is the string $x_1 \dots x_n y_1 \dots y_m$.

Thus, for any string x, we have $x\epsilon = \epsilon x = x$. Some more definitions follow.

Definition. Given a string $x = x_1 \dots x_n$ over Σ (i.e. $x_i \in \Sigma$)we define:

1. The *length* of x is written $|x|$ and is the number of characters in x (here, $|x| = n$).
2. A *prefix* of x is a string $x_1 \dots x_i$ for some $1 \leq i \leq n$.
3. A *suffix* of x is a string $x_j \dots x_n$ for some $1 \leq j \leq n$.

And finally, we have:

Definition. A *language* is a set of strings over some alphabet Σ.

2.1.1 Operations on languages

Since a language is, in some sense, nothing but a set, we can define operations on languages just as we do with sets.

1. The union, intersection, and complement of a language are defined just as they are for sets.
2. Given two languages L_1 and L_2 over Σ, we define the language $L_1 \circ L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$, that is, $L_1 \circ L_2$ is the language obtained by concatenating all words in L_1 with all words in L_2.
3. Let $L^0 = \{\epsilon\}$. Define $L^i = L \circ L^{i-1}$ for $i \geq 1$. The *Kleene closure* of a language L over alphabet Σ is defined the language $L^* = L^0 \cup L^1 \cup L^2 \dots = \bigcup_{i=0}^{\infty} L^i$.
4. Note that $\epsilon \in L^*$, but sometimes we will not want to include the empty string in the Kleene closure, leading us to define the *positive closure* of a language. Given a language L, it's positive closure, denoted L^+, is given by $L^+ = L^1 \cup L^2 \dots = \bigcup_{i=1}^{\infty} L^i$.

In words, the Kleene closure of a set of strings is a new set containing all possible combinations of concatenations of strings from the original language. The Kleene closure of a language is also called the *Kleene star* of a language, and sometimes we'll be lazy and refer to it directly as just the *star* of a language.

Here's some examples:

1. $0^* = \{\epsilon, 0, 00, 000, \ldots\}$.
2. $0^* \cup 1^* = \{\epsilon, 0, 1, 00, 11, 000, 111, \ldots\}$.
3. $\{0 \cup 1\}^* =$ all strings over the alphabet $\{0, 1\}$.

2.1.2 Regular Expressions

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as $(5+3) \times 4$. Similarly, we can use *regular operations* to build up expressions describing languages, which are called *regular expressions*. An example is $(0 \cup 1)0^*$.

This'll become very clear in a second.

Definition. The *regular expressions over* Σ and the languages they denote are defined recursively as follows:

1. $\emptyset$ is a regular expression denoting the empty language $\emptyset$.
2. ϵ is a regular expressions and denotes the language $\{\epsilon\}$.
3. For each symbol $a \in \Sigma$, a is a regular expression and denotes the language $\{a\}$.
4. If p and q are regular expressions denoting the languages P and Q respectively, then the following are regular expressions:
   (a) $(p + q)$ corresponding to the language $P \cup Q$
   (b) pq corresponding to the language $P \circ Q$
   (c) p^* corresponding to the language P^*
   (d) p^+ corresponding to the language P^+

Here are some examples of regular expressions and the languages that they correspond to:

Examples. Let $\Sigma = \{0, 1\}$.

1. $0^* = \{\epsilon, 0, 00, 000, \ldots\}$
2. $0^*1 = \{1, 01, 001, 0001, \ldots\}$
3. $(0 + 1) = \{0, 1\}$
4. $(0 + 1)^* = \Sigma^* =$ all strings over Σ
5. $1(0 + 1)^*1 + 1 =$ all strings starting and ending with 1

Those languages that correspond to some regular expression have several *nice* properties (which we'll see when we learn about finite automata), and thus have a special name:

Definition. A language is said to be a *regular language* if and only if it is denoted by a regular expression over a finite alphabet.

2.1.3 Deterministic Finite Automata

There exist two types of finite state automata: *deterministic* and *non-deterministic*. We'll see that these two are actually equivalent from a computational point of view, and each of them is in turn equivalent to a regular expression.

Informally, a deterministic finite automaton (DFA) is a machine with a control unit, an input tape, and a head that processes the input tape from left to right, such that:

- The control unit consists of a number of *states*, and the DFA is in one of these states at any given instance.
- Each time the head reads an input, the DFA can transition from one state to another.

Finite automata and their probabilistic counterpart Markov chains are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets.

Here's a more formal definition:

Definition. A *deterministic finite automaton* (DFA) is a machine $M = (S, \Sigma, \delta, s_0, F)$ comprising of a control unit, a tape for input, and a head that reads the tape left to right, where:

1. S is a *finite* set of states.
2. Σ is a finite alphabet for the symbols on the tape.
3. $\delta : S \times (\Sigma \cup \epsilon) \to S$ is the *state transition function*
4. s_0 is the initial state of the DFA.
5. $F \subseteq S$ is the set of *final* or *accepting* states.

In order to describe the state of a DFA at some point during computation, we define the following:

Definition. An *instantaneous description* (ID) of a DFA is a pair (s, w) where $s \in S$ represents the current state, and $w \in \Sigma^*$ represents the unused portion of the input tape, i.e. the symbol under the tape head followed by the rest of the string to the right.

Note that the *initial* ID is the ID given by (s_0, w) and an *accepting* ID is an ID of the form (s, w) for $s \in F$.

Having defined the notion of an ID, we'll set up notation to talk about how these IDs change throughout the computation by means of a binary relation.

Definition. If $\delta(s, a) = s'$, then we write $(s, aw) \vdash (s', w)$ for all $w \in \Sigma^*$, where $a \in \Sigma \cup \epsilon$.

Note that if, in the above definition, $a = \epsilon$, then we transition from state s to s' without reading any input.

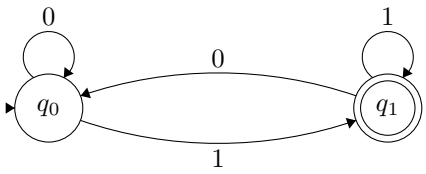
Definition. We use $\vdash^*$ to denote the reflexive, transitive closure of $\vdash$.

And finally:

Definition. The language $L(M)$ of a DFA M is the set of strings *accepted* by M, that is:

$$L(M) = \{w \in \Sigma^* \mid (s_0, w) \vdash^* (s, \epsilon) \text{ for some } s \in F\}$$

This was not done in class, but here's an example of a simple DFA. Suppose we have the alphabet $\Sigma = \{0, 1\}$. Consider:



In the picture above, q_0 is the initial state of the DFA, and q_1 is an accepting state (commonly denoted using the double circles in the above diagram).

The transition function is represented as arrows, e.g. $\delta(q_0, 1) = q_1$, and $\delta(q_0, 0) = q_0$ are represented by an arrow from $q_0 \to q_1$ for "input" 1, and similarly for the self-loop from $q_0 \to q_0$.

What's the language of this DFA? Indeed, the DFA accepts all strings that end in 1. Here's an example of a computation by this DFA on input 10110:

$$(q_0, 10110) \vdash (q_1, 0110) \vdash (q_0, 110) \vdash (q_1, 10) \vdash (q_1, 0) \vdash (q_0, \epsilon)$$

and as $q_0 \notin F$, we say that the DFA *rejects* the string 10110.

On the other hand, 101101 is accepted by the DFA:

$$(q_0, 101101) \vdash (q_1, 01101) \vdash (q_0, 1101) \vdash (q_1, 101) \vdash (q_1, 01) \vdash (q_0, 1) \vdash (q_1, \epsilon)$$

2.1.4 Nondeterministic Finite Automata

Put (very) informally, a nondeterministic finite automaton (NFA) is nothing but a DFA where we're allowed to be in multiple states at the same time. Spooky, isn't it? :)

Formally, we have:

Definition. A *nondeterministic finite automaton* (NFA) is a machine $M = (S, \Sigma, \delta, s_0, F)$ comprising of a control unit, a tape for input, and a head that reads the tape from left to right, where:

1. S is a *finite* set of states.
2. Σ is a finite alphabet for the symbols on the tape.
3. $\delta : S \times (\Sigma \cup \epsilon) \to \mathscr{P}(S)$ is the *state transition function*
4. s_0 is the initial state of the NFA.

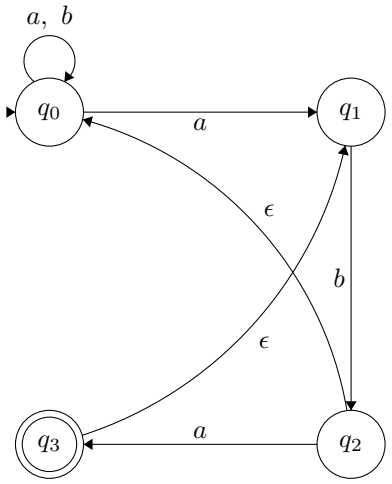
5. $F \subseteq S$ is the set of *final* or *accepting* states.

The *language* of a NFA is defined analogously to what we did for DFAs, and the definitions for IDs carry over as well.

Well, so what's different? Note that the definition above looks very similar to the one for DFAs, with one small (but significant!) change: the transition function has range $\mathscr{P}(S)$, which is the *power set* or the set of all subsets of S.

This makes rigorous our earlier idea of "being in multiple states at the same time".

Here's an example of a NFA:



What's the language of this NFA? Indeed, it accepts all strings that end in *aba*. Let's see a concrete example.

Suppose we have a string *ababa* provided as input to the NFA above. Our NFA starts out in the initial state q_0 as seen from the picture above. We denote one possible series of transitions of IDs:

$$(q_0, ababa) \vdash (q_1, baba) \vdash (q_2, aba) \vdash (q_3, ba) \vdash (q_1, ba) \vdash (q_2, a) \vdash (q_3, \epsilon)$$

and so *ababa* is in the language of the machine above.

Alternatively, another computation history is given by:

$$(q_0, ababa) \vdash (q_0, baba) \vdash (q_0, aba) \vdash (q_0, ba) \vdash (q_0, ba) \vdash (q_0, a) \vdash (q_0, \epsilon)$$

Thus, unlike a DFA, a NFA can have multiple computation histories for the same input string. As long as there exists one computation history in which the given string is accepted, we say that the NFA accepts that string!

2.1.5 Tying it all together

You might expect that being able to be in multiple states at the same time means that NFAs are more *powerful* than DFAs, i.e. can recognize some languages that DFAs cannot recognize. This, however, is not true.

It turns out that every NFA can be converted to a DFA; the issue, however, is that this transformation results in an exponential number of states in the DFA as compared to in the NFA. It also turns out that every language recognized by a DFA is a regular language.

Let's state this formally:

Theorem. Every NFA can be converted to a DFA recognizing the same language.

Theorem. Every language recognized by a DFA is a regular language.

We thus have the following correspondence:

$$\text{NFAs} \iff \text{Regular Expressions} \iff \text{DFAs}$$

Formal proofs were not given in class, but you can find them in Sipser's Introduction to the Theory of Computation (or in CS 1010 ☺).

2.2 The Knuth-Morris-Pratt Algorithm

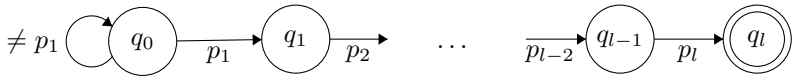
Let's get back to our original problem, namely string matching. Formally, our problem is as follows:

String Matching
Input: A pattern p and a text t, over some alphabet Σ. **Output:** The first exact occurrence of p in t.

2.2.1 First steps

Let's get started with constructing M_p.

1. First construct a *skeletal* DFA:



2. Observe that the state q_i of M_p corresponds to the prefix $p_1 \dots p_i$ of p.
3. Our machine M_p will start in state q_0 reading t_1, the first symbol in the text.

However, what do we do in case we're at state q_j with the next input symbol t_i, and $p_{j+1} \neq t_i$? Should we just start over from q_0? But then, this isn't any different from Camillo's algorithm!

Instead, we can harness the fact that DFAs have, in some sense, a *memory*, and use this fact to transition to an *appropriate* state in our machine M_p. This is all very vague, but will become clear soon once we learn about the *failure function*.

2.2.2 The Failure Function

Before proceeding, we recall some facts about our skeletal machine M_p. The state q_j of the skeletal machine M_p corresponds to the prefix $p_1 \dots p_j$ of p. Also, M_p starts in state q_0, and

so we need to position it *vis-á-vis* the text. Thus, the head of M_p starts off at t_1 on the input tape.

Suppose after having read $t_1 t_2 \dots t_k$ (the first k characters of the text t), we find that M_p is in state q_j. This implies that:

1. The last j symbols of $t_1 \dots t_k$ are $p_1 p_2 \dots p_j$.
2. The last m symbols of $t_1 \dots t_k$ are *not* a prefix of $p_1 \dots p_l$ for $m > j$, i.e. the suffix of size of j of $t_1 \dots t_k$ is the prefix of size j of p.

Now, if $t_{k+1} = p_{j+1}$, then we move to state q_{j+1} and our tape head moves right by one. But what if $t_{k+1} \neq p_{j+1}$? In this case, where should we transition to?

If you think about it, you'll realize that we want M_p to enter the state q_i for highest i such that $p_1 \dots p_i$ is a suffix of $t_1 \dots t_{k+1}$. In order to determine this i, we construct that we'll call the *failure function* for the pattern p. Formally, we have:

Definition. The function f such that $f(j)$ is the largest integer $s < j$ for which $p_1 \dots p_s$ is a suffix of $p_1 \dots p_j$ is said to be the *failure function* for the pattern p.

What exactly does this function compute? In words, $f(j)$ is the number of positions we can "fall back" to keep looking for the pattern p. Also, note that $s < j \implies p_1 \dots p_s$ is a *proper* prefix of $p_1 \dots p_j$.

Here's an example of the failure function f for the pattern $p = aabbaab$ over $\Sigma = \{a, b\}$.

i	1	2	3	4	5	6	7
$f(i)$	0	1	0	0	1	2	3

More concretely, suppose $i = 2$. Then the prefix we're dealing with is aa, and the only proper prefix of this prefix is a, which happens to be a suffix as well. So $f(2) = 1$.

Similarly aab is the longest proper prefix of $aabbaab$ that is also a suffix of $aabbaab$, and hence $f(7) = 3$.

2.2.3 How do we use the failure function?

We'll now look at an algorithm that utilizes failure function for a pattern p, in order to find an occurrence of p in text t.

First, define f^n recursively as follows: let $f^1(j) = f(j)$. We say $f^n(j) = f(f^{n-1}(j))$.

Thus, from our example in §2.2.2, we have $f^2(6) = 1$. Intuitively, $f^n(j)$ is just f applied n times to j.

Suppose that M_p is in state j having read $t_1 \dots t_k$, and $t_{k+1} \neq p_{j+1}$. At this point, M_p applied the failure function repeatedly to j, and two cases arise:

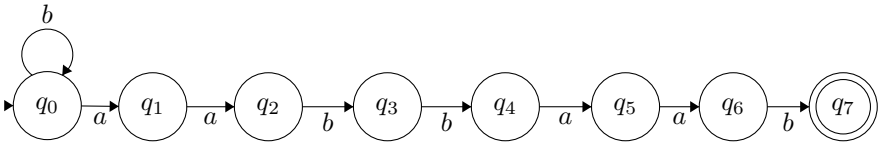
1. $f^m(j) = u$ and t_{k+1} is precisely p_{u+1}, or
2. $f^m(j) = 0$ and t_{k+1} is different from p_i for all $i \in \{1, \dots, j\}$.

In the first case above, we want M_p to enter state q_{u+1}. In the second one, M_p enters the state q_0. But in both cases, the tape head proceeds to the cell on the input tape with t_{k+2}.

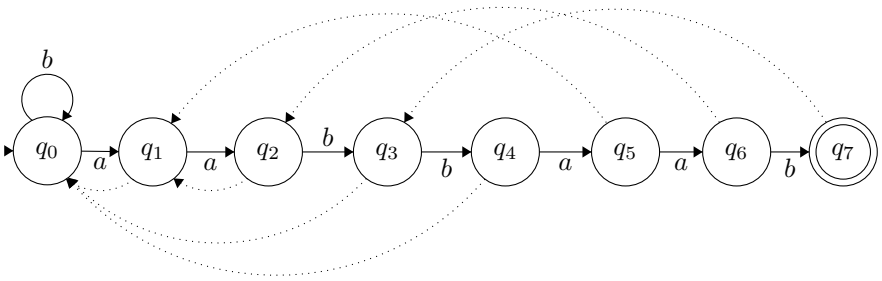
In Case 1, it's easy to see that if $p_1 \dots p_j$ was the longest prefix of p that is a suffix of $t_1 \dots t_k$, then $p_1 \dots p_{f^m(j)+1}$ is the longest prefix of p that is a suffix of $t_1 \dots t_{k+1}$. In Case 2, no prefix of p is a suffix of $t_1 \dots t_{k+1}$.

The algorithm is now clear: M_p proceeds reading t_{k+2} and so on and so forth, operating in the fashion described above, until it reaches the final state q_l, in which case it accepts. If it never reaches the final state, then the given text t has no occurrence of p.

Let's go back to our earlier example from §2.2.2 where $p = aabbaab$. We have a skeletal DFA for p as follows:



Having computed the failure function, we can add in the following arrows:



Let $t = abaabaabbaab$.

Initially M_p is in state q_0. On reading the first symbol of t (i.e. t_1 which is an a), M_p enters q_1. Since there is no transition from q_1 on the second input symbol of t (i.e. t_2 which is b), M_p enters state q_0. More explicitly, M_p goes back to the state given by the output of the failure function from q_1.

Now, $u = 0$ and $p_{u+1} = p_1 = a \neq t_2$, and so Case 2 (from the discussion in § 2.2.3) prevails and M_p remains in state 0. From here M_p continues consuming characters in the input string and following the corresponding arrows. If the machine ever reaches q_7, the pattern p has been found in the text t.

2.2.4 Computing the failure function

Having seen the failure function in action, let's now see how to compute it. That is, given a string $p = p_1 \dots p_l$, how do we compute its failure function?

1: **function** FAILURE FUNCTION($p = p_1 \dots p_l$)
2: $f(1) \leftarrow 0$

```
3:      i ← 0
4:      for j ∈ {2, ..., l} do
5:          i ← f(j − 1)
6:          while p_j ≠ p_{i+1} and i > 0 do
7:              i ← f(i)
8:          end while
9:          if p_j ≠ p_{i+1} and i = 0 then
10:             f(j) ← 0
11:         else
12:             f(j) ← i + 1
13:         end if
14:     end for
15: end function
```

2.2.5 Proof of correctness for the failure function

How can we be sure our algorithm for constructing the failure function is correct? By proof, of course! We'll refer to the pseudocode above in our proof using line numbers (L#).

Let's prove the definition of the failure function by induction: that for a pattern p of length l, $f(j) = i$ is the largest integer $i < j$ such that $p_1 p_2 \dots p_i = p_{j-i+1} p_{j-i+2} \dots p_j$.

Assume that this induction hypothesis is true for all $f(k)$ such that $k < j$.

Proof. **Base case:** $f(1) = 0$. Empty string equality, if you will, but it checks out.

Induction step: Our algorithm compares p_j with $p_{f(j-1)+1}$ in L6 using the assignment from L5. This splits our analysis into two cases:

- **Case 1:** $p_j = p_{f(j-1)+1}$

  The L6 and L9 blocks do not get executed, and we continue to L12. By the induction hypothesis, $f(j-1) = i$ is the largest i such that $p_1 \dots p_i = p_{j-i+1} \dots p_j$. Since the comparison from L6 ensures that the next character in p matches the character after the $j-1$th prefix, L12 assigns $f(j)$ correctly.

- **Case 2:** $p_j \neq p_{f(j-1)+1}$

  The L6 loop finds the largest i such that

  $$\begin{cases} p_1 \dots p_i &= p_{j-i} \dots p_{j-1} & i\text{th prefix matches } (j-i)\text{th suffix} \\ p_{i+1} &= p_j & \text{next characters match} \end{cases}$$

  If such an i exists, the L6 loop will consider all possible i which satisfies the above equalities. Note that $f(k)$ is always less than k, so L6 will check these i from greatest to least. Based on this ordering, it finds the largest such i. So L12 assigns $f(j)$ correctly.

  If no such i exists, i eventually reaches 0 from L7. So L10 assigns $f(j)$ correctly, as the next characters do not match.

In all cases, our algorithm assigns failure function values correctly. This concludes our proof!

2.2.6 Proof of runtime for the failure function

What about the runtime of this algorithm? Let's take a look at a brief informal proof using *amortized analysis*.

Informal Proof. It looks like we have nested loops here. Although it's apparent that the L4 loop (pun not intended) runs in time linear in the length of p, the L6 loop is less clear.

Let's examine i as a limited resource. With careful analysis, one can discover that i is incremented by the combination of L4, L5, and L12. $f(j)$ is set to $i+1$ in L12, and in the next iteration of the L4 loop where j has been incremented, i is set to $f(j-1)$ in L5. Thus, this combination results in a single increment to i.

One can also find that i only decreases in L7. In fact, we can say that i is decreased every time the L6 block is executed. The converse is also true: that L6 is executed every time i is decreased. So the total cost of L6 (over the entire algorithm, not within the L4 loop) is proportional to the number of times i decreases.

The L4 loop can only run $l-1$ times, so L5 and L12 can only be executed $l-1$ times each. Since theses lines control the increase of i, the cost of the L6 loop must be proportional to the runtime of the L4 loop. Note that we've separated the total cost of L6 from the L4 loop, so these time complexities are now additive!

$$O(l) \text{ (L4 loop)} + O(l) \text{ (L6 loop)} = O(l)$$

So, our algorithm's overall time complexity is $O(l)$, or linear in the length of the pattern!

2.2.7 Constructing a DFA for matching a pattern in a text

Recall that we denote our text by t and our pattern by p. We wish to construct a DFA that recognizes the language $\Sigma^* p$, and which makes exactly one state transition per input symbol.

ALGORITHM FOR DFA WITH LANGUAGE $\Sigma^* p$

Input: A pattern $p = p_1 \dots p_l$ over Σ.
Output: A DFA M such that $L(M) = \Sigma^* p$.

We proceed in the following fashion:

1. Use the FAILURE FUNCTION algorithm for p.
2. Let $M = (S, \Sigma, \delta, 0, \{l\})$ where $l = |p|$. Let $S = \{0, 1, \dots, l\}$.
3. **Constructing** δ:
   (a) For $j = 1$ to l, we have $\delta(j-1, p_j) = j$.
   (b) For each $a \in \Sigma$, $a \neq p_1$, we have $\delta(0, a) = 0$.

(c) For $j = 1$ to l, for each $a \in \Sigma$ and $a \neq p_{j+1}$, we have $\delta(j, a) = \delta(f(j), a)$.

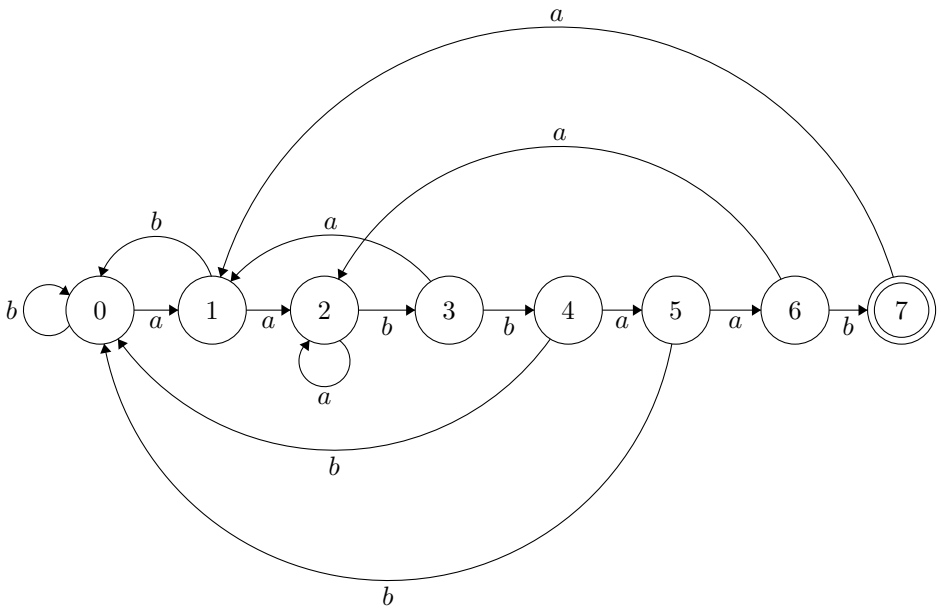
Question: Why do we want to think of the pattern p being at the end? Or, why don't we want a DFA for $p\Sigma^*$?

Sorin: Because we want to find the first exact occurrence!

Example. Let's return to our earlier example for the failure function. We're given a string $p = aabbaab$. We computed the failure function last time, which is given by:

i	1	2	3	4	5	6	7
$f(i)$	0	1	0	0	1	2	3

By following the algorithm given above, we obtain the following DFA:



2.3 The Burrows-Wheeler Transform

The *Burrows-Wheeler Transform* is a reversible string compression algorithm. Given an input string, it can rearrange its characters into a sequence which often groups identical characters together, allowing for the string to be represented in condensed notation. While it doesn't always achieve this perfectly, the benefit is that it incurs no additional space costs! (Well, one character, but that's negligible.) Let's take a look at the steps and an example.

The Algorithm:

Given a string $X = x_1 x_2 \dots x_n$,

1. Add an end-of-string character to the string. We'll often use \$.
2. Generate all *rotations* of X\$.

3. Sort the rotations in lexicographical order. Typically, the end-of-string character comes first.
4. From your sorted list of rotations $R_1, R_2, \ldots R_{n+1}$, output the string formed by all r_{n+1} characters (ending characters) in order.

Here's an example. Suppose we have the string *aardvark*.

Above: An aardvark.

1. Add \$ to the end of the string: *aardvark*\$
2. Obtain all rotations: *aardvark*\$ → *ardvark*\$*a* → *rdvark*\$*aa* → *dvark*\$*aar* → *vark*\$*aard* → *ark*\$*aardv* → *rk*\$*aardva* → *k*\$*aardvar* → \$*aarkvark*.
3. Sort these rotations:

$$\begin{array}{c} \$aardvark \\ aardvark\$ \\ ardvark\$a \\ ark\$aardv \\ dvark\$aar \\ k\$aardvar \\ rdvark\$aa \\ rk\$aardva \\ vark\$aard \end{array}$$

4. Obtain the output string from the last column: *k*\$*avrraad*

And so we see that $BWT(aardvark\$)$ is *k*\$*avrraad*!

You could imagine that if we wanted to store a string in the most efficient way possible, one relatively simple approach is to represent any repeat strings as the repeat character, followed by the number of its occurrences.

So our output string might be represented as: *k*\$*avr*2*a*2*d*. Now, this might not appear to improve anything. After all, our condensed output is 9 characters long... which was also the length of our input string... which was also the length of the condensed version of our input string: *a*2*rdvark*\$. But you can already begin to see some beneficial effects! Notice how the Burrows-Wheeler Transform grouped the *r*'s together, when they were separated to begin with. On much longer strings, the space benefits will be much more apparent when you can begin replacing repeats of length 3, 4, 5, and so on with 2-character condensed representations!

Ok, it's great that we can save some space when storing the characters in these strings. But what we really wanted to was to store *aardvark*\$, not *k*\$*avrraad*. How do we get our original string back?

2.3.1 The Inverse Transform

Thankfully, the Burrows-Wheeler transform is reversible! This means that there exists an inverse algorithm that takes any BWT output and recovers its original input! Let's introduce the steps and continue our *aardvark* example to see it in action.

The Algorithm:

Given a string $Y = BWT(X) = y_1 y_2 \dots y_n$,

1. Initialize a matrix where each row contains a character in Y
2. Sort the rows in the matrix in lexicographical order.
3. Append the contents of each row to the characters of Y. These strings become the new contents of the matrix. (Alternatively, you can think of this as appending the characters in Y to the front of each row)
4. Repeat 2 and 3 until each row is n characters long.
5. Sort the rows in lexicographical order.
6. Output the first row. Since our end-of-string character is lexicographically first, this will be our original input with the end-of-string character in front! (Recall that this is the last rotation we generated in the BWT algorithm.)

Let's try this on *k*\$*avrraad*.

First, we initialize our matrix:

k
\$
a
v
r
r
a
a
d

Then, we sort:

\$
a
a
a
d
k
r
r
v

Add in our input characters to the front of the strings:

k	$
$	a
a	a
v	a
r	d
r	k
a	r
a	r
d	v

Sort:

$	a
a	a
a	r
a	r
d	v
k	$
r	d
r	k
v	a

Add our input in:

k	$	a
$	a	a
a	a	r
v	a	r
r	d	v
r	k	$
a	r	d
a	r	k
d	v	a

Sort:

$	a	a
a	a	r
a	r	d
a	r	k
d	v	a
k	$	a
r	d	v
r	k	$
v	a	r

Add our input in:

k	$	a	a
$	a	a	r
a	a	r	d
v	a	r	k
r	d	v	a
r	k	$	a
a	r	d	v
a	r	k	$
d	v	a	r

$\vdots$

Several sorts and additions later, we have:

k	$	a	a	r	d	v	a	r
$	a	a	r	d	v	a	r	k
a	a	r	d	v	a	r	k	$
v	a	r	k	$	a	a	r	d
r	d	v	a	r	k	$	a	a
r	k	$	a	a	r	d	v	a
a	r	d	v	a	r	k	$	a
a	r	k	$	a	a	r	d	v
d	v	a	r	k	$	a	a	r

Finally, we sort again:

$	**a**	**a**	**r**	**d**	**v**	**a**	**r**	**k**
a	a	r	d	v	a	r	k	$
a	r	d	v	a	r	k	$	a
a	r	k	$	a	a	r	d	v
d	v	a	r	k	$	a	a	r
k	$	a	a	r	d	v	a	r
r	d	v	a	r	k	$	a	a
r	k	$	a	a	r	d	v	a
v	a	r	k	$	a	a	r	d

Taking the first row, we remove the end-of-string character, and voila! We have recovered *aardvark*, which was our original input to the Burrows-Wheeler Transform.

2.4 Suffix Trees

Suffix trees allow us to query strings in optimal time. In other words, using suffix trees, we can solve problems like whether a potential substring is contained within a second string in optimal time.

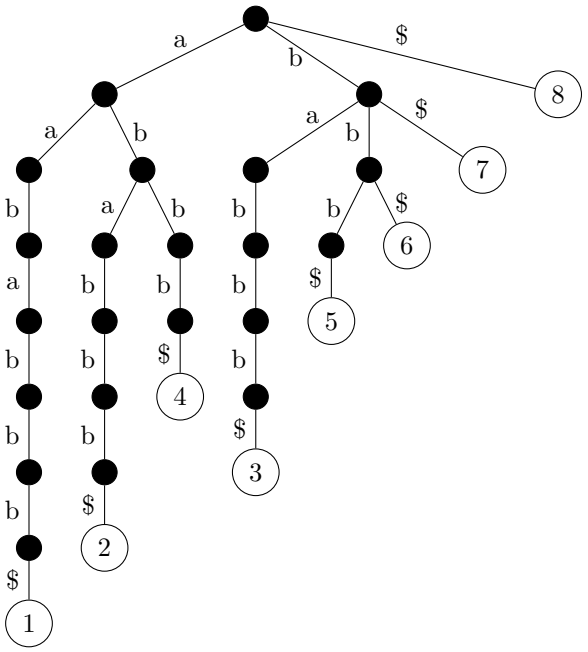
Suppose we have a string:

$$s = aababbb$$

Basic stringology: if a string has length n, then it will have n suffixes and n prefixes, and thus $\mathcal{O}(n^2)$ substrings (equivalent to picking two indices).

We can construct the suffix tree T_s for the string s using the following algorithm:

1. Add a \$ symbol to the end of s to denote the end of the string. The \$ symbol will be useful for indicating the ends of suffixes in the suffix tree.
2. Add a root node to T_s. Take the first suffix of s and add it to T_s as a path from the root node to a leaf with label 1.
3. Take the next suffix of s and add it to T_s as a path from the root node to a leaf with label 2, reusing nodes and edges already in the tree whenever possible.
4. Continue to add suffixes to the tree until all suffixes have been added, labeling each leaf with the starting position of the suffix it represents in the string.



Suffix trees are deterministic finite automata that accept the language consisting of all substrings of a string (or a set of strings). Think of the nodes of the tree as states, and think of the edges of the tree as directed edges with symbols from our string (alphabet) on them. Therefore, suffix trees allow us to check if a pattern p is present in a given string in time $\mathcal{O}(|p|)$. We'll denote the suffix tree associated to a string s by T_s.

How is this a DFA? Start parsing the input pattern and begin at the root node. At each letter, check if we can advance to a subsequent (lower) node in the suffix tree. If we can advance, then we continue. If we can't advance, we instead transition to an absorbing failure state. If we finish reading the input pattern and we are still in a node in the tree, then we accept the pattern, indicating that the pattern is in the string. If we finish reading the input pattern and we are in the failure state, then the pattern is not in the string.

We can make several modifications to this representation of a suffix tree to further optimize this data structure.

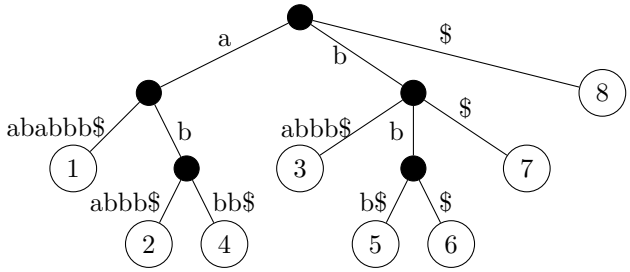
First, we notice that if we were to move down the leftmost branch in this tree, there's only one sequence of letters that the suffix tree accepts. So, instead of having so many individual edges and internal nodes, we can concatenate them all. In fact, we can do this for any subtree with one branch! Doing so creates a **compact suffix tree**. (Note: We sometimes refer to suffix trees that have not reduced their internal nodes in this way as **expanded suffix trees**.)

Once we have these multi-character edges, we can further optimize the way we store the suffix tree. If we have the string used to construct the tree in memory, we don't need to store characters in our tree at all. Instead, we could replace edge labels with position indices. For example, we can replace our long left-most edge with (2, 7). This modification allows us to retrieve the exact substring we need to make comparisons without having to store the full 6-character subsequence in the tree. By modifying compact suffix trees in this way, we can create **position suffix trees** for strings of length n that can be stored in $O(n)$ space. In contrast, both expanded and compact suffix trees require $O(n^2)$ space to store the entire tree.

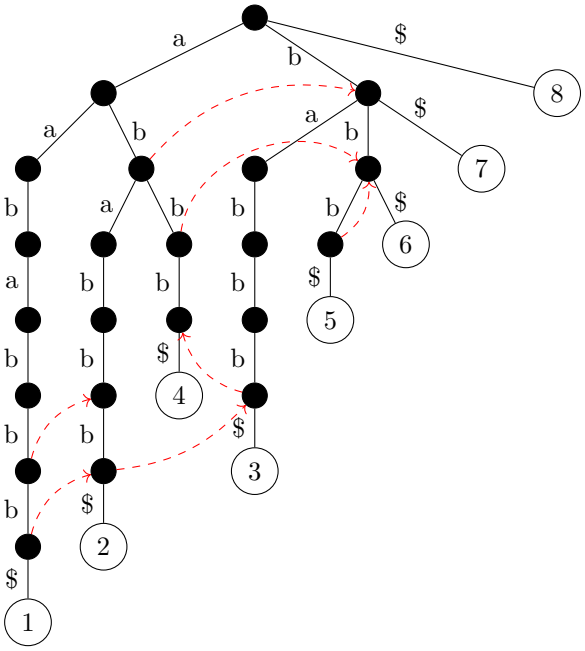
An additional improvement we can make to suffix trees is to add *suffix links* to a tree. A **suffix link** is a special edge which connects an internal node v to another node w, where v represents the string p and w represents the string $p_{2:end}$.

Put another way, if v represents the string $p = ab$, where a is a character and b is a string, then w represents b. A *suffix link* connects v to w.

Here's the compact suffix tree for the example above:



And here's an example of a few (not all!) of the suffix links in our original expanded suffix tree:



One of these suffix links can even be included in our compact suffix tree. See if you can figure out which one it is!

We can use suffix links to improve the runtime of our suffix tree construction algorithm from $O(n^2)$ to $O(n)$. If you are interested in learning about the algorithm for constructing suffix trees in linear time, take CS182!

In summary, a suffix tree T_s is a data structure that takes as input a string s of length n. It stores all the suffixes of s in an efficient way such that:

1. T_s stores the starting position of each suffix of s
2. T_s stores each substring of s
3. Each suffix of s can be thought of as a path label from the root node to some leaf, and vice-versa.
4. Every leaf is labelled with the starting position of the suffix for which it is the endpoint of the path label from the root.
5. Branching edges have different letters labelling them.
6. For any two suffixes S_i and S_j of s, we can find their common prefix $w_{i,j}$ using the suffix tree.