“Forward Values” vs “Reverse Values”

F(v) is the score of the best path from Q to v and R(v) is the score of the best path from v to F. Then F(v) + R(v) is the score of the best path from Q to F through v. The alignment can be delivered in linear space if we are only interested in one optimal alignment at a cost of O(2MN) as seen in the previous lecture. However, if we are interested in the set of alignments with a score ≤ t, we must think beyond this. Consider G_t = the subgraph of G induced by V_t where

\[ V_t = \{ v : F(v) + R(v) \leq t \} \]

Then we need F and R scores at the same time. We can compute F “with the grain” as we progress through the DP matrix, but the R scores are more of a problem:

- F for row i = f(row_{i-1})
- R for row i = g(row_{i+1})

We need to deliver values for both directions i. To do this, consider a k-ary tree that recursively divides M. The R rows of the matrix are cached for the current segment of the matrix at each level and we progress forward calculating F. As soon as we leave a segment at any level we must recompute and cache the rows for that segment. This ends up recalculating all the rows at each level, for a time complexity of O(kMN). The space requirement of each level is O(M^{1/k}N) and there are k levels, so the total space required is O(kM^{1/k}N). At the limit point, when k = \log_2 M this becomes O(Mn \log_2 M) time and O(N \log_2 M) space.

A few points: think about delivering an alignment. We want to deliver a particular path p. For the linear space case in which we halve M each time, we have ~2MN time. Now, if we instead use this “paneling” concept, ie use a k-ary tree where k = 2, then when we compute each panel… it doesn’t change. I thought we could improve on the asymptotic computation boundary of 2, but this doesn’t. An interesting problem: can we get that asymptote to be less than 2?
For $G_{ij}$, you can compute the whole row on the first level and in lower levels compute only within the leftmost and right-most paths to save space and time.

Approximate Pattern Matching

for normal sequences, a comparison $\mathbb{D}(A, B) = \text{optimum alignment of } A \text{ and } B$. We can talk about the same thing for patterns:

$$\mathbb{D}(A, P) = \text{opt } \{ \mathbb{D}(A, B) : B \in \text{Lang}(P) \}$$

In general, we are speaking about global alignments, but the other variations are also available in all of these discussions.

Problem: find the word $w$ in $P$ such that the alignment of $A$ and $w$ is optimal. There are three levels to this problem: we must find the cost, the alignment and the word. This is contrast to the standard alignment problem which is only concerned with cost and alignment.

There is a database variant: given $A$ and $P$ and a threshold $t$, find all substrings $A_{i..j}$ such that $\mathbb{D}(A[i..j], P) \leq t$. This raises the issue of $P$ being too short to generate a significant match, so perhaps we should think about $\mathbb{D}(A[i..j], P)$

$$|i-j|$$

We can also align two patterns: $\mathbb{D}(P, G) = \text{opt } \{ \mathbb{D}(A, B) : A \in \text{Lang}(P) \text{ and } B \in \text{Lang}(G) \}$

If the spaces of these regular expressions are far apart you may want to try to compare the average of the two spaces:

$$\frac{\mathbb{D}(A, G)}{|L(P)|}$$

This could be a project.

Looking at pattern matching for regular expressions and context-free languages, consider $F_A \quad F_R$ and find the best path from $\square$ to $\square$. This graph is no longer a DAG, since $F_R$ is arbitrary. We could invoke Dijkstra and incur log overhead extra to compute the path, but there is another way.
We can make an inductive construction for R:

This construct has the following properties:
- There is only one start and final state
- The start state has in-degree 0
- The final state has out-degree 0
- The out-degree and in-degree of any node is ≤ 2
- |v| ≤ |R|
- It is linear in the number of edges
- (V,E) is reducible

This construct works well for data structures as only 2 ptrs are needed at each node due to the edge degree limit.

Reducibility is the property that all back edges in the graph are nested: the loop interconnectedness parameter is 1. This means that every cycle-free subpath contains at most 1 back-edge. If you ignore the back edge in R*, then the graph is acyclic.

These considerations are related to program flow theory. People making compilers looked at this concept to error-correct code. The goal was to produce the closest program to the input that is syntactically correct.

Remember the alignment cross product:

\( A_1 = < [], V_1, E_1, [] : E_i -> [], [], [] > \)

\( A_1 \quad A_2 = < [], V_1 \times V_2, E_1 \times E_2, [], [] > \)

Where \([a \ b] \) and \(E_1 \times E_2\) are defined as

\[(s,t) [a \ b] (u,v) \text{ iff } \{ s \overset{a}{\rightarrow} u \text{ and } t \overset{b}{\rightarrow} v \}
\text{ or } s \overset{a}{\rightarrow} u \text{ and } t = v \text{ and } b = \text{ '-' } \]
\text{ or } s = u \text{ and } a = \text{ '-' } \text{ and } t \overset{b}{\rightarrow} v \]
Now, we can do the same thing, but with one machine being a regular expression. Let the
automaton of the reg exp be R. This creates a cross-automaton graph that has nested
back edges and therefore is no longer cycle-free, since R can have cycles. The number of
vertexes and edges is $O(|A| \times |R|)$. We could appeal to Dijkstra, but there is another way.
First note that any optimal path has no cycles because they simply add cost, so we can
excise them. There are back edges, however. We find the path in 2 passes.

Possible regular expression

Note that any path with a
cycle can be simply converted
into a cycle-less path by
excising the cycle.

However, the path may have
back edges.

First Pass:

$C(i,s) = \text{score of best path from } A[i..j] \text{ with word } s \text{ }$ 

$\text{Lang}(R)$

$= \min \{ C(i-1, t) + d(a,b) \}$ 
$C(i-1, s) + d(a,\text{'-'})$ 
$C(i, s) + d(\text{'-'}, b)$ 

Second Pass:

$C(i,s) = \min \{ C(i,s) \}$ 
$C(i,t) + d(\text{'-'}, t)$ 
$C(i,u) + u \text{ }$ s transition cost 

where $u$ is a node ahead of $s$ that has a back edge coming into $s$. Then the algorithm is:

for $i = 1$ to $M$

for $s$ in a topological order of the acyclic subgraph of $R$ (i.e. drop back edges)

calculate $C(i,s)$ as above for first pass

for $s$ in same topo order
calculate $C(i,s)$ as for second pass
Claim: this gives us the best alignment if $\delta \geq 0$. Because the graph is reducible, any row
has at most one back edge on the path of interest, so we include this in the second pass of
that row. This requires $O(|R|)$, so there is no need to go to Dijkstra. To actually deliver
an alignment, we can go cache as before and pay $O(M |R| \log M)$.

Context-free languages:

Define a CFL as $< \Sigma, V, P, S >$

where $\Sigma$ is the alphabet of terminals, $V$ is the set of nonterminals, $P$ is the set of
productions of nonterminals, and $S$ is the starting nonterminal. $P \subseteq \{ V \times (V + \Sigma) \}$

e.g. $\Sigma = \{a,b\}$
$V = \{T\}$
$P = \{ T \rightarrow a$
         $T \rightarrow b$
         $T \rightarrow aTa$
         $T \rightarrow bSb$
$S = T$

then e.g. the following operations are allowed:

S $\rightarrow$ aSa
aSa $\rightarrow$ abSba
abSba $\rightarrow$ abba

Note that we are assuming that the CFL is in Chomsky Normal Form (CNF), since in
theory any CFL can be put in CNF. The problem: given CFL $G$ and word $w$, is $w \in$
$\text{Lang}(G)$? Define a function $M$:

$$M(V, i, j) = w[i..j] \in \text{Lang}_G(V)$$

The various cases in which $V$ can yield $w[i..j]$ are:

\[
\begin{array}{c|c}
V & \text{or} & V \\
/ & \backslash & 1 \\
A & B & a \\
/ & \backslash & / \\
i & k & k+1 & j & i = j, w[i..j] = a \\
\end{array}
\]

\[
\begin{aligned}
i+1 & = \text{OR} \quad \text{OR} \quad (M(A, i, k) \text{ and } M(B, k+1, j)) \quad \text{OR} \quad (i=j \text{ and } V[a = w[i]])
\end{aligned}
\]

Then $w \in \text{Lang}(G)$ iff $M(S, 1, |w|)$ is true.