Advanced Data Structures for String Matching: Suffix Arrays and OASIS Algorithm

Nick Kiourtis

May 17, 2007

Suffix Array: Definitions

- Large String $A = a_0 a_1 \dots a_{N-1}$ of length N, Alphabet Σ, W query string of size $|P|$.
- $A_i = a_i a_{i+1} \dots a_{N-1}$ suffix of A starting at i.
- $Pos[k]$ is the start position of the kth smallest suffix in set $\{A_0, A_1, \dots, A_{N-1}\}$ (lexicographic ordering)
- Pos is the suffix array of A
- $A_{Pos[0]} < A_{Pos[1]} < \cdots < A_{Pos[N-1]}$
- If $|u| > p$ then $u^p = u_0 u_1 \dots u_{p-1}$, else $u^p = u$
- $u <_p v$ iff $u^p < v^p$ (similarly $\neq_p$, $\leq_p$, etc.)
- Because Pos is sorted lexicographically, for any choice of p, Pos is also ordered according to $\leq_p$

Suffix Array: Example

- $A = abracadabra$
- Suffix, Index

a	b	r	a	c	a	d	a	b	r	a	0
	b	r	a	c	a	d	a	b	r	a	1
		r	a	c	a	d	a	b	r	a	2
			a	c	a	d	a	b	r	a	3
				c	a	d	a	b	r	a	4
					a	d	a	b	r	a	5
						d	a	b	r	a	6
							a	b	r	a	7
								b	r	a	8
									r	a	9
										a	10

Suffix Array: Example (2)

- $A = abracadabra$
- Sorted Suffix, Index

a	10
a b r a	7
a b r a c a d a b r a	0
a c a d a b r a	3
a d a b r a	5
b r a	8
b r a c a d a b r a	1
c a d a b r a	4
d a b r a	6
r a	9
r a c a d a b r a	2

- Suffix Array $[10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2]$

Searching in Suffix Arrays

- $W = w_0 w_1 \dots w_{P-1}$, $P \leq N$.
- $L_W = \min(k : W \leq_P A_{Pos[k]}$ or $k = N)$
  $R_W = \max(k : A_{Pos[k]} \leq_P W$ or $k = -1)$
- W matches $a_i a_{i+1} \cdots A_{i+P-1}$ iff $i = Pos[k]$, $k \in [L_W, R_W]$
- Find L_W, R_W with binary search, using $O(\log(N))$ comparisons of strings of size at most P
- Total time $O(P \log(N))$

Algorithm 1

```
if W ≤_P A_Pos[0] then
        L_W ← 0
else if W >_P A_Pos[N−1] then
        L_W ← N
else
  {  (L, R) ← (0, N−1)
     while R − L > 1 do
       {  M ← (L + R)/2
          if W ≤_P A_Pos[M] then
                R ← M
          else
                L ← M
       }
     L_W ← R
  }
```

Optimizing Search Time

- $lcp(v, w)$ length of longest common prefix of v and w. We can obtain it when testing $v < w$
- $l = lcp(A_{Pos[L]}, W)$, $r = lcp(W, A_{Pos[R]})$. Initially, $l = lcp(A_{Pos[0]}, W)$, $r = lcp(W, A_{Pos[N-1]})$, and update l or r in every step
- This way we can save $h = \min(l, r)$ single symbol comparisons, when comparing $A_{Pos[M]}$ to W, because $A_{Pos[L]} =_l W =_r A_{Pos[R]}$, that is $A_{Pos[L]} =_h W$, $\forall k \in [L, R]$
- Worst case still $O(P \log(N))$ (e.g. search $ac_{N-2}b$ for $c_{P-1}b$)

Optimizing Search Time

- ALL possible triples (L, M, R) inside the loop are exactly $N-2$, and each has a unique midpoint $M \in [1, N-2]$.
  $0 \leq L < M < R \leq N-1$
- Let (L_M, M, R_M) one of them. Define
  $Llcp[M] = lcp(A_{Pos[L_M]}, A_{Pos[M]})$ and
  $Rlcp[M] = lcp(A_{Pos[M]}, A_{Pos[R_M]})$
- Consider an iteration of search loop for (L, M, R). Let $h = \max(l, r)$, Δh be the change in h after iteration
- Assume $h = \max(l, r) = l$, so we must consider 3 cases:
  1. $Llcp[M] = l$
  2. $Llcp[M] < l$
  3. $Llcp[M] > l$

  (if $\max(l, r) = r$ we would consider cases for $Rlcp[M]$)

Optimizing Search Time

For each case, determine whether L_W is in right half or left half of (L, M, R) and update value of either l or r

1. $Llcp[M] = l \;\; (lcp(A_{Pos[L]}, A_{Pos[M]}) = lcp(A_{Pos[L]}, W))$

   The first l symbols of $Pos[M]$ and W are equal. Compare $l+1, \ldots, l+j$ symbols until $W \neq_{l+j} Pos[M]$. If $W[l+j] < M[l+j]$ then $L_M \in (L, M)$ and $r = l + j$, else $L_M \in (M, R)$ and $l = l + j$. Since $l = h$ at the beginning of the loop, this step takes $\Delta h + 1$ single step comparisons

2. $Llcp[M] < l \;\; (lcp(A_{Pos[L]}, A_{Pos[M]}) < lcp(A_{Pos[L]}, W))$

   W matched l symbols of L and $< l$ symbols of M, thus $L_M \in (L, M)$ and $r = Llcp[M]$

3. $Llcp[M] > l \;\; (lcp(A_{Pos[L]}, A_{Pos[M]}) > lcp(A_{Pos[L]}, W))$

   $A_{Pos[M]} =_{l+1} A_{Pos[L]} \neq_{l+1} W$ and $A_{Pos[M]} =_l W =_l A_{Pos[L]}$, so $L_W \in (M, R)$ and l is unchanged

Optimizing Search Time

- Thus, the use of arrays $Llcp$ and $Rlcp$ reduces the number of single symbol comparisons to at most $\Delta h + 1$ for each iteration.
- $\sum \Delta h \leq P$
- Total number of single symbol comparisons is at most $P + \lceil \log_2(N-1) \rceil$
- Worst case running time $O(P + \log(N))$

Algorithm 2

$l \leftarrow lcp(A_{Pos[0]}, W)$
$r \leftarrow lcp(A_{Pos[N-1]}, W)$
if $l = P$ **or** $w_l \leq a_{Pos[0]+l}$ **then**
    $L_W \leftarrow 0$
else if $r < P$ **or** $w_r \leq a_{Pos[N-1]+r}$ **then**
    $L_W \leftarrow N$
else
  { $(L, R) \leftarrow (0, N-1)$
    while $R - L > 1$ **do**
      { $M \leftarrow (L+R)/2$
        if $l \geq r$ **then**
            if $Lcp[M] \geq l$ **then**
                $m \leftarrow l + lcp(A_{Pos[M]+l}, W_l)$
            else
                $m \leftarrow Lcp[M]$
        else
            if $Rcp[M] \geq r$ **then**
                $m \leftarrow r + lcp(A_{Pos[M]+r}, W_r)$
            else
                $m \leftarrow Rcp[M]$
        if $m = P$ **or** $w_m \leq a_{Pos[M]+m}$ **then**
            $(R, r) \leftarrow (M, m)$
        else
            $(L, l) \leftarrow (M, m)$
      }
    $L_W \leftarrow R$
  }

Sorting

- Sorting Pos array is done in $\lceil \log_2(N+1) \rceil$ stages
- In H^{th} stage, suffixes are sorted according to $\leq_H$ order. In next stage, suffixes are sorted according to $\leq_{2H}$ order
- First sort is according to the first symbol of each suffix, and the result is stored in Pos array and in BH array which demarcates the partitioning of suffixes into m_1 buckets (each bucket holds suffixes with the same first symbol)
- Let A_i, A_j be suffixes that belong in same bucket after H^{th} step, $A_i =_h A_j$. We want to compare them according to the next H symbols
- But the next H symbols of A_i are the first H symbols of A_{i+H} and we know the relative order according to the $\leq_H$ relation

Sorting

- Start at the first bucket (contains the smallest suffixes according to $\leq_H$). Let A_i be the first suffix ($Pos[0] = i$).
- Consider A_{i-H} (if $i - H < 0$, ignore A_i and take the suffix of $Pos[1]$, etc.)
- Since A_i starts with smallest H–symbol string, A_{i-H} should be the first in its $2H$–bucket, so move A_i to beginning of its bucket and mark this fact.
- Keep track of the number of suffixes that have been moved from bucket
- Basically the algorithm scans the suffixes as they appear in $\leq_H$ order and for each A_i, if A_{i-H} exists, it moves it to the next available space in its H–bucket

Sorting Algorithm

- Maintain 3 integer arrays $Pos, Prm, Count$ and 2 boolean arrays $BH, B2H$ of length $n+1$
- After stage H, $Pos[i]$ contains start position of i^{th} $\leq_H$–smallest suffix, $Pos[Prm[i]] = i$. $BH[i] = 1$ iff $Pos[i]$ contains the leftmost suffix of an H–bucket ($A_{Pos[i]} \neq_H A_{Pos[i-1]}$). Initially need time $O(N)$
- In stage $2H$: Reset $Prm[i]$ to point to the leftmost cell of the H–bucket containing the i^{th} suffix rather than to its precise place in the bucket, and set $Count[i] = 0$ for all i
- Now we scan the Pos array in increasing order, one H–bucket at a time. If l, r are the left and right boundaries, for i from l to d do:
- Define $T_i = Pos[i] - H$ (if $T_i \leq 0$ go to next i)
- Increment $Count[Prm[T_i]]$ (to keep track of how many suffixes have been moved in the H–bucket that countains the suffix that starts at position T_i)

Sorting Algorithm (2)

- Set $Prm[T_i] = Prm[T_i] + Count[Prm[i]] - 1$ (to make this the next suffix in the bucket, without changing the Pos array)
- Set $B2H[Prm[T_i]] = 1$ (to mark the suffix as moved)
- Before moving to the next bucket, find all the moved suffixes and reset their $B2H$ fields such that only the leftmost one of each $2H$-bucket is set to 1. This way, the $B2H$ fields mark correctly the beginning of the $2H$ buckets
- In the end update Pos as the inverse of Prm and set BH to $B2H$
- All these steps can be done in linear time $O(N)$
- Since there are $\lceil \log_2(N+1) \rceil$ stages, sorting requires $O(N \log(N))$ time in the worst case

Finding Longest Common Prefixes

- We compute the longest common prefixes between the suffixes starting at each midpoint M and its left and right boundaries L_M, R_M during the sorting
- Key idea for adjacent buckets: Assume that after step H we know the $lcps$ between suffixes in adjacent buckets (after step 1, $lcps = 0$)
- The $lcps$ between suffixes in newly adjacent buckets must be at least H and at most $2H - 1$ (since at stage $2H$, buckets are partitioned according to $2H$ symbols)
- If A_p, A_q are in the same H–bucket but in distinct $2H$–buckets, then $lcp(A_p, A_q) = H + lcp(A_{p+H}, A_{q+H})$, and $lcp(A_{p+H}, A_{q+H}) < H$
- However, if $A_{Pos[i]}$ and $A_{Pos[j]}$, $i < j$, have a lcp less than H and Pos is in H–order, then

$$lcp(A_{Pos[i]}, A_{Pos[j]}) = \min_{k \in [i, j-1]} (lcp(A_{Pos[k]}, A_{Pos[k+1]}))$$

Interval Tree

- $O(N)$ space height balanced tree structure that records the minimum pairwise lcp over a collection of intervals in the suffix array. Helps compute the lcp between any two suffixes in $O(\log(N))$
- Define $height(i) = lcp(A_{Pos[i-1]}, A_{Pos[i]})$, $1 \leq i \leq N-1$, Pos final sorted order. These values are computed in an array $Hgt[i]$ inductively with the sort
- $Hgt[i]$ achieves its correct value at stage H iff $height(i) < H$, otherwise it is undefined and $Hgt[i] = N+1$
- If $height(i) < H$, then $A_{Pos[i-1]}$, $A_{Pos[i]}$ must be in different buckets (or else they would have the same H preffix and thus $height(i) \geq H$)
- Lemma: If $H \leq height(i) < 2H$ then $height(i) = H + \min(Hgt^H[k] : k \in [\min(a,b)+1, \max(a,b)])$, where $a = Prm^{2H}[Pos^{2H}[i-1]+H]$, $b = Prm^{2H}[Pos^{2H}[i]+H]$

Interval Tree

- Initially, $Hgt[i] = 0$ if $A_{Pos^1}[i-1] \neq A_{Pos^1}[i]$, and $n+1$ otherwise
- At the end of stage $2H > 1$, we have $Pos^{2H}, Prm^{2H}, BH^{2H}$. From previous lemma, we can compute Hgt^{2H} from Hgt^H:

  for $i \in [1, N-1]$ such that $BH[i]$ and $Hgt[i] > N$ do
  $\{$
  $a \leftarrow Prm[Pos[i-1] + H]$
  $b \leftarrow Prm[Pos[i] + H]$
  $Set(i, H + Min\_Height(\min(a,b)+1, \max(a,b)))$
  $\}$
- Consider a balanced and full binary tree, $N-1$ leaves that correspond to $Hgt[i]$ values (left to right). Tree has height $O(\log(N))$ and $N-2$ interior vertices.
- Tree is "current" if for every interior vertex v, $Hgt[v] = \min(Hgt[left(v)], Hgt[right(v)])$

Interval Tree

- $Min\_Height(i, j)$ computes $\min(Hgt[k] : k \in [i, j])$ in $O(\log(N))$ time as follows:
- Let $nca(i, j)$ be the nearest common ancestor of leaves i, j (can be found in $O(\log(N))$ time). If P is the set of vertices on the path from i to $anc(i, j)$ without $nca(i, j)$, and Q the similar path from j, then $Min\_Height(i, j)$ is the minimum of
  1. $Hgt[i]$
  2. $Hgt[w]$, such that $right(v) = w$ and $w \notin P$ for some $v \in P$
  3. $Hgt[w]$, such that $left(v) = w$ and $w \notin Q$ for some $v \in Q$
  4. $Hgt[j]$
- All those vertices are $O(\log(N))$ and their min can be computed in $O(\log(N))$ time
- $Set(i, h)$ sets $Hgt[i] = h$ and then makes T current again by updating Hgt values of the interior nodes on the path from i to the root. Again this takes $O(\log(N))$ time

Interval Tree

- Overall time in stage H is $O(N + \log(N) \cdot Set_H)$, where Set_H is the number of indices i for which $height(i) \in [H, 2H-1]$
- $\sum Set_H = N$, so total time in all stages is at most $O(N \log(N))$
- Interval tree helps us compute $Llcp[M] = lcp(A_{Pos[L_M]}, A_{Pos[M]})$ and $Rlcp[M] = lcp(A_{Pos[M]}, A_{Pos[R_M]})$
- One way: The root of the tree is labeled $(0, N-1)$ and the remaining vertices are labeled (L_M, M) or (M, R_M)
- Another way: The trees interior nodes have labels (L_M, R_M), and in particular the leaves have labels $(i-1, i)$ for $i \in [1, N-1]$ (left to right order). For each interior vertex $left(L_M, R_M) = (L_M, M)$ and $right(L_M, R_M) = (M, R_M)$. Since the tree is full and balanced and the leaf $(i-1, i)$ holds the value of $Hgt[i]$, we have that $Hgt[(L, R)] = \min(height(k) : k \in [L+1, R]) = lcp(A_{Pos[L]}, A_{Pos[M]})$.
  Thus $Llcp[M] = Hgt[(L_M, M)]$ and $Rlcp[M] = Hgt[(M, R_M)]$

OASIS Overview

- OASIS is an algorithm that computes the local alignment between two sequences with the maximum possible score
- Local alignment: given two sequences of symbols $Q = q_1 q_2 \dots q_m$ and $T = t_1 t_2 \dots t_n$, a local alignment is some way of lining up any two subsequences of Q andn T.
- There are three types of alignment operations:
  1. Replacement, with either the same symbol or another symbol
  2. Deletion allows to skip a symbol in the target
  3. Insertion allows to skip a symbol in the query
- Alginments are given scores based on the sum of the scores of each operation involved in the alignment
- Every operation is generalized to a replacement $\alpha \rightarrow \beta$, where insertions are represented as $- \rightarrow \beta$ and deletions as $\alpha \rightarrow -$
- Scores are stored at a substitution matrix S, where $S_{\alpha,\beta}$ is the score of $\alpha \rightarrow \beta$

Smith-Watermann (S-W)Algorithm

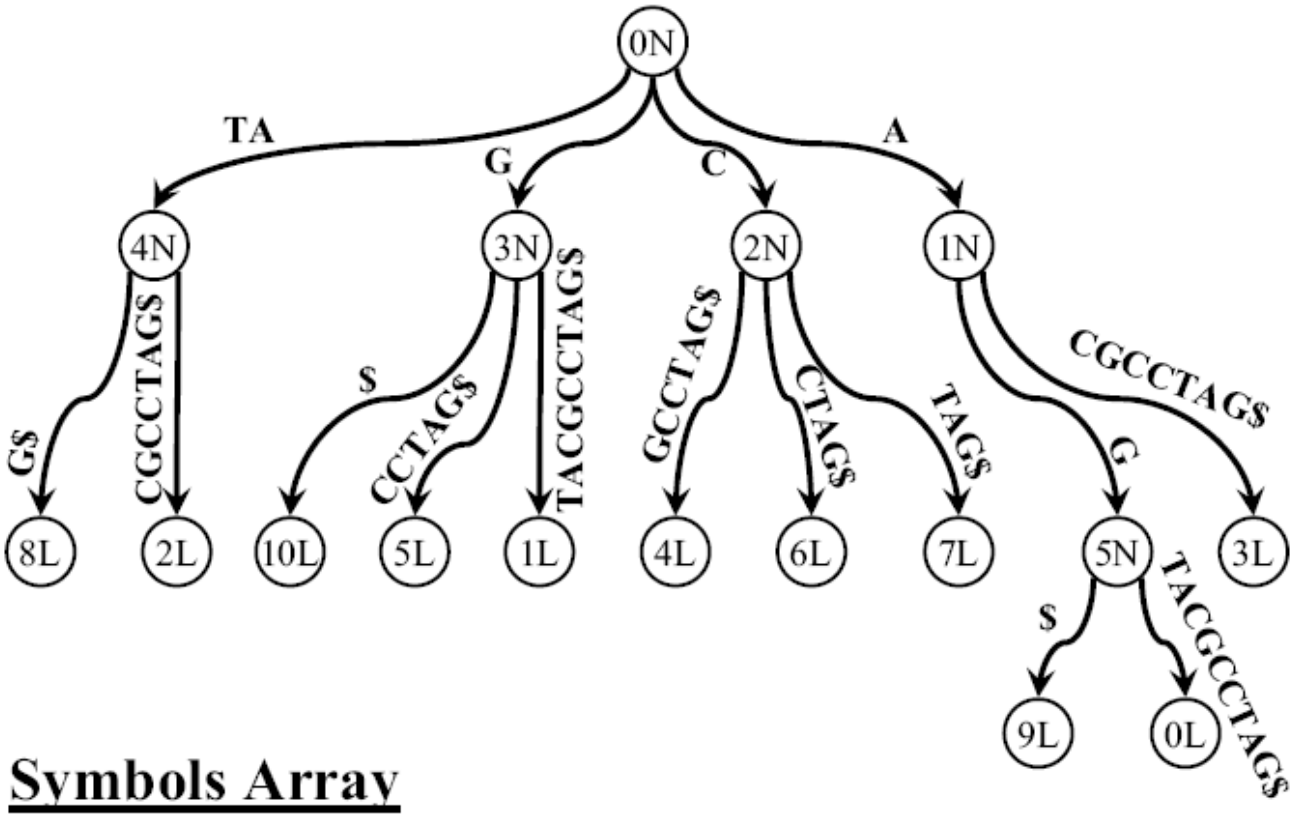
- The S-W algorithm finds the local alignment with the max possible sxore using a dynamic programming algorithm that takes $O(mn)$ time. It generates an $m \times n$ matrix G where each entry $G_{i,j}$ stores the score of the maximum alignment between a query Q and target T ending at q_i and t_j
- $$G_{i_j} = \max \begin{cases} 0, \text{ “start over”} \\ G_{i-1,j-1} + S(q_i \rightarrow t_j), \text{ Replacement} \\ G_{i-1,j} + S(q_i \rightarrow -), \text{ Insertion} \\ G_{i,j-1} + S(- \rightarrow t_j), \text{ Deletion} \end{cases}$$

Smith-Watermann (S-W)Algorithm

- $G_{i_j} = \max \begin{cases} 0, \text{ “start over”} \\ G_{i-1,j-1} + S(q_i \rightarrow t_j), \text{ Replacement} \\ G_{i-1,j} + S(q_i \rightarrow -), \text{ Insertion} \\ G_{i,j-1} + S(- \rightarrow t_j), \text{ Deletion} \end{cases}$
- Example: Q =TACG and target T =AGTACGCCTAG

	A	G	T	A	C	G	C	C	T	A	G
T	0	0	↖1	0	0	0	0	0	↖1	0	0
A	↖1	0	0	↖2	←1	0	0	0	0	↖2	←1
C	0	0	0	↑1	↖3	←2	↖1	↖1	0	↑1	0
G	0	0	0	0	0	↖4	↖3	←2	↖1	0	↖2

Suffix Tree



Symbols Array

Position	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	11
Symbol	A	G	T	A	C	G	C	C	T	A	G	$

OASIS Algorithm

- Basic idea is to perform a best-first search for local alignments, where the expansion of the nodes in the search space is driven by a suffix tree. Nodes are expanded "like" S-W and are placed in a priority queue. Nodes that are likely to produce better score are at the top of the queue.
- OASIS consists of three subroutines $Initialize$, $Expand$ and $OASIS$ (the last is the main function)
- The $Initialize$ function takes as input the suffix tree T (built on the sequence database), the query Q, the arbitrary substitution matrix S and $minScore$, which is a minimum alignment score
- The function returns a vector H with all alignments with scores greater than or equal to $minScore$, in reverse order of score. H is called a "heuristic vector". The function also returns the priority queue, which is a list of nodes that it will expand one by one. At this point the priority queue only contains the root of the suffix tree (called the $seed$).

OASIS Algorithm

- Each h_i in H represents the maximum possible score of $q_{i+1}q_{i+2}\ldots q_m$ with an arbitrary target.
- We calculate these values assuming non-positive values for insertions and deletions
- H_m is set to zero, because the remaining part o the query is empty.
- Then by induction, we calculate the remaining values $H_{i-1} = H_i + (\text{ the max score for the replacement of } q_{i-1})$
- The root of the tree is returned as a search node of OASIS.

OASIS Search nodes

OASIS search nodes have the following properties:

1. sp: A pointer to a node in the suffix tree
2. Z: A vector $[z_0 z_1 \cdots z_m]^T$, where z_i is the score of the strongest alignment between the sequence $path(sp)$ and any subsequence of Q ending at q_i. z_i is set to $-\infty$ if the alignment has been pruned. This vector is analogous to a column of the S-W matrix
3. $maxScore$: The maximum score alignment found along this path. it is the score of the strongest alignment between any prefix of the sequence $path(sp)$ and any subsequence of the query
4. f: The maximum prossible score that can be achieved by further expanding this node
5. g: The maximum score in Z, or the best score ending at node sp in the suffix tree
6. tag: Indicates the status of each search node: ACCEPTED, VIABLE or UNVIABLE

OASIS Search nodes

- A node is tagged ACCEPTED when the strongest possible alignment of the query with this node (or any of its descendants) has been found, and the alignment score passes the $minScore$ threshold. When these nodes reach the top of the priority queue, we return the alignment on-line, since we are certain, by the ordering of the priority queue, hat no subsequent alignments will be stronger
- A node is tagged VIABLE when a stronger alignment other than that already found along this path is possible, and the $minScore$ threshold is reached
- A node is tagged UNVIABLE if no possible expansion of this node can result in an alignment with the necessary strength. These nodes are pruned from the search tree, while the other two are added to the priority queue

OASIS Search nodes

- As we noted earlier each node stores scores for alignments ening at each position of the query (Z vector)
- The sum of these scores is used to organize the search nodes in a priority queue PQ, that is, the priority queue is ordered by the f–value
- Because the node at the head of the PQ is always expanded first (remember: it has the largest f–value), it is clear that a node is only expanded when it cane be guaranteed that no other node on the PQ can produce a stronger alignment
- That's why OASIS is a best-first search technique

OASIS *Initialize*

Algorithm 2 $Initialize(T, Q, S, minScore)$

{compute maximum alignment score beginning after each position in query}

$$H \leftarrow \begin{bmatrix} h_0 \leftarrow h_1 + \max_{\forall \beta} S(q_1 \rightarrow \beta) \\ \cdots \\ h_i \leftarrow h_{i+1} + \max_{\forall \beta} S(q_{i+1} \rightarrow \beta) \\ \cdots \\ h_m \leftarrow 0 \end{bmatrix}$$

{initialize first entry in priority queue}

$seed.sp \leftarrow Root[T]$

$$seed.Z \leftarrow \begin{bmatrix} \cdots \\ z_i \leftarrow \begin{cases} 0 & \text{if } h_i \geq minScore \\ -\infty & \text{if } h_i < minScore \end{cases} \\ \cdots \end{bmatrix}$$

$seed.maxScore \leftarrow Seed.g \leftarrow \max(seed.Z)$

$seed.f \leftarrow \max(H)$

if $seed.maxScore = 0$ **then**

  $seed.tag \leftarrow VIABLE$

  $PQ \leftarrow \{seed\}$

else

  $PQ \leftarrow \{\}$

end if

Return H, PQ

OASIS *Initialize*

- The function returns the 'heuristic vector" vector H with all alignments with scores greater than or equal to $minScore$, in reverse order of score. It also returns the priority queue $PQ = \{seed\}$, $seed.sp \leftarrow Root[T]$.
- Because the path length of the root is 0, all the values in Z are 0, or $-\infty$, in cases where the alignment can be pruned. This is done when entry h_i is less that the value of $minScore$ (which implies that this starting alignment point cannot yield a strong alignment)

	Z	H	$x = 0N$
	0	4	$maxScore = 0$
T	0	3	$f = 4$
A	0	2	$g = 0$
C	0	1	
G	$-\infty$	0	

OASIS *Expand*

Algorithm 3 $Expand(psn, stn, H, Q, S, minScore)$

$node.sp \leftarrow stn$
$node.maxScore \leftarrow psn.maxScore$
$C = c_1 c_2 \dots c_n \leftarrow IncomingPath(stn)$
$G \leftarrow$
$$\begin{bmatrix} g_{0,0} \leftarrow psn.z_0 & g_{0,1} \leftarrow -\infty & \dots & g_{0,n} \leftarrow -\infty \\ g_{1,0} \leftarrow psn.z_1 & g_{1,1} & \dots & g_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m,0} \leftarrow psn.z_m & g_{m,1} & \dots & g_{m,n} \end{bmatrix}$$
for $j \leftarrow 1 \dots n$ **do**
  for $i \leftarrow 1 \dots m$ **do**
    $g_{i,j} = \max \begin{pmatrix} g_{i-1,j-1} + S(q_i \rightarrow c_j), \\ g_{i-1,j} + S(q_i \rightarrow -), \\ g_{i,j-1} + S(- \rightarrow c_j) \end{pmatrix}$
    if $g_{i,j} \leq 0 \vee g_{i,j} + h_i \leq node.maxScore \vee g_{i,j} + h_i < minScore$ **then**
      $g_{i,j} \leftarrow -\infty$
    end if

    if $g_{i,j} > node.maxScore$ **then**
      $node.maxScore \leftarrow g_{i,j}$
    end if

  end for

  $gValues \leftarrow$ the j^{th} column of G
  $fValues \leftarrow gValues + H$

  if $node.maxScore \geq \max(fValues) \wedge node.maxScore \geq minScore$ **then**
    $node.tag \leftarrow ACCEPTED$
    $node.f \leftarrow node.g \leftarrow node.maxScore$
    Return $node$
  else if $max(fValues) < minScore$ **then**
    $node.tag \leftarrow UNVIABLE$
    Return $node$
  end if

end for

$node.f \leftarrow \max(fValues)$
$node.g \leftarrow \max(gValues)$

if $IsALeaf(stn)$ **then**
  if $minScore \leq node.maxScore$ **then**
    $node.tag \leftarrow ACCEPTED$
    $node.f \leftarrow node.g \leftarrow node.maxScore$
  else
    $node.tag \leftarrow UNVIABLE$
  end if
else
  $node.Z \leftarrow gValues$
  $node.tag \leftarrow VIABLE$
end if

Return $node$

OASIS *Expand*

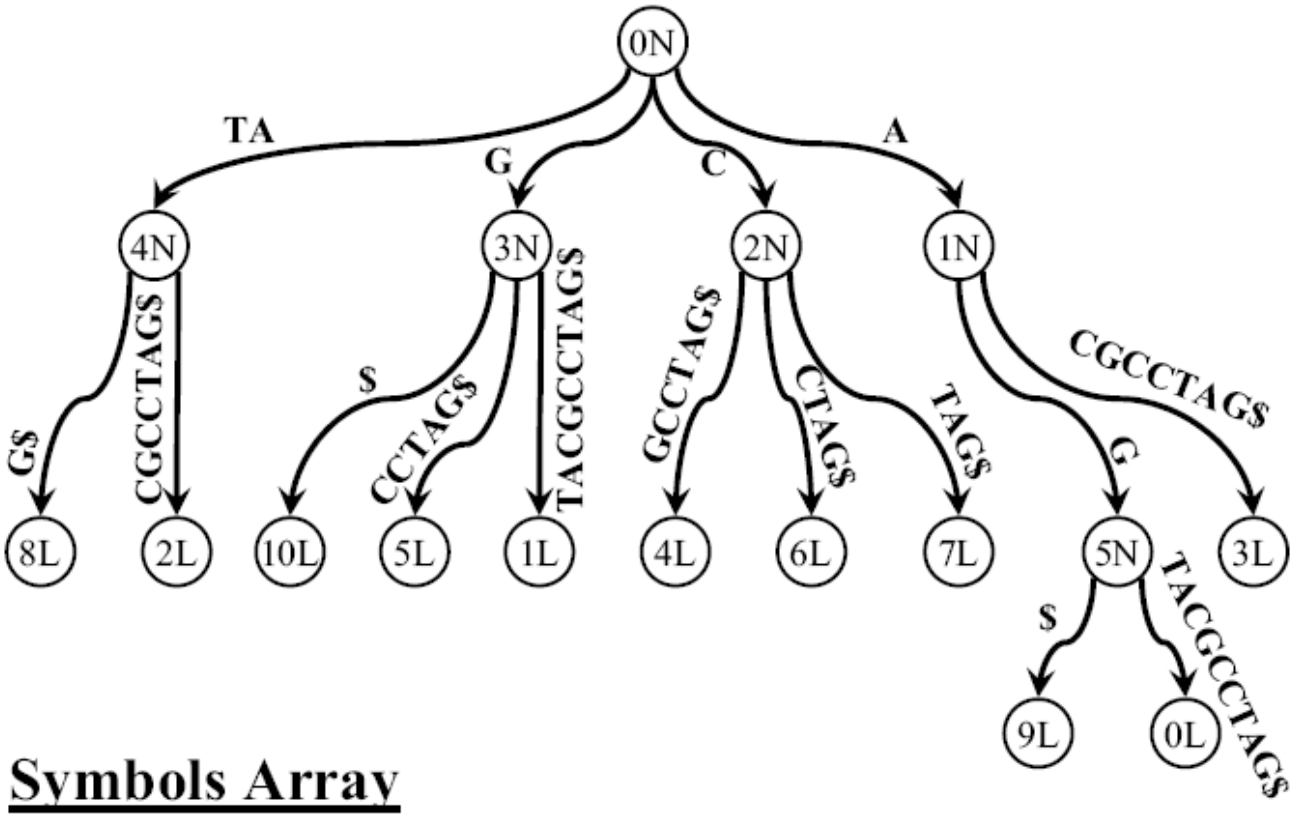
- This function is the core of OASIS
- It is the function that computes columns like in the S-W algorithm, and to do that it uses the Z column of the parent node as seed
- After computing a value $g_{i,j}$, OASIS performs alignment pruning (seting score in Z vector to $-\infty$) on alignments that are no longer viable.

OASIS *Expand*

Three cases for pruning:

1. $g_{i,j} \leq 0$: To avoid duplicationg work, nodes with negative alignment are pruned. Consider a partial local alignment between a portion of a query $q_a q_{a+1} \ldots q_b$ and a portion of suffix $t_c t_{c+1} \ldots t_d$. If the max alignment score beginning at $q_a \rightarrow t_c$ and ending at $q_b \rightarrow t_d$ is negative, then the score of any alignment between $q_a q_{a+1} \ldots q_b \ldots q_{c'}$ and $t_c t_{c+1} \ldots t_d \ldots t_{d'}$ will have a lower score than the alignment between $q_{b+1} \ldots q_{c'}$ and $t_{d+1} \ldots t_{d'}$. This second alignment will be expanded alogn another tree path
2. There is an existing alignment which is just as good, i.e. $g_{i,j} + h_i \leq node.MaxScore$: We cannot find an extension to this alignment with a better score that the strongest alignment already found in this search path
3. Threshold failure, $g_{i,j} + h_i \leq minScore$: No possible extension to this alignment can be equal to or greater than $minScore$

Suffix Tree



Symbols Array

Position	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	11
Symbol	A	G	T	A	C	G	C	C	T	A	G	$

Expansion of node 1N

	G	(Z)	H	F	$x = 1N$
	-	A			$\leftarrow Target(headPos = 3)$
	0	$-\infty$	4	$-\infty$	$maxScore = 1$
T	0	$\nwarrow (-1)$	3	$-\infty$	$f = 3$
A	0	$\nwarrow 1$	2	3	$g = 1$
C	0	$\uparrow (0)$	1	$-\infty$	$tag = VIABLE$
G	$-\infty$	$\nwarrow (-1)$	0	$-\infty$	

Node is VIABLE because $f \geq minScore$

Expansion of node 2N results in a f value of 2 and g value of 1.
Expansion of node 3N results in a f value of 1 and g value of 1, so this node is marked ACCEPTED

Expansion of node 4N

	G		(Z)	H	F	$x = 4N$
	-	T	A			$\leftarrow Target(position = 8-9)$
	0	$-\infty$	$-\infty$	4	$-\infty$	$maxScore = 2$
T	0	$\nwarrow 1$	$\leftarrow (0)$	3	$-\infty$	$f = 4$
A	0	$\uparrow (0)$	$\nwarrow 2$	2	4	$g = 2$
C	0	$\nwarrow (-1)$	$\uparrow (1)$	1	$-\infty$	$tag = VIABLE$
G	$-\infty$	$\nwarrow (-1)$	$-\infty$	0	$-\infty$	

Node is VIABLE because $f \geq minScore$

Expansion of node 2L

	G			H	F	$x = 2L$
	A	C	G			$\leftarrow Target(position = 3-5)$
	$-\infty$	$-\infty$	$-\infty$	4	$-\infty$	$maxScore = 4$
T	$-\infty$	$-\infty$	$-\infty$	3	$-\infty$	$f = 4$
A	2	$\leftarrow 1$	$\leftarrow (0)$	2	$-\infty$	$g = 4$
C	$-\infty$	$\nwarrow 3$	$\leftarrow (2)$	1	$-\infty$	$tag = ACCEPTED$
G	$-\infty$	$\uparrow (2)$	$\nwarrow 4$	0	4	

Node is ACCEPTED because $f = g$

Expansion of nodes

- At first $PQ = \{Root[T]\}$
- After first expansion $PQ = \{(4N/4), (1N/3), (2N/2), (3N/1)\}$
- After another step $PQ = \{(2L/4), (8L/2), (2N/2), (3N/1)\}$
- The top element is tagged ACCEPTED and we have found the maximum local alignment

OASIS Main Loop

Algorithm 1 $OASIS(T, Q, S, minScore)$

$H, PQ = Initialize(T, Q, S, minScore)$
while $\neg Empty(PQ)$ **do**
  $searchNode \leftarrow PQ.Pop()$
  if $searchNode.tag = VIABLE$ **then**
    for $successor \in Successors(searchNode.sp)$ **do**
      $newNode \leftarrow Expand(searchNode, successor, H, Q, S, minScore)$
      if $newNode.tag = VIABLE \vee newNode.tag = ACCEPTED$ **then**
        $PQ.Add(newNode)$
      end if
    end for
  else if $searchNode = ACCEPTED$ **then**
    Return on-line all sequences containing $Path(searchNode.sp)$
  end if
end while