Final Report
Prateep Mukherjee(00876583)

Paper title: Top-*K* Color Queries for Document Retrieval
Authors: Marek Karpinski & Yakov Nekrich
Conference/Year: SODA '11.

1. Problem description & Nature of the results (Time & Space Complexity): Document listing/retrieval problems are extremely important in today's information age. Given that the number of web-pages, blogs and research articles are increasing at an exponential rate every second, it is necessary to design efficient, scalable and robust algorithms for information retrieval. The well-known example of such a scenario are search engines: an answer to a query is a sequence of documents output in the reverse order of their relevance. Static ranking of documents, based on their PageRank and hyperlinks, is an important part of estimating document relevance. Muthukrishnan[4] explored this area first and came up with the following important contribution: Document listing/reporting problems can be reduced to color range queries.

Lemma: Let, d1… dn be a set of documents, each being a string such that max_i |di| = L and the alphabets being in set Sigma. The Document Listing Problem(DLP) is to find a subset of documents which contain the pattern P. DLP is equivalent to reporting the number of unique colors in a range [a,b] of an array A, in which each element has a color value from the set C, ($|C| = \sigma$).
Proof: State-of-the-art methods [2,3,4] for solving the document-retrieval problems store suffixes of all patterns in all the documents in a generalized suffix tree. Edges of the suffix tree are labelled with strings, which correspond to a set of suffixes. The leaves of the suffix tree are labelled with documents, in which the suffixes belong to. Non-leaf nodes (v) of the suffix tree are also labelled with a set of documents, those whose suffixes contain the string at node v. Therefore, the algorithm to find the set of documents containing pattern P in the tree is as follows: start at the root(R) and find the shortest path to a non-leaf node(V), such that the path traversed($R \rightarrow V$) generates the entire pattern P. The set of documents at this node all contain the pattern P. The top-K documents are computed from this set using the range maximum query (RMQ) algorithm.

The mapping of the document retrieval problem to the color range query problem can now be defined as follows. The set of documents at the node all contain the pattern P. Therefore, the indices documents marked by the leftmost and rightmost children of V denote the lower and upper bounds of our search query. This reduces the document retrieval problem to finding the top-K documents in this range. □

The paper builds on the work of Muthukrishnan[4] by proposing an efficient algorithm for the *top K-color problem.* The problem statement is as follows: Given an array A where every element is assigned with a color c and an associated priority p(c), a query with a range [a,b] and a value K expects the algorithm to report K unique colors with the highest priorities among all colors that occur in A[a,...b], sorted in decreasing order of their priorities. The main contribution of this paper is the optimal time and space complexities of the involved data structures and the core algorithm in solving the *top K-color query* problem.

The data structures designed in this paper uses a new explicit technique for recursive, exponentially decreasing size sub-arrays combined with a new method for caching pre-computed answers. Specifically, the authors show that an array A can be stored in **O(N log σ)** bits data structure so that for any two indices a < b and for any integer K, K distinct colors with highest priorities among all colors occurring in A[a,b] can be reported in **O(K)** time. The authors first design a data structure with $O(N^{1/f} + K)$, f > 1, query time and then transform the latter into the proposed data structure with optimal query time. This transformation critically depends on an efficient method for obtaining solutions for pre-computed intervals, and recursively defined data structures, a.k.a. trees, with exponentially decreasing number of elements.

Summary:

	Query Time	Space Usage (in bits)
section 3	$O(\log^2 N + K)$	$O(N \log^2 N)$
section4	$O(N^{1/f} + K)$	$O(N \log N)$
section 5	$O(K)$	$O(N \log N)$
section 6	$O(K)$	$O(N \log \sigma)$

2. Related Work: Matias et al. [3] described the first data structure for this problems. They used a generalized suffix tree data structure for listing and counting queries a given pattern P from a list of documents[d1,dn]. The generalized suffix tree(GST) is simply a compact trie of all suffixes of each of the documents. Each leaf node in the tree is labeled with the list of documents which have a suffix represented from the root to itself in the tree. In order to respond to the queries, the data structure performs a Least Common Ancestor(LCA) query. Their data structure answers document listing queries in O(|P| log s + docc) time, where |P| is the length of P and docc is the number of reported documents.

Muthukrishnan [4] describes an O(N logN) bits data structure that answers document listing query in optimal O(|P| + docc) time. The data structures of [6, 7] further improve the space usage by storing the documents in compressed form; it takes $O(\log^{\varepsilon} N)$ time [6] or O(log s) time to report each document.

Hon et al [2] reports the relevant documents with respect to a pattern P, instead of all the documents. Their approach also uses the GST data structure, however they boost the running time using Range Maximum Query(RMQ) data structure to report the K maximum occurrences of string P in O(|P| + KlogK) time. Their data structure uses linear space (i.e., O(N logN) bits) and can report K most relevant documents in O(|P| + K logK) time.

3. Key Ideas/Techniques: The authors start with an augmented yet simple tree data structure (DS), and thereafter consequently improve their methods of traversal and retrieval to generate optimal space and query time.

(a) $O(N\log^2 N)$ Space, $O(\log^2 N + K)$ Time Data structure (Section 3)

Data Structure Used/Proposed: Wavelet Tree. The standard wavelet tree data structure uses bit vectors at each node to execute search queries in a string in logarithmic time. Wavelet tree is a balanced binary search tree, where for each node 0 encodes half of the symbols, and a 1 encodes the other half. In this way, at each level of the tree the input string is split into two halves, depending on how the encoding is performed. In this section, the authors use a variation of the aforementioned wavelet trees to execute range search queries in logarithmic time complexity.

Design: At each node, the color array C (C_v) is split into two equal parts such that every color in one part(C_0) has a smaller priority than any color in the second(C_1). Let us call the left child with color C_0 as node w, and the right child with color C_1 as u. Along with the color array, the element array A_v is also split into two groups such that elements with colors from C_1 are in A_u, and elements with colors in C_0 are in A_w. This separation is done using a bit array B_v at node v, $B_v[i] = 1$ if $A_v[i]$ belongs to C_1, = 0 otherwise. The idea is to partition the elements of the array Av based on the colors and first query the right child (having higher priority of colors) and then recurse over the left subtree.

Data Structures stored at each node (v):

- the subarray $A_v = A[a_v,b_v]$
- bit array B_v
- REP_v and $COUNT_v$ data structures to support color reporting and counting queries on A[a,b]

Algorithm:

- Initialize $a_v = a$ and $b_v = b$ at the root node v. Also compute B_v.
- Given A_v and B_v, the bounds of node u can be computed by counting the number of bits in B_v.
  i. a_u is the number of 1's in $A_v[1,a_v]$. Similarly, b_u is the number of 1's in $A_v[1,b_v]$.
  ii. a_w and b_w are also computed in a similar fashion, by counting numbers of 0's.
- Compute $[a_u,b_u]$ of right child from B_v and A_v and count the number of colors (say m_u) at node u in range $A[a_u,b_u]$.
- If $m_u >= K$, report the top K colors in range $[a_u,b_u]$. Else, report m_u colors. Decrease K by m_u, and then compute $[a_w,b_w]$ for the left node w.
- Set v = w, and recursively search the left subtree.

Proof of Correctness: Each node of the wavelet tree DS is subdividing range of the entire array A into two halves. After that, the bounds [a,b] in the original array A is percolated to Au and Aw using the number of 1's and 0's in the bit array. This reduces the top-K color problem in A[a,b] to Au[au,bu] and Aw[a_w,b_w]. Also, since minimum priority of colors in C1 is greater than maximum priority in C0, the algorithm searches the right subtree in order to get the top K-colors. In case, it finds it, the number of unique colors is listed or reported. In case the query does not find a result, it reiterates over the left subtree. As we are reducing the search space to half while going from a parent to a child, the range sizes also gets reduced proportionately. Therefore, using the COUNT and REP data structures at each node, the algorithm converges at an exponential rate proportional to the number of elements in A.

Complexity Analysis:

Total number of visited nodes is O(logN) .

The total number of bits required is $O(N*\log N * \log N) = O(N\log^2 N)$, where C_v requires O(logN) bits and A_v requires N bits.

Space Complexity: $O(N\log^2 N)$ bits

Time Complexity: At each node v, the time complexity for color counting is O(logN) and color reporting is $O(K_v)$, where K_v is the number of colors in C_v. Thus, the total query time is $O(\log N+K_v)$. Therefore, **total time complexity for each query in such a tree is $O(\log^2 N + K)$.**

(b) O(NlogN) Space, $O(N^{1/f}+K)$ Time Data structure (Section 4)

Data Structure Used/Proposed: Wavelet tree with a constant number of levels (f). The DS described in the previous section searches every node of the tree in order to answer the color range queries. For most practical purposes and for efficiency, it suffices to look at certain levels of the tree only in order to answer the queries.

Design: Same as in (a). However, instead of looking at nodes at each level of the tree, one can only look at *"**important nodes**"*. A node v is *"important"* if it is situated on levels whose height are a constant factor (f) of logN, { i/f logN, i = [0,f] }. The trick used here is to only look at descendants of an important node, which are also important. These descendants are referred as **highest important descendants.** One can optimize the space used at each node by storing information only about its most important descendants. The maximum number of important descendants of a node is $O(N^{1/f})$, since $fN^{1/f}$ is the total number of nodes we look up in the tree.

Data Structures stored at each node (v):

- array E_v which counts elements with color c belonging to $C_v[i]$ of the array A_v
- REP_v and $COUNT_v$ data structures to support color reporting and counting queries on A_v

Algorithm:

Let $v_i(1<=i<=t)$ be the most important descendants of node v.

- Initialize the root node v with $a_v = a$, $b_v = b$, $i = t$.
- If we know [a,b] at node v, then find $[a_{vi},b_{vi}]$ at each of its important descendent nodes v_i from E_v. For each descendant node v_i, let m_i be the number of colors in $A[a_{vi},b_{vi}]$ and $r_i = \text{sum}\{i=1..t\}\ m_j$.
- If $r_i < K$, visit v_i and report all m_i colors in $A[a_{vi},b_{vi}]$ and proceed to v_{i-1} in decreasing order.
- If $r_i >= K$, we have to look at the next level of important nodes. Therefore $K = K - r_{i+1}$ and we recursively look at level [(i+1)/f logN].

Proof of Correctness: The strategy is to search over the important descendant children of a node in order to get the result. Now, r_i is the cumulative sum of all the number of colors in range [i,t]. Therefore, if $r_i < K$, the algorithm reports all the colors in $A[a_{vi},b_{vi}]$ and moves to its previous sibling. However, if $r_i >= K$, it falls through to the next important level of the tree and recursively looks at every node at this level. The array E at each node v allows us to compute the bounds in the child nodes in $O(\log N_v)$ time and $O(N_v \log N_v)$ bits of space.

Complexity Analysis:

Total number of visited nodes in this case is $O(fN^{1/f}) = O(N^{1/f})$.

At most one color counting and one reporting query is performed at each node.

Time complexity: $O(N^{1/f}\ \log N+K)$ using f levels.

Space complexity: For all nodes at same level of the tree, O(NlogN) bits of space for storing

REP_v, $COUNT_v$ and E_v, where N is the length of array A. Thus, total space complexity is **O(NlogN)** bits, as we traverse a constant number of levels of the tree.

(c) O(NlogN) Space, O(K) Time Data structure (Section 5)

Data Structure Used/Proposed: Wavelet tree with $\log\sqrt{N}$ levels.

Design: In (b), if K = $\Omega(\sqrt{N})$, the queries can be answered in $O(\sqrt{N}+K) = O(K)$ time. So, for this section, we assume K <= $\sqrt{N}$. The first claim the authors make is that query in a range with endpoints $O(\sqrt{N})$ apart can be computed in O(K) time. Therefore, each range [a,b] is computed as a union of 3 sub-intervals: $[a,a_1]$, $[a_1,b_1]$, $[b_1,b]$, such that range $[a_1,b_1]$ satisfy the aforementioned claim. Thus, there are two design details discussed in this section: (1) computation of top-K colors in a $O(\sqrt{N})$ range in O(K) time; and (2) compute the top K colors in ranges $[a,a_1]$ and $[b_1,b]$ in O(K) time. These are discussed in order.

Let, J = $\{i\sqrt{N}\}$ denote the set of all$\sqrt{N}$ ranges. A data structure is stored for each subarray A[i1,i2] such that i_2 follows i_1 in J. Since each range$[i_1,i_2]$ contains roughly $\sqrt{N}$ elements, each such range query can be answered in $O(N^{1/4} + K)$ time. Subdividing each such range $[i_1,i_2]$ using a similar design as before, the range queries can be computed in $O(N^{1/8} + K)$ time complexity. Therefore, top K-colors in each range $[i_1,i_2]$ is computed by from lists $L(N^{p(l)}, i_1,i_1+2^x)$ and $L(N^{p(l)}, i_2-2^x,i_2)$, where $x = \log(i_2-i_1)$, $p(l) = (½)^l$ and L(m,a,b) is the list containing the top m colors in A[a,b].

Next, the authors focus on the details to compute top K colors in [a,a1] and [b,b1]. First, one computes the maximum l such that $N^{p(l)} >= K$. In order to query the range [a,a1], and similarly [b,b1], the authors use a wavelet tree with f = 6, as discussed in (b). p(l) decreases exponentially as l increases. Thus, the query of reporting top K colors in range A[a,a1] is performed in O(K) time. Similarly, the same query for range A[b,b1] can be reported in O(K) time. The final step is combination of these lists, which is also O(K).

Data Structures stored at each node(v):

- F(a,b) : supports top-K color queries in O(K) time in the case when K <= $\log^{⅓} N$
- R(m,a,b) : data structure as implemented in (b), with a followed by b in J.
- L(m,a,b) : list for storing top-m colors in range A[a,b]
- C(i,j) : set of all colors that occur in A[i,j]
- M(i,j) : set in which every color in C(i,j) is replaced by rank of its priority.

  $M(i,j) = \{ prank(c,C(i,j)) \mid c \in C(i,j) \}$
- Tbl(i,j) : lookup table where (i,j) is color c ∈ C which corresponds to M(i,j)

Algorithm:

- Check whether the number of distinct colors in [a,b] exceeds K. The data structure in (b) is used to report colors in [a,b] until K colors are reported or the procedure terminates. K is set to the output number of colors at this step.
- Find the largest value l such that $N^{p(l)} >= K$.
- Find the minimum range [a1,b1] that one has to compute, $a1 = a/N^{p(l)}$ and $b1 = b/N^{p(l)}$.
- First compute the top-K colors in [a1,b1] from L.
- If K > $\log^{1/3} N$, compute the top-K colors in [a,a1] and [b,b1] from R
- If K <= $\log^{1/3}N$, compute the top-K colors in [a,a1] and [b,b1] from F
- Combine ranges $[a,a_1]$, $[a_1,b_1]$, $[b_1,b]$ to report the top-K colors in A[a,b]. The colors can be sorted in decreasing order using radix sort in O(K).

Proof of Correctness: The solution is built in a bottom-up fashion on the wavelet tree. In the first step, the smallest interval is computed and the result is reported. In the next step, the solution to its parent node is computed by combining the result from 3 such intervals. In this way the algorithm traverses upto the root of the tree, where it reports the top-K colors in A[a,b]. The number of levels of the tree is O(log log N), since we are reducing an O(log N) size array into half each time. Depending on the value of K, the number of levels we need to go are computed initially. This is essentially important, as the length of array A might be huge. So recurring all the way down to the bottom of the tree will give stack overflow. Also, K is handled differently based on its ratio to O(log N). The reason is because R is a 3-dimensional data structure, and hence too many of these might overflow memory. So when K is sufficiently large ($K > \log^{1/3} N$) and we have to go O(log log N) levels, R is used to compute the top-K values. However, when ($K <= \log^{1/3} N$) and l = O(log log N),the data structure F is used. Construction of F takes O(N) time in the beginning, however query time is O(K).

Complexity Analysis: All lists L use o(N) bits. R, as discussed in (b), uses O(NlogN) bits of space. Finally, F also uses O(NlogN) bits.

Space complexity: O(NlogN) bits

Time complexity: As discussed in the design section, each of the two steps involved in query takes O(K) time. Therefore, the total time complexity using this Data structure is **O(K).**

(d) **O(N log σ) Space, O(K) Time Data structure (Section 6)**

Data Structure Used/Proposed: Same Data structure as in Section 5, with chunks of the array of different length.

Design: Let the size of the color array C is .**The authors assume in this section that σ = o(N).** The authors consider two separate cases (i) $\sigma^2 >= \log N$ and (ii) $\sigma^2 < \log N$. In (i), the array A is subdivided into chunks L_i, each chunk containing σ^3 elements. In (ii), the chunks are of length $\sigma^2 \log N$ size. Thereafter, similar algorithm of computing a range as a union of three chunks is used.

Data Structures stored at each node(v): Same as in (c)

Algorithm: The same recursive algorithm, as described in (c) is used here. Thus the time complexity remains the same.

Proof of Correctness: The entire array is divided into chunks. If the bounds of the query range fall inside one chunk, we can directly apply the recursive algorithm described in (c). Else, the original range [a,b] can be decomposed into 3 chunks [a,a'], [a'+1,b'] and [b'+1,b] such that [a',b'] fall in one chunk. Range query in [a'+1,b'] is done using the algorithm in (c). The other two ranges [a,a'] and [b'+1,b] can be reported using the chunk data structures.

Complexity Analysis: The algorithm works exactly the same as discussed in (c) with different length of sub-arrays. Therefore the time complexity remains same. Space complexity, however, is optimized as we are working on smaller chunks.

Time complexity: O(K)

Space complexity: O(Nlog σ), where is the number of different colors.

4. Online queries

In the methods described before, the algorithm must report top K colors in the query range, and the value of K is specified apriori. This condition can be relaxed by modifying our method to report top colors from a specified interval until the user terminates the reporting procedure or all colors have been reported. The trick here is as follows. Assume $K_i = 2^i$. (I) The algorithm reports K_i colors incrementally, that is for i = 0, 1, 2 and so on. (II) After one execution, the algorithm erases the colors reported in the previous

iteration and reports the new colors to the user (K_{i+1} - K_i). The above two steps are interleaved in order to provide an online version of the range query problem.

5. Document retrieval

The authors provide an application of the above data structures on different flavors of Document retrieval problems. The three major versions of the generalized DLP (mentioned in Section 1) are as follows:

a) **Ranked Document Listing :-**
Statement Documents [d_1 … d_n], each having a priority value, are stored in a data structure, so that for any pattern P and any fixed K, the algorithm must return K most highly ranked documents that contain P ordered by their rank.
Solution A suffix tree is built for the pattern P. The locus v of P can be reported in O(|P|) time. Thereafter a and b be the minimum and maximum indices of the leaf descendants of the node v. Finally, the algorithm reports the top K most highly ranked documents in A[a,b] in O(|P| + K) time.

b) **Ranked t-Mine Problem :-**
Statement Documents [d_1 … d_n], each having a priority value, are stored in a data structure, so that for any pattern P and any fixed K and t, the algorithm must report K most highly ranked documents that contain P at least t times ordered by their rank.
Solution For any pattern P, the algorithm described in Section 3(c) identifies indexes a_p and b_p in array A^t, containing O(N/t) elements. Therefore, the algorithm divides the entire array A into t chunks, and searches the pattern P over each of those chunks. It is to be noted that O(sum_t (N/t)) = O(N log N).

c) **Most relevant DLP :-**
Statement Documents d_1...d_n, are stored in a data structure, so that given any input pattern P and a value K, the algorithm reports the K most relevant documents with respect to a metric rel(d,P), in descending order of rel(d,P).
Solution The authors maintain a max-heap data structure to get the maximum element in A[ai,bi]. Then repeat for K times the following: (i) extract the maximum element from the heap and append it to the output list, (ii) remove element from the heap, pick the next highest element and add it to the heap. The heap contains |P| elements and extracting the maximum element from the heap is O(1). Finding the next largest element is O(K) time. Thus, the total time complexity is O(|P|+K).

6. References:

[1] P. K. Agarwal, S. Govindarajan, S. Muthukrishnan Range Searching in Categorical Data: Colored Range Searching on Grid, Proc. 10th ESA 2002, 17-28.
[2] W.-K. Hon, R. Shah, J. S. Vitter, Space-Efficient Framework for Top-k String Retrieval Problems, Proc. 50th IEEE FOCS 2009, 713-722.

[3] Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv, Augmenting Suffix Trees, with Applications, Proc. 6th ESA 1998, 67-78.
[4] S. Muthukrishnan, Efficient Algorithms for Document Retrieval Problems, Proc. 13th ACM-SIAM SODA 2002, 657-666.
[5] R. Grossi, A. Gupta, and J. S. Vitter, High-order Entropy-compressed Text Indices, Proc. 14th ACM-SIAM SODA 2003, 841-850. [Wavelet tree]
[6] K. Sadakane, Succinct Data Structures for Flexible Text Retrieval Systems, Journal of Discrete Algorithms, 5(1), 12-22 (2007).
[7] N. V¨alim¨aki and V. M¨akinen, Space-Efficient Algorithms for Document Retrieval, Proc. 18th *CPM 2007, 205-215.*
[8] T. Gagie, S. J. Puglisi, A. Turpin, Range Quantile Queries: Another Virtue of Wavelet Trees, Proc. 16th SPIRE 2009, 1-6.
[9] *Gonzalo Navarroy, Yakov Nekrichz, Top-k document retrieval in optimal time and linear space, Proc. 23rd ACM-SIAM SODA 2012, 1066-1077.*
[10]. M. L. Fredman, D. E. Willard, Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths, J. Comput. Syst. Sci. 48(3), 533-551 (1994).