CS 840: Project

TOP-k RETRIEVAL ON TEXT DOCUMENTS

by

Alexandre Daigle
20549314

CS 840: Advanced topics in Data Structures
David R. Cheriton School of Computer Science
University of Waterloo

due December 18, 2014

Waterloo, Ontario, Canada, 2014

Abstract

Given a collection of text documents $\mathcal{D}$, over an alphabet of size σ, of total length n, one common operation that is often required in practice is to retrieve the best documents matching a query pattern. This problem is the top-k document listing problem and has been addressed by multiple papers [1, 3]. In 2012, Navarro and Nekrich presented a new technique to translate a top-k problem into multidimensional geometric search [6, 7]. With this framework, we apply two different results. First, we propose a modification to support query in the bag-of-words model in $O\left(\sum_{w \in W} |w| + k\right)$ time, where W is the collection of words or expressions to search. Secondly, we apply this data structure to the top-k completion problem.

Introduction

Efficient retrieval of information in very big corpus is an important concern and multiple problems in this regards have been studied. In many cases, we are only interested in the most relevant documents from this corpus based on a given query, where the relevancy can be estimated using various measurement. In any case, getting the top-k documents based on one of these measurements given a query pattern P is often a good solution. We formally define the problem as follows.

Definition Given a collection of documents $\mathcal{D} = \{d_1, d_2, \dots d_m\}$, a non-negative integer k and a weight function $w(S, d)$ that compute the weight of the string S in document d, the top-k document listing problem is to output the k highest ranked document that contains the query pattern P.

There are two main approaches used to answer this problem. The first and more classic way is to use a tree representation of the content of the documents and to query this data structure using the supported operations. A such example of these representations are the suffix tree and the prefix tree[1]. These data structures are used to store every suffix (or prefix) of the documents in size linear to the length of the whole corpus. The second standard – and most popular in practice – way of solving the problem is to use inverted index. In an inverted index, we store a map of every word (called *term*) found in the documents and for each, we have a sorted array that list the document and position containing the term. This second technique is very popular in search engine communities because of the facility

[1]A prefix tree is also called a trie, mostly in the information retrieval community, or a Patricia tree

to support various form of parallelism but also because the implementation of an inverted index is usually faster than those of tree representation.

However, for this project and the selected paper, the first classical representation was chosen. The following section presents the main results from Navarro and Nekrich [7], section 3 shows a modification to support queries in the bag-of-words model. Section 4 presents another problem, top-k completion, that can use the described data structure to quickly and efficiently return results.

Overview of the technique

We present the data structure used to answer top-k document listing queries in $O(|P|+k)$ with $O(n)$-word space[2] in order to transfer the problem to two-dimensional range search. First we assume that the weight function $w(S, d)$ is given by an oracle in constant time.

Transferring the problem to a 2D grid

We store the documents in a generalized suffix tree, which stores every suffixes of the documents of $\mathcal{D}$ on which we append a different terminal character to each. The tree only requires $O(n)$ words space when using the compact representation that compress edges. We say that a node v is marked with document d either if v is a leaf whose suffix ends document d or if v is an internal node who has two different children that contains leaf nodes marked with document d in their subtree. We also store a pointer $ptr(v, d)$ in each of those nodes that points to its lowest ancestor u such that u is also marked with d (if none exists it points to a dummy node ν that is the parent of the root).

At this point, we can transfer the problem to a two-dimensional grid. We number the marked documents of the nodes with a pre-order traversal of the tree. This numbering is then used as the x-coordinate in the 2D grid. The y-coordinate corresponds to the depth of the node pointed by the pointer for the marked document i. We define the depth of ν as $depth(\nu) = 0$. The matching from the x-coordinates to the document number is done by a simple array of size n. Finally, we store $[l_v, h_v]$, the lowest and highest numbering for each node v.

[2]Throughout this paper, we use the RAM model as does the original paper from Navarro and Nekrich

Answering queries

We can now give the general idea to answer queries. The first step is to find the *locus* node p of P, which is defined as the highest node v such that P is a prefix of the path of v. We, then, report every points in the range of $[l_p, r_p] \times [0, depth(p) - 1]$. Finally, we find the best k matching points in this interval based on the weight function $w(\cdot, \cdot)$.

Remark The original paper uses a third dimension to store an additional parameter. This parameter can represent dates, PageRank values for documents or other values that doesn't depends on P. For this review, we decided to simplify to the representation with only two dimensions.

To support these operations efficiently we need to define a few additional data structures to use. For the first part, we must returns the points in the provided range without considering any of the weight. This is done by using a Range Minimum Query (RMQ) data structure on the y-coordinates. The data structure takes $O(n)$ bits and answer query of the form $rmq(a, b) = \arg\min_{a \leq i \leq b} Y[i]$ in constant time, where Y is an array – not stored – of the y-coordinated sorted by x-coordinate. To get all the values we ask for the first lowest points, which occur at position c. Then we can split the total range in two: $[a, c-1] \times [0, h]$ and $[c+1, b] \times [0, h]$. We can recursively find each of the points in the range until we reach points whose y-coordinate is bigger than h. It thus takes $O(occ)$ time to find all the points in the range where occ represents the number of points in $[a, b] \times [0, h]$ [5]. We express this result with the following lemma.

Lemma 1 *Let $v \leq m \leq n$. There exists a data structure that uses $O(v \log m)$ additional bits and report points in a two dimensional range for a set S of v points on an $m \times n$ grid in $O(\log v + occ)$ time, to report the occ result.*

Finally, we need to sort the v candidates points in weighted order and return the k most relevant elements which we do in two steps. First by a lemma that achieves the goal but doesn't meet the space requirement and secondly we match the space requirement by further refining the data structure introduced by the lemma.

Lemma 2 *Assume that $m \leq n$ and let $0 < f < 1$ be a constant. There exists a data structure that uses $O(m \log m)$ additional bits of space and reports top-k points in a $m \times n$ grid in $O(m^f + k)$ time.*

The idea of the proof is to partition the points of $\mathcal{S} = [a, b]$ into $m^{f'}$ buckets, with $0 < f' < f$. The buckets each have a size of $m^{1-f'}$ except maybe the last one. Now we want to place the points in each buckets such that $\forall i < j, \forall x \in B_i, y \in B_j$, we have $x \geq y$. We then recursively split the buckets into smaller buckets. This gives us a tree of arity $m^{\Theta(f)}$. The idea to extract the top-k element is to pick the element from the left of the tree and to simply pick every element in a node if the size of the node is smaller than h. This requires examining $O(m^{f'})$ classes for a total of $O(m^{f'} \log m + k) \subseteq O(m^f + k)$ time. We finally only need to sort the k selected points either by a standard sorting algorithm if $O(k \log k) \subseteq O(m^f)$ or in $O(k)$ by a radix sort since the sets contains at most m distinct weights.

Depending on the values of m, we might exceed our space limitation. The idea at this point is to partition the space in stripes of different width. We use the intervals $\Delta_j = n^{(f^j)} \log^\delta n$ for $j = 1, \ldots, O(\log \log n)$ and $\frac{1}{2} < \delta < 1$. For each of the stripes we store the data structure described earlier by lemma 2 which gives $O((\Delta_j)^{f'} + k)$ query time. Finally, we use a reduction to rank space[2] to answer each of the smallest intervals which takes $o(n)$ words space. Thus the data structure uses $O(n)$ words of space and answer queries in $O(h+k)$ time. The complete result is summed up in the following theorem.

Theorem 3 *A set of n weighted points on a $n \times n$ grid can be stored in $O(n)$ words of space, so that for any $1 \leq k, h \leq n$ and $1 \leq a \leq b \leq n$, k most highly weighted points in the range $[a, b] \times [0, h]$ can be reported in decreasing order of their weights in $O(h + k)$ time.*

Bag-of-words queries

In the *bag-of-words* model, we consider a sentence as a set of the words it contains. This means that we do not care about the position of the words in relation to each other. This is a big contrast on the basic search of a sentence with the current implementation which finds the documents that contains the exact sentence, with correct order, of each of the words. In some cases, this is desirable, but in others, we would like to represent two words and list every documents containing both words, not necessarily consecutively. This is the problem we will address in this section.

Remark This problem can be solved easily with an inverted index, in which it is easier to support the bag-of-words model than the exact phrase model that we support by default with the suffix tree. We will explore a way to support the bag-of-words model with the suffix tree described in the paper reviewed[6].

In order to support this query, we will assume that we have a set of words $W = \{w_1, w_2, \dots, w_l\}$ and we want to output the list of documents containing all of these words in any order or position.

A naive solution is to simply compute the standard document listing for each word of W and then intersect the result. This solution would give a query time of $O\left(\sum_{w \in W} |w| + |W| \cdot k\right)$. The intersection takes at most $|W| \cdot k$ since each individual query can return k documents and there are $|W|$ such queries.

A better solution is to find the locus node of each of the words, find their two-dimensional range and then intersects the ranges together before continuing with the original algorithm. Two part of the algorithm requires further observation.

Computing the intersection

Before trying to compute any kind of intersection, we can first state the following claim about the result of the intersections.

Claim 4 *The intersection of the two-dimensional ranges* $[a_1, b_1] \times [0, h_1], [a_2, b_2] \times [0, h_2], \dots, [a_l, b_l] \times [0, h_l]$ *with* $0 \leq a_i \leq b_i \leq n$ *and* $0 \leq h_i \leq n$ *for* $1 \leq i \leq l$ *is:*

$$\begin{cases} \left[\max_{1 \leq i \leq l} a_i, \min_{1 \leq i \leq l} b_i\right] \times \left[0, \min_{1 \leq i \leq l} h_i\right] & \text{if } \max_{1 \leq i \leq l} a_i \leq \min_{1 \leq i \leq l} b_i \\ \emptyset & \text{otherwise.} \end{cases}$$

With this in mind, we can simply find 2 minimums (one for the b_i and one for the h_i) and one maximum (for the a_i) to compute the intersection.

Continuing the algorithm

In order to continue the algorithm, we must have a single range of the form $[a, b] \times [0, h]$. The previous claim gives us that the intersection has the correct form, and we can therefore continue the algorithm without changing anything.

We denote the range of the intersection by $[a', b'] \times [0, h']$. Since this new range has the same form as the one used by the algorithm, we can simply run the remaining of the algorithm with this new range instead. This reduce the cost of the merge of the $|W|$ results.

Analysis

Finding the locus of each word still take $O\left(\sum_{w \in W} |w|\right)$ time. Then we must find the minimums and the maximums in $|W|$ time and finally the last part of the algorithm still take $O\left(\sum_{w \in W} |w| + k\right)$. We then slightly decrease the bound from the naive approach to $O\left(\sum_{w \in W} |w| + |W| + k\right)$. Also, since each word of W contains at least one character, the term $|W|$ is totally dominated by the summation. Finally, if W_{string} is the complete query interpreted as one exact string, then we have $O\left(\sum_{w \in W} |w| + k\right) \subseteq O\left(|W_{string}| + k\right)$. Thus, the bag-of-words query model is not harder than exact string matching with the suffix tree approach.

Top-k completion

One common feature that is popular in today's application is auto-completion. We want the software to tell us what words to use in order to find the result we are looking for. Usually, the user has a vague idea of the word that may be useful for the search but the best word to use comes from the corpus itself. It makes perfect sense to let the software recommend us with the best words that matches our partial query based on the corpus. The problem is common for search engines, soft keyboard on phones and tablets, text editor, etc... We give the following problem statement.

Definition Given a string $p \in \Sigma^*$ and an integer k, a top-k completion query returns the k highest occurrences that starts with p in the collection $\mathcal{D}$ according to a weight function $w(p, d)$ for p in document d.

Here, the collection $\mathcal{D}$ could represent a dictionary if we want top-k completion based on English words or any other collection of documents. Of course, we would have to adjust the weight function $w(p, d)$ to best match the use case of the problem. In the case of an English dictionary, the corpus is basically static. The data structure shown for top-k retrieval thus becomes a good candidate for this problem. There is however one big difference between this problem and the document listing problem. In this problem, we care about the precise occurrences of the query and not only whether it is included in the document or not. To have the best results, we expect to report only the leaves of the subtree of the locus node which corresponds to reporting the leaves of the three-sided range $[a, b] \times [0, \infty)$. Note that it is important to filter internal nodes in the occurrence listing problem, otherwise we could have multiple partial suffix matching the same occurrence. Since we do not need to represent the internal nodes in the grid, we only map the leaves on the grid while still keeping the lowest numbering and highest numbering references, $[l_v, h_v]$, in each internal nodes v with the modified numbering which is only attributed to leaves.

Once the points to consider have been reported, we can apply the rest of the algorithm (thus skipping the RMQ) and sorting the points. The modified step is equivalent to answering a $[a, b] \times [0, \max_v depth(v)]$ and therefore follows the same analysis as the paper reviewed.

The modified algorithm now returns the best k occurrences given a query pattern under a weight function $w(\cdot, \cdot)$. Each of these occurrences are different and can be easily used for auto-completion which concludes the application of the original algorithm.

Conclusions

To conclude, our review of the paper from Navarro and Nekrich, we provided an overview of the 2012 paper, we presented a way to support queries in the bag-of-words model and we provided an application to the top-k completion problem which correspond to the occurrence listing problem in multiple documents. We showed in section 2 that the complexity of bag-of-words queries on the suffix tree and the two-dimensional grid is the same as the provided algorithm for exact queries. In section 3 we showed how to answer the occurrence listing problem by modifying the two-dimensional grid and using the same theorems as the original paper. We can see by both proposed result that the suffix tree based data structure can be applied to various problem. We didn't explicitly look at the dynamic variant of the data structure[7], but we have good reason to think that we could obtain the same results as the two provided.

References

[1] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 993–1002, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0757-4. doi:10.1145/2009916.2010048. URL `http://doi.acm.org/10.1145/2009916.2010048`.

[2] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling

and related techniques for geometry problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 135–143, New York, NY, USA, 1984. ACM. ISBN 0-89791-133-4. doi:10.1145/800057.808675. URL `http://doi.acm.org/10.1145/800057.808675`.

[3] Wing-Kai Hon, R. Shah, and J.S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Foundations of Computer Science, 2009. FOCS '09. 50th Annual IEEE Symposium on*, pages 713–722, Oct 2009. doi:10.1109/FOCS.2009.19.

[4] Roberto Konow and Gonzalo Navarro. Faster compact top-k document retrieval. *CoRR*, abs/1211.5353, 2012. URL `http://arxiv.org/abs/1211.5353`.

[5] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 657–666, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X. URL `http://dl.acm.org/citation.cfm?id=545381.545469`.

[6] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1066–1077. SIAM, 2012. URL `http://dl.acm.org/citation.cfm?id=2095116.2095200`.

[7] Gonzalo Navarro and Yakov Nekrich. Optimal top-k document retrieval. *CoRR*, abs/1307.6789, 2013. URL `http://arxiv.org/abs/1307.6789`.

[8] Gonzalo Navarro and SharmaV. Thankachan. Top-k document retrieval in compact space and near-optimal time. In Leizhen Cai, Siu-Wing Cheng, and Tak-Wah Lam, editors, *Algorithms and Computation*, volume 8283 of *Lecture Notes in Computer Science*, pages 394–404. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45029-7. doi:10.1007/978-3-642-45030-3_37. URL `http://dx.doi.org/10.1007/978-3-642-45030-3_37`.