A segment tree based Top-k RMQ algorithm and its application to the autocomplete problem

Mikhail S. Dvoretckii
MSc Program Student
Lomonosov Moscow State University;
Programmer, IQ Systems LLC
Address: 1, Leninskie Gory, Moscow, 119991, Russian Federation
E-mail: mike.dvorecky@gmail.com

Abstract

An important way of ensuring data quality is controlling data input. One of the methods of doing that is checking the input data against the corresponding reference data where applicable. This may be done via autocomplete. Since reference data is usually stored in a centralized fashion, autocomplete algorithms usually run in client-server architectures and face strict time requirements.

In this article, a new autocomplete task decomposition is formulated using an existing method based on range minimum queries (RMQ). The Top-k RMQ problem is formulated and used in the autocomplete problem decomposition. A segment tree based algorithm is proposed for the Top-k RMQ problem. While the conventional segment tree based RMQ algorithm when used in autocomplete (in the Top-k RMQ subproblem) repeatedly processes the same nodes on the tree, the proposed algorithm is adapted directly to the Top-k RMQ problem and does not require any node of the segment tree to be processed more than twice. A complexity analysis is made for both the new Top-k RMQ algorithm and the conventional segment tree-based RMQ approach. This analysis considers different implementations of priority queues used in these algorithms, specifically binary heaps and ordered arrays. The new algorithm has time complexity that is not lower than that of the conventional algorithms with any priority queue implementation.

To prove the practical value of the new algorithm, a series of experiments was conducted using the data from the All-Russian Classifier of Addresses – a practical source of reference data for Russian address inputs. The new algorithm demonstrates better time efficiency than the conventional one in all experiments with all priority queue implementations.

Key words: All-Russian Classifier of Addresses, autocomplete, segment tree, range minimum query (RMQ), top *k* range minimum query (Top-k RMQ), algorithm.

Citation: Dvoretckii M.S. (2017) A segment tree based Top-k RMQ algorithm and its application to the autocomplete problem. *Business Informatics*, no. 1 (39), pp. 48–54. DOI: 10.17323/1998-0663.2017.1.48.54.

Introduction

Business applications are constantly dealing with master data, i.e. data containing key business information not associated with specific business transactions [1]. Master data can be data on people, organizations, production processes, etc. For master data processing, it is important to maintain its uniformity in order to avoid generation of multiple records corresponding to similar things. To do this, master data is most often subject to validation according to various rules and is brought to some standard form that enables us to maintain quality of this data and exclude creation of duplicate records. On frequent occasions, this standard form in turn relies on some reference data. For exam-

ple, for information on addresses in Russia, the standard is the All-Russian Classifier of Addresses (ARCA) [2], namely, a hierarchical directory correlating all address objects with positional numerical codes which uniquely represent address objects to a street level. A standardized line interpretation of the address also corresponds to this.

Operators entering master data into the database possibly do not know its canonical form. The best solution in this case is to issue variants of possible input information during the entry using autocomplete algorithms. Since the reference data management is ideally centralized, the programs which validate the data corresponding to the reference data and issue prompts for them, operate in a client-server mode. In this connection, the algorithm developed on the server side must be very fast, since an apparent delay in obtaining prompts reduces the convenience and performance of the entry, and the server itself must simultaneously solve a variety of problems for the associated workstations.

This article proposes that we modify the existing approach to autocomplete algorithms based on the decomposition of the autocomplete problem into binary search and range minimum query. A new statement is introduced for the range minimum query problem, and its solution algorithm is proposed using the segment tree. The algorithm efficiency is proved by its comparison with the existing autocomplete algorithm using the segment tree for the Range Minimum Query using data from ARCA.

1. Autocomplete problem

The term "autocomplete" is taken to mean a set of problems which can be extended to the following content-related statement: the user request is an incomplete form of some line (or lines) from an a-priori known applicant list; appropriate lines are issued from the applicant list in a certain form to the user, moreover, preferably the one (those) which the user originally wanted to receive. Algorithms solving autocomplete problems are used in search systems (to provide the user with the most frequently encountered or most anticipated queries), in information retrieval systems (to offer variants of responses to the user while the query is formed) and in mobile devices (to increase typing speed on touch keyboards). Autocomplete is also used when entering data into databases, to maintain uniformity of similar data in accordance with the reference data and prevent the occurrence of errors and duplicates.

Currently research in the autocomplete area is primarily aimed at development of effective error-tolerant autocomplete algorithms. However, for many applications of the autocomplete problem, including corresponding reference data to monitor the data input, it is quite enough to solve it in a classical prefix statement. Stringent response time requirements are generally imposed on the autocomplete algorithms. Ideally, the user should not notice a pause between the query input and response output.

2. Statement of the autocomplete problem

Let us provide a formal statement of the autocomplete problem by prefix.

Finite aggregate W of the finite lines above finite alphabet Σ is given:

$$W = \left\{ w_i \mid w_i \in \Sigma^*, i = \overline{0, n-1} \right\},$$

on which weighting function $f(w): W \to \mathbf{R}$ is determined. For finite query $q \in \Sigma^*$ let us write down a set of its extensions from a number of lines in W via $Q \subseteq W$:

$$Q = \left\{ w_i \mid w_i \in W, \exists v_i \in \Sigma^* : w_i = q v_i \right\}.$$

Put another way, Q represents a set of lines from W, for which q is a common prefix.

We need to find such a subset of extensions $R_k \subseteq Q$, which includes $k \in \mathbf{N}$ lines from Q, having the highest weight. In cases where $|Q| \le k$ the task is trivial, and we assume that $R_k = Q$. If $|Q| > k$, then let us introduce set B_k of all subsets of Q, having power k in consideration:

$$B_k = \left\{ B \mid B \subset Q, |B| = k \right\}.$$

For sets B from B_k let us determine price functional $C(\cdot): B_k \to \mathbf{R}$:

$$C(B) = \sum_{w \in B} f(w).$$

Then the problem consists in finding such a subset B^*, which maximizes the price functional:

$$R_k = B^* = \arg\max_{B \in B_k} C(B).$$

3. Current approaches to solving the autocomplete problem

While the more popular methods of solving the autocomplete problem are based on the use of prefix trees for fast-access retrieval of results, and also use caching responses to provide prompt query handling through additional memory requirements, one of the approaches to solving the autocomplete problem with low memory requirements set forth in [3] is decomposition of this

problem into the problem of binary search by the direct index and Range Minimum Query (RMQ) in the maximization variant.

The range minimum query (RMQ) is a problem of finding a minimum element on a subset of consecutive array elements of comparable data structures. This problem is thoroughly studied primarily due to the fact that the problem of finding the lowest/least common ancestor (LCA) [4] reduces to it. Paper [5] describes some other applications of this problem, and also proposes its solution algorithm with time complexity $O(1)$ and space complexity $O(n)$ and proves its memory optimality with requirement $O(1)$ for the time complexity.

In order to solve the autocomplete problem, the RMQ problem is solved repeatedly in the maximization variant in order to obtain k responses with a maximum weight from a number of variants suitable by prefix. Moreover, multiple RMQ-queries are made to the segments including each other.

Algorithm 1 given below describes a universal plan of using the existing algorithms for solving the autocomplete problem using RMQ. Here W is a direct index, T is a RMQ-structure on the weight array of the variants, q is a query-prefix, k is the number of expected results, pq is a priority queue [6] of the segments in terms of the maximum weight of the variants thereon. In so doing, numerous RMQ-queries by segments including each other are performed, possibly leading to a large number of recurring actions.

Autocomplete_Algorithm 1 (W, T, q, k | Rk)
l, r ← Binary_search(W, q)
If Incorrect(l, r)
    Return
range ← Segment (l, r, RMQ(T, l, r))
pq ← Ø
pq.Add*(range)*
i ← 0
While i < k and no *pq*.Empty
    range ← *pq*.GetTop()
    Rk.Add(*range.maxpos*)
    If (*range.maxpos* != *range.l*)
        *range*1 ← Segment(*range.l*, *range.maxpos* – 1, RMQ(*T*, *range.l*, *range.maxpos* – 1))
        pq.Add(*range*1)
    If (*range.maxpos* != *range.r*)
        *range*2 ← Segment(*range.maxpos* + 1, *range.r*, RMQ(*T*, *range.maxpos* + 1, *range.r*))
        pq.Add(*range*2)
    i ← i + 1
End of While Cycle
End of Algorithm.

4. Approach to solving the autocomplete problem based on top k range minimum query (Top-k RMQ)

The problem of finding the top-k range minimum queries (Top-k RMQ) is set up in order to find a more effective algorithm for solving the autocomplete problem when using a similar decomposition principle, i.e. decomposition of the autocomplete problem into binary search problem and Top-k RMQ.

Therefore, the following statement is proposed. Let there be given a heap of numbers $A, |A| = n$, with elements $a_i, i = \overline{0, n-1}$, and two indexes l and r, $0 \le l \le r < n$.

For case $k > r - l$ the Top-k RMQ problem is solved by a set of all array A indexes in segment $[l, r] - R_k = \{l, ..., r\}$.

In case $k \le r - l$ the solution is a subset of indexing set $R_k \subset \{l, ..., r\}$ of power k, for which a sum of array elements A is minimum:

$$B_k = \{B \mid B \subset \{l, ..., r\}, |B| = k\},$$

$$C(B) = \sum_{i \in B} a_i,$$

$$R_k = B^* = \arg\min_{B \in B_k} C(B).$$

As in the case of the classical RMQ problem, the statement of the Top-k RMQ problem can be changed to the problem of finding a subset with a maximum sum without a structural change of the algorithm.

In this case, the autocomplete problem is decomposed by the method given in algorithm 2. By comparison, algorithm 1 can be considered a reduction of Top-k RMQ to RMQ.

Autocomplete_Algorithm 2(W, T, q, k | Rk)
l, r ← Binary_search(W, q)
If Incorrect(l, r)
    Return
Rk ← TopkRMQ(*T*, *l*, *r*, *k*)
End of Algorithm.

5. Original algorithm based on segment trees

The proposed algorithm is a modification of the variant of algorithm 1 which uses the segment tree [7] as the RMQ structure. This RMQ solution algorithm is not asymptotically the best one (query complexity $O(\ln n)$ for asymptotically better $O(1)$), but it is actively used for the

autocomplete problem [8]. When using the segment tree as RMQ structure for the autocomplete problem, multiple RMQ queries for segments including each other result in a multiple rescanning of the same nodes at high tree levels. Due to changing the problem statement from pure RMQ to Top-k RMQ, these extra actions can be excluded.

Algorithm 3 uses a segment tree to solve the Top-k RMQ problem in the maximization variant. In this case, not sub-segments, on which each next maximum is present (as in Algorithm 1), but nodes deriving from the way leading from the initial vertex to the maximum are put on the priority queue. In so doing, the segment tree property is used: the maximum in the internal vertex is always reached in at least one of its daughter vertices. When leaves are included in the segment tree 1, this property makes it possible to find the way from any vertex to the maximum in the segment corresponding to it.

Algorithm 3. Top-k RMQ based on the range tree

```
Find_beginning(T, l, r, pq)
If T.start > r or T.end < l
        Return
If T.start ≥ l and T.end ≤ r
        pq.Add(T)
Else
        Find_beginning(T.left, l, r, pq)
        Find_beginning(T.right, l, r, pq)

TopkRMQ(T, l, r, k | Rk)
pq ← Ø
Find_beginning(T, l, r, pq)
i ← 0
While i < k and no pq.Empty
        CurT ← pq.GetTop()
        While (CurT.start != CurT.end)
                If (CurT.left.max = CurT.max)
                        pq.Add(CurT.right)
                        CurT ← CurT.left
                Else
                        pq.Add(CurT.left)
                        CurT ← CurT.right
        End of While Cycle
        Rk.Add(CurT.start)
        i ← i + 1
End of While Cycle
End of Algorithm.
```

Contrary to algorithm 1, before the start of the solution search cycle, not one structure corresponding to the segment, but a set of segment tree vertices covering the segment at the highest level are put on the priority queue. In this case, a leaf corresponding to this value always exists for the vertex with the highest value, and instead of storing its location in the tree nodes, algorithm 3 provides for getting off to this leaf while putting all unconsidered vertices on the priority queue.

Keep in mind that as opposed to algorithm 1, where an RMQ query should have already been performed to put the segment on the priority queue, the new algorithm does not require any additional processing for the segment tree nodes. Furthermore, any segment tree node is considered no more than twice - when putting on the queue and selecting from it, while using the segment tree in algorithm 1 the root of this tree will be considered k times, if the segment size is no less than k.

6. Complexity assessment of the proposed algorithm

By analogy with [3], let us define the algorithm complexity with symbol $\hat{O}$ with the assumption of complexity of operations with priority queue $O(1)$. Then excluding the binary search of the complexity of executing Top-k RMQ query per the scheme set forth in algorithm 1 using a standard algorithm RMQ on the segment tree is $\hat{O}(k \ln n)$, since on each of k iterations no more than two RMQ queries are executed with complexity $O(\ln n)$. Algorithm 3 also has complexity $\hat{O}(k \ln n)$, inasmuch as on k iterations a descent from the initial vertex to the tree leaf of $\lceil \log_2 n \rceil + 1$ high is executed .

Let us assume that the priority queue is implemented using the binary heap [6]. Then the operations of adding elements to the queue and reaching the maximum have complexity $O(\ln l)$, where l is the queue length. Therefore, the computational complexity of executing Top-k RMQ by a classical method and algorithm 3 becomes equal to $O(k \ln k + k \ln n)$ and $O(k \ln n \cdot \ln(k \ln n))$, respectively. The highest asymptotic complexity of algorithm 3 is caused by the fact that in each internal vertex under consideration the daughter vertex not lying on the way to the maximum is put on the priority queue; that leads to overhead costs absent in algorithm 1, where in each of k steps no more than two segments are put on the priority queue.

It should by also noted that for the binary heap there is no reliable method of limiting its permissible dimension, as its structure prevents us from determining its minimum element faster than during operations $O(l)$. However, no more than k elements, one per step,

are to be extracted from the priority queue for both solution algorithms to solve Top-k RMQ problem. In this case, it is handier to use a simpler variant of the priority queue, namely ordered length k array. In this case, the operation of extracting the maximum is trivial: the shift is not required because of the assumption that the number of elements retrieved is limited, so that you can leave the maximum in place and move the top of the queue. An element is added with a linear insertion with complexity $O(k)$, and elements that are known to be out completely miss the queue (the best case). In this case, the complexities of the classical algorithm and algorithm 3 as a whole become equal to $O\left(k^2 + k \ln n\right)$ and $O\left(k^2 \ln n\right)$, respectively. For Algorithm 3, this also enables us to reduce the amount of memory allocated to the priority queue. While the classical approach leads to the fact that at any step there are no more $k+1$ segments in the priority queue, for algorithm 3 in the worst case the number of vertices put on the queue is limited from above $\lceil (k+2)\log_2 n \rceil$ (up to $2\log_2 n$ initial vertices, up to $\log_2 n$ vertices are added on each of k descents).

Note that the algorithm given above for using the array for the priority queue is not universal, since it uses a priori information on how many elements will be extracted from the queue. In general, the priority array-based queue is to be either shifted when retrieving an element (complexity $O(l)$) or shall use a circular structure (which formally limits a number of items which can be stored in the queue or complicates the expansion operation).

For small sizes of entries encountered in practice, the greater asymptotic complexity is not an evidence of lack of practicality of the algorithm. To prove the practical value of a new algorithm and the validity of the new problem statement, let us perform an experimental comparison of the considered algorithms as components of a real-world autocomplete problem.

7. Experimental results

The experimental comparison of algorithms was carried out within the programs that solve the autocomplete problem of reference address lines from the All-Russian Classifier of Addresses (ARCA) [2], under the following conditions:

✦ algorithms were software-implemented in C language and collected by compiler gcc5 using operating system Ubuntu 16.04 LTS with flag -lm;

✦ all components of the autocomplete algorithm, having nothing to do with Top-k RMQ problem are similarly implemented for both algorithms in both options;

✦ the time of solution of the autocomplete problem is measured from the time of receiving the query to the time of issuing responses;

✦ results were measured using the function clock_gettime() and high-resolution timer CLOCK_PROCESS_CPUTIME_ID (specified resolution is 1 ns);

✦ unloading of ARCA, including 1 222 662 address lines as a set of reference lines was used. The weight was calculated as the sum of a number of subordinated address elements and weights of upper elements with attenuation factors $0{,}1^t$, where t is the difference in levels between the current and the upper elements;

✦ a definite number of initial symbols from sequences of randomly selected lines from the reference set similar for both algorithms was taken as queries;

✦ $k = 10$ was taken in all experiments;

✦ All experiments were conducted using OS Ubuntu 16.04 LTS in sequence;

✦ Computer hardware parameters:

✧ Processor Intel Core i3-4010U, 1.7 GHz

✧ 4 GB RAM

Table 1 provides the experiment results. Note that the variants of algorithms with the priority queue in the form of an ordered array in all experiments proved to be more effective variants with a binary heap, and the new algorithm variants showed better results as compared to the variants of the classical approach.

Conclusion

The classical statement of an RMQ problem provides for obtaining only one minimum (maximum) value from the segment, and this is reflected by the existing algorithms for solution of the problem. For cases where it is necessary to extract more than one extreme value, transfer to a wider setting of Top-k RMQ makes it possible not only to more clearly extract this subproblem, but also to propose new algorithms for its solution, initially aimed at obtaining a set of results. Due to this, a new algorithm is created within the real autocomplete program that gives better results as compared to the algorithm created by the classical method based on the same data structure. ■

Table 1.

Experimental results of implementations of algorithms of solving autocomplete problems

Number of queries	Classical algorithm, binary heap, s	Classical algorithm, ordered array, s	New algorithm, binary heap, s	New algorithm, ordered array, s
Queries of length 4				
1000	0.017555518	0.015670170	0.005854641	0.003890637
10000	0.175507172	0.160802266	0.057827076	0.038593355
100000	1.748508783	1.575862406	0.578767750	0.389285590
1000000	17.494891590	15.630793271	5.787819119	3.892264716
Queries of length 10				
1000	0.017930115	0.016084878	0.006358041	0.004443302
10000	0.179062306	0.160433853	0.062990764	0.044009557
100000	1.786391754	1.604154774	0.630281282	0.440086321
1000000	17.832476482	15.973139661	6.303791885	4.402448944

References

1. Pavlyuts A. (2016) *Master-dannye i upravlenie imi. Chto eto takoe i komu ono neobkhodimo?* [Master data and master data management. What is it and who needs it?]. Available at: https://iqsystems.ru/master-data-basics-article/ (accessed 13 January 2017) (in Russian).
2. *Klassifikator adresov Rossii (KLADR)* [All-Russian Classifier of Addresses (ARCA)]. Available at: https://www.gnivc.ru/inf_provision/classifiers_reference/kladr/ (accessed 13 January 2017) (in Russian).
3. Hsu B.-J., Ottaviano J. (2013) Space-efficient data structures for Top-k completion. Proceedings of the *22nd International World Wide Web Conference. Rio de Janeiro, Brazil, 13–17 May 2013.* NY: ACM, pp. 583–594.
4. Fischer J., Heun V. (2006) Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. Proceedings of the *17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006), Barcelona, Spain, 5–7 July 2006.* Berlin, Heidelberg: Springer, pp. 36–48.
5. Fischer J., Heun V. (2007) A new succinct representation of RMQ-information and improvements in the enhanced suffix array. *Proceedings of the First International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies* (ESCAPE 2007). Hangzhou, China, 7–9 April 2007. Berlin, Heidelberg: Springer, pp. 459–470.
6. Knuth D. (1998) *The art of computer programming.* Vol. 3: Sorting and searching. Boston: Addison-Wesley Professional.
7. Matani D. (2011) *An O(klogn) algorithm for prefix-based ranked autocomplete.* Available at: http://www.dhruvbird.com/autocomplete.pdf (accessed 12 May 2016).
8. *lib-face on GitHub*. Available at: http://github.com/duckduckgo/cpp-libface (accessed 13 January 2017).

Алгоритм нахождения *k* наименьших элементов на отрезке (Top-k RMQ) на основе дерева отрезков и его применение в задачах автодополнения

М.С. Дворецкий

студент магистратуры
Московский государственный университет им. М.В. Ломоносова;
программист, ООО «IQ Systems»
Адрес: 119991, г. Москва, Ленинские горы, д. 1
E-mail: mike.dvorecky@gmail.com

Аннотация

Контроль данных при вводе является важным способом обеспечения их качества. Одним из методов такого контроля является сопоставление вводимых данных, которые должны соответствовать справочной информации, непосредственно с этой информацией в процессе ввода. Это приводит к необходимости решения задачи автодополнения. Поскольку справочная информация, как правило, хранится централизованно, задача автодополнения решается в архитектуре клиент-сервер и к алгоритму ее решения предъявляются жесткие требования по быстродействию.

В данной статье на основе существующей декомпозиции задачи автодополнения с использованием задачи поиска минимума на отрезке (RMQ) формулируется задача поиска k минимумов на отрезке (Top-k RMQ) и приводится алгоритм ее решения, использующий дерево отрезков. В то время как классический алгоритм RMQ по дереву отрезков при использовании в задаче автодополнения (в подзадаче Top-k RMQ) требует многократного посещения вершин дерева, близких к корню, предложенный алгоритм Top-k RMQ непосредственно адаптирован к этой задаче и не требует рассмотрения какой-либо вершины дерева отрезков более двух раз. Выполнен анализ сложности как алгоритма Top-k RMQ, так и классического алгоритма RMQ с использованием дерева отрезков. При этом учитываются различные варианты реализации приоритетных очередей, используемых в этих алгоритмах, а именно вариант двоичной кучи и простая приоритетная очередь на основе упорядоченного массива. Новый алгоритм имеет не меньшую вычислительную сложность, чем классический, при любой реализации приоритетной очереди.

Для доказательства ценности нового алгоритма произведено экспериментальное сравнение алгоритмов с использованием данных из Классификатора адресов России, представляющего собой реальный примера справочной информации. Во всех проведенных экспериментах новый алгоритм показал лучшие результаты по времени по сравнению с классическим.

Ключевые слова: Классификатор адресов России, автодополнение, дерево отрезков, поиск минимума на отрезке (RMQ), поиск k наименьших элементов на отрезке (Top-k RMQ), алгоритм.

Цитирование: Dvoretckii M.S. A segment tree based Top-k RMQ algorithm and its application to the autocomplete problem // Business Informatics. 2017. No. 1 (39). P. 48–54. DOI: 10.17323/1998-0663.2017.1.48.54.

Литература

1. Павлюц А. Мастер-данные и управление ими. Что это такое и кому оно необходимо? [Электронный ресурс]: https://iqsystems.ru/master-data-basics-article/ (дата обращения 13.01.2017).
2. Классификатор адресов России (КЛАДР) // ГНИВЦ. [Электронный ресурс]: https://www.gnivc.ru/inf_provision/classifiers_reference/kladr/ (дата обращения 13.01.2017).
3. Hsu B.-J., Ottaviano J. Space-efficient data structures for Top-k completion // Proceedings of the 22nd International World Wide Web Conference. Rio de Janeiro, Brazil, 13–17 May 2013. NY: ACM, 2013. P. 583–594.
4. Fischer J., Heun V. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE // Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006), Barcelona, Spain, 5–7 July 2006. Berlin, Heidelberg: Springer, 2006. P. 36–48.
5. Fischer J., Heun V. A new succinct representation of RMQ-information and improvements in the enhanced suffix array // Proceedings of the First International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE 2007). Hangzhou, China, 7–9 April 2007. Berlin, Heidelberg: Springer, 2007. P. 459–470.
6. Knuth D. (1998) The art of computer programming. Vol. 3: Sorting and searching. Boston: Addison-Wesley Professional, 1998.
7. Matani D. An $O(k \ln n)$ algorithm for prefix-based ranked autocomplete. [Электронный ресурс]: http://www.dhruvbird.com/autocomplete.pdf (дата обращения 12.05.2016).
8. lib-face на GitHub. [Электронный ресурс]: http://github.com/duckduckgo/cpp-libface (дата обращения 13.01.2017).