APPROXIMATE RANGE MODE AND RANGE MEDIAN QUERIES

Prosenjit Bose[*] Evangelos Kranakis[*] Pat Morin[*] Yihui Tang[†]
School of Computer Science, Carleton University
5302 Herzberg Building
1125 Colonel By Drive
Ottawa, Ontario, K1S 5B6, Canada
{jit,kranakis,morin,y_tang}@scs.carleton.ca

ABSTRACT. Mode and median are two of the most important statistics we use when we analyze data. In this paper, we consider data structures and algorithms for preprocessing a labelled list of length n so that, for any given i and j we can answer queries of the form: What is the mode or median label in the sequence of labels between indices i and j. Our results are on an approximate version of this problem. Using $O(n/(1-\alpha))$ space, our data structure can find in $O(\log\log_{\frac{1}{\alpha}} n)$ time an element whose number of occurrences is at least α times of that of the mode, for some user-specified parameter $0 < \alpha < 1$. Data structures are proposed to achieve constant query time for $\alpha = 1/2, 1/3$ and $1/4$, using storage space of $n \log n, n \log\log n$ and n, respectively. We also show that if the elements are comparable, an $O(n/(1-\alpha))$ space, $O(1)$ query time data structure can answer range median queries with a guaranteed accuracy of $\alpha \times \lfloor |j-i+1|/2 \rfloor$.

1 Introduction

Let $A = a_1, \ldots, a_n$ be a list of elements of some data type. We wish to construct data structures on A, such that we can quickly answer *range queries*. These queries take two indices i, j with $1 \leq i \leq j \leq n$ and require computing $F(a_i, \ldots, a_j) = a_i \circ a_{i+1} \circ \ldots a_{j-1} \circ a_j$. If the inverse of the operation "$\circ$" exists, then range queries have a trivial solution of linear space and constant query time. The operation "$\circ$" being arithmetic addition (subtraction being its inverse) as an example, we precompute all the partial sums $b_i = a_1 + \ldots + a_i, i = 1, \ldots, n$, and the range query $F(a_i, \ldots, a_j) = a_i + \ldots + a_j$ can be answered in constant time by computing $b_j - b_{i-1}$. Yao [7] (see also Alon and Schieber [1]) showed that if "$\circ$" is a constant time semigroup operator for which no inverse operation is allowed, and $a \circ b$ can be computed in constant time then it is possible to answer range queries in $O(\lambda(k, n))$ time using a data structure of size $O(kn)$, for any integer $k \geq 1$. Here $\lambda(k, \cdot)$ is a slowly growing function at the $\lfloor k/2 \rfloor$-th level of the primitive recursive hierarchy, as a few of examples, $\lambda(2, n) = O(\log n)$, $\lambda(3, n) = O(\log\log n)$ and $\lambda(4, n) = O(\log^* n)$.

Krizanc *et al* [5] studied the storage space query time tradeoffs for range mode and range median queries. These occur when F is the function that returns the mode or median of its input. Given a set of n elements, a *mode* is an element that occurs at least as frequently as any other element of the set. If the elements are comparable (for example real numbers), the *median* is defined to be the element in position $\lceil n/2 \rceil$ in the sorted sequence of the input. Note the trivial solution does not work for range mode or range median queries as no inverse exists for either the mode or the median. Yao's approach does not apply either because neither range mode nor range median is associative and therefore not a

[*]Supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) and MITACS (Mathematics of Information Technology and Complex Systems) grants.
[†]Supported in part by Ontario Graduate Scholarship.

semigroup query. Also, given two sets S_1 and S_2 and their modes (medians), the mode (median) of the union $S_1 \bigcup S_2$ cannot be computed in constant time. New data structures are needed for range mode and range median queries. Krizanc *et al* [5] gave a data structure of size $O(n^{2-\epsilon})$ that can answer range mode queries in $O(n^\epsilon \log n)$ time, where $0 < \epsilon \leq 1/2$ is a constant representing space-time tradeoff. For range median queries, it is shown in [5] that a data structure of size $O(n)$ can answer range median queries in $O(n^\epsilon)$ time and a faster $O(\log n)$ query time can be achieved using $O(n \log^2 n / \log\log n)$ space.

In this paper we consider the approximate version of range mode and range median queries. We show that if a small error is tolerable, range mode and range median queries can be answered much more efficiently in terms of storage space and query time. Given a sequence $S = a_i, a_{i+1}, \ldots, a_j$, an element is said to be an *approximate mode* of S if its number of occurrences is at least α times that of the actual mode of S, where $0 < \alpha < 1$ is a user-specified approximate factor. An *approximate median* of S is an element whose rank is between $\alpha \times \lfloor |j - i + 1|/2 \rfloor$ and $(2 - \alpha) \times \lfloor |j - i + 1|/2 \rfloor$. Clearly, there could be several approximate medians.

Table 1 summarizes the main results of this paper. We show that with an error of at most $1 - \alpha$, range mode queries can be answered in $O(\log \log_{\frac{1}{\alpha}} n)$ time using a data structure of size $O(n)$. We also show that constant query time can be achieved for $\alpha = 1/2, 1/3$ and $1/4$ using $o(n^2)$ space. We introduce a constant query time data structure for answering approximate range median queries.

Approximate Range Mode Queries			
Preprocessing Time	Storage Space	Query Time	α
$O(n \log_{\frac{1}{\alpha}} n)$	$O(\frac{n}{1-\alpha})$	$O(\log \log_{\frac{1}{\alpha}} n)$	$0 < \alpha < 1$
$O(n \log n)$	$O(n \log n)$	$O(1)$	$1/2$
$O(n \log n)$	$O(n \log \log n)$	$O(1)$	$1/3$
$O(n \log n)$	$O(n)$	$O(1)$	$1/4$

Approximate Range Median Queries			
Preprocessing Time	Storage Space	Query Time	α
$O(\frac{n \log n}{1-\alpha})$	$O(\frac{n}{1-\alpha})$	$O(1)$	$0 < \alpha < 1$

Table 1: Summary of results in this paper

2 Approximate range mode queries

Given a list of elements $a_1, \ldots, a_n$ and an approximate factor $0 < \alpha < 1$, the *approximate range mode queries* can be specified formally as follows.
INPUT: Two indices i, j with $1 \leq i \leq j \leq n$.
OUTPUT: An element x in $a_i, \ldots, a_j$ such that $F_x(a_i, \ldots, a_j) \geq \alpha \times F(a_i, \ldots, a_j)$, where $F_x(a_i, \ldots, a_j)$ is the frequency[1] of x in $a_i, \ldots, a_j$ and $F(a_i, \ldots, a_j) = \max_x F_x(a_i, \ldots, a_j)$ is the number of occurrences of the mode of $a_i, \ldots, a_j$.

Our data structure is based on the observation that given a fix left end i of a query range, as the right end j of the range increases, the number of times the query answer has to change as j varies from i to n is $O(\log_{\frac{1}{\alpha}}(n - i))$. This is because we can output the same element as the approximate mode as long as no other element's frequency exceeds $1/\alpha$ times of that of the current approximate mode. When

[1]We use frequency and the number of occurrences interchangeably throughout the paper.

the actual mode's frequency has exceeded $1/\alpha$ times of that of the approximate mode, the approximate mode has to be replaced and the new approximate mode is the actual mode.

For example, given the list of 20 elements shown in Figure 1 and approximate factor $\alpha = 1/2$, b is an approximate mode of $a_1, \ldots, a_9$ because b occurs 2 times in the query range, while the actual mode, a, occurs 4 times in the same query range. But this is no longer true for query $a_1, \ldots, a_{10}$, as the number of b's occurrences is still 2 while the actual mode, a occurs 5 times ($F_a(a_1, \ldots, a_{10}) = 5$). In this case, either a or c ($F_c(a_1, \ldots, a_{10}) = 3$) is a valid approximate mode.

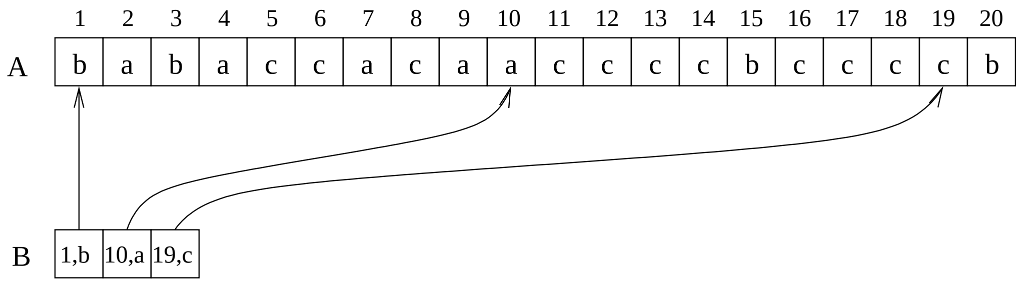

Figure 1: $\alpha = 1/2$. A lookup table of size 3 is used for answering queries $a_1, \ldots, a_j$, $j = 1, \ldots, 20$. For example, a is an approximate mode of $a_1, \ldots, a_{15}$ because a occurs at least 5 times in the query range ($F_a(a_1, \ldots, a_{15}) = 5$) while no other element occurs more than 10 times until $j = 19$ ($F_c(a_1, \ldots, a_{19}) = 11$).

Suppose a is chosen to be the new approximate mode, it remains an approximate mode as the right end of the query range $(i.e., j)$ increases till $j = 19$ at which point the actual mode, c occurs 11 times ($F_c(a_1, \ldots, a_{19}) = 11$). Since no other element (a or b) occurs more than or equal to half of the actual mode ($F_a(a_1, \ldots, a_{19}) = 5, F_b(a_1, \ldots, a_{19}) = 3$), c is now the only approximate mode. Since an approximate mode remains so till another element occurs more than $1/\alpha$ times the current approximate mode, the number of approximate modes that have to be stored is much less than the number of elements of the original list. As an example shown in Figure 1, instead of storing the complete original array of 20 elements, a table of 3 approximate modes is used to answer all approximate range mode queries $a_1, \ldots, a_j, 1 \leq j \leq 20$.

Given an approximate factor α, all approximate range mode queries with a_1 being the left end: $a_1, \ldots, a_j$ $(1 \leq j \leq n)$ can be answered using $O(\log_{\frac{1}{\alpha}} n)$ storage space. The data structure is a lookup table $B = a_{c_1}, \ldots, a_{c_L}(1 \leq c_1 < c_2 < \ldots < c_L \leq n)$ in which we store L approximate modes. The first entry is always a_1 ($c_1 = 1$). The second entry a_{c_2} is the first element in A that occurs $\lceil 1/\alpha \rceil$ times, *i.e.*, $F_{a_{c_2}}(a_1, \ldots, a_{c_2}) = \lceil 1/\alpha \rceil$ and $F_{a_{c_2}}(a_1, \ldots, a_{c_2}) > F_{a_i}(a_1, \ldots, a_{c_2})$ for $\forall i \neq c_2$. In general, the kth entry in the table is the first element in A that occurs $\lceil 1/\alpha^{k-1} \rceil$ times as the right end of the query range increases. Note that a_{c_k} is an approximate mode of $a_1, \ldots, a_j$ for any $c_k \leq j < c_{k+1}$ since a_{c_k} occurs at least $\lceil 1/\alpha^{k-1} \rceil$ times in $a_1, \ldots, a_j$ ($F_{a_{c_k}}(a_1, \ldots, a_j) \geq F_{a_{c_k}}(a_1, \ldots, a_{c_k}) = \lceil 1/\alpha^{k-1} \rceil$) while no other element occurs more than $1/\alpha^k$ times in the same range ($F_x(a_1, \ldots, a_j) < F_{c_{k+1}}(a_1, \ldots, a_{c_{k+1}}) = \lceil 1/\alpha^k \rceil$).

The last approximate mode in the table, a_{c_L}, occurs at least $\lceil 1/\alpha^{L-1} \rceil$ times in $a_1, \ldots, a_n$. It follows immediately that the number of approximate modes stored in the lookup table, L is at most $\log_{\frac{1}{\alpha}} n + 1$.

To answer approximate range mode query $a_1, \ldots, a_j$, binary search is used to find in $O(\log \log_{\frac{1}{\alpha}} n)$ time the largest c_k in B that is less than or equal to j and output a_{c_k} as the answer.

Lemma 1. *There is a data structure of size $O(\log_{\frac{1}{\alpha}} n)$ that can answer approximate range mode queries*

$a_1, \ldots, a_j$ $(1 \leq j \leq n)$ *in* $O(\log\log_{\frac{1}{\alpha}} n)$ *time.*

An immediate application of Lemma 1 is a data structure for answering approximate range mode queries with arbitrary end points. The data structure is a collection of n lookup tables $(T_i, i = 1, \ldots, n)$, one table for each left end point. An array of n pointers is used for locating a table in $O(1)$ time. A query $a_i, \ldots, a_j$ can be answered by first locating table T_i in $O(1)$ time, and then searching in T_i to find the approximate mode of $a_i, \ldots, a_j$, which takes $O(\log\log_{\frac{1}{\alpha}} n)$ time since T_i contains at most $O(\log_{\frac{1}{\alpha}}(n-i)) = O(\log_{\frac{1}{\alpha}} n)$ approximate modes.

Corollary 1. *There is a data structure of size* $O(n\log_{\frac{1}{\alpha}} n)$ *that can answer approximate range queries in* $O(\log\log_{\frac{1}{\alpha}} n)$ *time.*

2.1 An Improvement Based on Persistent Search Trees

We have seen that by maintaining a lookup table T_i of size $O(\log_{\frac{1}{\alpha}} n)$ for each left end point i $(1 \leq i \leq n)$ and using a total storage space of $O(n\log_{\frac{1}{\alpha}} n)$, any approximate range mode query $a_i, \ldots, a_j$ can be answered in $O(\log\log_{\frac{1}{\alpha}} n)$ time. Given a fixed left end point i, storing an answer for each right end point j is not necessary due to the fact the answer to the query changes less frequently as j varies. The approximate modes of two query ranges with adjacent right end points are unlikely to be different. In this section, we pursue this idea and show that storage of a complete lookup table for each left end point is not necessary because of the similarity between two tables with adjacent left end points.

To see how the approximate range mode changes gradually as the end points of query range moves, we need to keep track of the range within which the current approximate mode remains a valid approximation of the actual mode and its number of occurrences in that range. As the query range changes, the frequency of the current approximate mode may also change. Once it drops below a predetermined threshold value (f_{low}, the calculation of which will be discussed next), a new approximate mode is chosen and the query range updated.

As shown in Table 2, each entry in the lookup table is a 5-tuple $(f_{low_r}, f_{high_r}, q_r, ans_r, f_{ans_r})$. Given an approximate factor α, $[f_{low_r}, f_{high_r}]$ are precomputed for $r = 1, 2, \ldots, 2\lceil\log_{\frac{1}{\alpha}} n\rceil$ and remain the same for all tables.

Frequency Range	Query Range	Answer
...		
$[f_{low_r},\ f_{high_r}]$	q_r	(ans_r, f_{ans_r})
$[f_{low_{r+1}},\ f_{high_{r+1}}]$	q_{r+1}	$(ans_{r+1}, f_{ans_{r+1}})$
...		

Table 2: $f_{low_1} = 1$, $f_{high_1} = 1$, $f_{low_{r+1}} = f_{high_r} + 1$, $f_{high_{r+1}} = f_{low_r}/\alpha + 1$, $F(a_i, \ldots, a_{q_r}) = f_{high_r}$, $f_{ans_r} = F_{ans_r}(a_i, \ldots, a_{q_r})$, $f_{low_r} \leq f_{ans_r} \leq f_{high_r}$.

The ith table, T_i corresponds to all the range queries with the same left end i. A counter is set for each element to keep track of its frequency as the right end j varies. Given the fixed left end i, as the right end j proceeds, ans_r is the first element whose frequency in $a_i, \ldots, a_j$ reaches f_{high_r}, and q_{r+1} is the rightmost point up to which ans_r remains a valid approximate mode, *i.e.,* no other element has a frequency higher than f_{high_r}/α. Given a query $a_i, \ldots, a_j$ with $q_r \leq j < q_{r+1}$, ans_r is a valid approximate mode since its frequency is at least f_{high_r} while no other element has a frequency higher than $f_{high_{r+1}} - 1 = f_{low_r}/\alpha$. To see how the subsequent tables are built based on T_i with minimum number of changes, the right end of the query range is fixed, as the left end of the query range proceeds, ans_r's frequency may decrease, but it remains a valid approximate mode as long as $f_{ans_r} \geq f_{low_r}$ and it is copied to the next table along with a possibly modified f_{ans_r} (Note that f_{ans_r} is needed only for bookkeeping purpose). The only time that ans_r must change for a table is when its frequency drops below f_{low_r}. At this point we update ans_r and the new approximate mode is the first element whose frequency reaches f_{high_r} with respect to the current left end point of query range. The query range q_r is also updated to reflect the change on the approximate mode ($F_{ans_r}(a_i, \ldots, a_{q_r}) = f_{hgih_r}$).

T_i	T_1	T_2	T_3	T_4	T_5
a_i	b	a	b	a	c
$[1,1]$	$1, (\mathbf{b}, 1)$	$2, (\mathbf{a}, 1)$	$3, (\mathbf{b}, 1)$	$4, (\mathbf{a}, 1)$	$5, (\mathbf{c}, 1)$
$[2,3]$	$7, (\mathbf{a}, 3)$	$7, (a, 3)$	$7, (a, 2)$	$7, (a, 2)$	$8, (\mathbf{c}, 3)$
$[4,5]$	$10, (\mathbf{a}, 5)$	$10, (a, 5)$	$10, (a, 4)$	$10, (a, 4)$	$12, (\mathbf{c}, 5)$
$[6,9]$	$17, (\mathbf{c}, 9)$	$17, (c, 9)$	$17, (c, 9)$	$17, (c, 9)$	$17, (c, 9)$
$[10,13]$	$20, (\mathbf{c}, 11)$	$20, (c, 11)$	$20, (c, 11)$	$20, (c, 11)$	$20, (c, 11)$

T_i	T_6	T_7	T_8	T_9	T_{10}
a_i	c	a	c	a	a
$[1,1]$	$6, (\mathbf{c}, 1)$	$7, (\mathbf{a}, 1)$	$8, (\mathbf{c}, 1)$	$9, (\mathbf{a}, 1)$	$10, (\mathbf{a}, 1)$
$[2,3]$	$8, (c, 2)$	$10, (\mathbf{a}, 3)$	$10, (a, 2)$	$10, (a, 2)$	$13, (\mathbf{c}, 3)$
$[4,5]$	$12, (c, 4)$	$14, (\mathbf{c}, 5)$	$14, (c, 5)$	$14, (c, 4)$	$14, (c, 4)$
$[6,9]$	$17, (c, 8)$	$17, (c, 7)$	$17, (c, 7)$	$17, (c, 6)$	$17, (c, 6)$
$[10,13]$	$20, (c, 10)$	—	—	—	—

T_i	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}
a_i	c	c	c	c	b
$[1,1]$	$11, (\mathbf{c}, 1)$	$12, (\mathbf{c}, 1)$	$13, (\mathbf{c}, 1)$	$14, (\mathbf{c}, 1)$	$15, (\mathbf{b}, 1)$
$[2,3]$	$13, (\mathbf{c}, 3)$	$13, (c, 2)$	$16, (\mathbf{c}, 3)$	$16, (c, 2)$	$18, (\mathbf{c}, 3)$
$[4,5]$	$14, (c, 4)$	$17, (\mathbf{c}, 5)$	$17, (c, 4)$	$19, (\mathbf{c}, 5)$	$19, (c, 4)$
$[6,9]$	$17, (c, 6)$	$19, (\mathbf{c}, 7)$	$19, (c, 6)$	—	—
$[10,13]$	—	—	—	—	—

T_i	T_{16}	T_{17}	T_{18}	T_{19}	T_{20}
a_i	c	c	c	c	b
$[1,1]$	$16, (\mathbf{c}, 1)$	$17, (\mathbf{c}, 1)$	$18, (\mathbf{c}, 1)$	$19, (\mathbf{c}, 1)$	$20, (\mathbf{b}, 1)$
$[2,3]$	$18, (c, 3)$	$18, (c, 2)$	$19, (\mathbf{c}, 2)$	—	—
$[4,5]$	$19, (c, 4)$	—	—	—	—
$[6,9]$	—	—	—	—	—
$[10,13]$	—	—	—	—	—

Table 3: An example showing the data structure for answering $\frac{1}{2}$-approximate range mode queries on a list of 20 elements. Updates are in bold.

Table 3 shows the data structure for answering approximate range mode queries on the same list as in Figure 1. As an example, to look up the approximate mode of $a_4, \ldots, a_{12}$, we search in T_4 and find the entry with the largest q_r that is smaller than 12: $\{[4,5], 10, (a, 4)\}$. This tells us that, in the sequence $a_4, \ldots, a_{12}$, a occurs at least 4 times ($F_a(a_4, \ldots, a_{12}) \geq F_a(a_4, \ldots, a_{10}) = 4$) and no element occurs more than 8 times ($F_x(a_2, \ldots, a_{12}) \leq F(a_2, \ldots, a_{17}) - 1 = 8$).

After T_1 is built, T_i $(i \geq 2)$ is built based on T_{i-1} with updates that are necessary to satisfy conditions listed in Table 2. The number of the updates made is given by the following lemma.

Lemma 2. *If the rth row of the table is updated in T_i, then it does not need to be updated in T_k for any $i < k < i + 1/\alpha^{\lfloor r/2 \rfloor}$.*

Proof. When the rth row is updated in T_i we set ans_r to be the first element such that $F_{ans_r}(a_i, \ldots, a_{q_r}) = f_{high_r}$. Its frequency f_{ans_r} is initially f_{high_r} in T_i. Although f_{ans_r} may decrease as i increases, ans_r does not need to be updated again until f_{ans_r} drops below f_{low_r}, which takes at least $f_{high_r} - (f_{low_r} - 1) = f_{high_r} - f_{high_{r-1}} = 1/\alpha^{\lfloor r/2 \rfloor}$ steps. □

Note that there are no more than $2\lceil \log_{\frac{1}{\alpha}} n \rceil$ rows in a table and every time we build a new table, the first row needs to be updated. Lemma 2 shows that the rth $(r \geq 2)$ row changes no more than $\alpha^{\lfloor r/2 \rfloor} n$ times during the construction of all n tables. The total number of updates we have to make is given by the following theorem.

Theorem 1. *The total number of updates we have to make is $O(n/(1-\alpha))$.*

Proof.

$$\begin{aligned} \text{Total number of updates} &\leq n + \sum_{r=2}^{2\lceil \log_{\frac{1}{\alpha}} n \rceil} \alpha^{\lfloor r/2 \rfloor} n \\ &= O(\frac{n}{1-\alpha}) \end{aligned}$$

□

Theorem 1 says that, the majority of the table entries can be reconstructed by referring to other tables. In other words, although n lookup tables are needed to answer approximate range mode queries, many of them share common entries. A persistent search tree [4, 6] is used to store the tables space efficiently. It has the properties that the query time is $O(\log m)$ where m is the number of entries in each table, and the storage space is $O(1)$ per update. In the case of approximate range mode queries, although each table can have as many as $2\lceil \log_{\frac{1}{\alpha}} n \rceil$ entries, many tables share the same entries and the number of (different) nodes in the persistent tree is $O(n/(1-\alpha))$, one for each update, and the query time of each node is $O(\log \log_{\frac{1}{\alpha}} n)$.

To keep track of the frequency of each element, we first build a balanced binary search tree of distinct elements in the list. This can be done by walking through the list and for each element in the list, check whether it is already in the search tree. If not, we insert a node into the tree and add a pointer from the element in the list to the newly added node in the search tree. If the element is already in the search tree, a pointer to that node is added. Each element can be processed in $O(\log n)$ time, and walking through the whole list can be done in $O(n \log n)$ time.

The idea presented in [3] leads to an algorithm that maintains a counter for each element in the search tree: The counters are stored in a doubly linked list of groups, each group is a collection of elements with equal counter value. Groups are ordered according to their counter values. Note that it takes $O(1)$ time to increment or decrement an element's counter: The element is deleted from the current group and inserted to the adjacent group if the difference in the value between the two groups' counters is one, otherwise a new group is created and the element is added to that group. The tables are built one row at a time:

```
For $r = 1$ to $2\lceil \log_{\frac{1}{\alpha}} n \rceil$:

  1. $i = j = 1$.
  2. Initialize the counters to zero by placing them in a common group with value zero.
  3. For $j = 1$ to $n$:
     (a) Increment $a_j$'s counter.
     (b) If $a_j$'s counter, $F_{a_j}(a_i, \ldots, a_j)$ equals $f_{high_r}$,
         i. Store $a_j$ in $T_i$.
         ii. Increment $i$.
         iii. Decrement $a_i$'s counter.
```

Because i can be incremented at most $n-1$ times, Step 3 can be executed at most n times. In Step $3(a)$ it takes $O(1)$ time to locate and increment a_j's counter. Each part of Step $3(b)$ also takes $O(1)$ time. Therefore the total preprocessing time is $O(n \log_{\frac{1}{\alpha}} n)$.

Theorem 2. *There exists a data structure of size $O(n/(1-\alpha))$ that can answer approximate range mode queries in $O(\log \log_{\frac{1}{\alpha}} n)$ time, and can be constructed in $O(n \log_{\frac{1}{\alpha}} n)$ time.*

2.2 Lower bounds

Next we show there is no faster worst case algorithm to compute the approximate mode for any fixed approximate factor α. To see this, let A be a list of $n/\lceil \frac{1}{\alpha} \rceil$ elements and $B = A \ldots A = b_1, \ldots, b_n$ is a list obtained by repeating A $\lceil \frac{1}{\alpha} \rceil$ times. The problem of testing whether there exist two identical elements in A (also called *element uniqueness*) can be reduced to asking if the mode of B occurs more than $\lceil 1/\alpha \rceil$ times. In the case of approximate range mode query, the answer to query $b_1, \ldots, b_n$ is an element whose frequency is greater than 1 if and only if the actual mode of B occurs more than $\lceil \frac{1}{\alpha} \rceil$ times.

In the algebraic decision tree model of computation, the problem of determining whether all the elements of A are unique is known to have an $\Omega(n \log n)$ lower bound. However, this problem can also be solved by doing a single approximate range mode query $b_1, \ldots, b_n$ after preprocessing B, which implies the same lower bound holds for approximate range mode queries.

Theorem 3. *Let $P(n)$ and $Q(n)$ be the preprocessing and query times, respectively, of a data structure for answering approximate mode queries, we have $P(n) + Q(n) = \Omega(n \log n)$.*

On the other hand, $\Omega(n)$ storage space is required by any data structure that supports approximate range mode queries since the original list can be reconstructed by doing queries $(a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n)$, regardless of what value α is.

2.3 Constant Query Time

Yao[7] (see also Alon *et al.*[1]) showed that if a query $a_i, \ldots, a_j$ can be answered by combining answers of queries $a_i, \ldots, a_x$ and $a_{x+1}, \ldots, a_j$ in constant time, then $\Theta(n\lambda(k, n))$ time and space is both necessary and sufficient to answer range queries in at most k steps. We adapt the same approach to develop constant query time data structures for some special cases of approximate range mode queries. Namely, the approximate factor $\alpha = 1/k$ where k is some positive integer.

The following lemma says that, if we can partition the range $a_i, \ldots, a_j$ into k intervals and we know the mode of each interval, then one of these is an approximate mode, for $\alpha = 1/k$.

Lemma 3. *If $\{B_1, \ldots, B_k\}$ is a partition of $a_i, \ldots, a_j$ then $max_p F(B_p) \geq F(a_i, \ldots, a_j)/k$.*

Proof. By contradiction. Otherwise for any element x we have $F_x(a_i, \ldots, a_j) = \sum_{p=1}^{k} F_x(B_p) \leq k \times max_p F(B_p) < F(a_i, \ldots, a_j)$. □

Yao[7] and Alon *et al.*[1] gave an optimal scheme of using a minimum set of intervals such that any range $a_i, \ldots, a_j$ can be covered by at most k such intervals.

Lemma 4. *(Yao[7], Alon et al.[1]) There exists a set of $O(n\lambda(k,n))$ intervals such that any query range $a_i, \ldots, a_j$ can be partitioned into at most k of these intervals. Furthermore, given i and j, these at most k intervals can be found in $O(k)$ time.*

Given Lemma 3 and Lemma 4, we immediately obtain a constant query time solution for answering approximate range mode queries with approximate factor $1/k$. By precomputing the mode of each interval, a query can be answered by first fetching the partition of the query range, which is a set of at most k intervals, and then outputting the one with the highest frequency among k modes of these intervals.

Theorem 4. *There exists a data structure of size $O(n\lambda(k,n))$ that can answer approximate range mode in $O(k)$ time, for $\alpha = 1/k$.*

The results in Theorem 4 can be further improved using a table lookup trick for $k \geq 4$. We partition the list into $n/\log n$ blocks of size $\log n$, $B_i = a_{(i-1)\log n+1}, \ldots, a_{i \log n}, i = 1, \ldots, n/\log n$. By Lemma 4, there exists a set of $O((n/\log n)\lambda(2, n/\log n)) = O(n)$ intervals such that any range with both ends at the boundaries of the blocks can be covered with at most 2 of these intervals. The exact modes of these intervals are precomputed. Inside a block, exact modes of 2 intervals are precomputed for each element, one interval is between the element and the beginning of the block and the other interval between the element and the end of the block. There are $2n$ such intervals and computing the mode of each interval costs $O(\log n)$ time. Any query range that spans more than one block can be partitioned into at most 4 intervals. The first one is the (possibly partial) block in which the range starts; the last one is the the (possibly partial) block in which the range ends and the other (at most) two intervals in between cover all the remaining blocks (if any). Of these intervals the modes are all precomputed, and the one with the highest frequency is a $1/4$-approximation of the actual mode.

It remains to show that a query within a block can also be answered in $O(1)$ time. This is done by recursively partitioning the $\log n$ block into $\log n/\log\log n$ blocks of size $\log\log n$. The same method above is used to preprocess these blocks, and the result is a data structure of $O(n)$ size that can answer any query that spans more than one $\log\log n$-block in $O(1)$ time.

To answer queries within a $\log\log n$-block, a different approach is used. Note that we can normalize each block by replacing each element with the index of its first occurrence within the block. Because such index is a non-negative integer that is at most $\log\log n$ and each block consists of $\log\log n$ such values, there are at most $(\log\log n)^{\log\log n}$ different blocks. Among all $n/\log\log n$ blocks of size $\log\log n$, many are of the same type. Thus, preprocessing of each block is unnecessary, and storage space can be reduced by preprocessing a block once and reusing the results for all blocks of the same type. The data structure used is a $\log\log n \times \log\log n$ matrix that can answer range mode query in constant time. All the queries in blocks of the same type are done in the same matrix. There are at most $(\log\log n)^{\log\log n}$ possible matrices which require $O((\log\log n)^{\log\log n}(\log\log n)^2) = o(n)$ storage space.

Theorem 5. *There exists a data structure of size $O(n)$ that can answer approximate range mode queries in $O(1)$ time, for $\alpha = 1/4$.*

3 Approximate Range Median Queries

In this section, we consider approximate range median queries on a list of comparable elements $A = a_1, \ldots, a_n$. Given an approximate factor $0 < \alpha < 1$, our task is to preprocess A so that, for any indices $1 \leq i \leq j \leq n$, we can quickly return an element of $a_i, \ldots, a_j$ whose rank is between $\alpha \times \lfloor \frac{(j-i+1)}{2} \rfloor$ and $(2-\alpha) \times \lfloor \frac{(j-i+1)}{2} \rfloor$.

The idea behind our algorithm is that, if a query $a_i, \ldots, a_j$ spans many blocks, then the contribution of the first and last block is minimal and can be ignored. Instead, we could simply answer the (precomputed) median of the union of the internal blocks. On the other hand, since we are using many different block size, we can choose a block size so that $a_i, \ldots, a_j$ spans just enough blocks the strategy above needs to give a valid approximation. This ensures that we do not have to precompute too many medians.

3.1 $O(\frac{nlogn}{1-\alpha})$ Preprocessing Time

To simplify the presentation we assume $n = 2^k$ for some integer $k \geq 1$. We preprocess A and build k lookup tables $T_1, \ldots, T_k$ as follows. To build T_i $(1 \leq i \leq k)$, we partition A into 2^i blocks each of size $n/2^i$:

$$B_{i_j} = a_{(j-1)\times\frac{n}{2^i}+1}, \ldots, a_{j\times\frac{n}{2^i}}, j = 1, \ldots, 2^i$$

T_i has 2^i entries $(T_{i_j}, j = 1, 2, \ldots, 2^i)$, each corresponds to a block B_{i_j} and contains a pointer to a list of $\lceil \frac{2(1+\alpha)}{1-\alpha} \rceil$ elements of A.

$$T_{i_j}(p) = Median(B_{i_j} \ldots B_{i_{j+p-1}}), p = 1, \ldots, \lceil \frac{2(1+\alpha)}{1-\alpha} \rceil,$$

where $Median(B_{i_j} \ldots B_{i_{j+p-1}})$ is the median of $B_{i_j} \ldots B_{i_{j+p-1}}$, which can be computed in $O(\frac{pn}{2^i})$ time [2]. There are $k = \log n$ tables to be computed. It follows that:

$$\begin{aligned}\text{The total preprocessing time} &= \sum_{i=1}^{\log n} \sum_{j=1}^{2^i} \sum_{k=1}^{\lceil \frac{2(1+\alpha)}{1-\alpha} \rceil} O\left(\frac{kn}{2^i}\right) \\ &= O\left(\frac{n\log n}{1-\alpha}\right)\end{aligned}$$

In the following subsections, we give the preprocessing time, storage space and query time of our data structure for answering approximate range median queries.

3.2 $O(\frac{n}{1-\alpha})$ Storage Space

As shown in Section 3.1, the data structure for answering approximate range median queries is a set of lookup tables. Each table T_i $(1 \leq i \leq \log n)$ is of size $O(\frac{2^i(1+\alpha)}{1-\alpha})$ and the total space needed to store all $\log n$ tables is $\sum_{i=1}^{\log n} O(\frac{2^i(1+\alpha)}{1-\alpha}) = O(\frac{n(1+\alpha)}{1-\alpha}) = O(\frac{n}{1-\alpha})$.

3.3 $O(1)$ Query Time

Next we show how to compute an approximate range median of $a_i, \ldots, a_j$.

1. Compute the length of the query $L = j - i + 1$, and then locate table T_p in which to continue the search: $p = \lceil \log \frac{(1+\alpha)n}{(1-\alpha)L} \rceil$.
2. Compute $b_i = \lceil \frac{i}{n/2^p} \rceil$ and $b_j = \lfloor \frac{j}{n/2^p} \rfloor$, since $p = \lceil \log \frac{(1+\alpha)n}{(1-\alpha)L} \rceil \leq \log \frac{(1+\alpha)n}{(1-\alpha)L} + 1 = \log \frac{2(1+\alpha)}{(1-\alpha)L}$, *i.e.*, $2^p \leq \frac{2(1+\alpha)n}{(1-\alpha)L}$, we have

$$b_j - b_i = \lfloor \frac{j}{n/2^p} \rfloor - \lceil \frac{i}{n/2^p} \rceil \leq \frac{j-i}{n/2^p} \leq \frac{j-i}{\frac{(1-\alpha)L}{2(1+\alpha)}} < \frac{2(1+\alpha)}{1-\alpha}$$

   i.e., $Median(B_{p_{b_i}} \ldots B_{p_{b_j}})$ is stored in the list to which a pointer is stored in T_{p_j}.
3. Output $T_{p_{b_i}}(b_j - b_i) = Median(B_{p_{b_i}} \ldots B_{p_{b_j}})$ as the answer.

Because each of the three steps above takes $O(1)$ time, the time required for answering the approximate range median query is $O(1)$.

Theorem 6. *There exists a data structure of size $O(n/(1-\alpha))$ that can answer approximate range median queries in $O(1)$ time, and can be built in $O(n \log n/(1-\alpha))$ time.*

4 Conclusion

Range query problems demand reporting an answer very quickly. Efficient algorithms have been proposed for various types of range queries [1, 7]. In this paper we study the approximate version of the two types of range queries first investigated by Krizanc *et al* [5]. We propose data structures that can be used to answer approximate range mode queries and approximate range median queries both space and time efficiently. The approximation guarantees are explicit, and apply without regard to the distribution of the input. The preprocessing time needed for constructing the data structures are also shown to be very economical.

References

[1] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report 71/87, Tel-Aviv University, 1987.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[3] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 348–360, 2002.

[4] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[5] D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation (ISAAC 2003)*, 2003.

[6] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[7] A. C. Yao. Space-time tradeoff for answering range queries. In *Proceedings of the 14th annual ACM Symposium on the Theory of Computing*, pages 128–136, 1982.