Finding Frequent Elements in Compressed 2D Arrays and Strings

Travis Gagie[1], Meng He[2], J. Ian Munro[2], and Patrick K. Nicholson[2]

[1] Department of Computer Science and Engineering, Aalto University, Finland
[2] Cheriton School of Computer Science, University of Waterloo, Canada

Abstract. We show how to store a compressed two-dimensional array such that, if we are asked for the elements with high relative frequency in a range, we can quickly return a short list of candidates that includes them. More specifically, given an $m \times n$ array A and a fraction $\alpha > 0$, we can store A in $\mathcal{O}\big(mn(H+1)\log^2(1/\alpha)\big)$ bits, where H is the entropy of the elements' distribution in A, such that later, given a rectangular range in A and a fraction $\beta \geq \alpha$, in $\mathcal{O}(1/\beta)$ time we can return a list of $\mathcal{O}(1/\beta)$ distinct array elements that includes all the elements that have relative frequency at least β in that range. We do not *verify* that the elements in the list have relative frequency at least β, so the list may contain false positives. In the case when $m = 1$, i.e., A is a string, we improve this space bound by a factor of $\log(1/\alpha)$, and explore a space-time trade off for verifying the frequency of the elements in the list. This leads to an $\mathcal{O}(n \min(\log(1/\alpha), H+1)\log n)$ bit data structure for strings that, in $\mathcal{O}(1/\beta)$ time, can return the $\mathcal{O}(1/\beta)$ elements that have relative frequency at least β in a given range, without false positives, for $\beta \geq \alpha$.

1 Introduction

An α-*majority* in an array is an element that has relative frequency at least α, for $0 < \alpha < 1$. Finding α-majorities in strings is a well-studied problem: Misra and Gries [12], Demaine, López-Ortiz and Munro [4] and Karp, Shenker and Papadimitriou [10] all independently discovered the same algorithm that, given α, makes one pass over the string to build a list of $1/\alpha$ candidate α-majorities, and then makes a second pass to verify them, all using $\mathcal{O}(1/\alpha)$ words of working memory. Recently, this problem has been considered in the approximate setting, using the term *heavy hitters* instead of α-majorities, where false positives and even false negatives are allowed [3, 7]. Very recently, Durocher, He, Munro, Nicholson and Skala [5] showed how to store a string in $\mathcal{O}(\log(1/\alpha)\log n)$ bits[3] per character such that, given a range, the list of α-majorities in that range can be returned in $\mathcal{O}(1/\alpha)$ time; an improvement in both time and space over the previous best result [11]. As with Misra and Gries' algorithm, they first build a list of $\mathcal{O}(1/\alpha)$ candidates and then verify them, but this time using a wavelet tree [9] for batched rank queries. In this paper we extend Durocher et al.'s result

[3] We use $\log n$ to denote $\log_2 n$.

in three directions: we consider two-dimensional arrays, we achieve compression, and we show how to answer queries faster when asked only for the β-majorities, for some fraction $\beta > \alpha$.

In Section 2 we describe a data structure for storing an $m \times n$ array A such that later, given a rectangular range in A and a fraction $\beta \geq \alpha$, in $\mathcal{O}(1/\alpha)$ time we can return a list of $\mathcal{O}(1/\beta)$ distinct array elements that *includes* all the elements that have relative frequency at least β in that range. We do not consider how to *verify* which of the $\mathcal{O}(1/\beta)$ candidates are truly β-majorities, so the list may contain false positives. We assume throughout that A's elements are from the alphabet $\{1, \ldots, mn\}$. This data structure occupies $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits, where H is the entropy of the elements' distribution in A. We then show how, by increasing the space by a factor of $\mathcal{O}(\log 1/\alpha)$, we can build the list of candidates in $\mathcal{O}(1/\beta)$ time.

In Section 3 we show that if A is a string, then our space bounds are reduced by a factor of $\log(1/\alpha)$. We also explore a space-time trade off for verifying the candidates in a string. Ultimately, we present a data structure that occupies $\mathcal{O}(n \min(\log(1/\alpha), H+1)\log n)$ bits, and can return all the $\mathcal{O}(1/\beta)$ verified β-majorities in a range in $\mathcal{O}(1/\beta)$ time. If we reduce the space to $\mathcal{O}(n(H+1))$ bits, we can still return the verified β-majorities in $\mathcal{O}(\log\log n/\alpha)$ time.

2 Two-Dimensional Arrays

For the moment, assume that we are always willing to spend $\mathcal{O}(1/\alpha)$ time to compute the list, regardless of β. We store a Huffman-coded copy of A and a bit-vector that supports select in $\mathcal{O}(1)$ time [2], with a 1 marking the position of the first bit of each codeword. This takes a total of $mn(H + \mathcal{O}(1))$ bits and allows us $\mathcal{O}(1)$-time access to any element in A.

For the sake of simplicity, assume m and n are powers of 2; otherwise, we pad A with null values to make this true. Let I be the set

$$\left\{[i_1..i_2] \subseteq [1..m] : i_2 - i_1 + 1 = 2^k, i_1 = t2^{k-1}, (i_1, i_2, t, k \in \mathbb{Z}^*)\right\} \ ,$$

and J be the set

$$\left\{[j_1..j_2] \subseteq [1..n] : j_2 - j_1 + 1 = 2^k, j_1 = t2^{k-1}, (j_1, j_2, t, k \in \mathbb{Z}^*)\right\} \ .$$

For each vertical interval $[i_1..i_2] \in I$ and horizontal interval $[j_1..j_2] \in J$ with $\ell_1\ell_2 > 1/\alpha$, where $\ell_1 = i_2 - i_1 + 1$ is the length of $[i_1..i_2]$ and $\ell_2 = j_2 - j_1 + 1$ is the length of $[j_1..j_2]$, we store a list of the at most $9/\alpha$ distinct array elements that each occurs at least $\alpha\,\ell_1\ell_2$ times in $A[i_1 - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ (we assume these indices are valid; border cases can be handled similarly), in non-increasing order by frequency. Figure 1 shows an example. For any $\beta \geq \alpha$, the first $9/\beta$ elements in this list include all β-majorities for any range that is contained in $A[i_i - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ but contains $A[i_1..i_2, j_1..j_2]$. We call $A[i_1..i_2, j_1..j_2]$ a *block*, and we define the *size* of the block to be $\ell_1 \times \ell_2$.

Finally, we build a Huffman code for the all the blocks' lists and store them encoded, together with another bit-vector that supports select in $\mathcal{O}(1)$ time, with

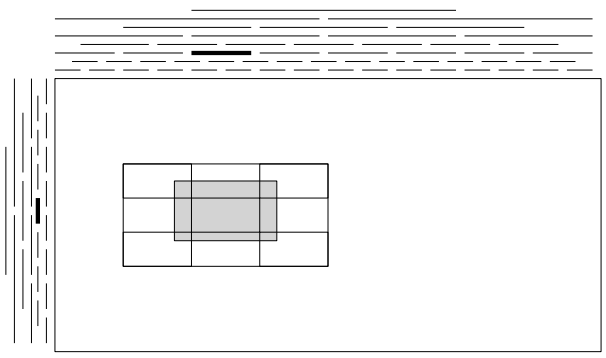

Fig. 1. Consider a vertical interval $[i_1..i_2] \in I$ (shown as thick line at left) of length $\ell_1 = i_2 - i_1 + 1$ and a horizontal interval $[j_1..j_2] \in J$ (shown as thick line at top) of length $\ell_2 = j_2 - j_1 + 1$, with $\ell_1\ell_2 > 1/\alpha$. The first $9/\beta$ elements in the list we store include all β-majorities for any range that is contained in $A[i_i - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ but contains $A[i_1..i_2, j_1..j_2]$. One such range is shown in grey, with $A[i_1 - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ shown as the 9-part rectangle containing it and $A[i_1..i_2, j_1..j_2]$ as the center part.

a 1 marking the position of the first bit of the first codeword in each list. We now describe how our structure answers queries, and then analyse its space usage.

Answering Queries: Given a rectangular query range in A, we can compute the block size and dimensions of the *query block*, i.e., the largest block contained in the query range, in $\mathcal{O}(1)$ time. Once we have the block size and dimensions, the offset of the query block in the Huffman-coded block lists is easily obtained based on the position of the query rectangle, and we can access the corresponding list in $\mathcal{O}(1)$ time using select. Thus, in $\mathcal{O}(1/\beta)$ time we can retrieve the list, and report the first $\mathcal{O}(1/\beta)$ candidates, if the query block has size greater than $1/\alpha$. If we are asked to find the β-majorities in a smaller range, then we scan the range with Misra and Gries' algorithm [12] in $\mathcal{O}(1/\alpha)$ time.

To reduce the time bound from $\mathcal{O}(1/\alpha)$ to $\mathcal{O}(1/\beta)$ when the query block is smaller than size $1/\alpha$, we store $\log(1/\alpha)$ instances of our data structure. The k-th instance is set up to return candidate $(1/2^k)$-majorities in $\mathcal{O}\big(2^k\big)$ time. To find candidate β-majorities, we use the $\lceil \log(1/\beta) \rceil$-th instance[4].

Space Analysis: Since I and J contain $\mathcal{O}(m)$ and $\mathcal{O}(n)$ intervals, respectively, it is not difficult to see that we use a total of $\mathcal{O}((mn/\alpha)\log(mn))$ bits. In fact, we will show that because we do not store lists for ranges with size $1/\alpha$ or less, the $1/\alpha$ factor in the space bound is reduced to $\log(1/\alpha)$. Moreover, encoding the lists using Huffman codes reduces the $\log(mn)$ factor to $H + 1$. The following three lemmas solidify these arguments, bounding the space occupied by our structure.

[4] Some of the block lists in the k-th instance are prefixes of lists in the $(k + 1)$-th instance. However, removing this redundancy does not asymptotically reduce space.

Lemma 1. *Any block B of size 2^b overlaps $\mathcal{O}\big(\alpha\, 2^b \log(1/\alpha)\big)$ blocks of size at most 2^b.*

Proof. The number of such blocks that B overlaps is at most 9 times the number it contains (including itself), so we count the latter. For $k \le b$, there are $k+1$ possible rectangular shapes with size 2^k and sides whose lengths are powers of 2: i.e., 1×2^k, $2 \times 2^{k-1}$, $\ldots$, $2^k \times 1$. For each such shape, B contains at most $4 \cdot 2^b/2^k$ blocks with that shape. Therefore, since $\sum_{k=\log(1/\alpha)}^{b}(k+1)(2^b/2^k) = \mathcal{O}\big(\alpha\, 2^b \log(1/\alpha)\big)$, B overlaps $\mathcal{O}\big(\alpha\, 2^b \log(1/\alpha)\big)$ blocks of size at most 2^b. □

Lemma 2. *Any array element c occurs in blocks' lists $\mathcal{O}(\log(1/\alpha))$ times as often as it occurs in A.*

Proof. Let s be the total size of all the blocks B such that c is in B's list but not in the list of any larger block that overlaps B. Some of the choices of B could overlap but c still occurs at least $\alpha\, s/9$ times in A. On the other hand, it follows from Lemma 1 that c occurs in $\mathcal{O}(\alpha\, s \log(1/\alpha))$ blocks' lists. □

Lemma 3. *The Huffman-coded lists take a total of $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits.*

Proof. Suppose we encode the lists with a Huffman code based on A, which cannot give better compression than using one based on the lists themselves. Since encoding A takes $mn(H + \mathcal{O}(1))$ bits, it follows from Lemma 2 that encoding the lists takes $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits. □

Recall that if we are always willing to spend $\mathcal{O}(1/\alpha)$ time to find the candidates, regardless of β, then we store only one instance of our data structure and, by Lemma 3, this occupies $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits. Otherwise, we store a data structure for each of the $\log(1/\alpha)$ fractions, $1/2, 1/4, \ldots, \alpha$, and use a total of $\mathcal{O}\big(mn(H+1)\log^2(1/\alpha)\big)$ bits. We get the following theorem:

Theorem 1. *Given an $m \times n$ array A and a fraction $\alpha > 0$, A can be stored in $\mathcal{O}\big(mn(H+1)\log^2(1/\alpha)\big)$ bits, where H is the entropy of the elements' distribution in A, such that later, given a rectangular range in A and a fraction $\beta \ge \alpha$, a list of $\mathcal{O}(1/\beta)$ distinct array elements that includes all the elements that have relative frequency at least β in that range can be returned in $\mathcal{O}(1/\beta)$ time. Alternatively, A can be stored in $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits, and return the same list in $\mathcal{O}(1/\alpha)$ time, regardless of β.*

3 Improvements for Strings

Space Reduction: In the special case when A is a string — i.e., when $m = 1$ — a range of given size can have only one shape. Using the following lemma instead of Lemma 1, we reduce our space bounds by a $\log(1/\alpha)$ factor.

Lemma 4. *In one dimension, any block B of size 2^b overlaps $\mathcal{O}\big(\alpha\, 2^b\big)$ blocks of size at most 2^b.*

Proof. The number of such blocks that B overlaps is at most 3 times the number it contains (including itself), which is $\sum_{k=\log(1/\alpha)}^{b} 2^b/2^k = \mathcal{O}(\alpha\, 2^b)$. □

Theorem 2. *Given a string A of length n and a fraction $\alpha > 0$, A can be stored in $\mathcal{O}(n(H+1)\log(1/\alpha))$ bits, where H is the 0th-order empirical entropy of A, such that later, given a range in A and a fraction $\beta \geq \alpha$, a list of $\mathcal{O}(1/\beta)$ distinct characters that includes all those with relative frequency at least β in that range can be returned in $\mathcal{O}(1/\beta)$ time. Alternatively, A can be stored in $\mathcal{O}(n(H+1))$ bits, and return the same list in $\mathcal{O}(1/\alpha)$ time, regardless of β.*

Verifying the Elements: Barbay et al. [1] showed how to store A in $nH+o(n)(H+1)$ bits and answer rank queries on it in $\mathcal{O}(\log\log n)$ time. Combining this with Theorem 2, we can easily verify each candidate c in $\mathcal{O}(\log\log n)$ time using a rank_c query at either end of the range. In the sequel, we show how to verify the $\mathcal{O}(1/\beta)$ candidates returned by the data structure in Theorem 2, in $\mathcal{O}(1)$ time per candidate, using $\mathcal{O}(n\min(\log(1/\alpha), H+1)\log n)$ bits.

For each block $A[j_1..j_2]$ in the string, we store a copy S of the substring $A[j_1-\ell, j_2+\ell]$, where $\ell = j_2 - j_1 + 1$, replacing each character with its rank in the list for $A[j_1..j_2]$; characters not in the list are replaced with 0. Thus, each substring S contains characters drawn from the alphabet $\{0,\ldots,\sigma\}$, where $\sigma \leq \lceil 3/\alpha \rceil$. Furthermore, recall that the list for $A[j_1..j_2]$ is sorted by the characters' frequency in the substring $A[j_1-\ell, j_2+\ell]$ in non-increasing order. This means that, with the exception of 0, the characters' lexicographic order in S is the same as their order by frequency. We call S the *rank sequence* for the block $A[j_1..j_2]$.

Observation 1 *Let S be a sequence of m characters, drawn from the alphabet $\Sigma = \{0, 1, ..., \sigma\}$, where, ignoring character 0, character i is the i-th most frequent character in S. There exists a skewed wavelet tree T, representing the sequence S using Elias gamma coding [6], that has the following properties:*

1. *The leaf representing i has depth $\mathcal{O}(\log(i+2))$ in T (the root has depth 0).*
2. *For $1 \leq k \leq \sigma$, the leaves representing $1, ..., k$ can be traversed in $\mathcal{O}(k)$ time.*
3. *The wavelet tree T occupies $\mathcal{O}(m(H(S)+1))$ bits, where $H(S)$ denotes the 0th-order empirical entropy of the sequence S.*

We store the concatenation of the rank sequences of all blocks of size 2^b, in ascending order of block starting position, as a string Y_b, for $\log(1/\alpha) < b \leq \log n$, noting that $|Y_b| = \mathcal{O}(n)$. Using the wavelet tree from Observation 1 to represent these strings will allow us to count the frequency of candidate elements efficiently. However, at this point we have not built concatenated strings for blocks of size smaller than $1/\alpha$. We now explain how to construct $Y_{b'}$, for block sizes $2^{b'}$, where $1 \leq b' \leq \log(1/\alpha)$, in order to verify the frequency of candidates in these smaller blocks in $\mathcal{O}(1/\beta)$ time.

Recall that the first result in Theorem 2 keeps $\log(1/\alpha)$ copies of the data structure, constructed for values in the set $F = \{1/2, 1/4, ..., \alpha\}$. Let $D_{\alpha'}$ denote the data structure constructed for $\alpha' \in F$. The string $Y_{b'}$ is constructed using the block lists from $D_{\alpha'}$, where α' is the smallest value in F such that $D_{\alpha'}$ stores

candidate lists for blocks of size $2^{b'}$. We now argue that representing all of the $\log n$ strings using Observation 1 requires $\mathcal{O}(n \min(\log(1/\alpha), H+1) \log n)$ bits.

Consider the rank sequence S of a block B of size 2^b, and the original substring S' of A, to which S corresponds. Applying a many-to-one mapping to a string's alphabet — e.g., from characters to their ranks or 0 — cannot increase its entropy, so $H(S) \leq H(S')$. Moreover, if we partition a string, then the average of the substrings' entropies, weighted by length, is at most the string's entropy [8], so $H(Y_b) = \mathcal{O}(H)$ for $1 \leq b \leq \log n$. Furthermore, it is clear that $H(Y_b) = \mathcal{O}(\log(1/\alpha))$, since the alphabet size is at most $\lceil 3/\alpha \rceil$. Since there are $\log n$ strings, the space is $\mathcal{O}(n \min(\log(1/\alpha), H+1) \log n)$ bits overall.

Verifying the candidates for a block of size 2^b involves traversing the first $\mathcal{O}(1/\beta)$ leaves of the skewed wavelet tree representing Y_b, which takes $\mathcal{O}(1/\beta)$ time by Observation 1. At each leaf, which represents a candidate, at most two $\mathcal{O}(1)$ time rank queries on a bit vector can be used to determine the frequency of the candidate in a range, in $\mathcal{O}(1/\beta)$ time overall.

Theorem 3. *Given a string A of length n and a fraction $\alpha > 0$, A can be stored in $\mathcal{O}(n \min(\log(1/\alpha), H+1) \log n)$ bits, where H is the 0th-order empirical entropy of A, such that later, given a range in A and a fraction $\beta \geq \alpha$, the list of $\mathcal{O}(1/\beta)$ distinct characters with relative frequency at least β in that range can be returned in $\mathcal{O}(1/\beta)$ time. Alternatively, A can be stored in $\mathcal{O}(n(H+1))$ bits, and return the same list in $\mathcal{O}(\log \log n/\alpha)$ time.*

References

1. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: Proc. ISAAC. pp. 315–326 (2010)
2. Clark, D., Munro, J.: Efficient Suffix Trees on Secondary Storage (extended abstract). In: Proc. SODA. p. 383 (1996)
3. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. of Alg. 55(1), 58–75 (2005)
4. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Proc. ESA. pp. 348–360 (2002)
5. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. In: Proc. ICALP. vol. 6755, pp. 244–255 (2011)
6. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. on Inf. Theory 21(2), 194–203 (1975)
7. Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.: Computing iceberg queries efficiently. In: Proc. VLDB. pp. 299–310 (1998)
8. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. on Alg. 3(2) (2007)
9. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA. pp. 841–850 (2003)
10. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Data. Sys. 28(1), 51–55 (2003)
11. Karpinski, M., Nekrich, Y.: Searching for frequent colors in rectangles. In: Proc. CCCG. pp. 11–14 (2008)
12. Misra, J., Gries, D.: Finding repeated elements. Sci. Comp. Prog. 2, 143–152 (1982)