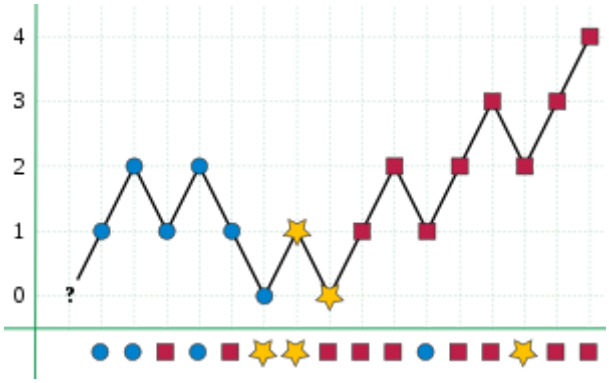
Boyer–Moore majority vote algorithm

The **Boyer–Moore majority vote algorithm** is an algorithm for finding the majority of a sequence of elements using linear time and constant space. It is named after Robert S. Boyer and J Strother Moore, who published it in 1981,[1] and is a prototypical example of a streaming algorithm.



The state of the Boyer–Moore algorithm after each input symbol. The inputs are shown along the bottom of the figure, and the stored element and counter are shown as the symbols and their heights along the black curve.

In its simplest form, the algorithm finds a majority element, if there is one: that is, an element that occurs repeatedly for more than half of the elements of the input. A version of the algorithm that makes a second pass through the data can be used to verify that the element found in the first pass really is a majority.

If a second pass is not performed and there is no majority the algorithm will not detect that no majority exists. In the case that no strict majority exists, the returned element can be arbitrary; it is not guaranteed to be the element that occurs most often (the mode of the sequence). It is not possible for a streaming algorithm to find the most frequent element in less than linear space, for sequences whose number of repetitions can be small.[2]

Contents

Description

The algorithm maintains in its local variables a sequence element and a counter, with the counter initially zero. It then processes the elements of the sequence, one at a time. When processing an element x, if the counter is zero, the algorithm stores x as its remembered sequence element and sets the counter to one. Otherwise, it compares x to the stored element and either increments the counter (if they are equal) or decrements the counter (otherwise). At the end of this process, if the sequence has a majority, it will be the element stored by the algorithm. This can be expressed in pseudocode as the following steps:

- Initialize an element m and a counter i with $i = 0$
- For each element x of the input sequence:
  - If $i = 0$, then assign $m = x$ and $i = 1$

    - else if $m = x$, then assign $i = i + 1$
    - else assign $i = i - 1$
- Return m

Even when the input sequence has no majority, the algorithm will report one of the sequence elements as its result. However, it is possible to perform a second pass over the same input sequence in order to count the number of times the reported element occurs and determine whether it is actually a majority. This second pass is needed, as it is not possible for a sublinear-space algorithm to determine whether there exists a majority element in a single pass through the input.[3]

Analysis

The amount of memory that the algorithm needs is the space for one element and one counter. In the random access model of computing usually used for the analysis of algorithms, each of these values can be stored in a machine word and the total space needed is $O(1)$. If an array index is needed to keep track of the algorithm's position in the input sequence, it doesn't change the overall constant space bound. The algorithm's bit complexity (the space it would need, for instance, on a Turing machine) is higher, the sum of the binary logarithms of the input length and the size of the universe from which the elements are drawn.[2] Both the random access model and bit complexity analyses only count the working storage of the algorithm, and not the storage for the input sequence itself.

Similarly, on a random access machine the algorithm takes time $O(n)$ (linear time) on an input sequence of n items, because it performs only a constant number of operations per input item. The algorithm can also be implemented on a Turing machine in time linear in the input length (n times the number of bits per input item).

Correctness

If there is a majority element, the algorithm will always find it. For, supposing that the majority element is m, let c be a number defined at any step of the algorithm to be either the counter, if the stored element is m, or the negation of the counter otherwise. Then at each step in which the algorithm encounters a value equal to m, the value of c will increase by one, and at each step at which it encounters a different value, the value of c may either increase or decrease by one. If m truly is the majority, there will be more increases than decreases, and c will be positive at the end of the algorithm. But this can be true only when the final stored element is m, the majority element.

See also

- Element distinctness problem, the problem of testing whether a collection of elements has any repeated elements
- Majority function, the majority of a collection of Boolean values
- Majority problem (cellular automaton), the problem of finding a majority element in the cellular automaton computational model

References

1. Boyer, R. S.; Moore, J S. (1991), "MJRTY - A Fast Majority Vote Algorithm", in Boyer, R. S. (ed.), *Automated Reasoning: Essays in Honor of Woody Bledsoe* (http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA131702), Automated Reasoning Series, Dordrecht, The Netherlands:

Kluwer Academic Publishers, pp. 105–117, doi:10.1007/978-94-011-3488-0_5 (https://doi.org/10.1007%2F978-94-011-3488-0_5). Originally published as a technical report in 1981.

2. Trevisan, Luca; Williams, Ryan (January 26, 2012), "Notes on streaming algorithms" (http://theory.stanford.edu/~trevisan/cs154-12/notestream.pdf) (PDF), *CS154: Automata and Complexity*, Stanford University.
3. Cormode, Graham; Hadjieleftheriou, Marios (October 2009), "Finding the frequent items in streams of data", *Communications of the ACM*, **52** (10): 97, doi:10.1145/1562764.1562789 (https://doi.org/10.1145%2F1562764.1562789), "no algorithm can correctly distinguish the cases when an item is just above or just below the threshold in a single pass without using a large amount of space".
