Gregable.com

Majority Voting Algorithm

Find the majority element in a list of values

Oct 6, 2013

I haven't done an algorithms post in awhile, so the usual disclaimer first: If you don't find programming algorithms interesting, stop reading. This post is not for you.

Problem Statement

Imagine that you have a non-sorted list of values. You want to know if there is a value that is present in the list for more than half of the elements in that list. If so what is that value? If not, you need to know that there is no majority element. You want to accomplish this as efficiently as possible.

One common reason for this problem could be fault-tolerant computing. You perform multiple redundant computations and then verify that a majority of the results agree.

Simple Solution

Sort the list, if there is a majority value it must now be the middle value. To confirm it's the majority, run another pass through the list and count it's frequency.

The simple solution is `O(n lg n)` due to the sort though. We can do better!

Boyer-Moore Algorithm

The Boyer-Moore algorithm is presented in this paper: Boyer-Moore Majority Vote

Algorithm. The algorithm uses `O(1)` extra space and `O(N)` time. It requires exactly 2 passes over the input list. It's also quite simple to implement, though a little trickier to understand how it works.

In the first pass, we generate a single candidate value which is the majority value if there is a majority. The second pass simply counts the frequency of that value to confirm. The first pass is the interesting part.

In the first pass, we need 2 values:

1. A `candidate` value, initially set to any value.
2. A `count`, initially set to `0`.

For each element in our input list, we first examine the `count` value. If the count is equal to 0, we set the `candidate` to the value at the current element. Next, first compare the element's value to the current `candidate` value. If they are the same, we increment `count` by 1. If they are different, we decrement `count` by 1.

In python:

```
candidate = 0
count = 0
for value in input:
  if count == 0:
    candidate = value
  if candidate == value:
    count += 1
  else:
    count -= 1
```

At the end of all of the inputs, the `candidate` will be the majority value if a majority value exists. A second O(N) pass can verify that the `candidate` is the majority element (an exercise left for the reader).

Explanation

To see how this works, we only need to consider cases that contain a majority value. If the list does not contain a majority value, the second pass will trivially reject the candidate.

First, consider a list where the first element is not the majority value, for example this list with majority value `0`:

```
[5, 5, 0, 0, 0, 5, 0, 0, 5]
```

When processing the first element, we assign the value of 5 to `candidate` and 1 to `count`. Since 5 is not the majority value, at some point in the list our algorithm must find another value to pair with every 5 we've seen so far, thus `count` will drop to `0` at some point before the last element in the list. In the above example, this occurs at the 4th' element:

List Value:

```
[5, 5, 0, 0, ...
```

Count value:

```
[1, 2, 1, 0, ...
```

At the point that `count` returns to `0`, we have consumed exactly the same number of 5's as other elements. If all of the other elements were the majority element as in this case, we've consumed 2 majority elements and 2 non-majority elements. This is the largest number of majority elements we could have consumed, but even still the majority element must still be a majority of the *remainder* of the input list (in our example, the remainder is ... `0, 5, 0, 0, 5]`). If some of the other elements were not majority elements (for example, if the value was 4 instead), this would be even more true.

We can see similarly that if the first element was a majority element and `count` at some point drops to `0`, then we can also see that the majority element is still the majority of the remainder of the input list since again we have consumed an equal number of majority and non-majority elements.

This in turn demonstrates that the range of elements from the time `candidate` is first assigned to when `count` drops to `0` can be discarded from the input without affecting the final result of the first pass of the algorithm. We can repeat this over and over again discarding ranges that prefix our input until we find a range that is a suffix of our input where `count` never drops to `0`.

Given an input list suffix where count never drops to `0`, we must have more values that equal the first element than values that do not. Hence, the first element (`candidate`) must be the majority of that list and is the only possible candidate for the majority of the full input list, though it is still possible there is no majority at all.

Fewer comparisons

The above algorithm makes 2 passes through our list, and so requires `2N` comparisons in the worst case. It requires another `N` more if you consider the comparisons of `count` to 0. There is another, more complicated, algorithm that operates using only `3N/2 - 2` comparisons, but requires `N` additional storage. The paper (Finding a majority among N votes) also proves that `3N/2 - 2` is optimal.

Their approach is to rearrange all of the elements so that no two adjacent elements have the same value and keep track of the leftovers in a "bucket".

In the first pass, you start with an empty rearranged list and an empty "bucket". You take elements from your input and compare with the last element on the rearranged list. If they are equal you place the element in the "bucket". If they are not equal, you add the element to the end of the list and then move one element from the bucket to the end of the list as well. The last value on your list at the end of this phase is your majority candidate.

In the second pass, you repeatedly compare the candidate to the last value on the list. If they are the same, you discard two values from the end of the list. If they are different, you discard the last value from the end of the list and a value from the bucket. In this way you always pass over two values with one comparison. If the bucket ever empties, you are done and have no majority element. If you remove all elements from the rearranged list without emptying the bucket your candidate is the majority element.

Given the extra complexity and storage, I doubt this algorithm would have better real performance than Boyer-Moore in all but some contrived cases where equality comparison is especially expensive.

Distributed Boyer-Moore

Of course, Gregable readers probably know that I like to see if these things can be solved in parallel on multiple processors. It turns out that someone has done all of the fun mathematical proof to show how to solve this in parallel: Finding the Majority Element in Parallel.

Their solution boils down to an observation (with proof) that the first phase of Boyer-Moore can be solved by combining the results for sub-sequences of the original input as long as both the `candidate` and `count` values are preserved. So for instance, if you consider the following array:

`[1, 1, 1, 2, 1, 2, 1, 2, 2]` (Majority = 1)

If you were to run Boyer-Moore's first pass on this, you'd end up with:

```
candidate = 1
count = 1
```

If you were to split the array up into two parts and run Boyer-Moore on each of them, you'd get something like:

split 1:

```
[1, 1, 1, 2, 1]
candidate = 1
count = 3
```

split 2:

```
[2, 1, 2, 2]
candidate = 2
count = 2
```

You can then basically run Boyer-Moore over the resulting `candidate`, `count` pairs the same as you would if it were a list containing only the value `candidate` repeated `count` times. So for instance, Part 1's result could be considered the same as [1, 1, 1] and Part 2's as [2, 2]. However knowing that these are the same value repeated means you can generate the result for each part in constant time using something like the following python:

```
candidate = 0
count = 0
for candidate_i, count_i in parallel_output:
  if candidate_i = candidate:
    count += count_i
  else if count_i > count:
    count = count_i - count
```

```
        candidate = candidate_i
      else:
        count = count - count_i
```

This algorithm can be run multiple times as well to combine parallel outputs in a tree-like fashion if necessary for additional performance.

As a final step, a distributed count needs to be performed to verify the final candidate.

Blog of Greg Grothaus.

Software Engineer at Google. Outdoor adventurer type. Opinions expressed here are mine alone.