Divide & Conquer strikes back: maximum-subarray in linear time

Ovidiu Daescu and Shane St. Luce
Department of Computer Science
University of Texas at Dallas

I. Abstract

Given an array of n numbers, the maximum-subarray problem can be solved in linear time by a simple incremental algorithm that scans the input array from left to right. On the other hand, the divide & conquer solution for this problem has super-linear running time. In this short note we present a modification of the divide & conquer algorithm that takes only linear time and thus is optimal.

II. Introduction

Given an array A of n real numbers, the *maximum-subarray* problem asks to find a nonempty, contiguous subarray of A whose values have the largest sum [1]. The problem can be solved in $O(n)$ time with a simple, incremental (dynamic programming) algorithm, that scans A from left to right (Kadane's algorithm [2]; see also Pb. 4.1-5 in [1]). Since there is an obvious $\Omega(n)$ lower bound, this algorithm is optimal.

The well known divide & conquer approach to solve the maximum-subarray problem involves splitting the array in half by the median index and making recursive calls on each of the two subarrays to find the maximum subarray on the left half and the maximum subarray on the right half. The combine step searches for the maximum subarray that begins in the left half of the array and ends in the right half. An overall maximum is then reported as the maximum of the three (left, right, and cross). Since the combine step requires a scan from the middle index of A to the left and to the right, a linear term to the recurrence solution is added due to this step, resulting in a final recurrence of

$$T(n) = 2T(n/2) + cn$$

with time complexity of $\Theta(n \log n)$. See [1] for a detailed description. Thus, the known divide & conquer algorithm is not optimal, which is a bit surprising, considering that divide & conquer usually outperforms dynamic programming.

A Linear Time Solution: We present a faster divide & conquer algorithm that matches the $O(n)$ time of its dynamic programming counterpart and thus is optimal. To this end, we change the combine phase of the divide and conquer algorithm to a series of comparisons by returning additional information about the ends of the subarrays. Doing so reduces the time complexity of the combine step to $\Theta(1)$, and $\Theta(n)$ for the overall algorithm.

III. A Linear Time Divide & Conquer Solution

We start by noticing that in order to reduce the running time of the divide & conquer algorithm introduced earlier we should be able to perform the combine phase in sub-linear time. Our solution actually reduces the time complexity of the combine phase to $\Theta(1)$. To do so, we return additional information about the (sum of the) numbers in the recursive calls that can help compute the maximum subarray crossing the middle index in constant time. Specifically, if u is the current node and v and w are the left and right children of v, we compute the maximum contiguous subarray at u that starts in v and ends in w in $\Theta(1)$ time.

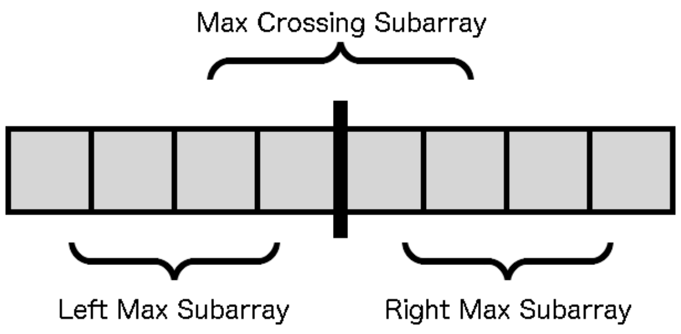

Fig. 1. Maximum Subarray Problem

The standard divide & conquer algorithm is given in Algorithm 1 with the combine phase detailed in Algorithm 2. A visual representation of the three subarrays can be viewed in Figure 1.

Two observations are important to make about the maximum crossing subarray. First, the maximum crossing subarray can be divided into two parts, the portion before the middle index and the portion after the middle index. Thus, these two parts of the subarray are located in the left and right subarrays, respectively. This is also evident from the two for loops in Algorithm 2 and visualized in Figure 2.

Second, the portion of the maximum crossing subarray in the left subarray of A is the maximum subarray ending at the middle index, and the portion in the right subarray of A is the maximum subarray beginning at the middle index. We refer to each of these maximum subarrays as the prefix and suffix of their corresponding halves of the array, as seen in Figure 3.

Algorithm 1 Divide-Conquer-Combine Algorithm

```
1: function FindMaxSubarray($A, low, high$)
2:     if $low = high$ then
3:         return $(low, high, A[low])$
4:     else
5:         $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$
6:         $L \leftarrow$ FindMaxSubarray($A, low, mid$)
7:         $R \leftarrow$ FindMaxSubarray($A, mid+1, high$)
8:         $C \leftarrow$ FMCS($A, low, mid, high$)
9:         if $L.maxSum >= R.maxSum$ and
10:            $L.maxSum >= C.maxSum$ then
11:            return $L$
12:        else if $R.maxSum >= L.maxSum$ and
13:            $R.maxSum >= C.maxSum$ then
14:            return $R$
15:        else
16:            return $C$
17:        end if
18:    end if
19: end function
```

Algorithm 2 Find Maximum Crossing Subarray

```
1: function FMCS($A, low, mid, high$)
2:     $leftSum \leftarrow -\infty$
3:     $sum \leftarrow 0$
4:     for $i \leftarrow mid$ downto $low$ do
5:         $sum \leftarrow sum + A[i]$
6:         if $sum > leftSum$ then
7:             $leftSum \leftarrow sum$
8:             $maxLeft \leftarrow i$
9:         end if
10:    end for
11:    $rightSum \leftarrow -\infty$
12:    $sum \leftarrow 0$
13:    for $j \leftarrow mid + 1$ to $high$ do
14:        $sum \leftarrow sum + A[j]$
15:        if $sum > rightSum$ then
16:            $rightSum \leftarrow sum$
17:            $maxRight \leftarrow j$
18:        end if
19:    end for
20:    $maxSum \leftarrow leftSum + rightSum$
21:    return $(maxLeft, maxRight,$
    $maxSum)$
22: end function
```

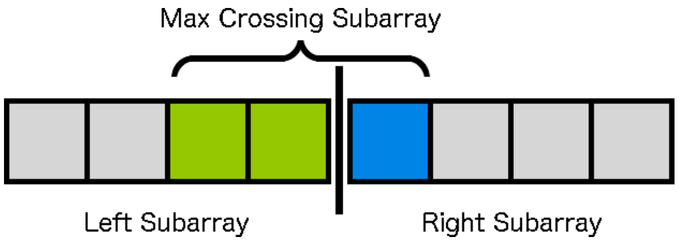

Fig. 2. Maximum Crossing Subarray using Prefix and Suffix

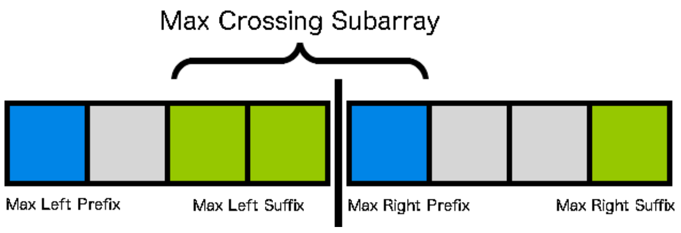

Fig. 3. Left and Right Prefixes and Suffixes

Algorithm 3 Alternate Algorithm

```
1: function FMS-Compare($A, low, high$)
2:     if $low = high$ then
3:         return $(A[low], A[low], A[low], A[low])$
4:     else
5:         $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$
6:         $Left \leftarrow$ FMS-Compare($A, low, mid$)
7:         $Right \leftarrow$ FMS-Compare($A, mid+1, high$)
8:         return Compare($A, Left, Right$)
9:     end if
10: end function
11:
12: function Compare($A, L, R$)
13:     $totalSum \leftarrow L.totalSum + R.totalSum$
14:     $maxPrefix \leftarrow$ max($L.maxPrefix,$
15:         $L.totalSum + R.maxPrefix$)
16:     $maxSuffix \leftarrow$ max($R.maxSuffix,$
17:         $R.totalSum + L.maxSuffix$)
18:     $maxSum \leftarrow$ max($L.maxSum, R.maxSum,$
19:         $L.maxSuffix + R.maxPrefix$)
20:     return $(totalSum, maxSum, maxPrefix,$
21:         $maxSuffix)$
22: end function
```

Observation: For a node u in the recursion tree, with left child v and right child w, the maximum crossing subarray is the union of the maximum left suffix (from v) and the maximum right prefix (from w).

Since each node u could be either a left or a right child of its parent we need to compute both the prefix and suffix of u.

Let $totalSum$, $maxSum$, $maxPrefix$, and $maxSufix$ denote the sum of the entries for the current array, its maximum-subarray, maximum-prefix, and maximum-suffix. Assuming a subarray must have at least 1 element and an array with only 1 element is considered the base case, we can return the value of the single cell as $totalSum$, $maxSum$, $maxPrefix$, $maxSuffix$. Otherwise, the $totalSum$, $maxSum$, $maxPrefix$, $maxSuffix$ tuples from the left child and the right child of a node are returned in L and R, respectively.

Lemma 1. *For a node u in the recursion tree, with left child v and right child w, the total sum, maximum subarray, maximum prefix and maximum suffix of u can be found in constant time from the $totalSum$, $maxSum$, $maxPrefix$, $maxSuffix$ tuples stored at v and w.*

Proof. Let L and R store the $totalSum$, $maxSum$, $maxPrefix$, $maxSuffix$ tuples returned from v and w, respectively. Then $totalSum$ at u is simply $L.totalSum + R.totalSum$.

Suppose the maximum prefix of the array at u does not cross the middle of the array. Then, the maximum prefix of that array must be the same as the maximum prefix of the left subarray, stored at v. If the maximum prefix indeed crosses the middle, then it must include the entire left subarray as well as the maximum prefix of the right subarray, stored at w. Then, the maximum prefix of the array stored at u is the maximum of $L.maxPrefix$ and $L.totalSum + R.maxPrefix$. See Figure 4 for an illustration. Following similar arguments, the maximum suffix at u is the maximum of $R.maxSuffix$ and $R.totalSum + L.maxSuffix$.

Lastly, the $maxSum$ of the array at u is determined by comparing $L.maxSum$ and $R.maxSum$ with the maximum crossing sum, $L.suffix + R.prefix$. Since the calculation after the recursive calls involves a constant number of comparisons and additions, the overall time to find the $totalSum$, $maxSum$, $maxPrefix$, $maxSuffix$ tuple at u is constant. $\square$

It follows from Lemma 1 that we achieve a constant time for the combine step and ultimately a recurrence of

$$T(n) = 2T(n/2) + d$$

for the overall algorithm, where d is a constant, which solves for linear time. The resulting algorithm is given in Algorithm 3. Note that the indexes of the maximum subarray are not shown in Algorithm 3 for simplicity, but they can be returned as well when making the final comparisons in the COMPARE function.

Theorem 1. *The maximum subarray problem can be solved in $O(n)$ time by divide & conquer.*

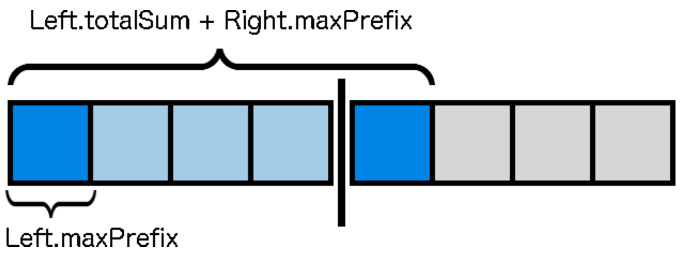

Fig. 4. Calculating the Prefix

IV. Conclusion

By recognizing that the maximum crossing subarray can be computed in constant time from additional information computed at the children in the recursion tree, we have designed an optimal, divide & conquer algorithm for the maximum-subarray problem, matching the time of the corresponding dynamic programming algorithm.

Like merge-sort, the maximum-subarray algorithm we presented can be implemented bottom-up, and thus no actual recursive calls are needed, making the divide & conquer algorithm competitive in practice, against the dynamic programming counterpart.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.
[2] J. Bentley, "Programming pearls: algorithm design techniques," *Commun ACM*, pp. 865–871, 1984.