Article

Not peer-reviewed version

Two Kadane algorithms for the maximum sum sub-array problem

Joseph B. Kadane *

Posted Date: 17 October 2023

doi: 10.20944/preprints202310.1061.v1

Keywords: dynamic programming; Kadane's algorithm; linear algorithm; maximum subarray problem; miscommunication

Article

Two Kadane Algorithms for the Maximum Sum Sub-Array Problem

Joseph B. Kadane

Carnegie Mellon University; kadane@stat.cmu.edu

Abstract: The algorithm now known as Kadane's algorithm for the maximum subarray problem uses many of the ideas that Kadane proposed, but it is not the algorithm that Kadane intended. This paper compares the two algorithms in question. They are both linear in time, employ just a few words of memory, and use a dynamic programming structure. The upshot is two light-weight algorithms for the maximum subarray problem.

Keywords: dynamic programming; Kadane's algorithm; linear algorithm; maximum subarray problem; miscommunication

1. History

In the late 1970's Jon Bentley and Michael Shamos (Computer Science) and I (Statistics) jointly taught a seminar course at Carnegie Mellon on the stochastic analysis of algorithms. The idea was to examine the relative usefulness of worst-case analysis (growing from minimax ideas of von Neumann and Morgenstern [1]) and average case analysis (growing from Savage's [2] Bayesian ideas). Although worst-case analyses were the dominant paradigm in computer-science, it seemed too pessimistic. For example, linear programming Dantzig [3] has a poor worst-case analysis (Klee and Minty)[4], but had been used successfully on very large problems.

The course was loosely structured, in part to encourage discussion about whatever technical issues people wanted to discuss. One day Shamos took the floor to talk about a problem he and Bentley had been discussing. Ulf Grenander at Brown had been studying how to analyze two-dimensional array data. The maximum likelihood estimate under his model required finding a contiguous area with high likelihood. To simplify the problem in the hope of better understanding its structure, he proposed a one-dimensional problem: given a vector of numbers, find the contiguous subvector with the largest sum. Grenander knew that a brute force method was of order n^3, and had constructed an n^2 algorithm. Shamos had devised a divide-and-conquer $n \log n$ algorithm, and Bentley and Shamos reported that they were having difficulty proving that $n \log n$ was the best possible rate for this problem. (The details of these algorithms are given in Bentley [5]).

I had never heard of this problem before, but it felt to me that Shamos' algorithm was ignoring the contiguity constraint rather than using it as part of the solution. So I said "I wouldn't do it that way, I'd do it this way." I cannot reconstruct the description I gave of my proposal alteration, but it used contiguity in an essential way to implement a dynamic programming-type algorithm. This idea was linear in n, as it scanned the input a single time. And it required only a hand-full of memory locations. So this explained why Bentley and Shamos were having difficulty proving that $n \log n$ was the best possible rate: it isn't.

Bentley [5] recounts this history, and gives a linear algorithm he attributes to me. In broad outline, it is the method I proposed in class. But a key detail is different.

The remainder of this paper is organized as follows: Section 2 gives my way of thinking about the problem, and the algorithm I thought I was proposing. Section 3 gives Bentley's version and compares the algorithms, and Section 4 concludes.

2. Growing champions

There are two simple conditions that the adjacent subsequences with the largest sum must have:

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 17 October 2023 doi:10.20944/preprints202310.1061.v1

(a) a maximal adjacent subsequence cannot have a starting sub-subsequence with a negative sum. Eliminating such a starting sub-subsequence and starting over must result in a larger sum for the subsequence, so the original subsequence cannot be optimal.
(b) after eliminating starting sub-subsequences with negative sums, the ensuing sub-subsequence must start non-negatively, so including it in the subsequence must increase (zeros don't affect the sum) the resulting sum. So an optimal subsequence must start immediately after the elimination of starting sub-subsequences with negative sums.

The algorithm is designed to exploit these ideas.

The Champ step eliminates negative starts (see (a) above) and then begins immediately (see (b) above).

Suppose the algorithm reports S as the largest sum among contiguous intervals. A user might want to know the starting and ending indices of the interval whose sum is S. Since Algorithm 1 is constructive, it can be modified to record the interval as it advances. This is implemented in Algorithm 2.

Algorithm 1 Linear algorithm based on Champ

```
MaxSoFar    : = − inf
Champ       : = − inf
For I = 1 to N do
    Champ     : =X[I] + Max(0.0, Champ)
    MaxSoFar  : = Max(MaxSoFar, Champ).
```

Algorithm 2 Algorithm 1 modified to report the start and end of the first interval whose sum is maximum.

```
MaxSoFar    : = − inf
Champ       : = − inf
Start       : = 1
End         : = 1
Cstart      : =1
For I = 1 to N do
      if    Champ < 0 then
            Cstart      : = I
            Champ       : = X[I]
      else  Champ       : = X[I] + Champ
      if    MaxSoFar < Champ
            Start       : = Cstart
            End         : = I
            MaxSoFar    : = Champ.
```

A user might want to know even more: the start and end of every interval whose sum is maximum. This might be inadvisable. For example, if the input vector is all zeros, the required storage for all possible optional intervals is of order n^2.

How do we know that Algorithm 1 (2) is correct? Every interval satisfying (a) and (b) is offered to MaxSoFar. Since the optimal interval must satisfy (a) and (b), the value of MaxSoFar after step N is optimal.

These algorithms were designed with the thought that the input vector would include both positive and negative elements. If the input is entirely positive, then the optimal contiguous sequence is the entire input, and the sum of the input is the optimal sum. But what happens with an entirely

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 17 October 2023 doi:10.20944/preprints202310.1061.v1

negative input? There are two possible kinds of answer a user might desire. The first is the largest of the input numbers (smallest in absolute value). Algorithms 1 and 2 deliver this result without change. The second kind of answer a user might want is the empty set. This can be offered by adding a line at the end of those algorithms (outside the loop) to report the empty set (designated however one wishes) if BestSoFar is negative.

3. The linear algorithm Bentley gave me credit for

This algorithm recursively calculates "BestEndingHere" at each stage. Formally it looks like this:

As in Algorithm 1, Algorithm 3 honors (a) by restarting in MaxEndingHere, and (b) by restarting immediately.

The next question is whether Algorithm 3 can be modified to give the start and end positions of an optimal subsequence. The following algorithm does that:

Algorithm 3 Same as Algorithm 4 in Bentley [5]

```
MaxSoFar            : = 0.0
MaxEndingHere       : = 0.0
for I = 1 to N do
      MaxEndingHere : =Max(0.0, MaxEndingHere + X[I])
      MaxSoFar      : = Max(MaxSoFar, MaxEndingHere).
```

Algorithm 4 Algorithm 3 modified to report the start and end of an optimal subsequence.

```
MaxSoFar            : = 0
MaxEndingHere       : = 0
Start               : = 1
End                 : = 1
Mstart              : =1
for I = 1 to N do
   if   MaxEndingHere + X[I] < 0.0 then
        Mstart          : = I
        MaxEndingHere   : = 0.0
   else MaxEndHere : = MaxEndhere + X[I]
   if   MaxSoFar < MaxEndingHere then
        MaxSoFar        : = MaxEndingHere
        Start = MStart
        End = I.
```

The correctness of Algorithm 3 can be seen by induction. If MaxEndingHere at I-1 is correct, then so is MaxEndingHere at I. Algorithm 3 has found various application, for example in Aygun [6].

An input vector that has only negative numbers leads, using Algorithm 3, to a MaxSoFar of zero, and the empty set. In this particular, it differs in behavior from Algorithm 1. A result of 0 could also occur if the input is non-positive and includes at least one 0. Hence an Algorithm 3 result of 0 is ambiguous.

These algorithms look very similar. The driving mechanism of Algorithm 1 is

$$\text{Champ} = \text{X[I]} + \text{Max}(0.0, \text{Champ}), \tag{1}$$

while that of algorithm 3 is

$$\text{MaxEndingHere} = \text{Max}(0.0, \text{MaxEndingHere} + \text{X[I]}. \tag{2}$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 17 October 2023 doi:10.20944/preprints202310.1061.v1

To make them look even more similar, (1) can be rewritten as

$$\text{Champ} = \text{Max}(\text{X[I]}, \text{Champ} + \text{X[I]}. \tag{3}$$

Nonetheless, they are different. The relationship between the two algorithms is given in the following proposition:

$$\text{MaxEndingHere} = \text{Max}(0, \text{Champ}). \tag{4}$$

The proof of 4 is given in the Appendix.

Table 1 compares them.

Table 1. Comparison of Algorithms 1 and 3.

	Algorithm 1	**Algorithm 3**
Time	0(n)	0(n)
Space	0(1)	0(1)
Correct?	Yes	Yes
Modify to report start and end	Algorithm 2	Algorithm 4
Negative input	Either empty set or largest element	Empty set only

I slightly prefer Algorithms 1 and 2 to Algorithms 3 and 4 because they handle the all-negative-input case more smoothly.

4. Discussion

Memories are tricky things. Bentley (private communication) was writing his column some five years after the seminar. He believes that, at the time of the seminar he understood my algorithm to be Algorithm 1, but that his later reconstruction of it resulted in Algorithm 3. That it took 40 years to recognize that such a misunderstanding had occurred is entirely on me.

We now have two similar but different light-weight algorithms for the maximum subarray problem. Even though algorithmically they are virtually identical, they reflect different ways of thinking about the maximum subarray problem. That there are two is a gain in our knowledge, and raises new questions. Are there other such algorithms that are similarly light-weight, or are these two unique in some sense? Can the ideas behind these algorithms aid in the two-dimensional problem of Grenander, either with an exact algorithm or heuristically? Algorithms are continuously fascinating.

Appendix A. Proof of 4

Proof. Proof by induction on I.

$$\text{At } I = 1, \text{Champ}[1] = \text{X}[1]; \text{MaxEndHere}[1] = \text{Max}(0, \text{X}[1]) = \text{Max}(0, \text{Champ}[1]).$$

Suppose the proposition is true at I.

Case 1: $\text{Champ}[\text{I}] \geq 0$. Then by the inductive hypothesis,

$$\text{MaxEndingHere}[\text{I}] = \text{Champ}[\text{I}] \geq 0$$

so

$$\begin{aligned} \text{MaxEndingHere}[\text{I}+1] &= \text{Max}(0, \text{MaxEndingHere}[\text{I}] + \text{X}[\text{I}+1] \\ &= \text{Max}(0, \text{Champ}[\text{I}] + \text{X}[\text{I}+1] \\ &= \text{Max}(0, \text{Champ}[\text{I}+1]. \end{aligned}$$

 doi:10.20944/preprints202310.1061.v1

Case 2: $\text{Champ}[\text{I}] < 0$. Then by inductive hypothesis, $\text{MaxEndingHere}[\text{I}] = 0$ then

$$\begin{aligned} \text{MaxEndingHere}[\text{I}+1] &= \text{Max}(0, \text{MaxEndsHere}[\text{I}] + \text{X}[\text{I}+1] \\ &= \text{Max}(0, \text{X}[\text{I}+1]). \end{aligned}$$

$$\text{Champ}[\text{I}+1] = \text{X}[\text{I}+1] + \text{Max}(0, \text{Champ}[\text{I}] = \text{X}[\text{I}+1].$$

Hence

$$\text{MaxEndingHere}[\text{I}+1] = \text{Max}(0, \text{Champ}[\text{I}+1]$$

□

References

1. von Neumann, J.; Morgenstern, O. *Theory of Games and Economic Behavior*; Princeton University Press: New Jersey, 1944.
2. Savage, L.J. *Foundations of Statistics*; J. Wiley and Sons: New York, 1954.
3. Dantzig, G. *Linear Programming and Extensions*; Princeton University Press, 1963.
4. Klee, V.; Minty, G.J. How good is the simplex algorithm? In *Inequalities III (Proceedings of the Third Symposium on Inequalities held at the University of California, Los Angeles, Calif., September 1–9, 1969, dedicated to the memory of Theodore S. Motzkin), Shisha, Oved (ed.)*; Academic Press: New York-London:, 1972; pp. 159–175.
5. Bentley, J. Algorithm Design Techniques. *Communications of the ACM* **1984**, *(9) 27*, 865–871.
6. Aygun, R.S. Using Maximum Sum Subarrays for Approximate String Matching. *Annals of Data Science* **2017**, *4*, 503–531.