Article

Modified Algorithm for 2D Maximum Sum Subarray Problem

Boris Shukhat

Independent Researcher, USA; b_shukhat@yahoo.com

Abstract

This paper presents a two-phase adaptive algorithm to solve the 2-Dimensional Maximum Sum Subarray Problem. By reframing the search order to establish a single-column baseline first, the algorithm generates mathematical pruning bounds in $O(NM)$ time. These bounds are utilized in a second phase to skip unpromising multi-column scans in $O(1)$ time. This "Two-Phase" approach achieves a quadratic best-case floor of $O(M^2 + NM)$, while significantly improving the expected performance across typical data distributions and maintaining the cubic worst-case. This adaptive strategy effectively bridges the gap between theoretical sub-cubic complexity and practical implementation.

Keywords: maximum sum sub-array; Kadane's algorithm; branch-and-bound; pruning heuristics; summed area table; data-dependent performance

---

1. Background

The classical approach to the 2D Maximum Sum Subarray Problem is an extension of the 1D problem famously solved by Kadane's algorithm [1,2]. Traditionally, the 2D variant is reduced to $O(M^2)$ 1D problems applied to all possible aggregated columns (strips). This exhaustive cubic approach remains the benchmark in algorithmic theory due to its simplicity and low constant-factor overhead.

While research, notably [3], has focused on lowering theoretical asymptotic bounds using complex transformations, these methods often encounter substantial practical limitations. This paper proposes a middle ground that prioritizes algorithmic engineering and practical effectiveness via branch-and-bound principles [4].

2. Formal Pruning Logic

The efficiency of the proposed algorithm relies on establishing a mathematical bound that allows for the skipping of vertical scans. We define $M_c(j)$ as the maximum 1D subarray sum of column j in the original matrix A.

Theorem 1. *For any aggregated column bounded horizontally by columns $[c_1, c_2]$, the maximum subarray sum S is strictly bounded by the sum of individual column maximums:*

$$S \leq \sum_{j=c_1}^{c_2} M_c(j)$$

Proof. Let $A_{i,j}$ denote the elements of the original matrix A. Consider any aggregated column defined by column indices $[c_1, c_2]$. A subarray of this aggregated column corresponds to a row range $[r_1, r_2]$.

The sum S is given by:

$$S = \sum_{i=r_1}^{r_2} \sum_{j=c_1}^{c_2} A_{i,j}$$

By reordering the summations, we express the total sum as the sum of column slices:

$$S = \sum_{j=c_1}^{c_2} \left(\sum_{i=r_1}^{r_2} A_{i,j} \right)$$

For any specific column j, the term $\sum_{i=r_1}^{r_2} A_{i,j} \leq M_c(j)$ by definition of the 1D column maximum. Summing this inequality across all columns in the range $[c_1, c_2]$ yields:

$$\sum_{j=c_1}^{c_2} \sum_{i=r_1}^{r_2} A_{i,j} \leq \sum_{j=c_1}^{c_2} M_c(j)$$

Thus, $S \leq \sum_{j=c_1}^{c_2} M_c(j)$, proving that the sum of pre-calculated column maximums provides a rigorous upper bound for the maximum subarray sum of the aggregated column. □

Corollary 1. *If the upper bound established in Theorem 1 is less than or equal to the current global maximum sum found by the algorithm, the $O(N)$ vertical scan for the corresponding aggregated column $[c_1, c_2]$ can be safely bypassed.*

3. Two-Phase Adaptive Search

For simplicity, this implementation focuses exclusively on determining the value of the maximum sub-region sum; it does not explicitly track or return the coordinates (r_1, r_2, c_1, c_2) of the resulting sub-matrix.

Phase 1: Baseline and Bound Preparation: The algorithm begins with a mandatory $O(NM)$ pass to initialize the search environment. It performs a row scan of each row i to find its maximum $M_r(i)$, simultaneously constructing the *rowMaxPrefixSum* bound array. It then performs a column scan of each column j to find $M_c(j)$, simultaneously constructing the *columnMaxPrefixSum*. The results from both row and column baseline scans update the $globalMaxSum$, establishing a high pruning floor immediately. Simultaneously, a 2D Summed Area Table (SAT) [5] is constructed.

Phase 2: Multi-Column Pruning: This phase executes the adaptive search by iterating through all multi-column pairs $L < R$. For each pair, if the upper bound derived from *columnMaxPrefixSum* is no greater than $globalMaxSum$, the entire region is bypassed. Otherwise, the vertical worker performs a scan, utilizing the SAT for constant-time row-sum retrieval and *rowMaxPrefixSum* to terminate early if no improvement over $globalMaxSum$ is mathematically possible.

4. Implementation

Algorithm 1 PrunedKadane1D Vertical Worker

```
function PRUNEDKADANE1D(slice, rowMaxPrefixSum, globalMaxSum)
    currSum ← 0, localMaxSum ← −∞, N ← slice.length
    for i = 0 to N − 1 do
        currSum ← currSum + slice[i]
        if currSum > localMaxSum then
            localMaxSum ← currSum
        end if
        if currSum < 0 then
            currSum ← 0, startRow ← i + 1
            if startRow < N then
                upperBound ← rowMaxPrefixSum[N] − rowMaxPrefixSum[startRow]
                if upperBound ≤ globalMaxSum then
                    break
                end if
            end if
        end if
    end for
    return localMaxSum
end function
```

Algorithm 2 Adaptive 2D Search Manager

$globalMaxSum \leftarrow -\infty, N \leftarrow RowCount, M \leftarrow ColCount$
Precompute SAT ▷ $O(NM)$; can be performed inside the row scan loop
$rowMaxPrefixSum \leftarrow$ new Array of size $N+1, rowMaxPrefixSum[0] \leftarrow 0$
for $i = 0$ to $N-1$ **do** ▷ Phase 1: Row Scan
  $rowMax \leftarrow$ Kadane1D on row i
  $rowMaxPrefixSum[i+1] \leftarrow rowMaxPrefixSum[i] + rowMax$
  if $rowMax > globalMaxSum$ **then**
    $globalMaxSum \leftarrow rowMax$
  end if
end for
$columnMaxPrefixSum \leftarrow$ new Array of size $M+1, columnMaxPrefixSum[0] \leftarrow 0$
for $j = 0$ to $M-1$ **do** ▷ Phase 1: Column Scan
  $colMax \leftarrow$ PRUNEDKADANE1D(column $j, rowMaxPrefixSum, globalMaxSum$)
  $columnMaxPrefixSum[j+1] \leftarrow columnMaxPrefixSum[j] + colMax$
  if $colMax > globalMaxSum$ **then**
    $globalMaxSum \leftarrow colMax$
  end if
end for
for $L = 0$ to $M-1$ **do** ▷ Phase 2: Multi-Column Search
  for $R = L+1$ to $M-1$ **do**
    $upperBound \leftarrow columnMaxPrefixSum[R+1] - columnMaxPrefixSum[L]$
    if $upperBound \leq globalMaxSum$ **then**
      continue
    end if
    for $i = 0$ to $N-1$ **do**
      $currSlice[i] \leftarrow SAT[i+1][R+1] - SAT[i][R+1] - SAT[i+1][L] + SAT[i][L]$
    end for
    $res \leftarrow$ PRUNEDKADANE1D$(currSlice, rowMaxPrefixSum, globalMaxSum)$
    if $res > globalMaxSum$ **then**
      $globalMaxSum \leftarrow res$
    end if
  end for
end for
return $globalMaxSum$

5. Complexity Analysis

The computational complexity is data-dependent, oscillating between a cubic worst-case and a quadratic best-case. By establishing a high global maximum and all prefix sum bounds early in Phase 1, the solver achieves a best-case floor of $O(M^2 + NM)$ when the majority of multi-column search regions are successfully pruned.

Table 1. Complexity Summary.

Case	Complexity	Condition
Worst-Case	$O(M^2N)$	Late discovery of global max or loose heuristics
Empirical Benchmark	Pruned $O(M^2N)$	Uniformly distributed values (stress-test environment)
Best-Case	$O(M^2 + NM)$	All multi-column regions pruned after Phase 1

6. Experimental Results and Discussion

Empirical evaluation on matrices populated with values drawn from a uniform distribution ($[-100, 100]$) demonstrates that the pruning heuristics are effective even in high-entropy environments (which can be considered a stress-test). Java-based benchmarks on 1000×1000 matrices indicate a consistent 20% reduction in total execution time compared to a standard non-adaptive implementation. If the matrix were sparse (mostly negative with a few positive "islands"), the time reduction would jump from 20% to

over 80%. While the column-skipping rate is lower in perfectly uniform noise, the vertical worker's ability to terminate early via the row-maximum bounds provides significant computational savings.

7. Performance Comparison

Compared to Takaoka's sub-cubic algorithm [3], this adaptive approach offers superior practical scalability and maintains an $O(NM)$ space efficiency. This is particularly critical for "tall" matrices where $N \gg M$, as Takaoka's method requires $O(M^2)$ auxiliary memory.

Table 2. Algorithm Metric Comparison.

Metric	Kadane 2D	Takaoka	Adaptive Pruned
Complexity	$O(M^2N)$	$O\left(M^2N\sqrt{\frac{\log\log M}{\log M}}\right)$	$O(M^2N)$ to $O(M^2+NM)$
Memory	$O(NM)$	$O(M^2)$	$O(NM)$

8. Extensibility

8.1. Parallelization

The dual-loop structure over column indices L and R is naturally parallel. An atomic update mechanism for the record maximum allows parallel workers to immediately benefit from pruning triggers discovered by any thread.

8.2. Multi-Dimensional Case

The pruning logic can be generalized to d-dimensional arrays ($d > 2$). By nesting the $(d-1)$-dimensional pruning bounds, an algorithm could navigate d-dimensional space with a best-case polynomial floor of $O(N^{2d-2})$.

8.3. Greedy Search Order

A "Greedy Search" optimization could involve using a Max-Heap to prioritize column ranges based on their pre-calculated bounds. Evaluating high-potential regions first maximizes the probability of early pruning in the remaining search space.

9. Conclusion

This adaptive approach bridges the gap between theoretical sub-cubic models and standard cubic implementations. By utilizing dual-level prefix sum heuristics and Summed Area Tables, the algorithm achieves a best-case floor of $O(M^2+NM)$ and significantly improves performance across a variety of input distributions.

Supplementary Materials: The following supporting information can be downloaded at the website of this paper posted on Preprints.org.

Acknowledgments: The author acknowledges the use of artificial intelligence tools for assistance in technical writing and LaTeX document preparation. The author specifically utilized an AI-assisted execution environment for Python to empirically verify the pruning rates and algorithmic benchmarks cited in this work. The algorithms and their verification remain the sole responsibility of the author.

References

1. Bentley, J. (1984). "Programming Pearls: Algorithm Design Techniques." *Communications of the ACM*, 27(9), 865–873.
2. Kadane, J. B. (2023). "Two Kadane Algorithms for the Maximum Sum Subarray Problem." *Algorithms*, 16(11), 519.
3. Takaoka, T. (2002). "Efficient algorithms for the maximum subarray problem by distance matrix multiplication." *ENTCS*, 66(1), 191–200.
4. Land, A. H., and Doig, A. G. (1960). "An automatic method of solving discrete programming problems." *Econometrica*, 28(3), 497–520.
5. Crow, F. C. (1984). "Summed-area tables for texture mapping." *ACM SIGGRAPH Computer Graphics*, 18(3), 207–212.