Freie Universität Berlin

SMAWK in Dynamic Programming

Bachelor's Thesis

Author:
Xiaoxiao Shen

Student ID:
5579686

Supervisor:
Prof. Dr. László Kozma

June 20, 2025

Abstract

The SMAWK algorithm efficiently computes the row minima of an $m \times n$ totally monotone matrix in $O(m + n)$ time. In certain dynamic programming problems, such matrices arise implicitly, allowing SMAWK to serve as an optimization technique for such problems. This thesis explores how and when SMAWK can be applied within dynamic programming contexts. We first present the algorithm itself. We then apply SMAWK to two distinct problems: the Web Proxy Placement Problem and a variant of the Line Breaking Problem. In addition to the theoretical analysis, we implement the algorithm and analyze its practical performance in the context of the Web Proxy Placement Problem, confirming its efficiency compared to the naive approach.

Contents

1 Introduction

In many algorithmic applications, especially in the context of dynamic programming, one often implicitly encounters the task of computing the row minima of a matrix. While this problem appears straightforward and usually takes $O(mn)$ time for an $m \times n$ matrix, in many practical cases the matrix exhibits additional structure that can be used to achieve significant speed-ups. In particular, if the matrix is *totally monotone*, it can be solved in just $O(m + n)$ time using the SMAWK algorithm, originally developed by Aggarwal, Klawe, Moran, Shor, and Wilber [1] — the name SMAWK itself being an acronym of their initials. There, the algorithm was first introduced in the field of computational geometry, where it was used to compute the farthest point from each vertex of a convex polygon. The property of total monotonicity arises implicitly in many dynamic programming problems, for example, in cost functions. This structure is not always explicitly recognized, but identifying and exploiting total monotonicity in such cases allows us to reduce the overall computational complexity of the program. Applications where SMAWK can be utilized are widespread across different domains. For instance, problems in image processing [2] and bioinformatics [3]. In this thesis, we focus on two other representative applications: *The Web Proxy Placement Problem*, a resource allocation problem on network topologies [4], and *The Line Breaking Problem*, a classical example in text formatting [5].

This thesis investigates how the SMAWK algorithm can be used to efficiently solve row-minima subproblems in dynamic programming settings. The focus lies not only on the theoretical properties of the algorithm, but also on its practical applicability and effectiveness. Specifically, the work examines to what extent SMAWK leads to measurable performance improvements over naive approaches, and under which conditions its use is most beneficial in real-world algorithmic scenarios.

The thesis is structured as follows: Chapter 2 introduces the theoretical foundations necessary for understanding the SMAWK algorithm. Chapter 3 explains the SMAWK algorithm in detail. Chapter 4 discusses the Web Proxy Placement Problem as a concrete application. This chapter illustrates how totally monotone matrices arise naturally in a dynamic programming formulation and how SMAWK can be applied to improve performance. Chapter 5 introduces a second application with a different character, aimed at showcasing the versatility of the SMAWK algorithm. Chapter 6 provides a practical performance analysis with experimental results based on the Web Proxy Placement Problem. It compares the efficiency of the SMAWK-optimized solution against the naive dynamic programming approach.

2 Theoretical Foundations

In this chapter, we introduce fundamental theoretical concepts that are necessary for understanding the SMAWK algorithm. These foundations were first introduced by [1], while additional observations are based on [6].

Let M be an $m \times n$ matrix with entries from $\overline{\mathbb{R}}$ (i.e., real numbers and possibly $\pm\infty$), where m denotes the number of rows and n the number of columns. Let $j(i)$ denote the index of the leftmost column containing the minimum value in row i of M.

Definition 2.1. *v A matrix M is called monotone if for all row indices i, the following condition holds:*

$$j(i) \leq j(i+1).$$

Definition 2.2. *A matrix M is called totally monotone if every 2×2 submatrix of M – formed by selecting any two distinct rows and columns, is monotone.*

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 27 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 25 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix} \qquad \begin{bmatrix} 12 & 21 & 38 & 76 & 89 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 20 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

(a) monotone, but not totally monotone (b) totally monotone

Figure 2.1: Example of monotone and totally monotone matrices. The blue cells indicate the leftmost minima in each row. The red-marked entries highlight a 2×2 submatrix, that is not monotone.

The most prominent special case of totally monotone matrices is the class of *Monge* matrices.

Definition 2.3. *A matrix M is called Monge if, for any two row indices $i < i'$ and any two column indices $j < j'$, we have:*

$$M[i,j] + M[i',j'] \leq M[i,j'] + M[i',j].$$

Lemma 2.1. *Every Monge matrix is totally monotone, but not necessarily vice versa.*

$$\begin{bmatrix} 12 & 21 & 38 & 76 \\ 47 & 14 & 14 & 29 \\ 21 & 8 & 20 & 10 \\ 68 & 16 & 29 & 15 \end{bmatrix}$$

Figure 2.2: Example of a totally monotone matrix that is not Monge. The blue cells indicate the leftmost minima in each row. The red-marked entries highlight a 2×2 submatrix, that violates the Monge property.

Proof. Assume that M is a matrix that is not totally monotone. This means there must exist a submatrix of M that violates the monotonicity condition. There exist indices $i < i'$ and $j < j'$ such that the following inequalities hold: $M[i,j] > M[i,j']$ and $M[i',j'] \geq M[i',j]$. Adding both sides gives: $M[i,j] + M[i',j'] > M[i,j'] + M[i',j]$. This contradicts the Monge property. Thus, any matrix that is not totally monotone cannot be Monge (proof by contradiction). □

Lemma 2.2. *The transpose M^T of any Monge array M is also Monge.*

Proof. We know that the following inequality holds for all $i < i'$ and $j < j'$:

$$M[i,j] + M[i',j'] \leq M[i,j'] + M[i',j].$$

The transpose M^T of M is defined as $M^T[i,j] = M[j,i]$. Substituting this into the inequality, we obtain:

$$M^T[j,i] + M^T[i',j'] \leq M^T[i,j'] + M^T[i',j].$$

Thus, M^T also satisfies the Monge property. □

To make use of total monotonicity in the SMAWK algorithm, we rely on some key observations. The following lemmas describe properties of totally monotone matrices that will later be used in the design of the SMAWK algorithm.

Lemma 2.3. *For a totally monotone matrix M, if $j < j'$ and $M[i',j] \leq M[i',j']$, then for all $i \leq i'$, we have $j(i) \neq j'$.*

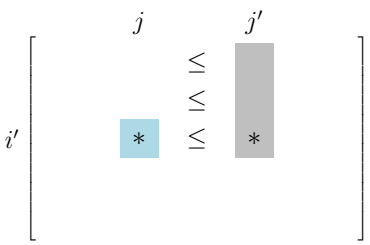

Figure 2.3: The $*$ - marked cells represent $M[i,j]$ (left) and $M[i,j]$ (right). The gray cells indicate entries $M[i,j]$ for all $i \leq i'$, which cannot contain the leftmost minimum.

Proof. By total monotonicity, the condition $M[i', j] \leq M[i', j']$ implies that for all $i \leq i'$ we have $M[i, j] \leq M[i, j']$. Entry j' cannot be the row minimum for row i. Thus, for all $i \leq i'$, we conclude that $j(i) \neq j'$. □

Lemma 2.4. *For a totally monotone matrix M, if $j < j'$ and $M[i, j] > M[i, j']$, then for all $i' \geq i$ we have $j(i') \neq j$.*

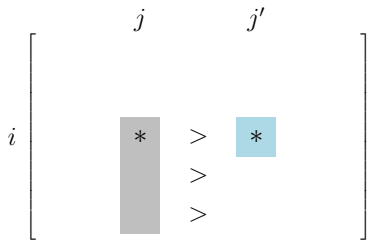

Figure 2.4: The $*$ - marked cells represent $M[i, j]$ (left) and $M[i, j]$ (right). The gray cells indicate entries $M[i', j]$ for all $i' \geq i$, which cannot contain the leftmost minimum.

Proof. By total monotonicity, the condition $M[i, j] > M[i, j']$ implies that for all $i' \geq i$ we have $M[i', j] > M[i', j']$. Entry j cannot be the row minimum for any row i'. Thus, for all $i' \geq i$, we conclude that $j(i') \neq j$. □

3 The SMAWK Algorithm

The SMAWK algorithm computes all row minima of an $m \times n$ totally monotone matrix by recursively applying the two subroutines: *Reduce* and *Interpolate*. The algorithm's general structure and pseudocode follow established descriptions [7, 8]. We assume that matrix indices start at 1.

- If $m = 1$, the row minimum is found by linear search.
- If $n > m$, the *Reduce* subroutine is applied. This step reduces the problem of finding the row minima of an $m \times n$ totally monotone matrix to finding the row minima of a $m \times n'$ matrix, where $n' \leq m$.
- If $m \geq n$, the *Interpolate* subroutine is applied. This step reduces the problem to finding the row minima of a $\left\lfloor \frac{m}{2} \right\rfloor \times n$ matrix.

3.1 Reduce

Given an $m \times n$ totally monotone matrix, the goal of the *Reduce* subroutine is to eliminate at least $n - m$ unnecessary columns by leveraging total monotonicity, while ensuring that all row minima are preserved.

The subroutine maintains a stack S of remaining column indices. The fundamental idea is to compare column entries within the same row. An entry or column is considered *dead* if it cannot contain a row minimum.

The algorithm follows these steps:

- It iterates over each column C_j one by one, from left to right.
- If the stack is empty, the current column C_j is added to the stack, and its first entry (row 1) is marked as its *head*.
- Each column in the stack has a designated *head*, which is the topmost surviving entry in that column. Any entry above the *head* is *dead*, as it cannot contain a row minimum. The *head* of a column corresponds to its position within the stack: if there are k elements in S, the column at the top is assigned *head* row k.
- Let $M[i, \ell]$ be the *head* of the top column C_ℓ in S. The algorithm compares $M[i, \ell]$ with $M[i, j]$ (the entry of the i-th row of C_j):
  - If $M[i, \ell] > M[i, j]$, then $M[i, \ell]$ and everything below is *dead* by Lemma 2.4. Since everything above the *head* is also *dead*, C_ℓ is popped from the stack and C_j competes against the next column in the stack.

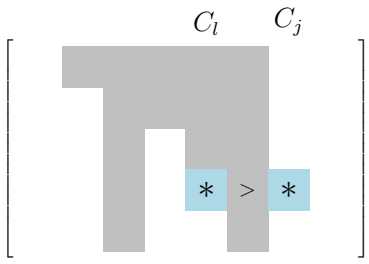

Figure 3.1: Illustration of why column C_ℓ is considered dead. The entry $M[i, \ell]$ (left) is greater than $M[i, j]$ (right). The gray cells represent cells, that are dead.

– If $i < n$ and $M[i, \ell] \leq M[i, j]$ push the current column C_j onto S. By Lemma 2.3 this means that all entries above $M[i, j]$ are *dead*. This does not exclude the possibility that a row minimum could be found below $M[i, j]$. Thus the *head* position of C_j is set to $i + 1$.

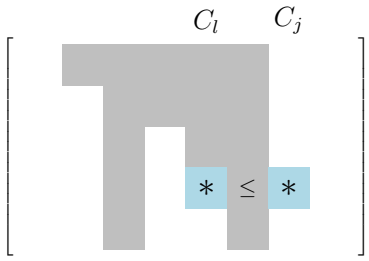

Figure 3.2: Illustration of why the entries above $M[i, j]$ in C_j are considered dead. The entry $M[i, \ell]$ (left) is less than or equal to $M[i, j]$ (right). The gray cells represent cells, that are dead.

– For $i \geq n$, since there are no more rows to process. In this case, C_j is no longer considered.

After *Reduce*, SMAWK is then recursively applied to the reduced matrix M', which consists of the columns indexed by S.

The following pseudocode presents the *Reduce* procedure, following the description above:

Algorithm 1 Reduce Subroutine

Require: totally monotone $m \times n$ matrix M, where $n > m$
Initialize: $S \leftarrow [\,]$
for $j = 1$ **to** n **do**
  while $|S| > 0$ **and** $M[|S|, S.\text{top}] > M[|S|, j]$ **do**
    $S.\text{pop}()$
  end while
  if $|S| < m$ **then**
    $S.\text{push}(j)$
  end if
end for
Let M' be the submatrix of M containing only columns indexed by S
Call SMAWK on M'

3.2 Interpolate

The *Interpolate* step begins by recursively applying the SMAWK algorithm to a submatrix M' which consists only of the even-indexed rows of M.

For each odd-indexed row i, its minimum must lie within the column range $[j(i-1), j(i+1)]$, where $j(i-1)$ and $j(i+1)$ are the previously computed minima of its adjacent even-indexed rows. For the special cases: if $i = 1$, there is no preceding even-indexed row, so the search range is $[1, j(2)]$ and if $i = m$, there is no following even-indexed row, so the search range is $[j(m-1), n]$. The minimum for each odd-indexed row is found by performing a linear search within this restricted range.

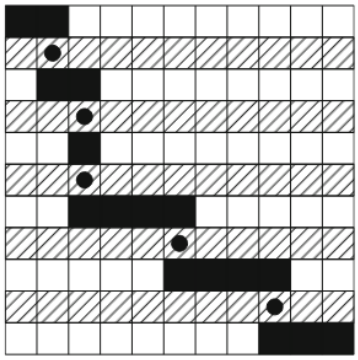

Figure 3.3: Row minima of even-indexed rows (•) are computed first. The black rectangles indicate the search intervals for odd-indexed rows.

The following pseudocode presents the *Interpolate* procedure, following the description above:

Algorithm 2 Interpolate Subroutine

Require: totally monotone $m \times n$ matrix M, where $n \leq m$
1: Let M' be the submatrix of all even-indexed rows from M
2: Call SMAWK on M'
3: **for all** odd i in $1, \ldots, m$ **do**
4: **if** $i = 1$ **then**
5: $\ell \leftarrow 1, r \leftarrow j(2)$
6: **else if** $i = m$ **then**
7: $\ell \leftarrow j(m-1), r \leftarrow n$
8: **else**
9: $\ell \leftarrow j(i-1), r \leftarrow j(i+1)$
10: **end if**
11: Find $j(i) \in [\ell, r]$ by linear search
12: **end for**

3.3 Example of the Algorithm

In this section, we demonstrate one full iteration of the SMAWK algorithm on a small matrix. This example illustrates how *Reduce* and *Interpolate* work together.

Consider the following 4×5 matrix:

$$\begin{bmatrix} 13 & 10 & 20 & 13 & 19 \\ 26 & 20 & 29 & 21 & 25 \\ 58 & 48 & 49 & 39 & 42 \\ 100 & 86 & 82 & 65 & 61 \end{bmatrix}$$

1. Reduce

1.1 *The stack evolution is as follows:*

Current Column	Compared Entry	Action	Stack
1	–	push 1	[1]
2	$M[1,1]$ vs. $M[1,2]$	pop 1	[ ]
	–	push 2	[2]
3	$M[1,2]$ vs. $M[1,3]$	push 3	[2, 3]
4	$M[2,3]$ vs. $M[2,4]$	pop 3	[2]
	$M[1,2]$ vs. $M[1,3]$	push 4	[2, 4]
5	$M[2,4]$ vs. $M[2,5]$	push 5	[2, 4, 5]

1.2 *We apply SMAWK recursively to the submatrix consisting of the columns* $[2, 4, 5]$*:*

$$\begin{array}{c} \ 2 \quad 4 \quad 5 \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 10 & 13 & 19 \\ 20 & 21 & 25 \\ 48 & 39 & 42 \\ 86 & 65 & 61 \end{bmatrix} \end{array}$$

2. Interpolate

2.1 *Recursively apply SMAWK onto the matrix consisting of all even-index rows:*

$$\begin{array}{c} \ 2 \quad 4 \quad 5 \\ \begin{array}{c} 2 \\ 4 \end{array} \begin{bmatrix} 20 & 21 & 25 \\ 86 & 65 & 61 \end{bmatrix} \end{array}$$

3. Reduce

3.1 *The stack evaluation is as follows:*

Current Column	Compared Entry	Action	Stack
2	–	push 2	[2]
4	$M[2,2]$ vs. $M[2,4]$	push 4	[2,4]
5	$M[4,4]$ vs. $M[4,5]$	pop 4	[2]
	–	push 5	[2,5]

3.2 *We apply SMAWK recursively to the submatrix consisting of the columns* $[2,5]$*:*

$$\begin{array}{c} \\ 2 \\ 4 \end{array}\begin{array}{c} \begin{array}{cc} 2 & 5 \end{array} \\ \begin{bmatrix} 20 & 25 \\ 86 & 61 \end{bmatrix} \end{array}$$

4. Interpolate

4.1 *Recursively apply SMAWK onto the matrix consisting of every second row:*

$$\begin{array}{c} \\ 4 \end{array}\begin{array}{c} \begin{array}{cc} 2 & 5 \end{array} \\ \begin{bmatrix} 86 & 61 \end{bmatrix} \end{array}$$

5. Base Case

Our matrix consists of only one row. We find the row minima via linear search:

$$\begin{array}{c} \\ 4 \end{array}\begin{array}{c} \begin{array}{cc} 2 & 5 \end{array} \\ \begin{bmatrix} 86 & 61 \end{bmatrix} \end{array}$$

4.2 *Find row minima for every odd row via linear search in the restricted area:*

$$\begin{array}{c} \\ 2 \\ 4 \end{array}\begin{array}{c} \begin{array}{cc} 2 & 5 \end{array} \\ \begin{bmatrix} 20 & 25 \\ 86 & 61 \end{bmatrix} \end{array}$$

Row minima:

$$\begin{array}{c} \\ 2 \\ 4 \end{array}\begin{array}{c} \begin{array}{cc} 2 & 5 \end{array} \\ \begin{bmatrix} 20 & 25 \\ 86 & 61 \end{bmatrix} \end{array}$$

2.2 *Find row minima for every odd row via linear search in the restricted area:*

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{ccc} 2 & 4 & 5 \end{array} \\ \begin{bmatrix} 10 & 13 & 19 \\ 20 & 21 & 25 \\ 48 & 39 & 42 \\ 86 & 65 & 61 \end{bmatrix} \end{array}$$

Final Result:

$$\begin{bmatrix} 13 & 10 & 20 & 13 & 19 \\ 26 & 20 & 29 & 21 & 25 \\ 58 & 48 & 49 & 39 & 42 \\ 100 & 86 & 82 & 65 & 61 \end{bmatrix}$$

3.4 Time Complexity

The runtime of the *Reduce* algorithm (see Algorithm 1) is determined by the number of push and pop operations performed. Each column index j is processed exactly once, leading to at most n push operations. Since each column is pushed at most once, the total number of pop operations is also at most n. The total number of operations is bounded by at most $2n$. Thus, the initial column reduction operation of the algorithm is $O(n)$. After *Reduce* at most m columns remain. The algorithm then recursively processes a matrix with $m/2$ rows (due to *Interpolate*) and at most m columns.

The complexity of *Reduce* across all recursion levels leads to the recurrence:

$$T(m, n) \leq O(n) + T(m/2, m).$$

We conclude that *Reduce* runs in $O(m + n)$.

In the *Interpolate* algorithm (see Algorithm 2) each odd indexed row i with $1 \leq i \leq m$ searches within an interval $[\ell_i, r_i]$, where $\ell_1 = 1$, $r_m = n$, and $r_i = \ell_{i+1}$ for $i = 1, \ldots, m - 1$. The total number of comparisons across all rows is given by

$$\sum_{i=1}^{m}(r_i - \ell_i) = \sum_{i=1}^{m-1}(\ell_{i+1} - \ell_i) + (n - \ell_m) = (\ell_m - \ell_1) + (n - \ell_m) = n - \ell_1 = n - 1.$$

Thus, finding the minima for odd-indexed rows requires $O(n)$. It recursively calls SMAWK on every second row, thus $m/2$ rows, and due to the next reduction step, the number of columns is considered to be at most $m/2$.

The complexity of *Interpolate* across all recursion levels leads to the recurrence:

$$T(m, n) \leq O(n) + T(m/2, m/2).$$

This holds even if multiple *Interpolate* steps occur before a reduction. We conclude that *Interpolate* runs in $O(m + n)$.

The *Reduce* step needs $O(m + n)$ to reach the base case; the row minima are determined in an additional $O(m + n)$ time through the *Interpolate* step, working backwards. Thus, the overall time complexity remains $O(m + n)$.

4 The Web Proxy Placement Problem

In this chapter, we introduce the Web Proxy Placement Problem as a concrete example of how SMAWK can be applied to optimize a dynamic programming (DP) solution. We first present the problem and its naive dynamic programming approach. We then demonstrate how the Monge property arises in this context and how SMAWK can therefore be applied to reduce the computational complexity.

4.1 Problem Definition

In modern networks, long response times may occur. One approach to dealing with these issues is through the use of web proxies, which store cached copies of server content and are located closer to clients, thereby reducing latency.

We model our problem as a weighted graph $G = (V, E)$, where:

- V is the set of nodes $v_0, v_1, ..., v_n$ representing clients and web proxies.
- E is the set of edges. G follows a *directed line topology* meaning that edges are of the form (v_i, v_{i-1}) for $1 \leq i \leq n$, forming a linear path from v_n to v_0.
- One designated node $v_0 \in V$ represents the source of all information (i.e., the main server).
- Each node $v \in V$ has a weight w_v, which represents the frequency of requests generated by v.
- For any two nodes $v_i, v_j \in V$, let $d(v_i, v_j)$ denote the distance from v_i to v_j, defined as the sum of the edge distances between v_i and v_j.

In a network without proxies, the total traffic associated with requests is determined by the direct connection between clients and the server. The latency ℓ for a request from node v is given by:

$$\ell(v) = d(v, v_0) \cdot w_v.$$

Thus, the total system latency L is:

$$L = \sum_{v \in V} d(v, v_0) \cdot w_v = \sum_{v \in V} \ell(v).$$

To reduce latency, server data may be replicated and stored on *web proxies* distributed across the network. Let $S \subseteq V$ denote the set of web proxies, including the original server v_0 (i.e., $v_0 \in S$). The distance from any node v to its nearest proxy is defined as:

$$d(v, S) = \min_{s \in S} d(v, s).$$

The new total system latency with web proxies is then:

$$L_S = \sum_{v \in V} d(v, S) \cdot w_v.$$

Optimization Problem. Given a graph $G = (V, E)$ and a constraint on the number of proxies, $|S| = m + 1$ (where m is the number of additional proxies to be placed), the goal is to minimize the total latency of the system. While this implicitly determines an optimal set of proxy positions, our initial focus is on computing the minimal latency itself.

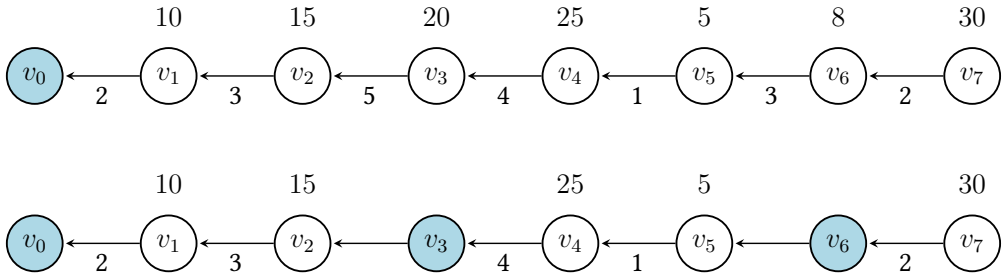

Figure 4.1: Example of the problem: The top configuration shows an unoptimized setup, resulting in a total system latency of 1464. The bottom configuration displays the optimal solution for $m = 2$, which reduces the total system latency to 280. Blue nodes represent the elements of the set S. Distances are labeled along the edges; weights are displayed above the nodes.

This problem is a variation of the *m-median problem*, which is NP-hard for general graphs. In the case of a directed line topology, it can be solved efficiently using dynamic programming. The traditional approach has a time complexity of $O(mn^2)$. However, as shown in [9] and [10], this can be further improved to $O(mn)$ by using the SMAWK algorithm.

4.2 Dynamic Programming Solution

Based on the problem formulation in the previous section, we now describe a dynamic programming approach to determine the minimal total latency.

Computing $a(i, j)$. Let the function $a(i, j)$ represent the total latency when all requests from v_{i+1} to v_{j-1} are served by a single web proxy placed at v_i. This function is defined for $i < j \leq n+1$ by:

$$a(i, j) = \sum_{\ell=i+1}^{j-1} w_\ell d(v_\ell, v_i) = \sum_{\ell=i+1}^{j-1} \sum_{q=i+1}^{\ell} w_\ell d_q,$$

where $d_q = d(v_q, v_{q-1})$.

To efficiently compute $a(i, j)$ in constant time, we use auxiliary arrays, which can be precomputed in $O(n)$ time.

We define:

$$D[j] = \sum_{q=0}^{j} d_q, \quad W[j] = \sum_{\ell=j}^{n+1} w_\ell, \quad X[j] = \sum_{q=0}^{j} \sum_{\ell=q}^{n+1} w_\ell d_q, \quad Y[j] = \sum_{\ell=j}^{n+1} \sum_{q=0}^{\ell} w_\ell d_q.$$

We compute D iteratively from left to right, as well as W from right to left:

$$D[0] = 0, \quad D[j] = D[j-1] + d_j \quad \text{for } j > 0.$$

$$W[n+1] = 0, \quad W[j] = W[j+1] + w_j \quad \text{for } j \leq n.$$

Once D and W are determined, we compute X and Y efficiently using:

$$X[0] = 0, \quad X[j] = X[j-1] + d_j W[j] \quad \text{for } j > 0,$$

$$Y[n+1] = 0, \quad Y[j] = Y[j+1] + w_j D[j] \quad \text{for } j \leq n.$$

Since each array is computed in a single pass over j, the total preprocessing time remains $O(n)$.

Lemma 4.1. *For all i, j with $0 \leq i < j \leq n+1$ the entry $a(i,j)$ satisfies:*

$$a(i,j) = Y[0] - Y[j] - X[i] + D[i] \cdot W[j],$$

where we assume $d_{n+1} = w_{n+1} = 0$ and $d_0 = w_0 = 0$.

Proof.

$$\begin{aligned}
Y[0] - Y[j] - X[i] + D[i] \cdot W[j] &= \sum_{\ell=0}^{n+1} \sum_{q=0}^{\ell} w_\ell d_q - \sum_{\ell=j}^{n+1} \sum_{q=0}^{\ell} w_\ell d_q - \sum_{q=0}^{i} \sum_{\ell=q}^{n+1} w_\ell d_q + \sum_{q=0}^{i} \sum_{\ell=j}^{n+1} w_\ell d_q \\
&= \sum_{\ell=0}^{j-1} \sum_{q=0}^{\ell} w_\ell d_q - \sum_{q=0}^{i} \sum_{\ell=q}^{j-1} w_\ell d_q \\
&= \sum_{q=0}^{j-1} \sum_{\ell=q}^{j-1} w_\ell d_q - \sum_{q=0}^{i} \sum_{\ell=q}^{j-1} w_\ell d_q \\
&= \sum_{q=i+1}^{j-1} \sum_{\ell=q}^{j-1} w_\ell d_q \\
&= \sum_{\ell=i+1}^{j-1} \sum_{q=i+1}^{\ell} w_\ell d_q \\
&= a(i,j)
\end{aligned}$$

□

Dynamic Programming Recurrence. We define a matrix F. Each entry $F[j,k]$ presents the minimum latency across the nodes v_0 through v_j when using k proxies (including v_0) where the k-th proxy is placed at v_j. We have the following base cases:

$$F[j,1] = a(0,j) \quad \text{for } j = 0,\ldots,n+1,$$

$$F[1,k] = 0 \quad \text{for } k > 0.$$

F is computed iteratively, filling in values column by column as k increases. To compute $F[j,k]$, we rely on the previously computed values for $k-1$ proxies.

Therefore, for $k \geq 2$, we compute $F[j,k]$ using:

$$F[j,k] = \min_{1 \leq i < j} \{F[i,k-1] + a(i,j)\}.$$

Optimal Solution. The minimal total latency when placing m proxies optimally is given by

$$\min_{1 \leq i < n+1} \{F[i,m] + a(i,n+1)\} = F[n+1,m+1].$$

The following pseudocode follows the description above:

Algorithm 3 Compute Minimal Latency

Require: $n, m, weights, distances$
1: Compute D, W, X, Y
2: **function** $a(i,j)$
3: **return** $Y_0 - Y_j - X_i + D_i \cdot W_j$
4: **end function**
5: Initialize F with ∞
6: **for** $j = 1$ to $n+1$ **do**
7: $F[j,1] \leftarrow a(0,j)$
8: **end for**
9: **for** $k = 2$ to $m+1$ **do**
10: **for** $j = 2$ to $n+1$ **do**
11: $F[j,k] \leftarrow \min_{1 \leq i < j}(F[i,k-1] + a(i,j))$
12: **end for**
13: **end for**
14: **return** $F[n+1,m+1]$

4.3 Improving Efficiency with SMAWK

In the previous section, we introduced a dynamic programming approach for solving the web proxy placement problem (see Algorithm 3). In particular, line 11 of the algorithm iterates over all possible positions i to find the value for $F[j,k]$.

In this section, we show how this step can be improved using the SMAWK algorithm.

Lemma 4.2. *Let A be an $n \times (n+1)$ matrix defined by*

$$a_{i,j} = \begin{cases} a(i,j) & \text{if } i < j \\ \infty & \text{otherwise} \end{cases}$$

Then, A is a Monge matrix.

Proof. We show that the function $a(i,j)$ satisfies the Monge property (Definition 2.3). That is, for all indices $i < i'$ and $j < j'$, the following inequality holds:

$$a_{i,j} + a_{i',j'} \leq a_{i,j'} + a_{i',j}$$

Case 1: $i < j$

$$\begin{aligned} a(i,j) + a(i',j') - a(i,j') - a(i',j) &= (Y[0] - Y[j] - X[i] + D[i] \cdot W[j]) \\ &\quad + (Y[0] - Y[j'] - X[i'] + D[i'] \cdot W[j']) \\ &\quad - (Y[0] - Y[j'] - X[i] + D[i] \cdot W[j']) \\ &\quad - (Y[0] - Y[j] - X[i'] + D[i'] \cdot W[j]) \\ \\ &= D[i] \cdot W[j] + D[i'] \cdot W[j'] - D[i] \cdot W[j'] - D[i'] \cdot W[j] \\ &= (D[i] - D[i']) \cdot (W[j] - W[j']) \\ &\leq 0 \qquad (D[i] \leq D[i'] \text{ and } W[j] \geq W[j']) \end{aligned}$$

Case 2: $i \geq j$
Since $a_{i,j} = \infty$ and $a_{i',j} = \infty$ (as $i' > i \geq j$), we have:

$$\begin{aligned} a_{i,j} + a_{i',j'} &\leq a_{i,j'} + a_{i',j} \\ \infty + a_{i',j'} &\leq a_{i,j'} + \infty \\ \infty &\leq \infty \end{aligned}$$

□

Lemma 4.3. *Let M^k be a matrix initialized as: $M^k[i,j] = F[i,k-1] + a_{i,j}$ for any fixed k. Then M^k is Monge.*

Proof. We already established that $a_{i,j}$ satisfies the Monge property in Lemma 4.2, meaning that for all $i < i'$ and $j < j'$, we have:

$$a_{i,j} + a_{i',j'} \leq a_{i,j'} + a_{i',j}.$$

Now, if we add the values $F[i,k-1]$ and $F[i',k-1]$ to both sides of the inequality, we obtain:

$$(F[i,k-1] + a_{i,j}) + (F[i',k-1] + a_{i',j'}) \leq (F[i,k-1] + a_{i,j'}) + (F[i',k-1] + a_{i',j}).$$

Therefore, this implies:

$$M^k[i,j] + M^k[i',j'] \leq M^k[i,j'] + M^k[i',j]$$

□

Now that we have established that M^k is a Monge matrix, we can conclude that $(M^k)^T$ is also Monge (see Lemma 2.2), which implies total monotonicity (see Lemma 2.1).

Since $F[j,k]$ corresponds to the column minima of each column j in M^k, we can instead compute the row minima of $(M^k)^T$ using the SMAWK algorithm.

To compute all values of $F[j,k]$, we need to process $m-1$ matrices: $M^2, M^3, \ldots, M^{m+1}$. The optimal solution corresponds to the $(n+1)$th row minimum of $(M^{m+1})^T$.

A straightforward approach to optimizing this problem would be to first explicitly construct the matrix $(M^k)^T$ and then apply the SMAWK algorithm to compute its row minima. However, this method has a major drawback, as constructing the matrix requires $O(n^2)$ time (and space), which offsets the efficiency gained by SMAWK. To overcome this issue, we can use a lookup-based implementation of SMAWK that computes only the necessary entries of $(M^k)^T$ in $O(1)$ time as needed. We will discuss the implementation details later.

Using the SMAWK algorithm to compute the row minima of each $(M^k)^T$ takes $O(n)$ time per matrix. Since we have $m-1$ such matrices, the total runtime for this step is $O(mn)$.

Without using SMAWK, computing each value $F[j,k]$ would require a brute-force search over all possible previous proxy placements, leading to an $O(n^2)$ time complexity per k. Since this must be done for all $m-1$ proxies, the total runtime without SMAWK would be $O(mn^2)$.

Thus, applying SMAWK significantly improves the efficiency, reducing the total complexity from $O(mn^2)$ to $O(nm)$.

5 The Line Breaking Problem

In this chapter, we present a second application of the SMAWK algorithm, where the algorithm is used in a different way. Rather than providing a full solution or implementation details, the goal here is to give an intuitive understanding of how SMAWK can also be applied incrementally in dynamic programming. The naive dynamic programming solution has a runtime of $O(n^2)$, but by exploiting the Monge property, this can be reduced to $O(n)$ using SMAWK.

5.1 Problem Definition

Given a paragraph of text, the goal is to optimally break it into lines of a fixed width (*parwidth*). To model this problem, the text is represented as a sequence of syllables $s_1, s_2, ..., s_n$, where spaces and punctuation marks are considered part of the previous syllable.

A weight function $w(i, j)$ is defined for every possible line from s_{i+1} to s_j (where $i < j$). The function assigns a penalty based on how well the text fits within the paragraph width:

$$w(i, j) = (\text{len}(i, j) - \text{parwidth})^2,$$

where $\text{len}(i, j)$ represents the length of the line from the first character of s_{i+1} to the last character of s_j.

To reduce the total penalty, we define a sequence of line break positions:

$$l_0, l_1, ..., l_k,$$

where $l_0 = 0$, $l_k = n$ and each l_i represents the starting position of a new line at syllable s_{l_i+1}.

The objective is to minimize the total penalty:

$$\sum_{i=0}^{k-1} w(l_i, l_{i+1}).$$

While our solution implicitly determines an optimal set of line break positions, our primary focus is on computing the minimal total penalty itself.

This problem is an instance of the *Concave Least-Weight subsequence Problem*, which has been studied in different contexts [11].

5.2 Dynamic Programming Setup

To compute the minimum total penalty for breaking the text into lines, we define a function f, which represents the minimum penalty for formatting the text from syllable s_1 to s_j. We formulate the recurrence as:

$$f(j) = \min_{0 \le i < j} \left(f(i) + w(i, j) \right).$$

The idea is to iteratively compute the minimum latency for any line ending at s_j, whose line must start at some earlier position s_{i+1} assuming that the layout up to s_i was already optimal. Our base case is $f(0) = 0$ and $f(n)$ is the final result, the minimum total penalty for the entire text.

Lemma 5.1. *The weight function $w(i, j)$ can be computed in $O(1)$ by using an auxiliary array.*

Proof. We define $L[i]$ as the length starting from the first character of s_1 to the last character of s_i. The array L can be computed in $O(n)$ preprocessing time. Then, $\text{len}(i, j) = L[j] - L[i]$ and therefore $w(i, j) = (\text{len}(i, j) - \text{parwidth})^2$ can be computed in $O(1)$. □

The following pseudocode describes the dynamic programming approach to computing $f(n)$:

Algorithm 4 Compute Minimal Penalty

Require: Syllables $s_1, \ldots, s_n$, *parwidth*
1: Compute cumulative lengths $L[0], \ldots, L[n]$
2: **function** w(i, j)
3: **return** $(L[j] - L[i] - parwidth)^2$
4: **end function**
5: Initialize array f with $f[0] \leftarrow 0$
6: **for** $j = 1$ to n **do**
7: $f[j] \leftarrow \min_{0 \le i < j}(f[i] + \text{w}(i, j))$
8: **end for**
9: **return** $f[n]$

5.3 Improving Efficiency with SMAWK

This section shows how the dynamic programming method from the previous section can be further optimized by applying the SMAWK algorithm.

The weight function $w(i, j) = (\text{len}(i, j) - \text{parwidth})^2$ satisfies the Monge property. That is, for all $i < i'$ and $j < j'$, the following inequality holds:

$$w(i, j) + w(i', j') \le w(i, j') + w(i', j)$$

We define a cost matrix G as follows:

$$g_{i,j} = \begin{cases} f(i) + w(i, j) & \text{if } i < j \\ \infty & \text{otherwise} \end{cases}$$

Since adding a constant does not affect the Monge property (as shown in Lemma 4.3), the matrix G is also Monge. G is an $n \times n$ matrix, with row indices from 0 to $n - 1$ and column indices from 1 to n.

The column minimum of column j corresponds to $f(j)$. Thus, similar to the Web Proxy Placement Problem, this can be formulated as a column minima problem on a Monge matrix. Likewise, the matrix does not need to be explicitly constructed. Entries should instead be computed on demand in $O(1)$ time. However, unlike before, we cannot compute the row minima of G^T in a single pass. The challenge here is that evaluating any entry $g_{i,j} = f(i) + w(i, j)$ requires the value $f(i)$, which itself depends on earlier column minima. Arbitrary entries of G are not accessible in constant time. Instead, we adopt an incremental approach, where SMAWK is applied at most two times per iteration. The algorithm begins in the upper-left corner of the Monge matrix G and progresses rightward and downward with each iteration, gradually expanding the known region.

The following figure shows the matrix G during a typical iteration.

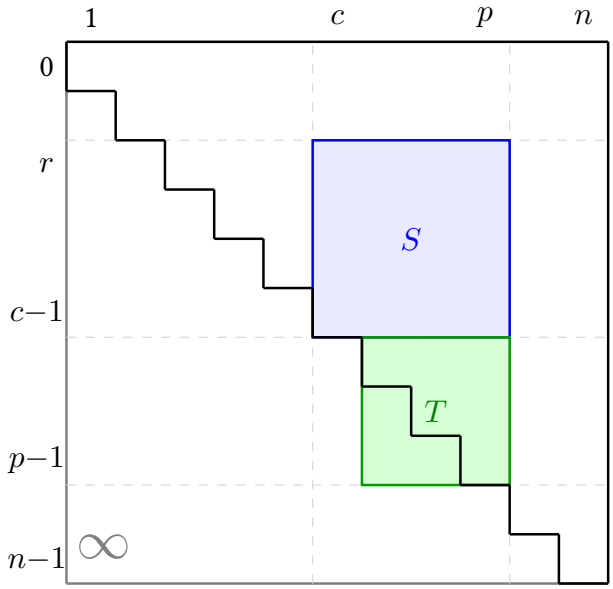

Figure 5.1: Submatrices S and T of the Monge matrix G, on which SMAWK is applied during a typical iteration.

Each iteration of the algorithm maintains the following invariants:

1. $r \geq 0$ and $c > r$,
2. all values $f(j)$ for $0 \leq j < c$ have already been computed,
3. for all columns $j \geq c$, the current column minima lie in rows $\geq r$.

A typical iteration consists of two SMAWK applications.

1. SMAWK on submatrix S^T We determine the column minima of S, consisting of rows $r, \ldots, c-1$ and columns $c, \ldots, p$. Since all values $f(i)$ for $i < c$ are known (by Invariant 2), all entries of S can be computed in constant time. Furthermore, due to Invariant 3, any entry in a row $i < r$ cannot be the column minimum and can be ignored.

The result of this step is a first approximation of the values $f(j)$ for $j = c+1, \ldots, p$. We can only guarantee that $f(c)$ is computed correctly here, since all relevant entries within its column are fully known within this submatrix.

2. SMAWK on submatrix T^T To verify the values for $f(j)$ with $j = c+1, \ldots, p$, we determine the column minima of submatrix T (only if $c < p$, otherwise this step is skipped). This matrix uses the newly available rows $c, \ldots, p-1$, its entries are computed based on the results of the first SMAWK pass.

Let $h(j)$ be the column minimum in column j of T. Not all values $h(j)$ are guaranteed to be correct, since the corresponding $f(j)$ values may not have been computed correctly in the first pass.

However, since $f(c)$ is known to be correct, the entire row c must also be correct. The first column of T only consists of the cell $g_{c,c+1}$, which implies that $h(c+1)$ is also correct.

If $h(c+1) > f(c+1)$, then $f(c+1)$ was computed correctly. In that case, the entire row $c+1$ is also correct, and we can proceed to verify $f(c+2)$ in the same way – and so on:

- If $h(j) \geq f(j)$ for all j, then all intermediate estimates for $f(i)$ were correct. The algorithm advances and continues with the next block (c is set to $p+1$).
- If there exists any j such that $h(j) < f(j)$, then $f(j)$ was computed incorrectly. In that case, the earliest such index j_0 is identified. The algorithm updates the current boundary $c := j_0$. Because of the total monotonicity, we know that all rows above c cannot contain the minimum, therefore it restricts future consideration to rows $\geq c+1$.

Repeating this process incrementally, the algorithm traverses the matrix diagonally — from top-left to bottom-right until $f(n)$, the final result has been computed correctly.

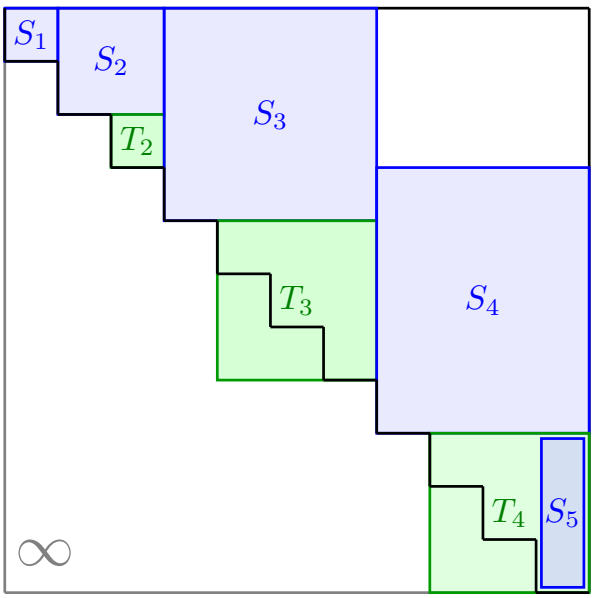

Figure 5.2: Example of matrix G during all iterations.

6 Performance Analysis

In this chapter, we evaluate the practical efficiency of SMAWK by implementing and analyzing its performance in the *Web Proxy Placement Problem*. We first present the implementations of our approach to solving the problem using three different methods. We also implemented two versions of the SMAWK algorithm to handle different scenarios. This is followed by the experimental setup with different parameters. Finally, we compare the results and analyze under which conditions SMAWK provides the greatest efficiency gains.

All algorithms were implemented in Python. Only a high-level description is provided here. Detailed code implementations are available at [12].

6.1 Implementation

SMAWK Implementation

- **Explicit-Matrix SMAWK:** This version requires a precomputed *totally monotone matrix* as input. The output is a list of column indices, where each entry corresponds to the column index of the minimum value in that row.

```
def smawk(matrix: list[list[float]]) -> list[int]:
```

Code Snippet 6.1: Function signature of the SMAWK implementation with explicit matrix

- **SMAWK with Lookup:** Instead of receiving a matrix, this version operates on a *virtual matrix* by using only the matrix dimensions $m \times n$ and a lookup function that dynamically computes individual matrix entries on demand.

```
def smawk_with_lookup(num_rows: int, num_columns: int,
lookup_function: Callable[[int, int], float]) -> list[int]:
```

Code Snippet 6.2: Function signature of the lookup-based SMAWK implementation

Both implementations follow the same fundamental SMAWK logic for computing row minima.

Web Proxy Implementation

- **Unoptimized Dynamic Programming:** The implementation follows Algorithm 3. The computation of an entry $F[j,k]$ requires iterating over all possible placements i with $1 \leq i < j$.

```
F[j][k] = min(F[i][k-1] + a_tilde(i, j) for i in range(1, j))
```

Code Snippet 6.3: Computing a single DP entry in the unoptimized dynamic program

- **Dynamic Programming with Explicit-Matrix SMAWK:** This version first constructs the matrix $(M^k)^T$ and then applies *Explicit-matrix SMAWK* to determine the required row minima.

```
# Construct (M^k)^T
M = [[float('inf') for _ in range(n+1)] for _ in range(n+2)]
for j in range(2, n+2):
    for i in range(1, j):
        M[j][i] = F[i][k-1] + a_tilde(i, j)

# Apply SMAWK to find the row minima
minima = smawk(M)
```

Code Snippet 6.4: Explicitly constructing and applying SMAWK on the matrix $(M^k)^T$

- **Dynamic Programming with SMAWK with Lookup:** This version follows the same logic as the previous approach but eliminates the need to explicitly construct the matrix $(M^k)^T$ by using the *SMAWK with Lookup*.

```
# Define lookup function for computing entries of (M^k)^T on demand
def lookup(j, i):
    if i >= j or j < 2:
        return float('inf')
    return F[i][k-1] + a_tilde(i, j)

# Apply SMAWK using the lookup function as matrix access
minima = smawk_with_lookup(n+2, n+1, lookup)
```

Code Snippet 6.5: SMAWK applied with on-demand lookups instead of full matrix construction

Note on matrix dimensions: In theory, $(M^k)^T$ is an $n \times n$ matrix indexed from $i = 2$ and $j = 1$, where i and j denote the row and column indices, respectively. In practice, we use a $(n + 2) \times (n + 1)$ matrix in Python, since here indices start at 0. This avoids offset adjustments and keeps the implementation consistent with the theoretical formulation.

6.2 Experimental Set Up

To evaluate the performance of different approaches to solving the Web Proxy Placement Problem, we conduct a series of experiments under varying input sizes based on different evaluation metrics.

Across all experiments, we evaluate the algorithms for n nodes ranging from 10 to 150, increasing in steps of 10. The number of proxies m is fixed at 10 for all n.

The experiment is performed using randomly generated weights w and distances d, their values are integers in the range $[1, 100]$. For the same experiment, all algorithms process the same input, ensuring fair comparability of the results. An assertion check is performed to verify that all methods return the same result, ensuring correctness.

The experiments were conducted in Python, and the results were visualized with *matplotlib*. The code was executed on a Lenovo ThinkPad T14s Gen2 laptop with 8 GB RAM and an 11th Gen Intel Core i5-1135G7 processor, running Fedora Linux via the command line.

We assess the algorithms based on the following key parameters:

- **Runtime:** The runtime measurements were performed using Python's built-in time module. The focus is on comparing the *Unoptimized Dynamic Programming* approach with the *SMAWK with Lookup* approach. Additionally, we include the *Explicit-Matrix SMAWK* variant to observe whether explicitly constructing the full cost matrix negates the theoretical benefits of SMAWK due to its preprocessing overhead.

  Furthermore, we include theoretical runtime plots $O(mn^2)$ and $O(mn)$. These serve as a reference to verify whether the empirical runtime measurements follow the expected growth. Details on their scaling are discussed in the results section.

- **Comparison Count:** A counter variable is initialized to track how often two matrix entries are compared in the process of determining row minima. This comparison count serves as a hardware-independent measure of algorithmic effort and reflects the theoretical complexity of the algorithm.

  This counting mechanism has been implemented in two approaches: *Unoptimized Dynamic Programming* and *Dynamic Programing with SMAWK with Lookup*, where the counter tracks comparisons made within the SMAWK algorithm.

  Since both *Explicit Matrix SMAWK* and *SMAWK with Lookup* perform comparisons in the same way, their comparison count remains identical. Therefore, our comparison focuses only on Dynamic Programming with and without SMAWK, rather than between different SMAWK variants.

  As in the runtime analysis, theoretical plots are included.

- **Number of accessed entries:** To visualize how many matrix entries are actually accessed by the SMAWK algorithm, we perform an additional experiment. Since SMAWK doesn't need to access all matrix entries to find row minima, we modified the lookup-based implementation to count the total number of lookups, including repeated accesses to the same entries, the number of unique matrix entries that were accessed at least once, and the total number of theoretically accessible entries across all M^k. These measurements allow us to compare the actual access behavior of the algorithm with the full size of the matrix space.

6.3 Results

In this section, we show and analyze the experimental results of the setup from our previous section.

The following figure shows the measured runtimes of the three implementation variants: *Unoptimized Dynamic Programming*, *Dynamic Programming with Explicit-Matrix SMAWK*, and *Dynamic Programming with SMAWK with Lookup*, as a function of the number of nodes n.

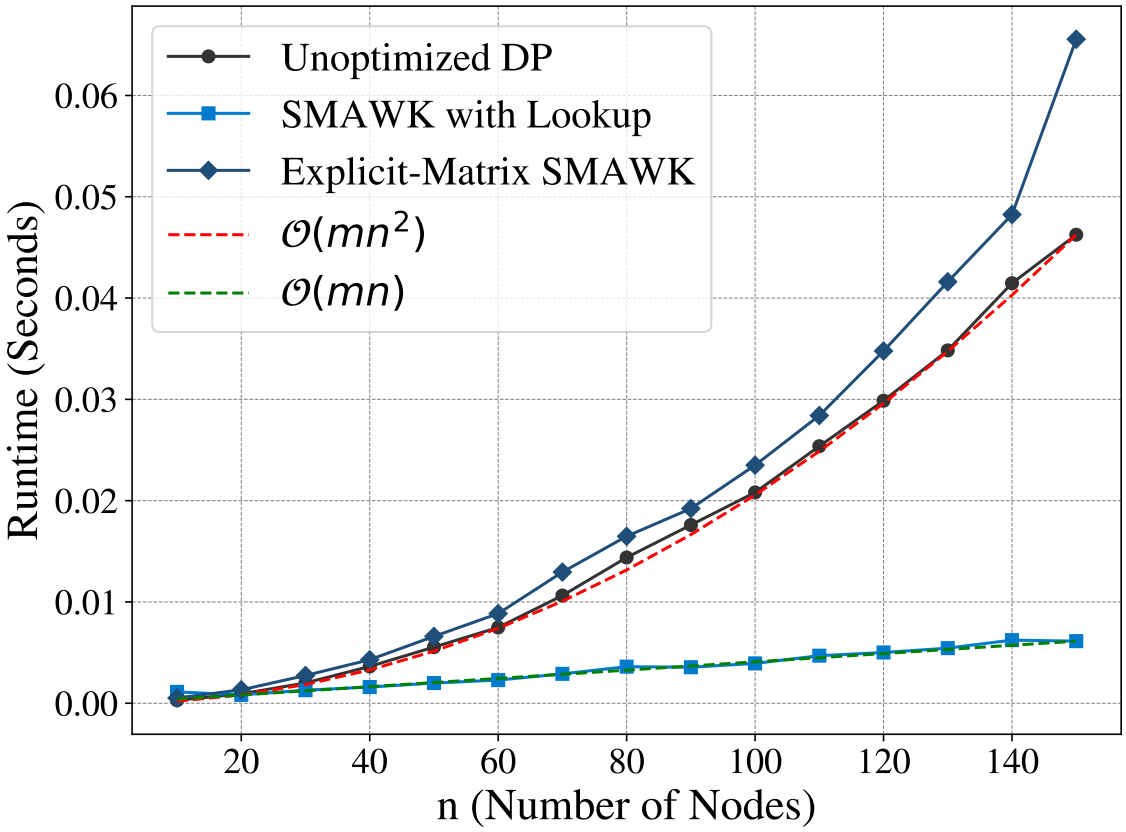

Figure 6.1: Runtime of three dynamic programming variants solving the Web Proxy Placement Problem.

The runtime of the *Unoptimized Dynamic Programming* approach is quadratic. In contrast, the optimized version using *SMAWK with Lookup* exhibits linear growth. Both variants align well with their theoretical time complexities of $O(mn^2)$ and $O(mn)$, respectively.

However, the *Explicit-Matrix SMAWK* implementation performs even worse than the unoptimized version. Since generating the matrix itself requires $O(n^2)$ time, it negates the benefits gained from SMAWK and makes it an inefficient alternative. Therefore, in cases where the matrix is not precomputed, the lookup-based SMAWK variant should be used, with efficiency gains growing with larger n.

Theoretical curves $O(nm)$ and $O(mn^2)$ were scaled by constants to align with the measured runtimes at $n = 150$ for the curves: *Unoptimized Dynamic Programming* and *SMAWK with Lookup*. The constants were computed independently for each variant by dividing the measured runtime at $n = 150$ by the corresponding theoretical expression evaluated at that point.

While the previous runtime analysis demonstrated the practical efficiency of the SMAWK variant, the following analysis focuses on a hardware-independent metric: the number of elementary comparison operations required to solve the Web Proxy Placement Problem. The following figure shows the number of comparisons performed by the *Unoptimized Dynamic Programming* and *Dynamic Programming with SMAWK* approaches as a function of the number of nodes n.

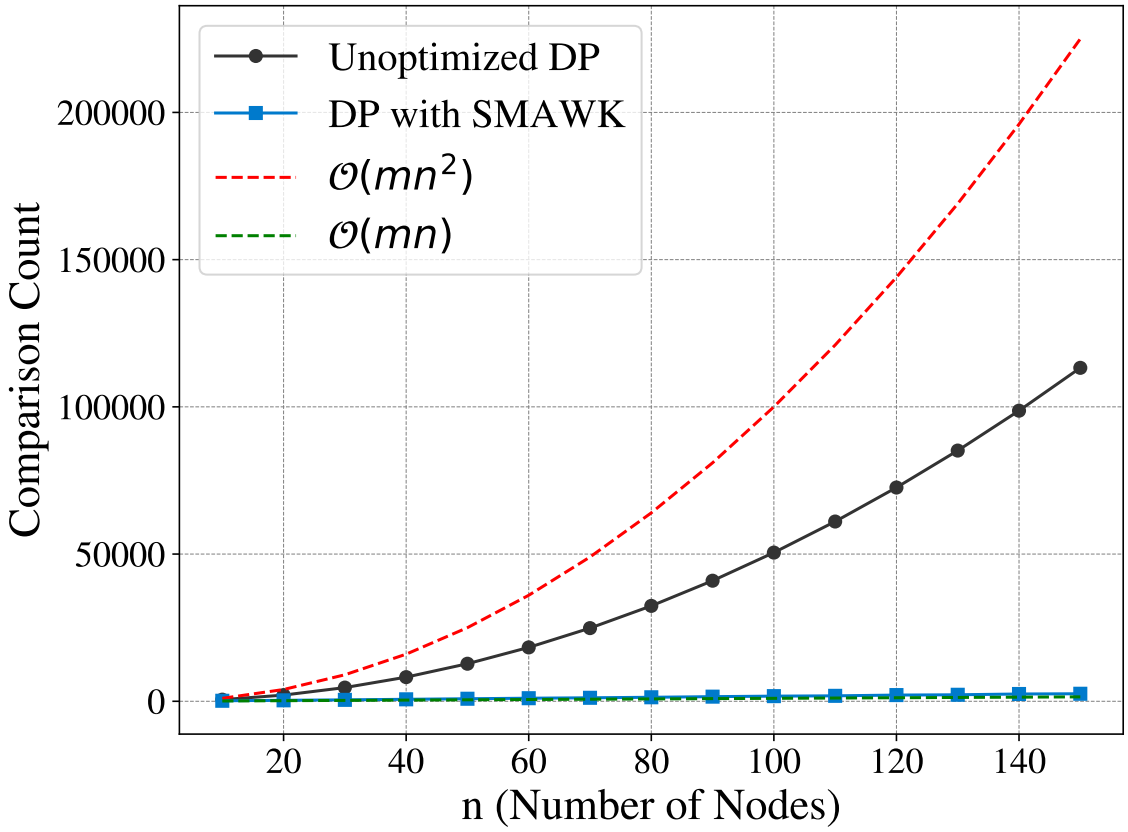

Figure 6.2: Comparison counts of DP approaches with and without SMAWK optimization.

The *Unoptimized Dynamic Programming* approach is quadratic, as expected from its theoretical complexity of $O(mn^2)$. However, the measured comparison count is slightly lower than the direct estimate. This is due to the fact that the minimum is not searched over all indices from 1 to n when computing $F[j, k]$, but rather from 1 to j.

In contrast, the SMAWK-optimized variant requires significantly fewer comparisons, also aligning with its theoretical $O(nm)$ complexity. This supports the runtime analysis by showing that SMAWK leverages total monotonicity to avoid unnecessary comparisons.

The theoretical bounds $O(mn^2)$ and $O(mn)$ are represented by the unscaled functions mn^2 and mn (with constant $m = 10$).

To provide a clearer picture of how efficiently SMAWK accesses the input matrix. The following figure compares, as a function of the number of nodes n, the total number of lookups and the number of unique entries accessed by the *Dynamic Programming with SMAWK with Lookup* implementation, as well as the total number of theoretically accessible entries across all matrices $(M^k)^T$.

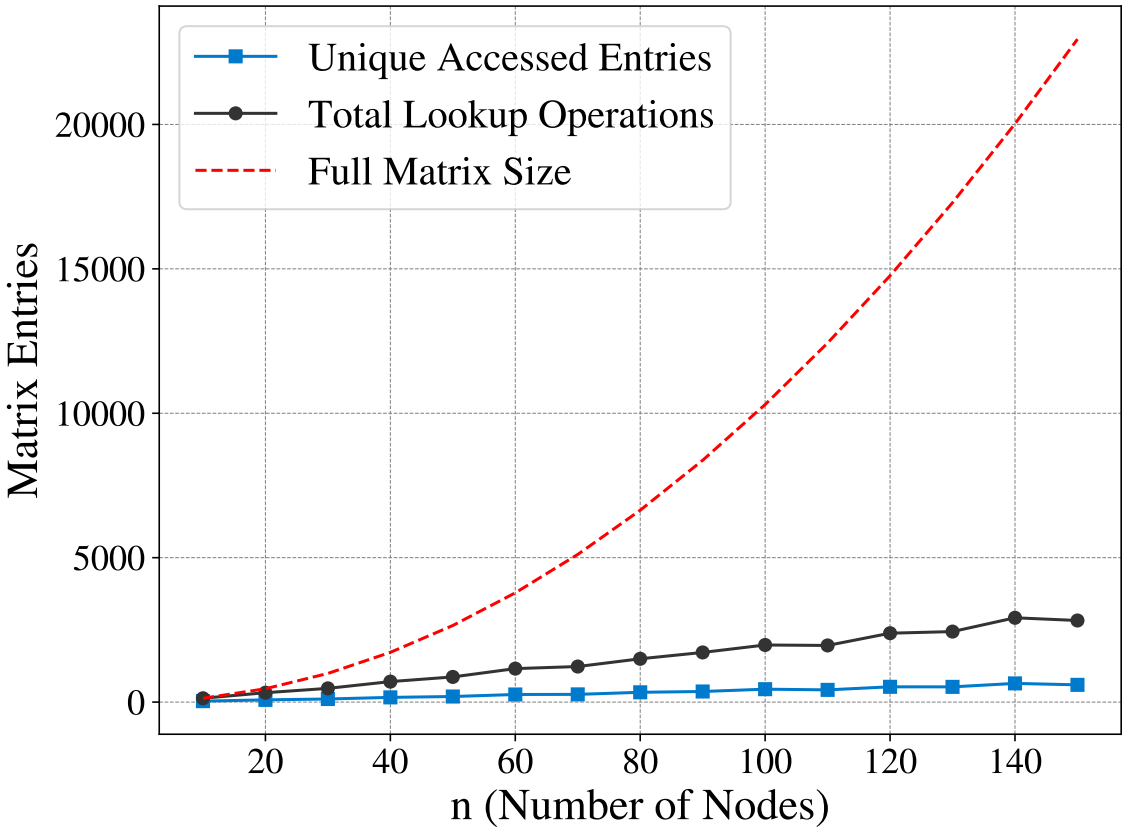

Figure 6.3: Matrix access statistics of the lookup-based SMAWK algorithm versus full matrix size.

The theoretical matrix size grows quadratically with n, since each of the m matrices M^k contains $O(n^2)$ entries. In contrast, both the total and unique lookups grow significantly more slowly and linearly. This demonstrates that SMAWK avoids the majority of possible evaluations, even though it still computes the exact minima in each row compared to the *Unoptimized Dynamic Programming* approach, which linearly scans each row.

The results highlight the strength of the lookup-based approach and further reinforce why the explicit-matrix variant is inefficient, as most of the computed entries are never accessed.

7 Conclusion

In this thesis, we explored how the SMAWK algorithm can be used to compute row minima in totally monotone matrices, and how this property can be exploited in dynamic programming to achieve significant performance improvements, by reducing this step from $O(mn)$ to $O(m + n)$.

We presented two representative applications: the Web Proxy Placement Problem and the Line Breaking Problem, which differ in the way SMAWK is integrated into the solution. We learned how problems involving column minima in Monge matrices can be transformed into row minima problems in totally monotone matrices, allowing us to use SMAWK in a wider range of scenarios. However, SMAWK only works under specific conditions. It requires the input to be totally monotone, and either known in advance or evaluable in constant time via a lookup function. In practice, we addressed this limitation in the Line Breaking Problem by applying SMAWK incrementally.

Totally monotone matrices often do not appear explicitly, but instead arise implicitly from the structure of the cost or weight functions used in dynamic programming. From a practical perspective, a central takeaway is the importance of using a lookup-based version of SMAWK. The algorithm should be embedded as a function that takes the matrix dimensions and a lookup function as input. This approach proved to be much more efficient, as confirmed in our experiment.

Bibliography

[1] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, November 1987. ISSN 0178-4617, 1432-0541. doi: 10.1007/BF01840359. URL http://link.springer.com/10.1007/BF01840359.

[2] M. Luessi, M. Eichmann, G. M. Schuster, and Aggelos K. Katsaggelos. New Results on Efficient Optimal Multilevel Image Thresholding. In *2006 International Conference on Image Processing*, pages 773–776. IEEE, October 2006. ISBN 978-1-4244-0480-3. doi: 10.1109/ICIP.2006.312426. URL https://ieeexplore.ieee.org/document/4106644/.

[3] Lawrence L Larmore and Baruch Schieber. On-line dynamic programming with applications to the prediction of RNA secondary structure. *Journal of Algorithms*, 12(3):490–515, September 1991. ISSN 0196-6774. doi: 10.1016/0196-6774(91)90016-R. URL https://www.sciencedirect.com/science/article/pii/019667749190016R.

[4] Bo Li, Xin Deng, Mordecai Golin, and Kazem Sohraby. *On the Optimal Placement of Web Proxies in the Internet: The Linear Topology*. January 1998. ISBN 978-1-4757-5397-4. doi: 10.1007/978-0-387-35388-3_28. Pages: 495.

[5] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, November 1981. ISSN 0038-0644, 1097-024X. doi: 10.1002/spe.4380111102. URL https://onlinelibrary.wiley.com/doi/10.1002/spe.4380111102.

[6] Jeff Erickson. Advanced Dynamic Programming. Technical report, University of Illinois at Urbana-Champaign, 2018. URL https://jeffe.cs.illinois.edu/teaching/algorithms/notes/D-faster-dynprog.pdf.

[7] Wolfgang Bein. Advanced Techniques for Dynamic Programming. In Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham, editors, *Handbook of Combinatorial Optimization*, pages 41–92. Springer, New York, NY, 2013. ISBN 978-1-4419-7997-1. doi: 10.1007/978-1-4419-7997-1_28. URL https://doi.org/10.1007/978-1-4419-7997-1_28.

[8] Lawrence L. Larmore. Monge Properties and Dynamic Programming. Technical report, University of Nevada, Las Vegas. URL https://web.cs.unlv.edu/larmore/Courses/CSC477/monge.pdf.

[9] Gerhard J. Woeginger. Monge strikes again: optimal placement of web proxies in the internet. *Operations Research Letters*, 27(3):93–96, October 2000. ISSN 0167-6377. doi: 10.1016/S0167-6377(00)00041-9. URL https://www.sciencedirect.com/science/article/pii/S0167637700000419.

[10] Mordecai Golin. DP Speedups II: The Monge Property. Lecture notes, Hong Kong University of Science and Technology, 2004. URL http://home.cse.ust.hk/~golin/COMP572/Notes/SMAWK.pdf.

[11] Robert Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418–425, September 1988. ISSN 0196-6774. doi: 10.1016/0196-6774(88)90032-6. URL https://www.sciencedirect.com/science/article/pii/0196677488900326.

[12] Xiaoxiao Shen. Source Code for SMAWK and its application, 2025. URL https://git.imp.fu-berlin.de/shex04/smawk.