# Fast Algorithms for the Maximum Convolution Problem *

Michael Bussieck †
Hannes Hassler ‡
Gerhard J. Woeginger §
Uwe T. Zimmermann †

**Abstract**

We describe two algorithms for solving the maximum convolution problem, i.e. the calculation of $c_k := \max\{a_{k-i} + b_i \mid 0 \leq i \leq n-1\}$ for all $k$ with respect to given sequences $(a_0, \ldots, a_{n-1})$, $(b_0, \ldots, b_{n-1})$ of real numbers. Our first algorithm with expected running time $O(n \log n)$ is mainly of theoretical interest while our second algorithm allows a simpler, more practicable implementation and showed quite fast performance in numerical experiments.

maximum convolution; probabilistic analysis

## 1 Introduction

For $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$ two sequences of real numbers, we consider the problem of computing the numbers $c_k$, $0 \leq k \leq n-1$ defined by

$$c_k = \max_{0 \leq i \leq n-1} (a_{k-i} + b_i) \tag{1}$$

(all indices in this exposition are taken modulo $n$). We call this problem the *Maximum Convolution* problem, MaxCon for short. A direct calculation of sequence $c$ from equation 1 obviously needs $O(n^2)$ time.

As the naming suggests, this problem is related to the standard *convolution problem* where the values $c_k$ are determined by $c_k = \sum_{i=0}^{n-1} a_{k-i} b_i$, i.e. with operations in the ring of real numbers $(\boldsymbol{R}, +, \cdot)$ instead of the semiring $(\boldsymbol{R}, \max, +)$.

However, the standard convolution problem can be solved in $O(n \log n)$ time by exploiting the resemblance of the problem to multiplication of polynomials and by applying the *Fast Fourier Transform* (see e.g. Aho, Hopcroft, Ullman [1]). Closer inspection reveals that a similar approach seemingly will not work for problem MaxCon. One of the main obstacles is that the operation 'max' allows no inverse elements. Therefore, for real numbers we may easily compute $x$ when $x \cdot y$ and $y$ is known, but we can not compute $x$ from $\max\{x, y\}$ and $y$.

The maximum convolution can be used to solve certain algebraic shortest path problems. Standard methods such as the Floyd-Warshall algorithm [2, 3, 8] assume a well

*This research was partially supported by the Christian Doppler Laboratorium für Diskrete Optimierung.
†TU Braunschweig, Abteilung für Mathematische Optimierung, Pockelsstraße 14, D-38106 Braunschweig, Germany
‡TU Graz, Institut für Angewandte Informatik und Kommunikationstechnologie, Klosterwiesgasse 32/I, A-8010 Graz, Austria
§TU Graz, Institut für Theoretische Informatik, Klosterwiesgasse 32/II, A-8010 Graz, Austria

organized cost structure, namely closed semirings $(R, \odot, \oplus)$. Unfortunately, many real problems, e.g. the shortest path problem with discounting [5, 7], which has applications to annual investment programs, do not satisfy this property. Lengauer and Theune [6] suggest a method for finding closed semirings from unstructured path problems. The use of the Floyd-Warshall algorithm for determining the shortest paths with discounting for costs in the Lengauer-Theune semiring requires a maximum convolution as multiplicative operation. This algorithm calls MAXCON $O(n^3)$ times, thus we need a fast solution for the MAXCON problem.

In order to describe the basic idea of our algorithms let $v_1 \geq v_2 \geq \ldots$ denote the ordered sequence of all values in the family $(a_i + b_j \mid 0 \leq i, j \leq n-1)$. Then, after initializing $c_k := -\infty$ for $k = 0, \ldots, n-1$, the loop

$$\text{For } \ell := 1 \text{ to } n^2 \text{ do if } v_\ell = a_i + b_j > c_{i+j} = -\infty \text{ then } c_{i+j} := v_\ell \tag{2}$$

assigns the correct values to $c$.

Based on this observation, in Section 2 we describe a fast algorithm with expected running time $O(n \log n)$ as proved in Section 3. In Section 4 we describe and analyze another version of this approach which allows a simpler implementation and turned out to be quite fast in numerical experiments. In Section 5, some numerical investigations are presented. Finally we give some concluding remarks in section 6.

# 2 Algorithm TACU

Our first algorithm TACU for the MAXCON problem is depicted in Figure 1.

**Algorithm Try-And-Clean-Up (TACU)**

(T1) Initialize $c_k \equiv -\infty$.

(T2) Compute the $n \ln n$ largest values $w_1, w_2, \ldots, w_{n \ln n}$ in the family $(a_i + b_j \mid 0 \leq i, j \leq n-1)$.

(T3) For $\ell := 1$ to $n \ln n$ do
If $w_\ell = a_i + b_j > c_{i+j}$ then $c_{i+j} := w_\ell$

(T4) For all $c_k = -\infty$ do
$c_k := \max_{0 \leq i \leq n-1}(a_{k-i} + b_i)$.

Figure 1: A fast algorithm for MAXCON

The values of all $c_k$ are determined in Steps (T3) and (T4). We classify the $c_k$ into *good* ones that receive a value in Step (T3) and into *bad* ones that must be handled in the clean-up Step (T4). Intuitively, Step (T3) tests whether the family $(a_{k-i} + b_i)_{i=0}^n$ contains one of the $n \ln n$ largest sums and if so, it assigns to $c_k$ the largest valid sum. Clearly, after Step (T3) all good $c_k$ are at their correct values. For bad $c_k$, Step (T3) fails and Step (T4) must do some extra computations according to the original definition. This proves correctness of the algorithm.

Next, we explain how to exactly perform Step (T2). In $O(n \log n)$ time we sort the numbers $a_i$ into a sequence $x_0 \geq x_1 \geq \ldots \geq x_{n-1}$ and the numbers $b_i$ into a sequence $y_0 \geq y_1 \geq \ldots \geq y_{n-1}$. Then the square matrix $M$ defined by $M_{ij} = x_i + y_j$ is a so-called *sorted* matrix since each row and each column is in nonincreasing order. Frederickson and Johnson [4] show how to determine the $k$-largest element in a sorted $n \times n$-matrix

in $O(n)$ time. We apply this technique to calculate the $(n \ln n)$-largest element $\rho$ of $M$ in linear time. Finally, we determine the border between elements greater or equal than $\rho$ and smaller ones. We start at $\rho$ and move leftward/downward and rightward/upward. Since this border has $2n$ elements, we need only $O(n)$ time.

To analyze the overall time complexity of TACU, we note that the initialization in Step (T1) takes $O(n)$ time. Step (T2) can be implemented to run in $O(n \log n)$ time as stated in the above paragraph and Step (T3) is a loop with $O(n \ln n)$ operations. Finally, the time complexity of Step (T4) depends on the number $B$ of bad $c_k$. For each bad $c_k$, it performs $O(n)$ operations. Summarizing, this gives an overall time complexity $O(n \log n + Bn)$.

In the next section, we will show (cf. Lemma 3) that the expected number $B$ of bad $c_k$ is $O(1)$ under the following mild assumption:

(As) We assume that for all sequences $a$ and $b$, all permutations of their elements occur with equal probability as input.

Together with Lemma 4, this will yield the following theorem.

**Theorem 1** *Under assumption (As), algorithm TACU solves the* Maximum Convolution *problem with expected running time $O(n \log n)$ and with worst case running time $\Theta(n^2)$.*

# 3 Analysis of TACU

We start with analyzing the following related matrix problem. Let $0 \leq t_1, \ldots, t_n \leq n$ denote $n$ integers that sum up to $n \ln n$ and fulfill $t_1 \geq t_2 \geq \ldots \geq t_n$. Let $M$ denote an $n \times n$-matrix with two-colored entries such that in the $i$-th column of $M$, exactly the first $t_i$ entries are colored red and all other entries are colored white. For two $n$-tuples $R = (r_i)$ and $C = (c_i)$ of numbers from 0 to $n-1$, we say that the pair $(R, C)$ *represents* some $k$, $0 \leq k \leq n-1$, if and only if there exist indices $i$ and $j$ such that

- entry $M_{ij}$ is colored red, and
- $r_i + c_j = k$ (this calculation again is done modulo $n$).

We will also say that in this case entry $M_{ij}$ or row $i$ or column $j$ represents $k$ under $(R, C)$. Finally, we define $\text{Val}(R, C)$ to be the number of different values $k$ represented by $(R, C)$.

Our first goal is to estimate the average of $\text{Val}(R_0, C)$ over all $n^n$ distinct $n$-tuples $C$ from below, where $R_0$ is some fixed permutation of the numbers $0 \ldots n-1$. This is done by applying the Inclusion-Exclusion principle to calculate the sum of all $\text{Val}(R_0, C)$ in the following way.

Let $i_1, \ldots, i_m$ denote the indices of an arbitrary set $I$ of columns. Then there are $t_{i_j}$ possibilities for choosing a number $c_{i_j}$ in $C$ such that the $i_j$-th column represents some fixed $k$, $0 \leq k \leq n-1$, and obviously there are $n$ possibilities to choose $k$. Together with the $n^{n-m}$ possibilities for choosing numbers $c_i$ for columns outside of $I$, this gives a total number of $n^{n-m+1} t_{i_1} t_{i_2} \cdots t_{i_m}$ possibilities that the columns in $I$ represent some common number. Now by the Inclusion-Exclusion principle,

$$\sum \text{Val}(R_0, C) = n^n \sum_i t_i - n^{n-1} \sum_{i \neq j} t_i t_j + n^{n-2} \sum_{i \neq j \neq k \neq i} t_i t_j t_k - \ldots + n(-1)^{n+1} \prod_{i=1}^{n} t_i$$

holds, where the sum in the lefthand side is taken over all $n^n$ distinct $n$-tuples for $C$. Simplifying and using $1 - x \leq e^{-x}$ yields

$$\sum \text{Val}(R_0, C) \quad = \quad n^{n+1} - n \prod_{i=1}^{n} (n - t_i) = n^{n+1} (1 - \prod_{i=1}^{n} (1 - \frac{t_i}{n}))$$

$$\geq \quad n^{n+1}(1 - e^{-\sum t_i/n}) = n^n(n-1)$$

which implies a lower bound of $n-1$ for the average of $\text{VAL}(R_0, C)$.

**Lemma 2** *For $t_1, \ldots, t_n$ and $M$ defined as above, the average value of* $\text{VAL}(R, C)$ *over all pairs of permutations $(R, C)$ of the numbers from $0$ to $n-1$ is at least $n-1$.*

**Proof.** It is sufficient to show that for some fixed permutation $R_0$ the average of $\text{VAL}(R_0, C)$ over all permutations $C$ is at least $n-1$.

We say that a permutation $\pi$ is *child* of some $n$-tuple $\nu$ iff $\nu$ can be transformed into $\pi$ in the following way: For each value that occurs in $\nu$ more than once, (i) we do not change its leftmost occurrence in $\nu$, but (ii) we overwrite all its other occurrences in $\nu$ by the corresponding value in $\pi$ (in other words, $\pi$ is child of $\nu$ iff the leftmost occurrence of each value in $\nu$ coincides with the unique occurrence of this value in $\pi$).

We claim that if $\pi$ is child of $\nu$, then $\text{VAL}(R_0, \pi) \geq \text{VAL}(R_0, \nu)$ holds: If $\nu$ assigns the same value to two columns (say to column $i$ and to column $j$, $i < j$), then $t_i \geq t_j$ implies that the right column $j$ only represents values that are also represented by the left column $i$. Since going from $\nu$ to $\pi$ leaves the leftmost occurrences unchanged, this cannot decrease $\text{VAL}(R_0, \nu)$.

Now let $\text{CHILD}(\nu)$ denote the number of the children of $n$-tuple $\nu$ and consider an enumeration $N = (\nu_1, \nu_2, \ldots, \nu_{n^n})$ of all $n$-tuples and an enumeration $P = (\pi_1, \pi_2, \ldots, \pi_{n!})$ of all permutations. We define rational numbers $w_{ij}$, $1 \leq i \leq n!$ and $1 \leq j \leq n^n$.

$$w_{ij} = \begin{cases} 0 & : \text{ if } \pi_i \text{ is not child of } \nu_j \\ 1/\text{CHILD}(\nu_i) & : \text{ if } \pi_i \text{ is child of } \nu_j \end{cases}$$

By the definition of the $w_{ij}$, $\sum_{i=1}^{n!} w_{ij} = 1$ holds for each $j$. Thus, the overall sum of all $w_{ij}$ equals $n^n$, and for reasons of symmetry we get that $\sum_{j=1}^{n^n} w_{ij} = n^n/n!$. Moreover, we have

$$\text{VAL}(R_0, \nu_j) \leq \sum_{i=1}^{n!} w_{ij} \text{VAL}(R_0, \pi_i)$$

since the $w_{ij}$ in the righthand side sum up to 1, and since only children of $\nu_j$ have non-zero coefficients. Finally, we use the above inequality to derive

$$\sum_{j=1}^{n^n} \text{VAL}(R_0, \nu_j) \leq \sum_{j=1}^{n^n} \sum_{i=1}^{n!} w_{ij} \text{VAL}(R_0, \pi_i) = \frac{n^n}{n!} \sum_{i=1}^{n!} \text{VAL}(R_0, \pi_i).$$

Thus, the $(n-1)$ lower bound for the average of $\text{VAL}(R_0, C)$ over all $n$-tuples $C$ is also a lower bound for the average of $\text{VAL}(R_0, C)$ over all permutations $C$. □

**Lemma 3** *Under assumption (As), the expected number $B$ of bad $c_k$ that have to be treated in Step (T4) of TACU is at most* 1.

**Proof.** We fix two non-decreasing sequences $a$ and $b$ and consider the corresponding sorted matrix $M$ as constructed in Step (T2) of TACU. If the $n \ln n$ largest values in $M$ are colored red, they form a configuration fulfilling the conditions of Lemma 2 with appropriate numbers $t_j$ satisfying $\sum t_j = n \ln n$.

Now assume TACU receives as input a permutation $R$ of $a$ and a permutation $S$ of $b$. Then some $c_k$ is good if and only if a red entry $m_{ij}$ with $k = r_i + s_j$ exists in $M$. Since by assumption (As) all pairs $(R, S)$ occur as input with equal probability, Lemma 2 implies that the expected value of good $c_k$ is at least $n-1$, or equivalently, the expected value $B$ of bad $c_k$ is at most 1. □

Finally, we want to consider the special case of MAXCON, where the input sequence $a$ is in non-increasing and $b$ is in non-decreasing order. It is straightforward to see that in this case, $c_k = \max_{0 \leq i \leq n-1}(a_{k-i} + b_i)$ boils down to $c_k = \max\{a_0 + b_k, a_{k+1} + b_{n-1}\}$, and MAXCON becomes solvable in $O(n)$ time.

Setting $a_i = n - i$ and $b_i = i$ for $0 \leq i \leq n-1$, we derive from the above that $c_k = \max\{n+k, 2(n-1)-k\}$. Thus, $c_k \leq \frac{5n}{3} - 1$ holds for $\frac{n}{3} - 1 \leq k \leq \frac{2n}{3} - 1$, whereas the $(n \ln n)$-th largest values in the set $\{a_i + b_j \mid 0 \leq i, j \leq n-1\}$ are all greater than $\frac{5n}{3} - 1$, for $n$ sufficiently large. In this case, TACU must handle $n/3 \in \Theta(n)$ bad $c_k$ and this leads to the following lemma.

**Lemma 4** *There exist input sequences $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n-1}$ for which TACU needs $\Theta(n^2)$ steps.* □

# 4 Algorithm FILL

In view of a practically efficient implementation, our first algorithm has certain disadvantages. In Step (T3) the running time is at least $O(n \ln n)$ although for nice instances of MAXCON $O(n)$ time will suffice. The preparations performed in Step (T2) use sophisticated algorithms with much overhead. If after Step (T3) we observe that the number of bad $c_k$ is more than expected it would be nice to extend the loop in Step (T3) in order to eliminate some of these bad $c_k$ before actually running the very slow $O(n)$-loops in Step (T4). However, the set of the $n \ln n$ largest numbers was generated in Step (T2) and cannot easily be extended afterwards.

For these reasons, we investigated several modifications (called FILL, FILL1 and FILL2) that circumvent these difficulties.

Our algorithm FILL successively generates the elements of the ordered sequence $v$ and runs the loop (cf. equation 2) until all components of $c$ are finite. The index $k$ of some *finite* $c_k$ is called *closed*, otherwise $k$ is called *open*.

In a preprocessing step, we sort the sequences $a$ and $b$. In the following we will assume that two corresponding permutations $r$ and $s$ are given such that $a_{r(0)} \leq a_{r(1)} \leq \ldots \leq a_{r(n-1)}$ and such that $b_{s(0)} \leq b_{s(1)} \leq \ldots \leq b_{s(n-1)}$. The set $V := \{(i,j) \mid i,j = 0, \ldots, n-1\}$ is a lattice with respect to the usual partial order of integer vectors. Now, $v(i,j) := a_{r(i)} + b_{s(j)}$ is called the *value* of $(i,j)$ and $\kappa(i,j) := r(i) + s(j)$ (index calculation modulo $n$) is called its *index*. Obviously, $(i,j) \leq (k,l)$ implies $v(i,j) \leq v(k,l)$, i.e. the total order given by the values extends the a priori known partial order. In the loop defined in equation 2, we have to scan the lattice in the order given by the values. It suffices to consider the upper half of the lattice, i.e. all elements in $V_{\geq} := \{(i,j) \mid i+j \geq n-1\}$, as for all $(i,j) \notin V_{\geq}$ there exists at least one $(k,l) \geq (i,j)$, $(k,l) \in V_{\geq}$ with $\kappa(k,l) = \kappa(i,j)$.

A *cover* of $W \subseteq V_{\geq}$ is a subset of $W$ containing all maximal elements of $W$, e.g. the maximum $(n-1, n-1)$ of the lattice is a cover of $V_{\geq}$. Obviously, if $T$ covers $W$ and $(i,j) \in T$ then the set

$$T(i,j) := \big(T \setminus \{(i,j)\}\big) \cup \big(\{(i-1,j),(i,j-1)\} \cap V_{\geq}\big) \tag{3}$$

covers $W \setminus \{(i,j)\}$. For example, $\{(n-2,n-1),(n-1,n-2)\}$ covers $V_{\geq} \setminus \{(n-1,n-1)\}$ for $n \geq 2$. If $T$ covers all unscanned elements $W \subseteq V_{\geq}$ then

$$\max\{v(k,l) \mid (k,l) \in T\} = \max\{v(k,l) \mid (k,l) \in W\}.$$

Therefore, it is sufficient to keep a cover $T$ of all unscanned elements of $V_{\geq}$. Algorithm FILL is summarized in Figure 2.

**Algorithm FILL**

(F1) $c \equiv -\infty$; $T := \{(n-1, n-1)\}$; $open := n$.

(F2) Find $v(i,j) := max\{v(k,l) \mid (k,l) \in T\}$.

(F3) If $v(i,j) > c_{\kappa(i,j)} = -\infty$
then $c_{\kappa(i,j)} := v(i,j)$; $open := open - 1$.

(F4) If $open = 0$ then stop.
$T := T(i,j)$; (according to equation 3)
and goto (F2).

Figure 2: A practicable fast algorithm for MAXCON

One possible improvement of FILL consists in keeping the new cover $T(i,j)$ of all unscanned elements as small as possible (since this will reduce the search time in Step (F2)). Clearly, the smallest possible new cover is the set of all maximal elements in $T(i,j)$, and these maximal elements form an antichain. If the previous cover $T$ was an antichain, then we only have to check whether the new elements $(i-1,j)$ resp. $(i,j-1)$ are already covered by some element in $T \setminus \{(i,j)\}$. With this the corresponding new Step (F4') reads:

(F4') If $open = 0$ then stop.
$T{:=}T \setminus \{(i,j)\}$;
If $i + j = n - 1$ then goto (F2);
If $T$ does not cover $\{(i-1,j)\}$ then $T := T \cup \{(i-1,j)\}$;
If $T$ does not cover $\{(i,j-1)\}$ then $T := T \cup \{(i,j-1)\}$;
goto (F2).

This modification of algorithm FILL consisting of Steps (F1), (F2), (F3) and (F4') will be called FILL1. Algorithm FILL1 guarantees that set $T$ always contains at most $n$ elements, since $T$ is an antichain.

A further improvement results from the observation that only the set of all unscanned elements with *open* indices needs to be covered. Algorithm FILL2 exploits this observation every time some new element is added to $T$ in Step (F4'). Let us assume that element $(i-1,j)$ is not covered by $T \setminus \{(i,j)\}$ and that element $(i,j-1)$ is covered by $T \setminus \{(i,j)\}$ (cf. Figure 3).

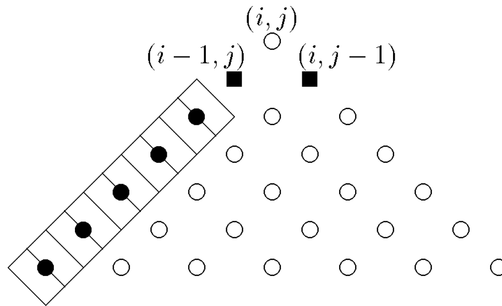


Figure 3: Lattice seen from $(i,j)$

After removal of $(i,j)$ from $T$ all elements represented by white circles are still covered

and consequently $(i, j-1)$ is not added to $T$. In order to cover all elements represented by black circles, we have to add the black square $(i-1, j)$. Since only open black circles have to be covered, it suffices to add the first open element $(i-k, j) \in V_{\geq}$. If any of the black circles $(i-l, j)$ with $l \in \{1, \ldots, k\}$ is already covered by $T$ or if all black circles are closed, $T$ remains unchanged.

The symmetric case where $(i-1, j)$ is covered but $(i, j-1)$ is not, can be treated in an analogous way. If both elements are not covered, one may choose one of them, add it to $T$ and then apply the same procedure as in the two cases before to the other (possibly closed) element. If both elements are covered, $T$ remains unchanged, anyway.

The modification of algorithm FILL1 which implements these treatment of closed elements in the construction of the new cover is called FILL2.

We conclude this section with several notes on the implementation and on the behaviour of FILL, FILL1 and FILL2. In all these algorithms the essential operations are

- Find the maximum of $T$;
- Delete the maximum of $T$;
- Add one or two elements to $T$.

We considered several different data structures for the implementation of $T$: *sorted lists*, *linked lists*, and *balanced binary trees*. Obviously, with balanced trees, an iteration of FILL1 runs in amortised time $O(\log n)$.

By similar arguments as in the analysis of algorithm TACU, we expect that after $n \ln n$ iterations in FILL1 resp. FILL2 only one index is open. However, there are examples with $O(n)$ iterations as well as with $O(n^2)$ iterations. For example, we consider

$$a_i = 2^i, \quad i = 0, \ldots, n-1; \qquad b_i = -2^{n-1-i}, \quad i = 0, \ldots, n-1.$$

The sequence of values generated in FILL1 is $a_{n-1} + b_{n-1}, a_{n-1} + b_{n-2}, \ldots, a_{n-1} + b_1, a_{n-2} + b_{n-1}, \ldots, a_{n-2} + b_2, \ldots, a_1 + b_{n-1}, a_{n-1} + b_0, \ldots$. The indices of the upper part of the corresponding lattice are given in Figure 4. After $n-1$ iterations exactly $n-1$ indices are closed, but the last index $n-1$ remains open for $\frac{n(n-1)}{2}$ iterations.

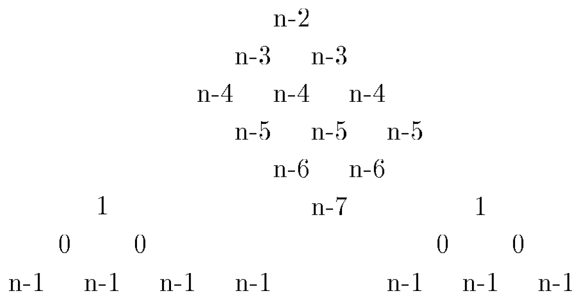


Figure 4: Indices in $V_{\geq}$

FILL2 only needs $n$ iterations, but still checks $O(n^2)$ indices. Obviously, TACU gives the solution of this problem in $O(n \ln n)$. There are other examples, in which TACU needs $O(n^2)$ but FILL1 runs in $O(n \ln n)$.

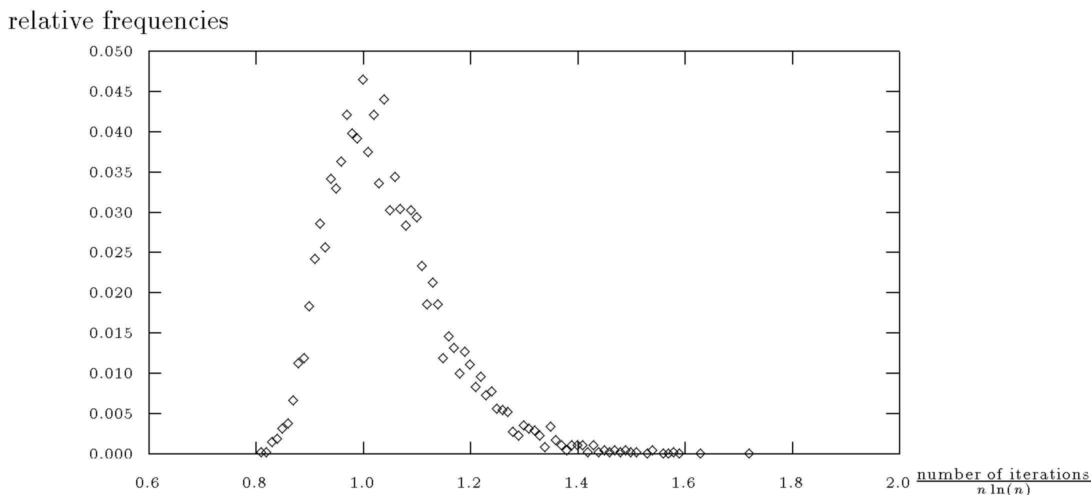


Figure 5: FILL1: $n = 100000$

# 5 Numerical Investigations

For numerical experiments, testdata (vectors $a$ and $b$) were randomly generated and sorted for $n = 1000, 10000, 100000$. All experiments were run on the HP 720 of the Abt. Math. Optimization at TU Braunschweig. FILL1/2 were coded in ANSI-C. With algorithm FILL1, the theoretical bound of lemma 3 is clearly observed (cf. Figure 5). The peak at $n \ln n$ becomes sharper with increasing $n$. For FILL2 the number of iterations is much lower with a peak close to $2n$ (cf. Figure 6).

On the other hand, the effort per iteration is higher in FILL2 than in FILL1. In particular in the last iterations when many indices are closed, the search for a suitable new element with open index descents deep into the lattice. In these cases, FILL2 calculates many indices which are not considered in FILL1. Therefore, one should dynamically restrict the length of such a search.

We counted the number of searchsteps occuring in the update of $T$ (cf. Figure 7). Let $c_{start}$, $c_{previous}$, and $c_{last}$ denote the number of searches during the first, previous to the last, and last $\frac{n}{100}$ iterations. We switch from FILL2 to FILL1 when $\frac{c_{last} - c_{start}}{c_{previous}} \geq f$, for an appropriate positive threshold $f$ (in numerical experiments the constant $f = 1.1$ turned out to be reasonable). Furthermore, we observed that the number of elements in $T$ remains much smaller than $n$ ($n = 100000$: $T$ usually contains not more than 350 elements). Thus, a binary tree implementation is not effective.

# 6 Conclusions

It is easy to see that for the special case where $a$ and $b$ only consist of values $(-\infty)$ and 0, the solution to the standard sum convolution problem with corresponding 0-1-sequences implies a solution to MAXCON. Therefore, in this case there exists an $O(n \log n)$ algorithm for MAXCON. Of course, the major open problem of constructing an algorithm for MAXCON with *subquadratic* worst case time complexity remains open.

Nevertheless, our numerical experience indicates that the proposed combination of algorithms FILL1/2 yields a fast algorithm for solving MAXCON. In fact, the average

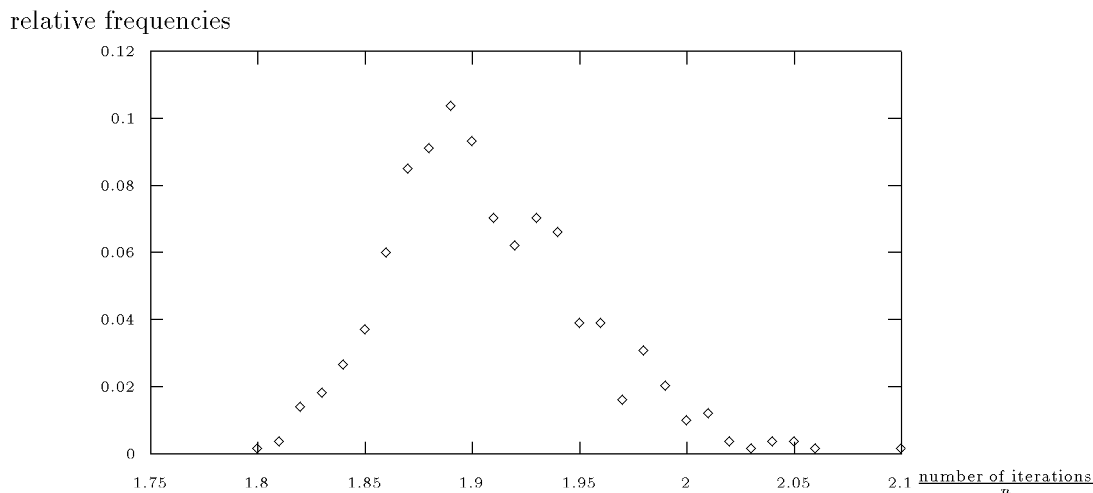


Figure 6: FILL2: $n = 100000$

running time (without sorting) is 15 sec for $n = 100000$ on the HP 720.

## References

[1] A.V.Aho, J.E.Hopcroft and J.D.Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.

[2] J.G.Fletcher, A more general algorithm for computing closed semiring costs between vertices of a directed graph, *Comm. ACM.* ***23***(6), 1980, 350–351.

[3] R.W.Floyd, Algorithm 97, Shortest path, *Comm. ACM.* ***5***(6), 1962, 345.

[4] G.N.Frederickson and D.B.Johnson, Generalized selection and ranking: Sorted matrices, *SIAM J. Comput.* ***13***, 1984, 14–30.

[5] M.Gondran and M.Minoux, Graphs and Algorithms, Wiley–Interscience Series in Discrete Mathematics, John Wiley & Sons, Chichester, 1984.

[6] T.Lengauer and D.Theune, Unstructured path problems and the making of semirings, in F.Dehne, J.-R.Sack and N.Santoro (Ed.): Algorithms and Data Structures, *Lecture Notes in Computer Science* ***519***, 1991, 189–200.

[7] M.Minoux, Structures algébrique généralisées des problèmes de cheminement dans les graphs, *RAIRO Rech. Oper.* ***10***(6), 1976, 33–62.

[8] S.Warshall, A theorem on boolean matrices, *Comm. ACM.* ***9***(1), 1962, 11–12.

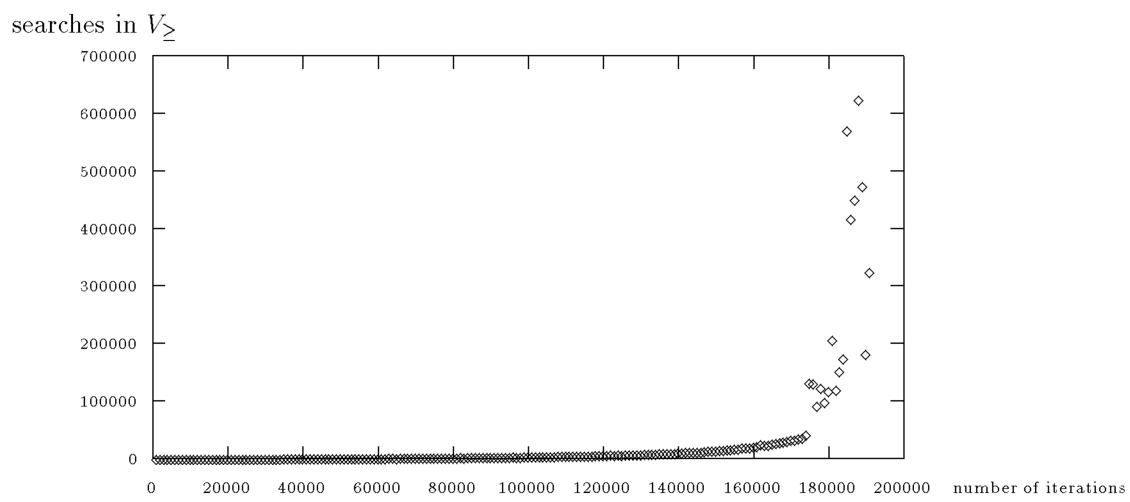


Figure 7: Number of searches per 1000 iterations, FILL2: $n = 100000$