Optimal Partitioning of Sequences

Fredrik Manne* and Tor Sørevik†

Abstract

The problem of partitioning a sequence of n real numbers into p intervals is considered. The goal is to find a partition such that the cost of the most expensive interval measured with a cost function f is minimized. An efficient algorithm which solves the problem in time $O(p(n-p)\log p)$ is developed. The algorithm is based on finding a sequence of feasible non-optimal partitions, each having only one way it can be improved to get a better partition. Finally a number of related problems are considered and shown to be solvable by slight modifications of our main algorithm.

*Norsk Hydro a.s., N-5020 Bergen, Norway, fmanne@bg.nho.hydro.com

†Department of Informatics, University of Bergen, N-5020 Bergen, Norway, Tor.Sorevik@ii.uib.no

1 Introduction and motivation

In parallel computing one often spends effort on certain kinds of precomputations in order to make the actual parallel algorithm execute faster. These precomputations fall into two main categories:

1. Precomputations in order to reduce dependencies among the different subtasks.

2. Determination of how to schedule subtasks to processors.

The goal of 1 is to get as many independent tasks as possible to make the problem more suitable for parallel computations, while the goal of 2 is to minimize the time when the last processor finishes. An example of 1 is to restructure a matrix to make it more suitable for parallel factorization [7]. In 2 there might be certain constraints on how to schedule the tasks to the processors. There might for example be a partial ordering on the subtasks or a release time when each task can first be executed.

While precomputations to reduce dependencies among subtasks are more ad hoc and problem specific, there exists a rich literature on how to schedule tasks to processors. For an overview see [4]. With no special constraints on the jobs or the processors, minimizing the total completion time is known to be NP-hard [3], even for two processors.

In this paper we present efficient algorithms for a number of restricted scheduling problems. In Section 2, we give a formal description of the main problem of the present paper and give a simple algorithm for obtaining a solution as close to the optimal as desired. In Section 3 we develop a new exact algorithm and prove its correctness. We also show that the running time of this algorithm is $O((n-p)p\log p)$. A number of related problems and their solutions are discussed in Section 4. In Section 5 we summarize and point to areas for future work.

2 Partitioning of sequences

We consider a restricted version of the scheduling problem. We assume that if job i is assigned to processor j then jobs with lower index than i must be assigned to processors numbered at most j and jobs with higher index than i must be assigned to processors numbered at least j. Our goal is to minimize the time before the last processor finishes.

To motivate why this particular problem is of interest consider the following example from Bokhari [2]:

In communication systems it is often the case that a continuous stream of data packages have to be received and processed in real time. The processing can among other things include demodulation, error correction and possibly decryption of each incoming data package before the contents of the package can be accessed [5]. Assume that n computational operations are to be performed in a pipelined fashion on a continuous sequence of incoming data packages. If we have n processors we can assign one operation to each processor and connect the processors in a chain. The time to process the data will now be dominated by the processor that has to perform the most time consuming operation. With this mapping of the operations to the processors, a processor will be idle once it is done with its operation and have to wait until the processor that has the most time consuming operation is done, before it can get a new data package. This is inefficient if the time to perform each task varies greatly. Thus to be able to utilize the processors more efficiently we get the following problem: Given n consecutively ordered tasks, each taking $f(i)$ time, and p processors. Partition the tasks into p consecutive intervals such that the maximum time needed to execute the tasks in each interval is minimized.

Bokhari [2] also described how a solution to this problem can be used in parallel processing as compared to pipelined. Mehrmann [9] shows how this particular partitioning problem arises when solving a block tridiagonal system on a parallel computer. We now give the formal definition of the problem:

Let the two integers $p \leq n$ be given and let $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ be a finite ordered set of real numbers. Let $R = \{r_0, r_1, ..., r_p\}$ be a set of integers such that $r_0 = 0 \leq r_1 \leq ... \leq r_{p-1} \leq r_p = n$. Then R defines a partition of $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ into p intervals: $\{\sigma_{r_0}, \sigma_1, ..., \sigma_{r_1-1}\}$, $\{\sigma_{r_1}, ..., \sigma_{r_2-1}\}$,..., $\{\sigma_{r_{p-1}}, ..., \sigma_{r_p-1}\}$. If $r_i = r_{i+1}$ then the interval $\{\sigma_{r_i}, ..., \sigma_{r_{i+1}-1}\}$ is empty.

Let f be a function, defined on intervals taken from the sequence $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$, that satisfies the following three conditions:

$$f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) \geq 0 \tag{1}$$

for $0 \leq i, j \leq n-1$, with equality if and only if $j < i$.

$$f(\sigma_{i+1}, \sigma_{i+2}, ..., \sigma_j) < f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) \tag{2}$$

for $0 \leq i \leq j \leq n-1$ and

$$f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) < f(\sigma_i, \sigma_{i+1}, ..., \sigma_{j+1}) \tag{3}$$

for $0 \leq i \leq j < n-1$.
The problem is then:

MinMax Find a partition R such that $\max_{i=0}^{p-1}\{f(\sigma_{r_i}, ..., \sigma_{r_{i+1}-1})\}$ is minimum over all partitions of $\{\sigma_0, ..., \sigma_{n-1}\}$.

Note that as the MinMax problem is stated there is no advantage in allowing an interval of zero length. We will, however, in Section 4.1 look at a related problem where we will need intervals of zero length. So far we have not said anything about the complexity of computing the cost function f. For most realistic problems we expect the two following criteria to be satisfied:

1. The function $f(\sigma_j)$ can be computed in time $O(1)$.
2. Given the value of $f(\sigma_l, \sigma_{l+1}, ..., \sigma_k)$ we can calculate f in time $O(1)$, where either k or l has been increased or decreased by one.

A straightforward example of such a function is $f(\sigma_l, \sigma_{l+1}, ..., \sigma_k) = \Sigma_{i=l}^{k}|\sigma_i|$, where $\sigma_i \neq 0$ for $0 \leq i \leq n-1$.

When there is no possibility for confusion we will use the simplified notation φ_j instead of $f(\sigma_{r_j}, ..., \sigma_{r_{j+1}-1})$. We will write $[r_j, r_{j+1}]$ for the sequence $\{\sigma_{r_j}, ..., \sigma_{r_{j+1}-1}\}$. From now on we assume that $r_0 = 0$ and $r_p = n$.

The first reference to the MinMax problem that we are aware of is by Bokhari [2] who presented an $O(n^3p)$ algorithm using a bottleneck-path algorithm. Anily and Federgruen [1] and Hansen and Lih [6] independently presented the same dynamic programming algorithm with time complexity $O(n^2p)$. The dynamic programming algorithm is as follows:

Let $g(i,k)$ be the cost of the most expensive interval in an optimal partition of $\sigma_i, \sigma_{i+1}, ..., \sigma_{n-1}$ into k intervals, $0 \leq i \leq n-k$ and $1 \leq k \leq p$. The cost of an optimal solution to the MinMax problem is then $g(0,p)$. The initial values are given by $g(i,1) = f(\sigma_i, ..., \sigma_{n-1})$ and $g(i, n-i) = \max_{i \leq j < n}\{f(\sigma_j)\}$. The following recursion

$\lambda := f(\sigma_0, \sigma_2, ..., \sigma_{n-1});$
$\tau := \max_{j=0}^{n-1}\{f(\sigma_j)\};$
Repeat
$\quad\rho := (\lambda + \tau)/2;$
$\quad$**If** we can find a partition such that $\max_{j=0}^{p-1}\{\varphi_j\} \leq \rho$
$\quad\quad$**then** $\lambda := \rho;$
$\quad\quad$**else** $\tau := \rho;$
Until $(\tau + \epsilon \geq \lambda);$

Figure 1: The Bisection algorithm

shows how $g(i,k)$ can be computed for $2 \leq k < n - i$:

$$g(i,k) = \min_{i \leq j \leq n-k}\{\max\{f(\sigma_i, ..., \sigma_j), g(j+1, k-1)\}\}$$

We can now solve the problem for $k = 1$ to p, and for each value of k for $i = n - k$ down to 0. This gives a time complexity of $O(n^2 p)$ and a space complexity of $O(n)$ to compute $g(0,p)$. Once $g(0,p)$ is known the actual placement of the delimiters can be computed in time $O(n)$.

The MinMax problem can also be solved by a bisection method. Since we can easily find an upper and lower limit on $\max_{j=0}^{p-1}\{\varphi_j\}$ we can get as good an approximation as we want by using bisection. We assume that $p < n$ and let ϵ be the desired precision. The algorithm is shown in Figure 1.

Note that if f is integer valued and $\epsilon = 1$ the bisection method will produce an optimal partition. We can determine if there exists a partition such that $\max_{j=0}^{p-1}\{\varphi_j\} \leq \rho$ in $O(n)$ time. This is done by adding as many elements as possible to the interval $[r_0, r_1]$ while keeping $\varphi_0 \leq \rho$. This process is then repeated for each consecutive interval. The overall time complexity of the bisection method is then $O(n \log((\lambda - \tau)/\epsilon))$ where $\lambda = f(\sigma_0, \sigma_1, ..., \sigma_{n-1})$ and $\tau = \max_{j=0}^{n-1}\{f(\sigma_j)\}$. The running time of this algorithm is polynomial in the input size but it is dependent on f and the values of σ. In Section 3 we will present a new algorithm for solving the MinMax problem which has running time $O((n-p)p \log p)$. This is more efficient than the bisection method if $\log((\lambda - \tau)/\epsilon)$ is large compared to $p \log p$.

3 A new algorithm

In this section we present an efficient iterative algorithm that finds an optimal partition for the MinMax problem. Our algorithm is based on finding a sequence of feasible non-optimal partitions such that there exists only one way each partition can be improved. This idea has also been used to partition acyclic graphs [8].

3.1 The MinMax algorithm

First we introduce a special kind of partition which is essential to our algorithm.

Definition 1 (A Leftist Partition (LP)) *Let $R = \{r_0, r_1, ..., r_p\}$ be a partition of $\{\sigma_0, ..., \sigma_{n-1}\}$ into p intervals where $[r_j, r_{j+1}]$ is a most expensive interval of cost $\varphi_j = \gamma$. Then R is a leftist partition if the following condition is satisfied:*

> *Let $[r_k, r_{k+1}]$ be the leftmost interval containing at least two elements. Then for any value of i, $k < i < p$, moving r_i one place to the left gives $\varphi_i \geq \gamma$.*

Let $R = \{r_0, ..., r_p\}$ define an LP such that $[r_j, r_{j+1}]$ is a most expensive interval of cost γ. If $r_{j+1} = r_j + 1$ so that the interval $[r_j, r_{j+1}]$ contains only one element, then the partition is optimal. Suppose therefore that the most expensive interval contains at least two elements. Our algorithm is then based on trying to find a new partition of lower cost by moving the delimiters r_i, $i \leq j$, further to the right. We do this by adjusting φ_i downward by removing solely the leading σ from $[r_i, r_{i+1}]$. The formal definition of the reordering step is as follows:

Reordering Step
$\gamma := \varphi_j$;
While $(\varphi_j \geq \gamma)$ **and** $(j > 0)$
    $r_j := r_j + 1$;
    If $(\varphi_j < \gamma)$
        then $j := j - 1$;
End While

If we terminate with $\varphi_0 \geq \gamma$ the reordering step was unsuccessful in finding a new partition of less cost than γ, and we restore our original partition.

We now show that given an LP, if we apply our reordering step the resulting partition will also be an LP.

Lemma 2 *The partition obtained after successful application of the reordering step to an LP is also an LP.*

Proof: Let $[r_j, r_{j+1}]$ be the interval from which we start the reordering step. Then prior to the reordering step $[r_j, r_{j+1}]$ is an interval of maximum cost γ. Let further $\omega \leq \gamma$ be the cost of a most expensive interval after the reordering and let r_l be the leftmost delimiter such that the position of r_l was changed during the reordering. (If $\omega = \gamma$ then there was more than one interval of cost γ in our original partition.) Note first that $r_i < r_{i+1}$, $l \leq i \leq j$, after the reordering step terminates. This is because the only way we can get $r_i = r_{i+1}$ is if there is an element σ_k that lies to the left of $[r_j, r_{j+1}]$ for which $f(\sigma_k) = \gamma$. But this would cause $\varphi_0 \geq \gamma$ when the reordering step terminates, contradicting the fact that we performed a successful application of the reordering step. We now look at each interval $[r_q, r_{q+1}]$ and show that it satisfies the conditions of an LP.

- $0 \leq q < l-1$ and $j < q < p$. These intervals remain unchanged and since $\omega \leq \gamma$ they still satisfy the condition of an LP.
- $q = l-1$. The interval $[r_{l-1}, r_l]$ satisfies the condition since the cost of this interval increased.
- $l \leq q \leq j$. Each of these intervals satisfies the condition since moving r_q one place to the left will give $\varphi_q \geq \gamma \geq \omega$.

Thus we see that after performing a successful iteration of the reordering step on an LP we still have an LP. This concludes the proof. □

If the reordering procedure fails to improve the partition, the following lemma tells us that the starting partition was optimal.

Lemma 3 *If the reordering step on an LP is unsuccessful then this LP is optimal.*

Proof: Let R be a partition to which the reordering step was applied but proved unsuccessful, and let $[r_j, r_{j+1}]$ be the interval of maximum cost $\varphi_j = \gamma$ on which the reordering step began. Let R' be a partition of $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ given by $0 = r'_0 \leq r'_1 \leq ... \leq r'_{p-1} \leq r'_p = n$. If $r'_j \leq r_j$ and $r'_{j+1} \geq r_{j+1}$, clearly the cost associated with R' is no smaller than that associated with R. In other words, R' cannot possibly cost less than R unless $r'_j > r_j$ or $r'_{j+1} < r_{j+1}$.

The reordering step starts by moving r_j one place to the right. Since the reordering step was unsuccessful there does not exist a partioning of $[r_0, r_j + 1]$ into j intervals such that the maximum cost is less than γ. It follows that if $r'_j > r_j$ the cost of R' is greater than or equal to γ.

From Definition 1 it follows that any partition of $[r_{j+1}-1, r_p]$ into $p-j-1$ intervals will be of cost at least γ. Thus if $r'_{j+1} < r_{j+1}$ the cost of R' will be at least γ. It follows that there exists no partition of $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ into p intervals of cost less than γ. $\Box$

We have now seen that a successful reordering of an LP gives a new LP and that an LP must be optimal if the reordering step is unsuccessful. It remains to find the first LP to start the algorithm. But this is easy: we set $r_j = j$ for $1 \leq j < p$. The $p-1$ leftmost intervals are each of length one and interval $[r_{p-1}, r_p]$ contains the rest of the numbers. Thus this placement of the delimiters gives an LP from which we can start. This also shows that there must exist an optimal LP since we can move each r_i at most $n-p$ places. Note that if the most expensive interval in the initial LP is any other than $[r_{p-1}, r_p]$ the starting LP is optimal.

The complete algorithm for solving the MinMax problem is given in Figure 2.

To simplify finding a most expensive interval we maintain φ_j for each interval in a heap ordered by decreasing cost. This heap is updated each time we move a delimiter.

3.2 The complexity of the MinMax algorithm

We now give both the time complexity of the worst case and the average case for the MinMax algorithm. In the worst case each of the $p-1$ interior delimiters is moved $n-p$ places. Thus we see that the total number of moves is never more than $(n-p)(p-1)$. We must find an interval of maximum cost before each iteration of the reordering step. Since we maintain the costs of the intervals in a heap this can be done in $O(1)$ time. The heap is updated each time we move a delimiter at a cost of $O(\log p)$. Thus we see that we never do more than $(n-p)(p-1)$ updates for a total cost of $O((n-p)p\log p)$. We note that for $p = n$ the algorithm has time complexity $O(n)$.

It is easy to construct examples for which the MinMax algorithm will perform $(n-p)(p-1)$ moves of the delimiters. Thus this is a sharp bound on the number of

```
r_0 := 0;
r_p := n;
For i := 1 to p − 1
    r_i := i;
Repeat
    i := {j | φ_j is maximum over all j};
    γ := φ_i;
    If (r_{i+1} = r_i + 1)
       then exit;
    While (φ_i ≥ γ) and (i > 0)
       r_i := r_i + 1;
       If φ_i < γ
          then i := i − 1;
    End While
Until (φ_0 ≥ γ);
Restore the last partition;
```

Figure 2: The MinMax algorithm

moves. However, we do not know if there exist problems where one has to do $\Theta((n-p)p)$ updates on the heap where each update costs $\Theta(\log p)$.

If each possible partition has the same probability of being optimal, the average number of moves in the MinMax algorithm is half the number of moves performed in the worst case. To see this let T be the set of all possible partitions of the form $\{r_0, r_1, ..., r_p\}$. The number of moves needed to reach a particular partition in T is $\Sigma_{i=1}^{p-1}(r_i - i)$. For each partition in T there exists a unique dual partition $\{0 = n - r_p, n - r_{p-1}, n - r_{p-2}, ..., n - r_1, n = n - r_0\}$. To reach this dual partition we need to make $\Sigma_{i=1}^{p-1}(n - r_{p-i} - i)$ moves. The number of moves needed to reach a partition and its dual is then $(n-p)(p-1)$, so that the total number of moves needed to reach each partition twice is $|T|(n-p)(p-1)$. Thus the average number of moves to reach a partition in T is $(n-p)(p-1)/2$, which is exactly half the number of moves needed in the worst case.

4 Related problems

We have now seen three different ways in which the MinMax problem can be solved. We have seen that the MinMax algorithm has lower complexity than the dynamic programming algorithm, and that for an approximate solution either the bisection method or the MinMax algorithm is the best choice depending on the data and the desired precision. In this section we will look at some related problems which at first might seem to be more difficult than the MinMax problem. For each of these problems we will show how it can be solved by simple modifications of the MinMax algorithm. For each problem we also show how the bisection method can be used.

4.1 Charging for the number of intervals

When scheduling tasks to processors there might be an extra cost associated with splitting the problem and/or combining the partial solutions in order to obtain a global solution. An example of an algorithm which solves such a problem can be found in [11]. The total execution time of these algorithms is then $\max_{j=0}^{p-1}\{\varphi_j\} + h(p)$ for $1 \leq p \leq n$. Here $h(p)$ is the extra cost of splitting/combining the problem. Since $\max_{j=0}^{p-1}\{\varphi_j\}+h(p)$ is a non-smooth integer function the only way to find its minimum is to evaluate the function for all values of p. In this section we show how this can be done by modifying the MinMax algorithm so that we solve the MinMax problem for each value of p between 1 and n.

As we presented the MinMax algorithm in Section 2 we never encounter intervals of length zero. However, neither the definition of an LP nor the proofs of Lemmas 2 and 3 prohibit us from redefining an LP to allow intervals furthest to the left of zero length. It follows that the MinMax algorithm will return an optimal partition from such an LP. Consider the trivial partition obtained by setting $r_j = 0$ for $0 \leq j < p$. If this is used as the starting partition for the MinMax algorithm we have to perform $(n - p/2)(p - 1)$ moves of the delimiters in the worst case.

Assume that we have some leftist partition where we have divided the sequence into k intervals $1 \leq k < n$. We now add a new delimiter in position zero thus giving the first interval length zero. This partition is still leftist and if we continue the MinMax algorithm we end up with an optimal partition for $k + 1$ intervals.

To solve the MinMax problem for each value of p we first evaluate the cost function for $p = 1$ before solving it for $p = 2$. In the last reordering step when we get $\varphi_0 \geq \gamma$ and it becomes clear that the previous partition was optimal, we evaluate the cost function for the optimal partition. This value is compared with the previous lowest value. When this is done we introduce a new empty interval in position zero and continue and solve the problem for $p = 3$. This is then repeated for each consecutive value of $p \le n$. In this way we will find the value of p which minimizes $\max_{j=0}^{p-1}\{\varphi_j\} + h(p)$. The maximum number of steps we have to move the delimiters is $n(n-1)/2$ giving an overall time complexity of $O(n^2 \log n)$.

If we use the bisection method for each value of p to find an optimal partition we get a total time complexity of $O(n^2 \log(\lambda - \tau)/\epsilon))$.

4.2 Bounds on the intervals

In this section we will consider the MinMax problem with the added restriction that each σ_i has a size and there is a limit on the maximum size of each interval. This is the case when scheduling jobs to processors and each processor has a limited amount of memory to store its job queue.

Let s be a function defined on continuous intervals of $\{\sigma_0, \sigma_1, ..., \sigma_{p-1}\}$ in the same way that f was defined in Section 2. We call $s(i,j)$ the *size* of the interval $\{\sigma_i, ..., \sigma_j\}$. Formally the problem now becomes:

Bounded MinMax Given positive real numbers $U_0, U_1, ..., U_{p-1}$, find an optimal partition for the MinMax problem with the constraint that $s(r_j, r_{j+1} - 1) \le U_j$ for $0 \le j \le p-1$.

Note that there might not exist a solution to the bounded MinMax problem. This can in fact be determined by testing if the starting partition satisfies the size constraints. The starting partition is given by setting the delimiters as far left as possible while maintaining the restrictions on the size of each interval. More formally the starting partition is given by $r_0 = 0, r_p = n$ and the following formula: $r_i = \min\{j \mid j \ge 0 \wedge s(j, r_{i+1} - 1) \le U_i \wedge s(r_k, r_{k+1} - 1) \le U_k, i < k < n\}$, $0 < i < n$. If this gives us $s(0, r_1 - 1) > U_0$ then there exists no solution to the bounded MinMax problem; otherwise, there exists a solution.

Let r_k be the leftmost nonzero delimiter, and let γ be the cost of a most expensive interval. We now define a leftist partition such that moving r_i to the left for $k \le i < p$ causes at least one of the following two events to occur:

1. $\varphi_i \geq \gamma$

2. $s(r_i, r_{i+1} - 1) > U_i$

Our reordering step is as follows:

$i := \{j \mid \varphi_j$ is maximum over all $j\}$;
$\gamma := \varphi_i$;
While $((s(r_i, r_{i+1} - 1) > U_i)$ **or** $(\varphi_i \geq \gamma))$ **and** $(i > 0)$
    $r_i := r_i + 1$;
    If $s(r_i, r_{i+1} - 1) \leq U_i$ **and** $(\varphi_i < \gamma)$
        then $i := i - 1$;
End While

If we terminate with $\varphi_0 \geq \gamma$ or $s(0, r_1 - 1) > U_0$ the reordering step was unsuccessful and we restore our original partition.

It is straightforward to modify the proofs of Lemmas 2 and 3 to show that repeated applications of the reordering step will converge to an optimal partition. Since each delimiter might be moved $O(n)$ times, the time complexity of this method is $O(np \log p)$.

The bisection method can also be used to solve this method without altering its time complexity.

Note that if $s(r_i, r_{i+1} - 1) = r_{i+1} - r_i$ (the number of elements in each interval) and each U_k is a positive integer then we can use the following starting partition: $r_i = \max\{i, n - \Sigma_{l=i}^{p-1} U_l\}$, $0 < i < p$. This gives a time complexity of $O((n-p)p \log p)$.

4.3 Cyclic sequences

In this section we consider the problem when the data is organized in a ring. This means that the sequence $\{\sigma_{r_{p-1}}, ..., \sigma_{n-1}, \sigma_0 ..., \sigma_{r_0 - 1}\}$ is now interval p. We assume that the data is numbered clockwise $\sigma_0, \sigma_1, ..., \sigma_{n-1}$. The problem now becomes: Find a partition $R = \{r_0, r_1, ..., r_{p-1}\}$ such that $\max_{j=0}^{p-1}\{\varphi_j\}$ is minimum.

Note that we no longer require $r_0 = 0$ and that we do not make use of r_p.

Our reordering step is similar to the one used in the MinMax algorithm. The main difference is that where in the MinMax algorithm we would have terminated when $\varphi_0 \geq \gamma$, we now continue and try to move r_0 to make the interval $[r_0, r_1]$ less expensive.

```
For i := 0 to p − 1
      r_i := i;
While (r_{p−1} ≠ 0)
      i := {j | φ_j is maximal over all j};
      γ := φ_i;
      If r_{(i+1)mod p} = ((r_i + 1) mod n)
         then exit;
      While (φ_i ≥ γ) and (r_{p−1} ≠ 1)
         r_i := (r_i + 1) mod n;
         If φ_i < γ
            then i := (i − 1) mod p;
      End While
End While
Restore the previous best partition;
```

Figure 3: The cyclic MinMax algorithm

We continue to run the algorithm until $r_{p-1} = 0$. The algorithm is as shown in Figure 3.

If the algorithm exits with $r_{i+1} = r_i + 1$, we know that the partition is optimal. As the following lemma shows this is also the case if the algorithm exits with $r_{p-1} = 0$.

Lemma 4 *When $r_{p-1} = 0$ we have encountered an optimal partition at some stage of the algorithm.*

Proof: Let $t_0, t_1, ..., t_{p-1}$ be an optimal placement of the delimiters such that $t_j < t_{j+1}$ for $0 \leq j < p-1$, and the most expensive interval is of cost γ. In the starting position we know that $r_j \leq t_j$ for $0 \leq j \leq p-1$. Let r_i be the first delimiter which is moved past t_i. Let further π be the cost that we are trying to get φ_i below. When $r_i = t_i$ we know that $\pi \leq \varphi_i$. We also know that $\varphi_i \leq \gamma$ since r_{i+1} has not been moved past t_{i+1}. This implies that $\pi \leq \gamma$. But since γ was the optimal solution we must have $\pi = \gamma$. Thus at some point of our algorithm we had a partition that was optimal. □

As soon as r_{p-1} has been moved to position 0 we know that at least one r_j has been moved passed t_j (since $t_{p-1} \leq n-1$) and that further moving of the delimiters will not decrease the cost of the most expensive interval. We can now restore the previous best partition knowing that it is optimal. It is clear that this algorithm has running time $O((n-p)p\log p)$.

To solve for all values of p we will have to restart the algorithm from scratch each time. This gives a running time of $O(n^3 \log n)$.

The initial problem can also be solved using the bisection method. First we partition $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ into p intervals such that no interval contains more than $\lceil n/p \rceil$ elements. Let φ_i be a most expensive interval. Then if we start the bisection method starting from each σ_j where $r_i \leq j \leq r_{i+1}$ we will find an optimal partitioning. To see this note that any partitioning that does not contain a delimiter in the interval $[r_i, r_{i+1}]$ will be of cost $> \varphi_i$. The time complexity of this method is $O(n^2/p\log((\lambda-\mu)/\epsilon))$. If we want to solve the problem for each value of p the time complexity becomes $O(n^2 \log n \log((\lambda-\mu)/\epsilon))$.

4.4 The MaxMin problem

When solving the MinMax algorithm we try to make the cost of the most expensive interval as low as possible by leveling out the cost of the intervals. Another way to level out the cost of the intervals is to ask for a partition such that the cost of the least expensive interval is as high as possible. The definition of this problem is as follows:

MaxMin Let f be as for the MinMax problem. Find a partition such that $\min_{j=0}^{p-1}\{\varphi_j\}$ is *maximum* over all partitions of $\sigma_0, ..., \sigma_{n-1}$.

Note that both the bisection method and the dynamic programming approach can be modified to solve the MaxMin problem.

As we did for the MinMax problem we first define a special kind of partition.

Definition 5 (A MaxMin Partition (MP)) *Let $R = \{r_0 = 0, r_1, ..., r_{p-1}, r_p = n\}$ define a placement of the delimiters such that $[r_j, r_{j+1}]$ is a least expensive interval of cost $\varphi_j = \gamma$. Then R is an MP if the following condition is satisfied:*

- *Moving r_i, $0 < i < p$, one place to the left gives $\varphi_{i-1} \leq \gamma$.*

Note that the starting partition for the MinMax algorithm also is an MP. This follows since moving r_i, $0 < i < p$, will make $r_{i-1} = r_i$ and thus $\varphi_{i-1} = 0$. If $[r_{p-1}, r_p]$ is a least expensive interval this MP is clearly optimal.

```
r_0 := 0;
r_p := n;
For i := 1 to p − 1
    r_i := i
Repeat
    i := max{j | φ_j is minimal over all j}
    γ := φ_i;
    While (φ_i ≤ γ) and (i ≠ p − 1)
          r_{i+1} := r_{i+1} + 1;
          If φ_i > γ
             then i := i + 1;
    End While
Until (φ_{p−1} ≤ γ);
Restore the last partition;
```

Figure 4: The MaxMin algorithm

The main difference between the MinMax algorithm and the MaxMin algorithm is that in the MaxMin algorithm we move the right delimiter of the current interval instead of the left. Also, once the current interval has reached the desired size we continue to work with the interval to its right. The algorithm is shown in Figure 4.

The reason for breaking ties by choosing the rightmost eligible interval is to avoid moving some r_k past r_{k+1}. To see that this has the desired effect we need the following lemma:

Lemma 6 *Let $r_j, \ldots, r_l$ be the delimiters that are moved during one iteration of the Repeat-Until loop in the MaxMin algorithm. Then $r_i < r_{i+1}$, $j \leq i \leq l$, after the iteration.*

Proof: Let γ be the initial cost of φ_{j-1}. We first show that $r_i \leq r_{i+1}$, $j \leq i \leq l$, after the iteration of the Repeat-Until loop. The proof is by induction on the number of delimiters moved in the while-loop. The first delimiter which is moved is r_j. Since initially $r_{j+1} > r_j$ and because r_j is moved only one place, the induction hypothesis remains true after r_j has been moved.

Assume that the while-loop has progressed to φ_{k-1}, $j < k$, and that $r_i \leq r_{i+1}$ for $j \leq i < k$. If $\varphi_{k-1} > \gamma$ or $k = p$ the reordering step is done and the induction hypothesis remains true. Assume therefore that $\varphi_{k-1} \leq \gamma$ and $k < p$. By the way j was chosen we have $\varphi_k > \gamma$. This implies that moving r_k until $r_k = r_{k+1}$ will give $\varphi_{k-1} > \gamma$. Therefore r_k is never moved past r_{k+1} and the induction hypothesis remains true. The main result now follows since after each iteration of the Repeat-Until loop we have $0 < \gamma < \varphi_i$ for $j - 1 \leq i \leq l$ (with the possible exception of φ_{p-1}). By the way f is defined this implies that $r_i < r_{i+1}$, $j \leq i < l$. □

The following two lemmas show that the MaxMin algorithm does produce an optimal partition. Their proofs are similar to the proofs of Lemma 2 and 3 and are omitted here.

Lemma 7 *The partition obtained after a successful application of the reordering step on an MP is also an MP.* □

Lemma 8 *If the reordering step on an MP is unsuccessful then the MP is optimal.* □

It is not difficult to show that the MaxMin algorithm has both the same upper limit on the time complexity in the worst case and the same average case time complexity as the MinMax algorithm.

5 Conclusion

We have shown algorithms for the MinMax problem and several variations of this problem. The time complexity of the algorithm for the initial problem was shown to be $O((n-p)p\log p)$. This was compared to other algorithms which either had worse time complexity or had a time complexity which was dependent on the f function and the size of the numbers in the sequence. We based our algorithm on finding sequences of non-optimal partitions, each having only one way it can be improved. This seems to be a useful technique in the design of algorithms.

In a recent development Olstad and Manne [10] have improved the dynamic programming approach for solving the MinMax problem to $O(p(n-p))$. However, since their method uses dynamic programming it *must* perform $p(n-p)$ steps on any input, where our algorithm might perform fewer than $(n-p)p\log p$ steps depending on the input.

A problem similar to the MinMax and MaxMin problems is to find the partition such that $\max_{j=0}^{p-1}\{\varphi_j\} - \min_{j=0}^{p-1}\{\varphi_j\}$ is minimum over all partitions. One could be lead to believe that it is possible to solve the MinMax problem and the MaxMin problem simultaneously thus giving an optimal solution. This however is not the case as the following example shows. Consider the sequence $\{3,3,1,5,2\}$ with $f(\sigma_l,...,\sigma_k) = \Sigma_{i=l}^{k}\sigma_i$. If we partition this sequence into $p = 3$ intervals the MinMax problem has the unique solution $f(3,3) = 6$, $f(1,5) = 6$ and $f(2) = 2$. While the MaxMin problem has unique solution $f(3) = 3$, $f(3) = 3$ and $f(1,5,2) = 8$. Thus we see that in general it is not possible to solve both problems at the same time.

An approximation to this problem can be found by first running the MinMax algorithm. Once a most expensive interval $[r_j, r_{j+1}]$ has been found, the MaxMin algorithm can be run to partition $[r_0, r_j]$ into j intervals and $[r_{j+1}, r_p]$ into $n-j-1$ intervals. This can also be done by running the MaxMin algorithm first followed by the MinMax algorithm.

Finally, we note that the MinMax problem can be generalized to higher dimensions than 1. In 2 dimensions the problem becomes how to partition a matrix A by drawing p_1 horizontal lines between the rows of A and p_2 vertical lines between the columns, such that the maximum value of some cost function f on each resulting rectangle is minimum. However, we do not know of any solutions to the partitioning problem for dimensions larger than one.

6 Acknowledgment

The authors thank Sven Olai Høyland for drawing their attention to the bisection approach in Section 2. They also thank Bengt Aspvall for constructive comments.

References

[1] S. Anily and A. Federgruen, *Structured partitioning problems*, Operations Research, 13 (1991), pp. 130–149.

[2] S. H. Bokhari, *Partitioning problems in parallel, pipelined, and distributed computing*, IEEE Trans. Comput., 37 (1988), pp. 48–57.

[3] M. R. Garey and D. S. Johnson, *Complexity results for multiprocessor scheduling under resource constraints*, SIAM J. Comput., 4 (1975), pp. 397–411.

[4] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Annals of Discrete Mathematics, 5 (1979), pp. 287–326.

[5] F. Halsall, *Data Communications, Computer Networks and OSI*, Addison Wesley, 1988.

[6] P. Hansen and K.-W. Lih, *Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing*, IEEE Trans. Comput., 41 (1992), pp. 769–771.

[7] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[8] F. Manne, *An algorithm for computing an elimination tree of minimum height for a tree*, Tech. Report CS-91-59, University of Bergen, Norway, 1992.

[9] V. Mehrmann, *Divide and conquer methods for block tridiagonal systems*, tech. report, Institut für Geometrie und Praktische Mathematik, RWTH Aachen, Germany, 1991.

[10] B. Olstad and F. Manne, *Efficient partitioning of sequences with an application to sparse matrix computations*, Tech. Report CS-93-83, University of Bergen, Norway, 1993.

[11] S. J. Wright, *Parallel algorithms for banded linear systems*, SIAM J. Sci. Statist. Comput., 12 (1991), pp. 824–842.