# Choosing Subsets with Maximum Weighted Average

David Eppstein* Daniel S. Hirschberg

Department of Information and Computer Science
University of California, Irvine, CA 92717

Tech. Report 95-12

March 17, 1995

**Abstract**

Given a set of $n$ real values, each with a positive weight, we wish to find the subset of $n - k$ values having maximum weighted average. This is equivalent to the following form of parametric selection: given $n$ objects with values decreasing linearly with time, find the time at which the $n - k$ maximum values add to zero. We give several algorithms showing that these problems can be solved in time $O(n)$ (independent of $k$). We also show that a slight generalization of the original problem, in which weights are allowed to be negative, is NP-complete.

*Work supported in part by NSF grant CCR-9258355 and by matching funds from Xerox Corp.

# 1 Introduction

A common policy in grading coursework allows students to drop a single homework score. The remaining scores are then combined in some kind of weighted average to determine the student's grade. We would like to be able to perform such calculations automatically. This problem has an easy linear time solution: simply try one by one each set of $n-1$ scores. The average for each set can be computed in constant time from the sums of all scores and of all weights.

However, consider the generalization of this problem in which not one but two scores may be dropped. More generally, of $n$ scores, suppose $k$ may be dropped. Now how can we maximize the weighted average of the remaining scores, more quickly than the naive $O(n^k)$ algorithm?

We formalize the problem as follows: We are given a set $S$ of $n$ *scores* $\langle v_i, w_i \rangle$, where $v_i$ denotes the *value* of a score and $w_i$ denotes the *weight* of the score. For later convenience we let $V$ and $W$ denote the sums of the values and weights respectively. We assume that all weights $w_i$ are positive. We wish to find a subset $T \subset S$, with $|T| = n - k$, maximizing the weighted average

$$A(T) = \frac{\sum_{i \in T} v_i}{\sum_{i \in T} w_i}. \tag{1}$$

In the coursework problem we started with, $v_i$ will typically be the grade received by the given student on a particular assignment, while $w_i$ will denote the number of points possible on that assignment.

Some similar problems have been studied before, but we were unable to find any reference to the minimum weighted average subset problem. Algorithms for finding a cycle or cut in a graph minimizing the mean of the edge costs [12, 13, 15, 27] have been used as part of algorithms for network flow [8, 11, 24] and cyclic staffing [17], however the averages used in these problems do not typically involve weights. More recently, Bern et al. [2] have investigated problems of finding the possible weighted averages of a point set in which the weight of each point may vary in a certain range. However that work does not consider averages of subsets of the points.

One intuitively appealing idea for solving the maximum weighted average problem is a greedy approach: find $i_1$ maximizing $A(S - i_1)$, then $i_2$ maximizing $A(S - i_1 - i_2)$, and so on. This would take total time $O(kn)$, and due to its simplicity would be suitable for practical implementations. Unfortunately, this technique does not work: consider the input $S = \{\langle 1,1 \rangle, \langle 0,2 \rangle, \langle 1,6 \rangle, \langle 1,6 \rangle\}$. The maximimum weight three-element

subset is $\{\langle 1,1\rangle, \langle 1,6\rangle, \langle 1,6\rangle\}$, dropping the score $i_1 = \langle 0,2\rangle$, but the maximum weight two-element subset $\{\langle 1,1\rangle, \langle 0,2\rangle\}$ does not drop $i_1$. The same input similarly serves as a counterexample for the "dual greedy" approach of selecting at each step a score to maximize the weighted average of the set selected so far.

Instead we give several algorithms for the problem, based on the following approaches. First, we show how to test for a particular candidate average $A$ whether some set $T$ has average $A$ or above, by a simple linear time algorithm. With this idea one can simply perform binary search for the true optimal value $A$ (in time $O(n \log W)$ if the inputs are integers). The ideas in this section also help us reformulate the problem as one of *parametric selection*: given $n$ objects with real values that decrease linearly as a function of time, choose the time at which the $n-k$ largest object values sum to zero. We describe two other general techniques for such problems, namely Newton iteration and Megiddo's parametric search method, and discuss why these techniques do not provide fully satisfactory solutions to our problem.

Second, we show that our problem falls into the class of *generalized linear programs*, defined by various computational geometers [1, 10, 21] for their applications to problems such as the minimum enclosing circle. A number of generalized linear programming algorithms are known to take time linear in the input size but exponential or subexponential in the *dimension* of the problem. For our weighted average problem, we show that the dimension is $k$, so for any fixed $k$ we can find in $O(n)$ time the set of $n-k$ scores with maximum weighted average. A version of Seidel's algorithm [25] provides a particularly simple randomized algorithm for the case $k = 2$, with expected running time approximately three times that of the algorithm for the easy case $k = 1$.

Third, we give another randomized algorithm, based on the linear time quickselect algorithm for finding the $k$ smallest values in a set. The idea is to solve the parametric selection problem defined above by simulating quickselect on the values of the objects as measured at the optimal time $t^*$ we are seeking. We do not have access to these values themselves but we can perform approximate comparisons on them by using our linear-time algorithm for testing a candidate average. Our method runs in $O(n)$ expected time for any $k$, however the constant factor improves for small $k$, so that this method is an improvement over Seidel's algorithm for $k \geq 4$.

Fourth, we show how to derandomize a variant of the third algorithm by applying a second technique from computational geometry, that of $\epsilon$-*cuttings*. The algorithm here is too complicated for practical use, but it

settles the theoretical question of the asymptotic complexity of the problem as well as demonstrating once again the usefulness of geometric ideas in seemingly non-geometric problems.

Finally, we show that if we generalize the problem somewhat, to allow negative weights, it becomes NP-complete, so no polynomial time solution is likely in this case.

## 2 Feasibility testing and parametric selection

Suppose that some $(n-k)$-element set $T \subset S$ has weighted average at least $A$. We can write this as an inequality of the form

$$A \leq A(T) = \frac{\sum_{i \in T} v_i}{\sum_{i \in T} w_i}. \tag{2}$$

Rearranging terms, and using the assumption that the $w_i$ are positive, we can rewrite this as

$$\sum_{i \in T} (v_i - A w_i) \geq 0. \tag{3}$$

Similar inequalities hold if we wish to know whether some $T$ has average strictly greater than $A$.

Define for each $i$ the decreasing linear function $f_i(t) = v_i - t w_i$, and define

$$F(t) = \max_{|T|=n-k} \sum_{i \in T} f_i(t). \tag{4}$$

Then $F(t)$ can be computed in linear time simply by selecting the $n-k$ largest (or equivalently $k$ smallest) values $f_i(t)$. Equation 3 can be rephrased as saying that some set has average at least $A$ iff $F(A) \geq 0$. We state this as a lemma:

**Lemma 1.** *For any value $A$, some set $T$ with $|T| = n-k$ has weighted average at least $A$ iff $F(A) \geq 0$, and some $T$ has weighted average greater than $A$ iff $F(A) > 0$. $F(A)$ can be computed in time $O(n)$.*

$F(A)$ is a piecewise linear decreasing function since it is the maximum of $\binom{n}{n-k}$ decreasing linear functions. Thus our original problem, of finding the maximum weighted average among all $n-k$ point sets, can be rephrased as one of searching for the root of $F(A)$. This can also be viewed as a parametric matroid optimization problem [7] in which the matroid in question is uniform.

```
Newton(S):
    choose an initial set T_0 ⊂ S
    i ← 0
    repeat
        let T_{i+1} be the set giving F(A(T_i))
        i ← i + 1
    until A(T_i) = A(T_{i-1})
    return A(T_i)
```

Figure 1. Newton iteration for the maximum weighted average.

By binary search we can find the position of a root of $F$ within an accuracy of $(1+\epsilon)$ in time $O(n \log 1/\epsilon)$. If the input values $v_i$ and $w_i$ are integers, the weighted average of any set is a rational number with denominator at most $W = \sum w_i$. Any two such numbers must differ by at least $1/W^2$, so if we find the optimal average $A^*$ within $O(1/W^2)$ error, any set $T$ with at least this average will be an optimal set and we can find the optimal average exactly. Thus we can solve our original weighted average problem in time $O(n \log W)$.

An alternate general method of finding the roots of functions is to use Newton iteration: start with an initial value $x_0$ and at each step compute $x_{i+1} = x_i - F'(x_i)/F(x_i)$. For this particular problem note that $F'(x)$ is the slope of the linear function $\sum_{i \in T} f_i(x)$, where $T$ is the set giving the value of $F(x)$. Thus this method can be implemented by the algorithm depicted in Figure 1. Each successive pair of values of $A(T)$ found by this algorithm is separated by a breakpoint of $F(T)$, and $F(T)$ can only have $O(n\sqrt{k}/\log^* k)$ breakpoints [23], so after polynomially many steps the algorithm terminates in the correct solution. (See e.g. [7] for the equivalence between the breakpoints of $F$ and the $k$-set problem studied in [23].) The worst case time of this algorithm may be larger than we wish, but it gives a simple method that may be useful in practice.

A third general method of solving our problem is provided by Megiddo's parametric search technique [22]. This method takes an algorithm for testing a parameter $A$, such as that of Lemma 1, and simulates the execution of that algorithm as it would run when given the optimal parameter $A*$ as input. At each step, any comparison involving $A*$ with some other value $\alpha$ is performed by running the same test algorithm on input $\alpha$. In general

this squares the running time (giving an $O(n^2)$ time algorithm in our case) but it can be made more efficient by simulating a *parallel* version of the test algorithm. (In our case what is needed is a parallel median-selection algorithm.) Typically all comparisons with $A*$ done in a single step of the parallel algorithm can be tested by binary search with $O(\log n)$ calls to the (sequential) test algorithm. A technique of Cole [4] further speeds up this idea, so that for our problem the resulting time bound should be $O(n \log n)$. However we find this unsatisfactory for two reasons. First, the time is still greater than we want. Second, and more importantly, the resulting algorithm is extremely complicated and not easily understood.

In subsequent sections we show how to use the feasibility testing function $F(A)$ with randomization and more direct forms of parametric search in order to improve the worst case or expected time to linear.

# 3 Generalized linear programming

Matoušek et al. [21] define a class of *generalized linear programming problems* that can be solved by a number of algorithms linear in the input size and exponential or subexponential in the *combinatorial dimension* of the problem. Their definition of a generalized linear program follows.

We assume we are given some $f$ taking as its argument subsets of some domain $S$ (in our case, subsets of the given set of scores), and mapping these sets to some totally ordered domain such as the real numbers. This function is required to satisfy the following two properties:

- If $A \subset B \subset S$, $f(A) \leq f(B)$.
- If $A \subset B \subset S$, $f(A) = f(B)$, and $s$ is any element of $S$, then $f(A+s) = f(A)$ iff $f(B+s) = f(B)$.

A *basis* of such a problem is a set $B \subset S$ such that for any proper subset $A \subset B$, $f(A) < f(B)$. The *dimension* of a problem is the maximum cardinality of a basis. The *solution* to the problem is a basis $B$ such that $f(B) = f(S)$.

In our problem, we let $S$ be the set of scores we are given. We define the objective function $f(A)$ as follows.

$$f(A) = \max_{B \subset A, |B|=k} A(S - B) \tag{5}$$

In other words, we consider a constrained version of our weighted average problem, in which the $k$ scores we drop are required to come from set $A$. If $|A| < k$, we define $f(A)$ to be a special value $(-\infty, |A|)$ less than any real number. The comparison between two such values $(-\infty, x)$ and $(-\infty, y)$ is defined to give the same result as the comparison between $x$ and $y$.

Any basis must consist of at most $k$ scores, so the dimension of this problem is $k$. To verify that this is a generalized linear program, we must prove that it satisfies the requirements above.

**Lemma 2.** *For the pair $(S, f)$ defined as above from our weighted average problem, any sets $A \subset B$ satisfy $f(A) \leq f(B)$.*

**Proof:** For $A$ and $B$ both having $k$ or more members, this follows immediately since the choices allowed in the maximization defining $f(B)$ are a superset of the choices allowed for $f(A)$. Otherwise, $f(A) = (-\infty, |A|)$; if $|B| \geq k$ then $f(B)$ is a real number greater than $f(A)$ by definition, and otherwise $f(B) = (-\infty, |B|) \geq f(A)$ since $|B| \geq |A|$. □

**Lemma 3.** *For the pair $(S, f)$ defined as above from our weighted average problem, any sets $A \subset B$ satisfying $f(A) = f(B)$, and any score $s = \langle v_i, w_i \rangle$, $f(A + s) = f(A)$ iff $f(B + s) = f(B)$.*

**Proof:** If $|A| < k$, then the assumption that $f(A) = f(B)$ forces $A$ and $B$ to be equal, and the lemma follows trivially. Otherwise, there must be some basis $C \subset A \subset B$ with $f(C) = f(A) = f(B)$, and $|C| = k$. Suppose that $C$ is non-optimal in $A + s$ or $B + s$. By an argument similar to that in Lemma 1, there is some other $k$-element set $D$ in $A + s$ or $B + s$ such that

$$\sum_{i \in S-D} f_i(A(S - C)) > 0. \tag{6}$$

Further, we know that

$$\sum_{i \in S-C} f_i(A(S - C)) = 0 \tag{7}$$

by the definition of $f_i$. Since the sum for $S - D$ is greater than the sum for $S - C$, there must be scores $i$, $j$ with $i \in D - C$, $j \in C - D$, and $f_i(A(S-C)) < f_j(A(S-C))$. Then $D' = C+i-j$ also satisfies inequality 6, showing that $D'$ is a better set to remove than $C$. We assumed that $C$ was optimal in $A$ and $B$, so it must be the case that $i = s$ and $D'$ shows that $C$ is non-optimal in both $A + s$ and $B + s$. Thus $C$ is optimal in $A + s$ iff it is optimal in $B + s$. □

**Corollary 1.** *The pair $(S, f)$ given above defines a generalized linear program, and the solution to the program correctly gives us the set of $k$ scores to drop in $S$ to maximize the weighted average of the remaining points.*

It is a curious phenomenon that although the generalized linear program class was defined to study *minimization* problems (such as finding the minimum radius of a circle containing a set of $n$ input points) the results above show that the same definition applies without reversing inequalities to our maximization problem.

As noted by Matoušek et al., many previously known linear programming algorithms can be adapted for solving generalized linear programs in time linear in $n$ and exponential or subexponential in $k$ [3, 21, 25]. For our purposes, such algorithms will be only interesting when $k$ is very small, and then only if the algorithm itself is relatively simple. Therefore the exponential algorithm of Seidel is preferable to the more complicated subexponential algorithms of Matoušek et al.

The procedure in Figure 2 is a modification of that of Seidel for our problem. As shown, we return a pair of values: the maximum weighted average $a$ of the set, and the value

$$x = \max_{\langle v_i, w_i \rangle \in B} f_i(a) \tag{8}$$

where $B$ is the basis of scores dropped from $S$ giving $a = A(S - B)$. If the actual basis $B$ is desired as output, it can be stored whenever we compute a new value of $a$.

Note that the recursive calls operate only on the subsequence of $S$ from 1 to $i - 1$. Recursive calls may permute this subsequence in place, rather than making a new copy of it, since those scores will only be revisited in other recursive calls which will re-permute them (in particular the set $S$ can be stored in a single array, pointers to which are used by all recursive calls).

**Theorem 1.** *Seidel's algorithm, modified as described above to solve our maximum weighted average problem, returns the optimal subset in expected time $O(k!\, n)$.*

**Proof:** We prove by induction that after each step $i$ in the loop, $a$ is the maximum of the values $A(S - B)$ for sets $B \subset S$ with $|B| = k$ such that $B$ contains only scores with indices in the range from 1 to $i$. The base case is trivial. At each successive step, if any set improves on the previous value of

```
Seidel(S, n, k, V, W):
    randomly permute scores 1 through n
    a ← (V − Σ_{1≤i≤k} v_i)/(W − Σ_{1≤i≤k} w_i)
    x ← max_{1≤i≤k} f_i(a)
    for i ← k + 1 to n do
        if f_i(a) < x then
            (a, x) ← Seidel(S, i − 1, k − 1, V − v_i, W − w_i)
            x ← max{x, f_i(a)}
    return (a, x)
```

Figure 2. Seidel's generalized linear programming algorithm, modified to solve the maximum weighted average problem.

$B$, it will be a set involving index $i$ by Lemma 3. A closer examination of that lemma shows that we can test $i$ by the comparison used in the **if** statement above. Then if there is an improvement to be made in $a$, it will be found by the recursive call which (by induction) finds the maximum $A(S - B)$ among sets $B$ meeting the conditions above and containing $\langle v_i, w_i \rangle$.

Thus each execution of the **if** statement in the loop gives rise to a recursive call if and only if every optimal basis for scores 1 through $i$ contains score $i$ itself. We consider the conditional probability that this is the case, subject to the condition that we know the first $i$ scores but not their permutation. Then because the initial permutation of the whole input sequence was randomly selected, any permutation of the given prefix of the sequence is also equally likely. Let $X$ denote the set of scores that belong to all optimal bases of the first $i$ scores. $|X| \leq k$ (since it is a subset of a base), so the conditional probability that $i$ belongs to $X$ is at most $k/n$. Therefore the **if** statement succeeds with at most this probability, and the expected time for the algorithm satisfies the recurrence

$$T(n,k) \leq O(n) + \sum_i \frac{k}{i} T(i, k-1), \tag{9}$$

which is easily shown to solve to $O(k!\, n)$. □

To compare the time algorithm more carefully with the obvious one for $k = 1$, let us count only the number of multiplications and divisions performed. In all the algorithms we consider, this will be proportional to the total amount of work. In the $k = 1$ algorithm, this number is $n$. In the

```
Seidel2(S, V, W):
    randomly permute scores 1 through n
    a ← (V − v1 − v2)/(W − w1 − w2)
    x ← max{f1(a), f2(a)}
    for i ← 3 to n do
        if fi(a) < x then
            find j < i maximizing (V − vi − vj)/(W − wi − wj)
            a ← (V − vi − vj)/(W − wi − wj)
            x ← max{fi(a), fj(a)}
    return a
```

Figure 3. Seidel's algorithm, special case for $k = 2$.

algorithm above, the expected number of multiplicative operations satisfies a recurrence $M(k,n) = kM(k-1,n)+n+o(n)$. In particular, this is always less than $2k!\,n$.

The version of this algorithm for $k = 2$ can be made particularly simple by replacing the recursive call with a loop implementing the usual algorithm for $k = 1$, as shown in Figure 3. For this version we return only $a$ and not $x$.

If we are computing the solutions for a sequence of problems having the same value of $n$ (such as the grades of a group of students in the same class) we can save some more time, without compromising the expected case analysis, by initially choosing one random permutation to use for all problems.

For $k = 2$, the expected number of multiplicative operations performed in either version of Seidel's algorithm is at most $3n + O(\log n)$. Thus we should expect the total time for this algorithm to be roughly three times that for the $k = 1$ algorithm.

## 4 Connection with computational geometry

Before describing more algorithms, we first develop some geometric intuition that will help them make more sense.

Recall that we have defined $n$ linear functions $f_i(A)$, one per score. We are trying to find the maximum weighted average $A^*$, and from that value it is easy to find the optimal set of scores to drop, simply as the set giving the $k$ minimum values of $f_i(A^*)$.

Consider drawing the graphs of the functions $y = f_i(x)$. This produces an arrangement of $n$ non-vertical lines in the $xy$ plane. The value $f_i(A^*)$ we are interested in is just the $y$-coordinate of the point where line $f_i$ crosses the vertical line $x = A^*$. We do not care so much about the exact coordinates of this crossing—we are more interested in its order relative to the similar crossing points for other lines $f_j$, as this relative ordering tells us which of $i$ or $j$ is preferable to drop from our set of scores.

By performing tests on different values of $A$, using Lemma 1, we can narrow the range in which the vertical line $x = A^*$ can lie to a narrower and narrower vertical slab, having as left and right boundaries some vertical lines $x = A_L$ and $x = A_R$. $A_L$ is the maximum value for which we have computed that $F(A_L) > 0$, and $A_R$ is the minimum value for which we have computed that $F(A_R) < 0$. If we use the algorithm of Lemma 1 on some value $A_L < A < A_R$, we can determine the relative ordering of $A^*$ and $A$; this results in cutting the slab into two smaller slabs bounded by the line $x = A$ and keeping only one of the two smaller slabs. For instance, the binary search algorithm we started with would simply select each successive partition to be the one in which the two smaller slabs have equal widths.

Any two lines $f_i$ and $f_j$ have a crossing $p_{ij}$ unless they have the same slope, which happens when $w_i = w_j$. If it happens that $p_{ij}$ falls outside the slab $[A_L, A_R]$, we can determine immediately the relative ordering of $f_i(A^*)$ and $f_j(A^*)$, as one of the two lines must be above the other one for the full width of the slab. If the two lines have the same slope, one is above the other for the entire plane and a fortiori for the width of the slab. We can express this symbolically as follows.

**Lemma 4.** *If $A_L \leq A^* \leq A_R$, $f_i(A_L) \geq f_j(A_L)$, and $f_i(A_R) \geq f_j(A_R)$, then $f_i(A^*) \geq f_j(A^*)$.*

In this formulation, the lemma can be proven simply by unwinding the definitions and performing some algebraic manipulation.

Define $A(i,j)$ to be the $x$-coordinate of the crossing point $p_{ij}$. If $f_i$ and $f_j$ have the same slope, $A(i,j)$ is undefined. If we use Lemma 1 to test the relative ordering of $A^*$ and $A(i,j)$, the resulting slab $[A_L, A_R]$ will not contain $p_{i,j}$ and so by the result above we can determine the relative ordering of $f_i(A^*)$ and $f_j(A^*)$. Symbolically $A(i,j) = (v_i - v_j)/(w_i - w_j)$, and we have the following lemma.

**Lemma 5.** *If $A(i,j) \leq A^*$ and $w_i \geq w_j$, or if $A(i,j) \geq A^*$ and $w_i \leq w_j$, then $f_i(A^*) \leq f_j(A^*)$.*

Again the lemma can be proven purely algebraically.

The algorithms below can be interpreted as constructing slabs containing fewer and fewer crossing points $p_{ij}$, until we know enough of the relative orderings of the values $f_i(A^*)$ to select the smallest $k$ such values. This will then in turn give us the optimal set of scores to drop, from which we can compute the desired maximum weighted average as the weighted average of the remaining scores.

For instance, one method of solving our problem would be to use binary search or a selection algorithm among the different values of $A(i,j)$. Once we know the two such values between which $A^*$ lies, all relative orderings among the $f_i(A^*)$ are completely determined and we can apply any linear time selection algorithm that uses only binary comparisons. (Each such comparison can be replaced with an application of Lemma 5.) However there are more values $A(i,j)$ than we wish to spend the time to examine. Instead we use more careful approaches that can eliminate some scores as belonging either to the set of $k$ dropped scores or the remaining set of $n-k$ scores, without first having to know their relative order compared to all other scores.

We note that similar methods have been applied before, to the geometric problem of selecting from a collection of $n$ points the pair giving the line with the median slope [5, 6, 18, 19, 26]. A geometric duality transformation can be used to transform that problem to the one of selecting the median $x$-coordinate among the intersection points of $n$ lines, which can then be solved by similar techniques to those above, of finding narrower and narrower vertical slabs until no points are left in the slab. The algorithm is dominated by the time to test whether a given $x$-coordinate is to the right or left of the goal, which can be done in time $O(n \log n)$. In our weighted average problem, the faster testing procedure of Lemma 1 and the ability to eliminate some scores before all pairwise relations are determined allow us to solve the overall problem in linear time.

# 5 Randomized linear time

We now describe a randomized algorithm which finds the subset with maximum weighted average in linear time, independent of $k$. The algorithm is more complicated than the ones we have described so far but should improve on e.g. Seidel's algorithm for values of $k$ greater than three.

The idea behind the algorithm is as follows. If we choose a random

member $i$ of our set of scores, and let $A^*$ denote the optimal average we are seeking, the position of $f_i(A^*)$ will be uniformly distributed relative to the positions of the other $f_j(A^*)$. For instance $i$ would have a $k/n$ chance of being in the optimal subset of scores to be removed. If we know that it is in this optimal subset, we could remove from our input those $j$ with $f_j(A^*) < f_i(A^*)$ and update $k$ accordingly. Conversely, if we know that score $i$ has to be included in the set giving the maximum weighted average, we would know the same about all $j$ with $f_j(A^*) > f_i(A^*)$ and we could collapse all such scores to their sum. In expectation we could thus reduce the input size by a constant fraction—the worst case would be when $k = n/2$, for which the expected size of the remaining input would be $3n/4$.

To compute the position of $f_i(A^*)$, and to find the scores to be removed or collapsed as described above, would require knowing the relative ordering of $f_i(A^*)$ with respect to all other values $f_j(A^*)$. For any $j$ we could test this ordering in time $O(n)$ by computing $F(A(i,j))$ as described in Lemma 5. We could compute all such comparisons by binary searching for $A^*$ among the values $A(i,j)$ in time $O(n \log n)$, but this is more time than we wish to take. The solution is to only carry out this binary search for a limited number of steps, giving the position of $f_i(A^*)$ relative to most but not all values $f_j(A^*)$. Then with reasonably high probability we can still determine whether or not score $i$ is to be included in the optimal set, and if this determination is possible we will still expect to eliminate a reasonably large fraction of the input.

We make these ideas more precise in the algorithm depicted in Figure 4. Much of the complication in this algorithm is due to the need to deal with special cases, and to the expansion of previously defined values such as $F(A)$ into the pseudo-code needed to compute them.

Let us briefly explain some of the notation used in our algorithm. $F(A)$ and $f_j(A)$ were defined earlier. Let $A^*$ denote the optimal average we seek; then as noted earlier the optimal subset is found by choosing the scores with the $n-k$ largest values of $f_j(A^*)$. $A(i,j)$ is defined as the "crossover point" for which $f_i(A(i,j)) = f_j(A(i,j))$. We initially let $[A_L, A_R]$ give bounds on the range in which $A^*$ can lie. (It would be equally correct to set $A_L = -\infty$ but the given choice $A_L = V/W$ allows some savings in that some crossover points $A(i,j)$ are eliminated from the range.) Note that $A_L \leq A^*$ since removing scores can only increase the overall weighted average.

In order to compare $f_i$ and $f_j$ at other values of $A$, we define $\sigma(i,j)$ as a value that is positive or negative if for $A > A(i,j)$, $f_i(A) < f_j(A)$ or $f_i(A) > f_j(A)$ respectively. We usually compute $\sigma(i,j)$ by comparing $w_i$ and

```
Random(S, k):
    A_L ← V/W
    A_R ← +∞
    while |S| > 1 do
        choose i randomly from S
        for ⟨v_j, w_j⟩ ∈ S do
            if w_i = w_j then
                σ(i,j) ← v_j − v_i
                A(i,j) ← −∞
            else
                σ(i,j) ← w_i − w_j
                A(i,j) ← (v_i − v_j)/(w_i − w_j)
        E ← {⟨v_j, w_j⟩ | σ(i,j) = 0}
        X ← {⟨v_j, w_j⟩ | A(i,j) ≤ A_L and σ(i,j) > 0}
            ∪{⟨v_j, w_j⟩ | A(i,j) ≥ A_R and σ(i,j) < 0}
        Y ← {⟨v_j, w_j⟩ | A(i,j) ≤ A_L and σ(i,j) < 0}
            ∪{⟨v_j, w_j⟩ | A(i,j) ≥ A_R and σ(i,j) > 0}
        Z ← S − X − Y − E
        repeat
            A ← median{A(i,j) | ⟨v_j, w_j⟩ ∈ Z}
            for ⟨v_j, w_j⟩ ∈ S do f_j(A) ← v_j − A w_j
            F(A) ← ∑(the largest |S| − k values of f_j(A))
            if F(A) = 0 then return A
            else if F(A) > 0 then A_L = A
            else A_R = A
            recompute X, Y, and Z
            if |X| + |E| ≥ |S| − k then
                remove max(|E|, |X| + |E| + k − |S|)
                    members of E from S
                remove Y from S
                k ← k − (number of removed scores)
            else if |Y| + |E| ≥ k then
                collapse max(|E|, |Y| + |E| − k) members of E
                    and all of X into a single score
        until |Z| ≤ n/32
    return v_1/w_1
```

Figure 4. Randomized algorithm for the maximum weighted average.

$w_j$, but we use a different definition if the weights are equal (corresponding to the geometric situation in which lines $y = f_i(x)$ and $y = f_j(x)$ are parallel).

Then set $E$ consists exactly of those scores that have the same value and weight as the random selection $\langle v_i, w_i \rangle$. Set $X$ consists of those scores for which $f_j(A_L) \geq f_i(A_L)$ and $f_j(A_R) \geq f_i(A_R)$, with one inequality strict. In other words these are the scores for which $f_j(A^*)$ is known by Lemma 4 to be greater than $f_i(A^*)$. Similarly $Y$ consists of those scores for which we know that $f_j(A^*) < f_i(A^*)$. Set $Z$ consists of those scores for which the relation between $f_i(A^*)$ and $f_j(A^*)$ is unknown.

If $Z$ were empty, we would know whether score $i$ itself should be included or excluded from the optimal subset, so we could simplify the problem by also removing all of either $X$ or $Y$. The purpose of the inner loop of the algorithm is to split the range $[A_L, A_R]$ in a way that shrinks $Z$ by a factor of two, so that this simplification becomes more likely.

In order to compare this algorithm to others including Seidel's generalized linear programming algorithm, we analyze the time in terms of the number of multiplicative operations. It should be clear that the time spent on other operations is proportional to this.

In what follows, we make the simplifying assumption that $E$ contains only score $i$. This is without loss of generality, as the expected time can only decrease if $E$ has other scores in it, for two reasons. First, $Z$ can initially have size at most $|S - E|$. Second, the fact that scores in $E$ are equivalent to score $i$ lets us treat them either as part of $X$ or as part of $Y$, whichever possibility allows us to remove more scores from our problem.

**Lemma 6.** *Let $n$ denote the size of $S$ at the start of an iteration of the outer loop of the algorithm. The expected number of scores removed or collapsed in that iteration is at least $49n/256$.*

**Proof:** Let $p$ be the position of $f_i(A^*) = v_i - A^* w_i$ in the sorted sequence of such values. Then $p$ is uniformly distributed from 1 to $n$, so with probability at least $7/8$, $p$ differs by at least $n/32$ from 1, $k$, and $n$. Consider the case that $p - k \geq n/32$. Then by the end of the inner loop, we will have at least $k$ scores in $Y$ and can collapse anything placed in $X$ during the loop, removing at least $n - p - n/32$ scores overall. Similarly if $k - p \geq n/32$ we will have at least $n - k$ scores in $X$ and can remove at least $p - n/32$ scores. The worst case happens when $k = n/2$, when the expected size of $X$ (in the first case) or $Y$ (in the second case) is $7n/32$. Thus we get a total expectation of $49n/256$. □

**Lemma 7.** *In a single iteration of the outer loop above, the expected number of multiplicative operations performed is at most* $371n/64$.

**Proof:** Let $n$ denote the size of $S$ at the start of the outer loop. Note that the only multiplicative operations are $n$ divisions in the computation of $A(i,j)$ in the outer loop, and $|S|$ multiplications in the computation of $F(\imath A)$ in the inner loop. The inner loop is executed at most five times per outer loop, so the worst case number of operations per iteration of the outer loop is $6n$.

To reduce this bound we consider the size of $S$ in each iteration of the inner loop. The analysis of the expected size of $S$ in each iteration is very similar to that in Lemma 6, with the $n/32$ bound on $Z$ replaced by the values $n$, $n/2$, $n/4$, $n/8$, and $n/16$. For the first three of these, we can prove no bound better than $n$ on the expected value of $|S|$. For the iteration in which $|Z| \leq n/8$, we have probability $1/2$ that $p$ has distance $n/8$ from 1, $k$, and $n$, and when $p$ is in this range we can expect to remove $n/8$ values, so the overall expected size of $S$ in the next iteration is $15n/16$. And for the iteration in which $|Z| \leq n/16$, we have probability $3/4$ that $p$ has distance $n/16$ from 1, $k$, and $n$, and when $p$ is in this range we can expect to remove $3n/16$ values in iterations up through this one, so the expected size of $S$ in the last iteration is $55n/64$. Adding these expectations gives the overall bound. □

**Theorem 2.** *The total expected number of multiplicative operations performed in the algorithm above is at most* $1484n/49 + O(1) \approx 30.3\,n$, *and the total expected time is* $O(n)$.

**Proof:** The time can be expressed as a random variable which satisfies a probabilistic recurrence

$$T(S) \leq 371|S|/64 + T(R) \tag{10}$$

where $R$ is a random variable with expected size $(1 - 49/256)|S|$. By the theory of probabilistic recurrences [16], the expected value of $T(S)$ can be found using the deterministic recurrence

$$T(n) = 371n/64 + T((1 - 49/256)n) \tag{11}$$

which solves to the formula given in the theorem. □

Although the constant factor in the analysis of this algorithm is somewhat disappointingly large, we believe this algorithm should be a reasonable choice in practice for several reasons. First, the bulk of the time in the algorithm is spent in the computations of $F(A)$, since other operations in the inner loop depend only on the size of $Z$, which is rapidly shrinking. Thus there is little overhead beyond the multiplicative operations counted above. Second, Lemma 7 assumes that initially $Z = S$, however due to bounds on $[A_L, A_R]$ from previous iterations of the outer loop, $Z$ may actually be much smaller than $S$. Third, the analysis in Lemma 6 assumed a pathologically bad distribution of the positions of $f_j(A^*)$ for $j \in Z$: it assumed that for $p$ close to $k$ these positions would always be between $p$ and $k$, while for $p$ far from $k$ these positions would always be on the far side of $p$ from $k$. In practice the distribution of positions in $Z$ is likely to be much more balanced, and the number of scores removed will be correspondingly greater. Fourth, for the application to grading, many weights are likely to be equal, which helps us in that there are correspondingly fewer values of $A(i,j)$ in the range $[A_L, A_R]$ and fewer multiplications spent computing $A(i,j)$. Fifth, the worst case for the algorithm occurs for $k = n/2$, but in the case of interest to us $k$ is a constant. For small $k$ the number of operations can be greatly reduced, as follows.

**Theorem 3.** *Let $k$ be a fixed constant, and consider the variant of the algorithm above that stops when $|Z| < n/16$. Then expected number of multiplicative operations used by this variant is at most $580n/49 + O(k) \approx 11.8n$.*

**Proof:** We mimic the analysis above. With probability $7/8$, $n/16 \leq p \leq 15n/16$. For such $p$, the expected number of scores removed is $7n/16$. Therefore the expected number of scores left after an iteration of the outer loop is $(1 - (7/8)(7/16))n = (1 - 49/128)n = 79n/128$. The same sort of formula also tells us how many scores are expected to be left after each iteration of the inner loop. As long as $|Z| \geq n/2$ we can't expect to have removed any scores, so the first two iterations have $n$ expected operations each. In the third iteration, $|Z| \leq n/4$, and with probability $1/2$ $n/4 \leq p \leq 3n/4$. For $p$ in that range, we would expect to have removed $n/4$ of the scores. Therefore in the third iteration we expect to have $(1-(1/2)(1/4))n = (1-1/8)n = 7n/8$ operations. Similarly in the fourth iteration we expect to have $(1 - (3/4)(3/8))n = 21n/32$ operations. We can therefore express our

expected number of operations as a recurrence

$$T(n) = 145n/32 + T(79n/128) \tag{12}$$

with the base case that if $n = O(k)$ the time is $O(k)$. The solution to the recurrence is given by the formula in the theorem. □

This method is already better than Seidel's when $k = 4$, and continues to get better for larger $k$, since the only dependence on $k$ is an additive $O(k)$ term in place of Seidel's multiplicative $O(k!)$ factor.

# 6 Deterministic linear time

Much recent work in theoretical computer science has focused on the difference between randomized and deterministic computation. From this work, we know many methods of *derandomization*, that can be used to transform an efficient randomized algorithm (such as our linear time algorithm described above) into a deterministic algorithm (sometimes with some loss of efficiency).

In our case, we have an algorithm that selects a random sample (a score $\langle v_i, w_i \rangle$) from our input, and eliminates some other scores by using the fact that the position of $f_i(A^*)$ in the list of all $n$ such values is likely to be reasonably well separated from 1, $k$, and $n$. We would like to find a similar algorithm that chooses this sample *deterministically*, so that it has similar properties and can be found quickly. Since the randomized algorithm is already quite satisfactory from a practical point of view, this derandomization process mainly has significance as a purely theoretical exercise, so we will not worry about the exact dependence on $n$ (e.g. as measured previously in terms of the number of multiplicative operations); instead we will be satisfied with any algorithm that solves our problem in time $O(n)$.

We return to the geometric viewpoint: graph the linear functions $y = f_i(x)$ to form a line arrangement in the $xy$-plane, which is cut into slabs (represented as intervals of $x$-coordinates) by our choices of values to test against $A^*$. We are trying to reduce the problem to a slab $[A_L, A_R]$ containing few crossings of the lines; this corresponds to having many pairs of scores for which we know the preferable ones to drop.

Many similar problems of derandomization in computational geometry have been solved by the technique of *$\epsilon$-cuttings*, and we use the same approach here.

A *cutting* is just a partition of the plane into triangles. If we are given a set of $n$ lines $y = f_i(x)$ in the $xy$-plane, an $\epsilon$-cutting for those lines is a cutting for which the interior of each triangle is crossed by a small number of the lines, at most $\epsilon n$ of them. For instance, if we start with the arrangement of all lines, and add diagonals to form a triangulation, we would get a cutting for which each triangle is crossed by no lines. Therefore it would be an $\epsilon$-cutting for any $\epsilon \geq 0$, however it would have far too high a complexity for our purpose.

Matoušek [20] showed that an $\epsilon$-cutting involving $O(1/\epsilon^2)$ triangles can be computed deterministically in time $O(n/\epsilon)$, as long as $1/\epsilon < n^{1-\delta}$ for a certain $\delta$. We will be choosing $\epsilon$ to be some fixed constant $(1/8)$, so the resulting cutting has $O(1)$ triangles and can be computed in linear time.

The idea of our algorithm is as follows. We first compute an $\epsilon$-cutting for the set of lines $y = f_i(x)$ (that is, a certain triangulation of the $xy$-plane). By binary search, we can restrict the optimal value $A^*$ we are seeking to lie in a range $[A_L, A_R]$ that does not contain the $x$-coordinate of any triangle vertex. Therefore if we consider the vertical slab $A_L \leq x \leq A_R$, the edges of triangles in the cutting either cross the slab or are disjoint from it. If a triangle crosses the slab, at most two of its three sides do so, and the top and bottom boundary lines of the triangle are well defined.

For each edge that crosses the slab, we consider the line $y = ax+b$ formed by extending that edge, and pretend that it is of the form $y = f_i(x)$ for some pair (which must be the pair $\langle b, -a \rangle$). We then use this pair to eliminate scores from $S$ similarly to the way the previous randomized algorithm used the pair $\langle v_i, w_i \rangle$. It turns out not to be important that $\langle b, -a \rangle$ might not be in $S$. (If it is absent from $S$ we have fewer special cases to deal with. However if the input includes many copies of the same score, it may be necessary for $\langle b, -a \rangle$ to be in $S$.)

Thus for each pair $\langle b, -a \rangle$ found in this way we compute sets $X$, $Y$, and $Z$ as before and eliminate either $X$ or $Y$ if the other of the two is large enough. Unlike the previous randomized algorithm, we do not need an inner loop to reduce the size of $Z$. Instead we use the definition of our $\epsilon$-cutting to prove that $|Z| \leq \epsilon n$. Further, for two edges bounding the top and bottom of the same triangle, the corresponding sizes of $X$ and $Y$ differ from one edge to the other by at most $\epsilon n$. Therefore at least one of the edges from the $\epsilon$-cutting, when used as a sample in this way, allows us to eliminate a constant fraction of the input.

These ideas are made more specific in Figure 5. Much of the code and notation is similar to that of our previous randomized algorithm, so we do

```
Deterministic(S, k):
    A_L ← V/W
    A_R ← +∞
    while |S| > 1 do
        compute a 1/8-cutting C of the lines y = v_i − x w_i
        T ← {vertices (x, y) in C with A_L < x < A_R}
        while |T| > 0 do
            A ← median{x | (x, y) ∈ T}
            compute F(A) and update A_L, A_R, and T
        for each triangle Δ_i ∈ C crossing slab A_L < x < A_R
            find the upper boundary line y = ax + b of Δ_i
            for ⟨v_j, w_j⟩ ∈ S do
                if w_i = −a then
                    σ(j) ← v_j − b
                    A(j) ← −∞
                else
                    σ(j) ← −a − w_j
                    A(j) ← (b − v_j)/(−a − w_j)
            E ← {⟨v_j, w_j⟩ | σ(j) = 0}
            X ← {⟨v_j, w_j⟩ | A(j) ≤ A_L and σ(j) > 0}
                ∪{⟨v_j, w_j⟩ | A(j) ≥ A_R and σ(j) < 0}
            Y ← {⟨v_j, w_j⟩ | A(j) ≤ A_L and σ(j) < 0}
                ∪{⟨v_j, w_j⟩ | A(j) ≥ A_R and σ(j) > 0}
            Z ← S − X − Y − E
            if |X| + |E| ≥ |S| − k then
                remove max(|E|, |X| + |E| + k − |S|)
                    members of E from S
                remove Y from S
                k ← k − (number of removed scores)
            else if |Y| + |E| ≥ k then
                collapse max(|E|, |Y| + |E| − k) members of E
                    and all of X into a single score
    return v_1/w_1
```

Figure 5. Deterministic linear time algorithm.

not explain it again in detail here.

**Lemma 8.** *For each boundary line $y = ax + b$ considered in the algorithm above, the set $Z$ computed in the algorithm has size at most $|Z| \leq n/8$.*

**Proof:** If a score $\langle v_j, w_j \rangle$ is in $Z$ then, by the definition of sets $X$, $Y$, and $Z$, it must be the case that the two lines $y = ax + b$ and $y = f_j(x)$ have their crossing point within the slab $A_L < x < A_R$. Thus in particular line $y = f_j(x)$ crosses triangle $\Delta_i$ near that crossing point. By the definition of an $\epsilon$-cutting, at most $\epsilon n = n/8$ lines can cross this triangle. □

**Lemma 9.** *Let line $y = ax + b$ be the top boundary line of a triangle $\Delta_i \in C$, and let $y = cx + d$ be the bottom boundary line of $\Delta_i$ (and hence the top boundary of a different triangle.) Then the sets $X_1$ computed for $y = ax + b$ and $X_2$ computed for $y = cx + d$ differ in size by at most $n/8$, and the same is true for the sets $Y$ computed for each line.*

**Proof:** Every score $\langle v_j, w_j \rangle$ in $X_1$ must also be in $X_2$. (Geometrically, this is trivial: if the line $y = f_j(x)$ passes above the top boundary of $\Delta_i$, it certainly passes above the bottom boundary.) If a score $\langle v_j, w_j \rangle$ is in $X_2$ is not in $X_1$, it must be the case that the line $y = f_j(x)$ crosses triangle $\Delta_i$. (Geometrically, if the line $y = f_j(x)$ passes above the bottom boundary of $\Delta_i$, but does not pass above the top boundary, it must either cross the top boundary or stay between the two, and in either case a portion of the line is contained in $\Delta_i$.) As in the previous lemma the definition of the $\epsilon$-cutting bounds the number of lines crossing $\Delta_i$ by $\epsilon n$. The same argument applies by symmetry to $Y_1$ and $Y_2$. □

**Lemma 10.** *Each iteration of the algorithm removes at least half the scores from $S$.*

**Proof:** Consider the sequence of lines $y = a_i x + b_i$ in order from bottom to top. As we proceed, the sizes of the sets $Y_i$ computed for each line increase by at most $n/8$ at each step, so there must be a step at which $k \leq |Y_i| \leq k + n/8$. At this step $X_i$ will be collapsed so that at most $k + n/4$ scores can remain, of which at most $n/4$ can be in the eventual optimal solution. Similarly if we proceed from top to bottom we can find a step at which $(n - k) \leq |X_i| \leq (n - k + n/8)$, at which point we will remove $Y_i$ and at most $n/4$ of the remaining scores can be omitted in our eventual optimal solution. Therefore overall there can be at most $n/2$ scores left. □

**Theorem 4.** *The algorithm described in Figure 5 returns the maximum weighted average among all $(n-k)$ element subsets of $S$, in time $O(n)$.*

**Proof:** Correctness follows from the same considerations used in our randomized algorithm for the same problem. Each iteration of the algorithm takes linear time, and removes a constrant fraction of the input. Therefore the times for all iterations add in a geometric series to $O(n)$. □

## 7 Negative weights

The methods above do not make any assumption about the values $v_i$, however the non-negativity of the weights $w_i$ was used in Lemma 1 and therefore in all our algorithms.

It is natural to consider a slight generalization of the problem, in which we allow the weights $w_i$ to be negative. The weighted average $V/W$ of a set $S$ is still well defined (as long as we make some consistent choice of what to do when $W = 0$) so it makes sense to seek the $(n-k)$ element subset maximizing this quantity. We can define $F(A)$ as before, and as before we get a convex piecewise linear function. Unlike the previous situation, however, there might be either zero or two values of $A$ for which $F(A) = 0$. In the situation in which $F(A)$ has two roots, it turns out that our problem can be solved by finding the larger of these two, by minor modifications of the algorithms we showed before. However it is not so obvious what to do when $F(A)$ has no roots.

Unfortunately, this problem turns out to be NP-complete, as we now show. We use a reduction from the following standard problem.

**Subset Sum.** Given a collection $S$ of positive integers $s_i$, and a value $t$, is there a subset of $S$ with sum exactly $t$?

As shown by Karp [9, 14], subset sum is NP-complete, although it can be solved in pseudo-polynomial time $O(nt)$ by a dynamic programming technique [9].

**Theorem 5.** *It is NP-complete to find the $(n-k)$ element subset of a collection of scores $\langle v_i, w_i \rangle$ maximizing the weighted average $\sum v_i / \sum w_i$, if one or more of the weights can be negative.*

**Proof:** Given an instance $(S, t)$ of the subset sum problem, we transform $S$ to the set of scores $\langle 1, 2s_i \rangle$. We also include $n$ additional "dummy" scores $\langle 1, 0 \rangle$ and one final score $\langle 1, 1 - 2t \rangle$. We then ask for the set of $n + 1$ scores maximizing the weighted average. We claim that the maximum possible average is $n + 1$ exactly when the subset sum problem is solvable.

Note that for any set containing $\langle 1, 1 - 2t \rangle$, $\sum w_i$ is odd (so nonzero). For any other set of $n + 1$ values there must be at least one score $\langle 1, s_i \rangle$, and the sum is positive (so nonzero). Therefore we need not worry about what to do when $\sum w_i$ is zero. Also note that any sum of weights must be an integer, so all weighted averages of subsets of the input are of the form $(n + 1)/x$ for some integer $x$.

Since all values $v_i$ are equal, this weighted average is maximized by finding a set for which $\sum w_i$ is positive and as small as possible. If some set $A \subset S$ has sum exactly $t$, we construct a set of scores by including all pairs $\langle 1, s_i \rangle$ for $s_i$ in $A$, together with $\langle 1, 1 - 2t \rangle$ and enough dummy scores to make the set have $n+1$ scores total. This set of scores then has $\sum w_i = 1$ and weighted average $n + 1$.

Conversely, suppose some set of scores has weighted average $n+1$. Therefore $\sum w_i = 1$. Since this sum is odd, it must include the pair $\langle 1, 1 - 2t \rangle$. Then the remaining weights must sum to $2t$, and the non-dummy scores from this set can be used to construct a set $A \subset S$ with sum exactly $t$. □

If weights of zero are not allowed, the dummy scores can be replaced by $\langle 1, \epsilon \rangle$ for any sufficiently small $\epsilon$.

Finally, we note that our original maximum weighted average problem can, like subset sum, be solved in pseudo-polynomial time. Suppose all the $v_i$ and $w_i$ are integers. Let $W_{\min}$ and $W_{\max}$ denote the largest and smallest possible sum of weights in any subset of our scores. We use a dynamic program to determine, for each possible sum of weights $w$ in the range $[W_{\min}, W_{\max}]$, the minimum and maximum sum of values $V_{\min}(w)$ and $V_{\max}(w)$ among all sets having that sum of weights. The maximum weighted average can then be found by comparing $V_{\min}(w)/w$ for negative $w$ with $V_{\max}(w)/w$ for positive $w$. The overall algorithm takes time $O(n(W_{\max} - W_{\min}))$.

## References


[1] N. Amenta. Helly theorems and generalized linear programming. *9th ACM Symp. Comp. Geom.* (1993) 63–72.

[2] M. Bern, D. Eppstein, L. Guibas, J. Hershberger, S. Suri, and J. Wolter. The centroid of points with approximate weights. Manuscript, 1995.

[3] K. L. Clarkson. A Las Vegas algorithm for linear programming when the dimension is small. *29th IEEE Symp. Foundations of Computer Science* (1988) 452–456.

[4] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM* 31 (1984) 200–208.

[5] R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Comput.* 18 (1989) 792–810.

[6] M. B. Dillencourt, D. M. Mount, and N. S. Netanyahu. A randomized algorithm for slope selection. *Int. J. Computational Geometry & Appl.* 2 (1992) 1–27.

[7] D. Eppstein. Geometric lower bounds for parametric matroid optimization. *27th ACM Symp. Theory of Computing*, to appear; tech. rep. 95-11, Dept. of Information and Computer Science, University of California, Irvine, 1995.

[8] T. R. Ervolina and S. T. McCormick. Two strongly polynomial cut cancelling algorithms for minimum cost network flow. *Disc. Appl. Math.* 46 (1993) 133–165.

[9] M. R. Garey and D. S. Johnson. Computers and Intractibility: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.

[10] B. Gärtner. A subexponential algorithm for abstract optimization problems. *33rd IEEE Symp. Foundations of Computer Science* (1992) 464–472.

[11] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by cancelling negative cycles. *J. ACM* 36 (1989) 873–886.

[12] M. Hartmann and J. B. Orlin. Finding minimum cost to time ratio cycles with small integral transit times. *Networks* 23 (1993) 567–574.

[13] K. Iwano, S. Misono, S. Tezuka, and S. Fujishige. A new scaling algorithm for the maximum mean cut problem. *Algorithmica* 11 (1994) 243–255.

[14] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press (1972) 85–103.

[15] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Disc. Math.* 23 (1978) 309–311.

[16] R. M. Karp. Probabilistic recurrence relations. *J. ACM* 41 (1994) 1136–1150.

[17] R. M. Karp and J. B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Disc. Appl. Math.* 3 (1981) 37–45.

[18] M. J. Katz and M. Sharir. Optimal slope selection via expanders. *5th Canad. Conf. Computational Geometry* (1993) 139–144.

[19] J. Matoušek. Randomized optimal algorithm for slope selection. *Inf. Proc. Lett.* 39 (1991) 183–187.

[20] J. Matoušek. Approximations and optimal geometric divide-and-conquer. *23rd ACM Symp. Theory of Computing.* (1991) 505–511.

[21] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *8th ACM Symp. Comp. Geom.* (1992) 1–8.

[22] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM* 30 (1983) 852–865.

[23] J. Pach, W. Steiger, and E. Szemerédi. An upper bound on the number of planar $k$-sets. *Disc. & Comp. Geom.* 7 (1992) 109–123.

[24] T. Radzik and A. V. Goldberg. Tight bounds on the number of minimum-mean cycle cancellations and related results. *Algorithmica* 11 (1994) 226–242.

[25] R. Seidel. Linear programming and convex hulls made easy. *6th ACM Symp. Comp. Geom.* (1990) 211–215.

[26] L. Shafer and W. Steiger. Randomizing optimal geometric algorithms. *5th Canad. Conf. Computational Geometry* (1993) 133–138.

[27] N. E. Young, R. E. Tarjan, and J. B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks* 21 (1991) 205–221.