A Linear Time Randomizing Algorithm for Searching Ranked Functions

Eitan Zemel

Abstract. Consider a set $F$ of $n$ functions defined on a common interval $U$. A ranked function over $F$ is defined from the functions of $F$ by using order information such as the $k$ largest function, the sum of $k$ largest functions, etc. We give a linear time randomizing algorithmic paradigm for finding local roots, optima, intersection points, etc., of ranked functions. The algorithm is generalized to the Cost Effective Resource Allocation Problem and to various variants of the Parametric Knapsack Problem.

Key words. Selection, Rank, Randomizing algorithm, Parametric methods.

1. Introduction. Simple elementary functions, such as linear or quadratic functions, are often pieced together into composite piecewise functions. Such composite functions offer an interesting algorithmic challenge. Although they can be as simple to evaluate as the elementary functions from which they are derived, piecewise functions are much harder to handle in searches such as for a root, a local minimum or maximum, an intersection of two functions, etc. The difficulty arises, of course, from the fact that one does not know a priori which of the elementary functions coincides with the composite function at the unknown target point. Had this information been available, one could have ignored the piecewise nature of the composite function and find the required point directly using the relevant elementary function. In this note we consider an algorithmic paradigm which can solve this difficulty in many cases. Indeed, in the situations for which the method is applicable, the computational time associated with such searches is of the same order of magnitude as that of evaluating the composite function involved, and is only linearly longer than the time of searching any of the original (non-piecewise) functions.

The composite functions which could be handled by our method are piecewise composed from simpler functions by rank information. Specifically, let $F = \{f_1, f_2, \ldots, f_n\}$ be a set of continuous functions defined over a common interval $U = [a, b]$. For each value of the argument $x \in U$, let $F(x)$ be the multiset of functions values $F(x) = \{f_1(x), f_2(x), \ldots, f_n(x)\}$. Let $\pi = \pi^*$ be an increasing

---

1 Supported, in part, by NSF Grant ECS-812141, and by the Mia Fischer David Foundation through the Israel Institute of Business Research, Tel Aviv University. IIBR working papers are intended for preliminary circulation of tentative research results. Comments are welcome and should be addressed directly to the author.

2 J. L. Kellogg Graduate School of Management, Northwestern University, Evanston, Illinois 60201. Part of this work was done when the author was visiting Tel Aviv University.

Received April 11, 1985; revised October 28, 1985 and February 21, 1986. Communicated by D. T. Lee.
permutation of \( F(x) \), i.e., \( f_{\pi_1}(x) \leq f_{\pi_2}(x) \leq \cdots \leq f_{\pi_n}(x) \) where ties are broken arbitrarily. A partition \( F^1_x, F^2_x, F^3_x \) of \( F \) is called a \( k \)-partition at \( x \) if

\[
F^1_x = \{ f_{\pi_1}(x), \ldots, f_{\pi_{k-1}}(x) \}, \quad F^2_x = \{ f_{\pi_k}(x) \} \quad \text{and} \quad F^3_x = \{ f_{\pi_{k+1}}(x), \ldots, f_{\pi_n}(x) \},
\]

i.e., \( F^1_x \) is the set of \( k-1 \) "smallest" functions at \( x \), \( F^2_x \) is the \( k \) largest function, etc. A composite function \( g(x) \) is called a \( k \)-ranked function if it can be expressed as a weighted sum of the function of \( F \) where for each \( x \), the weight of \( f_i \) depends only on the member of the \( k \)-partition containing that function, i.e.,

\[
g(x) = a_1 \sum_{F^1_x} f_i(x) + a_2 \sum_{F^2_x} f_i(x) + a_3 \sum_{F^3_x} f_i(x).
\]

Simple examples of functions which satisfy such a requirement are the \( k \) largest function, the sum of the \( k \) largest functions, the difference between the average of the \( k \) largest and the average of the \( n-k \) smallest functions, etc. The key feature is the fact that if the original functions of \( F \) are simple, say linear, quadratic, etc., then, given a \( k \)-partition of \( F \) at the target point \( x \), such ranked functions are themselves linear, quadratic, etc., and could be easily constructed and searched.

The type of search which can be handled by our method is exemplified by finding a root, a local optimum, an intersection of two functions, etc. The essential feature here is that, given an argument \( x \), one can decide rapidly whether the required point is to the left or right of \( x \) (see Section 2 for the exact requirement). Under these conditions, we give a randomizing algorithmic paradigm which identifies the required point in linear time with high probability. More specifically, one can show that for each predetermined confidence level, \( 0 < p < 1 \), there exists a constant \( c = c(p) \) such that

\[
\text{Pr[Algorithm requires more than } cn \text{ steps}] \leq p.
\]

Our methods are generalized to yield an efficient randomizing algorithm for various ratio knapsack problems and for the Cost Effective Effort Allocation Problem [2]. Similar methods yield an \( O(n \log n) \) randomizing algorithm for the Weighted Euclidean One Center Problem [5]. This latter problem has since been solved in linear deterministic time by Dyer [1].

The deterministic foundation on which our algorithm is based is the parametric approach of Megiddo [3]. Indeed, the methods of [3] can yield an \( O(n \log^3 n) \) deterministic algorithm for our problem. The algorithm uses internally induced randomizations in the spirit of Rabin [6]. Thus, all the probability statements provided are worst case statements and do not rely on any assumption concerning the distribution of problem instances. A key element in achieving the linear running time of the algorithm is the concept of an \( \varepsilon \)-partition achieved by a binary search which is not carried to conclusion. Without this element, the running time of the algorithm is \( O(n \log n) \).
We note that several specific cases of the problem discussed here, such as finding the root of the \( k \) largest function, or intersecting the \( k_1 \) largest function in one set with the \( k_2 \) largest function in another, can be solved in deterministic \( O(n) \) time by specialized algorithms. However, these methods do not seem to generalize to the other problems solved by the present algorithm.

The paper is organized as follows. In Section 2 we present the necessary assumptions and conventions. The algorithm is presented in Section 3 and the analysis is presented in Section 4. Finally, some generalizations are discussed in Section 5.

An earlier version of this manuscript was circulated as [8].

2. The Computational Model. In this section we define our notion of a computational step and spell out the necessary assumptions and conventions. We refer to the point we seek as a "target" over \( U \). Thus, the targets that we consider here are roots (i.e., solutions to the equation \( g(x) = 0 \)), local optima (including at an end point of the interval), or intersection points of two ranked functions \( g_i \) and \( g_j \) (\( k_1 \) and \( k_2 \) may be different).

A partial partition of \( F \) is a trio \( F^1, F^2, \) and \( F^3 \) of disjoint subsets of \( F \). We say that a partial partition \( (F^1, F^2, F^3) \) is consistent with respect to a point \( x \in U \) if \( F^i \subseteq F^j_x \), \( i = 1, 2, 3 \), for some \( k \)-partition \( (F^1_x, F^2_x, F^3_x) \) at \( x \). A partial partition is consistent with respect to an interval \( U' \subseteq U \) if it is consistent with respect to each point \( x \in U' \). A function \( f_i \) is called free with respect to a partial partition \( (F^1, F^2, F^3) \) if it is not included in the partial partition, i.e., if \( f_i \in F^i \forall (F^1 \cup F^2 \cup F^3) \). We denote by \( n' \) the number of free functions with respect to a given partial partition. A partial partition is complete if \( n' = 0 \), i.e., if it is an actual partition of \( F \).

We make the following assumptions:

A.1. If the target is a root, we require that \( g \) is positive at one endpoint of \( U \) and negative at the other. If the target is an intersection point, then we require that \( g_1 \) is above \( g_2 \) at one endpoint of \( U \) and below it at the other endpoint. No assumption is necessary when the target is an optimal point (but the problem is trivial if one of the endpoints of \( U \) is itself one of the required local optima).

A.2. There exists a constant \( s \) independent of \( n \) such that any pair of functions \( f_i, f_j \) of \( F \) intersect at most \( s \) times over \( U \). Furthermore, all the intersection points associated with a given pair could be identified in constant time.

A.3. Aggregation and evaluation: functions of the form

\[
h(x) = a_1 \sum_{S^1} f_j(x) + a_2 \sum_{S^2} f_j(x) + a_3 \sum_{S^3} f_j(x),
\]

where \( S^1, S^2, \) and \( S^3 \) are explicit disjoint subsets of \( F \) (as opposed to \( F^1, F^2, \) and \( F^3 \) which are defined only indirectly) could be evaluated in constant time after a set-up proportional to the number of functions in \( S^1 \cup F^2 \cup S^3 \). (If the target is an optimum, we require instead that the sign of the derivative of \( h \) could be evaluated.)
A.4. Given an interval $U' \subseteq U$ which satisfies the conditions of A.1 and a complete partial partition which is consistent over $U'$, one can identify a target in $U'$ in $O(n)$ steps. (In case of an intersection point we require a complete consistent partition for each function $g_1$ and $g_2$.)

A.5. There exists a constant $t$ independent of $n$ such that the number of targets $U$ does not exceed $t$.

Assumption A.1 implies that there exists at least one target over $U$. The first part of this assumption, together with Assumptions A.3 and A.4, constitute our definition of the notion of a computational step. For instance, if the functions in $F$ are linear, all these requirements are satisfied if we can perform the four basic arithmetic operations in constant time. For quadratic functions, we will also need to take square roots in constant time, etc. Assumptions A.1, A.2, A.3, and A.4 are standard for problems such as discussed here, and are required, say, if one were to use Megiddo's deterministic $O(n \log^2 n)$ algorithm referred to above. The only additional assumption we make here is Assumption A.5. This assumption is required for the probabilistic analysis of the running time of the algorithm. In its absence, the algorithm will still function as required (it would provide a target point over $U$), but the worst case effort may be quadratic. Assumption A.5 is more restrictive than one would like, but it still holds in a variety of interesting situations. For instance, $t = 1$ if we intersect two monotonic functions, one increasing and one decreasing, or if we look for the root of a monotonic function. ($g$ inherits monotonicity from the functions of $F$ if the constants $a_1$, $a_2$, and $a_3$ are of the same sign.) As another example, A.5 holds with $t = 2$ if we search for the root of a convex or concave function. ($g$ inherits its concavity from the function of $F$ if the coefficients satisfy, say, $0 \leq a_2 \leq a_3 \leq a_1$.)

3. The Algorithm. In this section we describe the algorithm. We start by discussing the basic building blocks from which it is constructed—namely, the concepts of a test, the rank and $\varepsilon$-rank of elements, and the process of eliminating functions. Then we show how these elements interact together to form the algorithm.

3.1. The Test. In the sequel we assume that an interval $U' = [a', b'] \subseteq U$, which satisfies the conditions of A.1, is given. Thus we know that $U'$ contains a target point of $g$. Let $x \in U'$ be arbitrary. We call the operation of deciding which of the subintervals $[a', x]$ or $[x, b']$ contains a target point a test at $x$. In general, if $U'$ contains several targets, it is possible that both these subintervals contain a target. However, from the point of view of the algorithm, all that is required is to identify one of these subintervals. For ranked functions of the type considered here, the operation of a test is very simple conceptually. In the case of roots, we need to compute the sign of $g$ at $x$; in the case of an intersection point, we need the sign of $g_1 - g_2$ at $x$; and in the case of an optimum, we need the sign of the derivative of $g$ at $x$. In all cases, this can be done in time proportional to the
number of free functions, as per Assumption A.3. In a typical binary search algorithm, a test allows us to reduce the interval of interest $U'$. The key idea in the parametric method of Megiddo is to use tests to reduce the number of free functions.

3.2. The Rank and Partition of a Function. Let $y$ be the target point which is eventually found by the algorithm. The rank of a function $f_i$ in $F$ is defined relative to this point. Let $F_1 \subseteq F$, $F_2 \subseteq F$, and $F_3 \subseteq F$ be a partition of the set of functions $F$ according to whether the values at $y$ are less than, equal to, and greater than $f_i(y)$, respectively. The rank of $f_i$ in $F$ is the ordered triplet of cardinalities $(|F_1|, |F_2|, |F_3|)$. Similarly, if we attach weight $w_i$ to each function $f_i$, $i = 1, \ldots, n$, the weighted rank of $f_i$ in $F$ is the triplet of total weight inside these sets. The rank of an element can be useful in the process of eliminating free functions, as is discussed below. Unfortunately, as $y$ is not known in advance, one cannot find the rank of $f_i$ directly. Nevertheless, we can still rank $f_i$ in a roundabout way by performing several tests at key points. Indeed, let $X' = \{x'_1, \ldots, x'_n\}$ be the set of roots of the equation $f_i(x) = f_j(x)$ inside $U'$. From Assumption A.2, $p_j \leq s$. Without loss of generality, we can assume that roots in which the functions $f_i$ and $f_j$ just touch (rather than cross) each other, have been eliminated. Thus, the roots in $X'$ induce a partition of $U'$ such that $f_i$ alternates, being above and below $f_j$ on subsequent intervals of this partition. Therefore, we can find whether $f_i(y)$ is above or below $f_j(y)$ by locating $y$ within the partition induced on $U'$ by the set $X'$. Let $X = \{x_1, \ldots, x_p\}$ be the union of the sets $X'$, $j = 1, \ldots, n$, $j \neq i$. Then $p \leq (n-1)s$. It follows that the complete partition of $F$ into the three subsets $F'_1$, $F'_2$, $F'_3$ can be identified if the location of $y$ is known relative to all the points in $X$. But this can be found, using binary search over $X$ in $\log(p)$ tests. Since $p = O(n)$, and since each test requires $O(n)$ computational steps, this can be the basis of an $O(n \log n)$ randomizing algorithm for our problem. An alternative $O(n \log n)$ randomizing algorithm can be obtained by using Reischuk’s randomizing parallel algorithm for selection [7], within the framework of Megiddo’s method [3]. To achieve linear time performance, we have to economize further. This can be accomplished by using a relaxed form of ranking which we call $e$-ranking. Before introducing this construct we remark that as a byproduct of ranking $f_i$, we derive a subinterval $U'' \subseteq U'$ which is guaranteed to contain $y$, namely that subinterval of the partition induced by $X$, which is identified as containing $y$ by the binary search. An interval of this type will be also identified as a byproduct of $e$-ranking. It is obvious that, once identified, $U''$ can replace $U'$ without causing any error in the algorithm.

3.3. $e$-Ranking and $e$-Partitioning. Let $0 < e < 1$ be arbitrary but fixed. An $e$-partition of $F$ by $f_i$ is any partition of $F$ into four subsets, $G^a_n$, $\alpha = 1, 2, 3, 4$, such that for $\alpha = 1, 2, 3$

$$G^a \subseteq F^a,$$

while for $\alpha = 4$

$$|G^a| \leq e|F|.$$
Clearly, there are numerous \( \epsilon \)-partitions associated with a given function \( f \) as the only requirement of \( G^1_i \) is on its cardinality. An \( \epsilon \)-rank of \( f \) is the ordered triplet of cardinalities \( (|G^1_i|, |G^2_i|, |G^3_i|) \) associated with one of these \( \epsilon \)-partitions. A weighted \( \epsilon \)-partition is similar to an \( \epsilon \)-partition, but with the requirement that the weight of \( G^1_i \) does not exceed \( \epsilon \) times the total weight. Although not unique, an \( \epsilon \)-rank gives some useful information on the "real" rank of a function \( f \), as for \( \alpha = 1, 2, 3 \), one has

\[
G^\alpha_i \subseteq F_i^\alpha \subseteq G^\alpha_i \cup G^\alpha_i
\]

and therefore,

\[
|G^\alpha_i| \leq |F_i^\alpha| \leq |G^\alpha_i| + \epsilon |F|.
\]

A quick and easy way to compute an \( \epsilon \)-rank of an element \( f \) is the following. Let \( X = \{x_1, \ldots, x_p\} \), \( p = (n-1)s \), be the set of intersections of \( f \) with the other \( n-1 \) functions of \( F \). By performing a test at the median element of \( X \), we establish the location of \( y \) in relation to one half of the points in \( X \). A second test at the median of the remaining set establishes the location of \( y \) with respect to an additional quarter of these points. Continuing in this fashion for a total of \( \log(s/\epsilon) \) tests, we are left with a subset of at most \( \epsilon n(n-1) \leq \epsilon n \) of the points of \( X \). But then the number of functions \( f_j \in F \) which cannot be put in one of the sets \( G^\alpha_i \), \( \alpha = 1, 2, 3 \), is at most \( \epsilon n \). For any prespecified \( \epsilon > 0 \), this amounts to \( O(n) \) steps overall, since \( s \) is a constant. We note that a similar number of steps is needed for deriving an \( \epsilon \)-weighted partition. This can be achieved by performing tests at the weighted medians of the appropriate sets.

3.4. Elimination of Free Functions. We now examine the process of elimination. Recall that \( y \) is the target point eventually found by the algorithm. Basically, elimination can take place if we can decide in which of the subsets \( F^1_i, F^2_i, \) or \( F^3_i \), a given function \( f \) belongs. This is so since, if that information is available, the appropriate weight of \( f \), in the definition of \( g \) is known, and the function can be aggregated with other functions in its subset and need not be further considered individually. Assume that the partition \( F^1_i, F^2_i, F^3_i \) of an arbitrary function \( f \in F \) is given. Recall that this partition is relative to \( y \). Thus, if \( |F^1_i| \geq k-1 \), then \( f \in F^2_i \). Moreover, every function \( f_j \) in \( F^1_i \cup F^2_i \), being as large as or even larger than \( f \) at \( y \), also belongs to that set. Similarly, if \( |F^3_i| \geq n-k \), then each function in \( F^1_i \cup F^2_i \) must be in \( F^3_i \). Finally, if neither condition holds, then \( F^1_i \subseteq F^2_y, F^2_i \subseteq F^3_y \), and the functions in \( F^2_i \) can be arbitrarily placed in \( F^1_y, F^2_y, \) and \( F^3_y \) so that the required cardinalities \( k-1, 1, \) and \( n-k \) for these sets, respectively, are achieved. In each case, a subset of the free functions is eliminated. A key question we still have to address is the cardinality of this subset.

In actuality, we only have information about the \( \epsilon \)-rank of \( f_i \), rather than about its true rank. However, we can still go through with the elimination process with only small modifications. For instance, assume that \( |G^1_i| \geq k-1 \). Then obviously \( |F^1_i| \), being larger than \( |G^1_i| \), must also satisfy this inequality. Thus, we can safely eliminate \( G^1_i \cup G^3_i \) (which is a subset of \( F^1_i \cup F^2_i \)). Note that the difference between
the magnitudes of $G_i^1 \cup G_i^2$ and $F_i^1 \cup F_i^2$ is at most $\varepsilon \cdot n$. A similar situation arises when $|G_i^2| > n - k$. If neither condition holds but $|G_i^2 \cup G_i^3| > k - 1$ then $G_i^1$ can be eliminated. Symmetrically, if $|G_i^1 \cup G_i^3| > n - k$ then $G_i^2$ can be eliminated. Finally, if none of the above-mentioned conditions holds, i.e., $k - 1 > |G_i^1| \geq k - 1 - \varepsilon n$ and $n - k > |G_i^2| \equiv n - k - \varepsilon n$, no elimination can take place. (Note that this situation never arises if the true rank of $f_i$ is known, i.e., if $\varepsilon = 0$.)

We are now in a position to state our algorithm:

### 3.5. Algorithm Search

Begin Search

Set: all functions are free: $U' = U$.

Do: until all free functions are eliminated:

1. Choose a random free function $f_i$.
2. Compute an $\varepsilon$-rank and $\varepsilon$-partition of $f_i$ in the set of free functions.
3. Eliminate, if possible, a subset of the free functions, and reduce the interval of search $U'$.

End Do

4. Find the required target over $U'$.

End Search

As per Assumption A.4, step 4 of the algorithm requires $O(n)$ steps. Also, each iteration through steps 1, 2, and 3 requires $O(n')$ steps, where $n'$ is the number of free functions at the beginning of the iteration. In the next section we show that the overall effort needed for the entire algorithm is $O(n)$ with high probability.

### 4. Analysis of the Algorithm

In this section we analyze the running time of Algorithm Search. We call each cycle through steps 1, 2, and 3 an iteration of the algorithm. Let $\alpha > 0$ be a constant to be specified later. Call an iteration a success if we eliminate at least $\alpha \cdot n'$ functions at step 3. We wish to assess the probability of success. Throughout this analysis we ignore the issues of rounding and treat expressions like $\alpha \cdot n'$ as integers. The reader should have no difficulty realizing that the errors introduced by this approximation are negligible and do not change the conclusions of this analysis.

Assume first that $y$ is the unique target inside $U$. Consider any permutation $\pi$ of the indices of $F(y)$ which is consistent with the order of the set $F(y)$, i.e., $f_{\pi_i}(y) \leq f_{\pi_{i+1}}(y) \leq \cdots \leq f_{\pi_{n'}}(y)$. Let $i = \pi_r$. Obviously, since $f_i$ is chosen randomly from $F$, $r$ is uniform over the integers $1$ to $n'$. Denote the integers in the set

$$\{ \pi_i : r \in \{k+\varepsilon \cdot n'\} \cup \{0, \ldots, (\alpha + \varepsilon) n'\} \cup \{n'-(\alpha + \varepsilon) n', \ldots, n'\} \}
$$

as forbidden. The probability that $r$ falls in the forbidden range is at most $2\alpha + 4\varepsilon$. It is easy to see that any choice of $f_i$ which is in the complement of the forbidden set must yield a success even for the worst possible choice of the set $G_i^4$.

Consider now the case of several targets in $U$. Since the order of the functions in $F$ may be different for each root, the forbidden regions with respect to each
target may be different. For each given target, the probability of falling in the forbidden range is as calculated in the previous section. However, \( y \) cannot be assumed fixed since its identity depends on the random steps of the algorithm. To overcome this difficulty, let \( A \) be the event:

\[ \{ f_i \text{ is in the forbidden range for a given target, } y \in U \} \]

and let \( B \) be the event:

\[ \{ f \text{ is in the forbidden range for at least one of the } t \text{ targets in } U \}. \]

Then, \( \Pr[A] \leq 2\alpha + 4\varepsilon \) and \( \Pr[B] \leq t \cdot \Pr[A] \leq t(2\alpha + 4\varepsilon) = q \). Clearly, \( q \) is an upper bound on the probability of failure. Choose the constants \( \alpha \) and \( \varepsilon \) so that \( q < 1 \). We recall that the work per iteration is \( c \log(s/\varepsilon)n' = c_i n' \) for some small constant \( c_i \), independent of success or failure.

Let \( S_n \) be the running time of the algorithm, with \( n \) functions left to go. \( S_n \) is a random variable whose distribution depends on the data (i.e., on \( F \) and \( U \)) in a very complex way. However, we can analyze the running time, \( T_n \), of a slightly modified version of the algorithm. The modified version proceeds precisely as the original algorithm, except that whenever success occurs, exactly \( \alpha \cdot n' \) functions are eliminated (as opposed to at least \( \alpha \cdot n' \) which are eliminated by our algorithm). Analyzing \( T_n \) is also complex since the probability of success in each step may vary. However, it can be easily observed to be stochastically dominated by a third random process, \( X_n \), which is defined exactly like \( T_n \), except that the probability of success in each iteration is exactly \( q \) (as opposed to at least \( q \) in \( T_n \)) and the events [iteration \( i \) is a "success"], for \( i = 1, 2, \ldots \), are independent. \( X_n \) can be easily analyzed. In fact, one can show that

\[ E(X_n) = \frac{c_1}{(1-q)\alpha} \cdot n = c_2 n, \]

and

\[ \text{Var}(X_n) = d^2 n^2 \]

for some easily obtained coefficient \( d \). Let \( 0 < p < 1 \) be given, and let

\[ c(p) = c_2 + \sqrt{1/p} \cdot d. \]

Then, by Chebyshev's inequality,

\[ (*) \qquad \Pr[X_n > c(p) \cdot n] \leq p. \]

Since \( X_n \) stochastically dominates \( T_n \), \( (*) \) holds also for \( T_n \) as announced.

5. Extensions. The method described in the previous sections has many generalizations. We briefly describe below two of these generalizations which can
be used for solving problems which have appeared elsewhere in the literature. The principles involved in the examples below are very similar to those discussed earlier in this paper. Therefore, we do not specify here the details necessary for the implementation of these generalizations. For a more detailed description, see [7]. The two cases considered below are problems with partial order specified, and problems with weighted functions.

Consider first the original problem of finding a target point \( y \in U \), but with some structure imposed on the set \( F \). Specifically, assume that this set is partitioned into \( m \) subsets, \( F_1, \ldots, F_m \), such that the sorted permutation at \( y \) of the functions in each subset is known in advance. This additional information allows us to speed up the computation, yielding, for appropriate values of \( m/n \), a sublinear algorithm.

The basic scheme is as follows. At each iteration, let \( n_i \) be the number of free functions in the subset \( F_i \), \( i = 1, \ldots, m \). Denote the total number of free functions by \( n' \). Assume \( k \leq n'/2 \). Let \( f_{n_i} \) be the \( 3n_i/4 \) largest function in \( F_i \) at the target point \( y \). We pick a subset \( F_i \), \( i = 1, \ldots, m \), with probability proportional to \( n_i \), and find the weighted rank of \( f_i \) in the set \( \{f_1, f_2, \ldots, f_m\} \), where the weight of \( f_i \) is \( n_i \). This information allows us to eliminate at least \( \alpha n' \) functions with high probability, where \( \alpha \) is a constant independent on \( m \) and \( n \). The cost per iteration of this procedure is \( O(m \log n/m) \) and the total number of steps required by the entire algorithm can be seen to be \( O(m \log^2 n/m) \) with high probability. A specific example is the Cost Effective Resource Allocation Problem:

\[
\max \sum_{i=1}^{m} \frac{f_i(x_i)}{g_i(x_i)},
\]

subject to

\[
\sum_{i=1}^{m} x_i \leq k,
\]

\[
x_i \geq 0, \quad \text{integer},
\]

where \( F = \{f_1, \ldots, f_m\} \) and \( G = \{g_1, \ldots, g_m\} \) are sets of concave and convex functions, respectively. The algorithm proposed here can solve this problem in \( O(m \log^2 m) \) steps with high probability as opposed to the \( O(m \log^2 m \log^2 k) \) worst case deterministic complexity [2].

As another extension consider weighted problems where a weight, \( w_i \geq 0 \), is attached to each function \( f_i \in F \). A case in point is the max ratio knapsack problem:

\[
\max \sum_{j \in N} c_j x_j + c_0
\]

subject to

\[
\sum_{j \in N} a_j x_j \leq a_0, \quad 0 \leq x_j \leq 1, \quad j \in N,
\]
with

\[ d_j > 0, \quad j = 0, 1, \ldots, n, \]

or the parametric knapsack problem:

\[ \min_{\lambda} \max_{x} \sum_{j \in N} (c_j - \lambda d_j)x, \]

subject to

\[ \sum_{j \in N} a_jx_j \leq a_0, \quad 0 \leq x_j \leq 1, \quad j \in N. \]

Both these problems can be solved in \( O(n \log^2 n) \) deterministically by the methods of [3]. By picking a variable \( x_i, i \in N \), randomly (with probability \( 1/n \)) and computing its weighted rank (where the weights are the \( a_i \)'s), we can get an \( O(n \log n) \) randomizing running time. Also, by picking \( x_i, i \in N \), with probability proportional to \( a_i \) and computing its \( \epsilon \)-weighted rank, we get an \( O(n \log \lambda) \) randomized running time, where \( \lambda \) is the ratio of the largest to smallest weight coefficient, i.e.,

\[ \lambda = \max_{i,j} \frac{|a_i|}{|a_j|}. \]

References