Check for updates

Winnowing Sequences from a Database Search

Piotr Berman* Zheng Zhang* Yuri I. Wolf†‡ Eugene V. Koonin† Webb Miller*§

Abstract

In database searches for sequence similarity, matches to a distinct sequence region (e.g. protein domain) are frequently obscured by numerous matches to another region of the same sequence. In order to cope with this problem, algorithms are developed to discard redundant matches. One model for this problem begins with a list of intervals, each with an associated score; each interval gives the range of positions in the query sequence that align to a database sequence, and the score is that of the alignment. If interval I is contained in interval J, and I's score is less than J's, then I is said to be dominated by J. The problem is then to identify each interval that is dominated by at least K other intervals, where K is a given level of "tolerable redundancy." An algorithm is developed to solve the problem in $O(N \log N)$ time and $O(N^*)$ space, where N is the number of intervals and N^* is a precisely defined value that never exceeds N and is frequently much smaller. This criterion for discarding database hits has been implemented in the Blast program, as illustrated herein with examples. Several variations and extensions of this approach are also described.

1 Introduction

As a protein or DNA sequence database grows, the number of entries showing similarity to a given query sequence will eventually exceed what a user can examine in detail. A large number of matches reported in one region of the query sequence may hide lower-scoring but informative matches occurring elsewhere. The situation is particularly critical with database programs that report matches in order of decreasing score and truncate the output after a fixed limit on output size is reached. Rules are needed to select a manageable subset of the matches that is likely to reveal all of the important search results.

Each "hit" for a query sequence aligns an interval of positions in that sequence with part or all of a database sequence. For some purposes, it may be adequate to discard any hit whose interval is contained in at least K other hit-intervals, where K is chosen in advance. A biologically more reasonable approach may be to discard a hit only if its interval is contained in more than K hit-intervals *of higher alignment score* [9].

Further levels of sophistication can be added by associating additional kinds of scores with each hit, and requiring that any "dominating" hit score higher on all criteria. For instance, database entries might be scored according to the completeness of their annotation, so that e.g. an entry with a literature citation would not be suppressed in favor of higher-scoring but anonymous hits. Similarly, database sequences could be scored according to the tractability of the originating organism as an experimental system. Finally, each database sequence could be scored by its phylogenetic distance from the query sequence, i.e., the height in the taxonomic tree of the most recent common ancestor of the two originating species. That way, the winnowed hits would retain information showing how sequence similarity falls off with increasing phylogenetic distance, as well as suggesting the age of a protein domain.

For winnowing, we start with a multiset H of intervals, each with M scores. ($M \geq 0$ is the number of criteria we are using for winnowing.) Let the i-th score of interval I be denoted I^i. In one formulation of winnowing, we identify as redundant each interval I for which there are at least K intervals J in H such that $I \subseteq J$ and that $I^i \leq J^i$ whenever $1 \leq i \leq M$. For this paper we impose the additional requirement that each J "strictly dominates" I, in the sense that either J is strictly larger as an interval, or $I^i < J^i$ for at least one of the scores. This additional requirement, which simplifies the analysis, in practice provides a desirable tie-breaking rule. Intervals that are redundant in this formulation will be called *K-contained*.

In one approach, winnowing would proceed in parallel with the database search procedure, so that at each point in time only the non-redundant hits would be retained. Let the hits be discovered in the the order $I_1, I_2, \ldots, I_N$, let N_i be the number of non-redundant hits among $I_1, \ldots, I_i$, and define $N^* = \max_i N_i$, so that this incremental approach would take $O(N^*)$ space. An apparent drawback is that an incremental algorithm might take more total computation time than a winnowing method that waits until all hits are known. In the next section we show that for many kinds of

*Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802.

†National Center for Biotechnology Information, National Library of Medicine, National Institutes of Health, Bethesda, MD 20894.

‡Permanent address: Institute of Cytology and Genetics, Russian Academy of Sciences, Novosibirsk 630090, Russia.

§Corresponding author: FAX:(814) 865-3176; e-mail: webb@cse.psu.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RECOMB '99 Lyon France

Copyright ACM 1999 1-58113-069-4/99/04...$5.00

winnowing procedures one can attain the space efficiency of an incremental algorithm while at most doubling the running time.

Our first result is a winnowing procedure, using the model described above with $M = 1$, which uses $O(N \log N^*)$ time and $O(N^*)$ space. This is far more efficient than the straightforward $O(N^2)$ method that inspects every pair of intervals to see if one dominates the other. For $M = 2$ we attain $O(KN \log^2 N^*)$ time and $O(N^*)$ space. These results are described in Section 3.

In Section 4 we develop winnowing algorithms for an alternative model under which redundancy requires only that each position in I be covered by at least K intervals of higher score. In the last section we describe results from using the original notion of winnowing in the Blast program [1, 2].

We will use the following shorthands: $A + a$ for $A \cup \{a\}$ and $A - a$ for $A - \{a\}$.

2 Offline, Batch, and Online Algorithms

The approaches to winnowing discussed in this paper fall under the following framework. We are given a multiset H, whose elements are called hits. (The term "multiset" means that we are allowing two entries to have identical endpoints and scores.) Also, we have some precise notion of a hit x being "redundant in A", denoted $x \rhd A$, where $A \subseteq H$. The problem is to find $H' = \{h \in H | h \not\rhd H\}$.

For Theorem 2, below, the only relevant properties of redundancy are:
(1) if $x \rhd A$, then $x \rhd A + y$, and
(2) if $x \rhd A$ and $y \rhd A$, then $x \rhd A - y$.

These properties are respectively equivalent to:
(1') if $x \rhd A$, then $x \rhd A \cup B$, and
(2') if $x \rhd A$ and $y \rhd A$ for each $y \in B$, then $x \rhd A - B$.

Condition (1) is straightforward, whereas condition (2) deserves a moment's thought. Consider the situation where redundancy is defined as in the Introduction, fix redundant hits x and y, and pick K hits in A that dominate x. If y is not one of those K hits, then clearly $x \rhd A - \{y\}$. Otherwise, any K hits in A that dominate y also dominate x.

Lemma 1 *If (1) and (2) are satisfied, then* $(A \cup B)' = (A' \cup B)'$ *for any* A *and* B.

Proof. Pick $x \in (A \cup B)'$ and suppose $x \notin (A' \cup B)'$. Then either $x \notin A' \cup B$ or $x \rhd A' \cup B$. In the former case, $x \in A - A'$, so $x \rhd A$, and hence $x \rhd A \cup B$ by (1'), a contradiction. In the latter case, $x \rhd A \cup B$ by (1'), so $x \notin (A \cup B)'$, another contradiction. Thus $(A \cup B)' \subseteq (A' \cup B)'$.

Conversely, pick $x \in (A' \cup B)'$ and suppose $x \notin (A \cup B)'$. Then $x \rhd A \cup B$. For each $y \in A - A'$, $y \rhd A$ by definition, so $y \rhd A \cup B$ by (1'), and from (2') we have $x \rhd A \cup B - (A - A') \subseteq A' \cup B$. By (1'), $x \rhd A' \cup B$, a contradiction. Thus $(A' \cup B)' \subseteq (A \cup B)'$. □

Let $h_1, h_2, \ldots, h_N$ be a fixed permutation of H. Set $H_0 = \emptyset$ and $H_i = (H_{i-1} \cup \{h_i\})'$. Lemma 1 implies that $H_N = H'$. Let $N^* = \max_i |H_i|$. An *offline* algorithm, i.e., one that starts from the complete set of data, by definition needs $\Omega(N)$ space, whereas computing $H_N = H'$ using the above recursion may reduce the space to $O(N^*)$. While the intermediate H_i are of no inherent interest to us, Lemma 1 lets us derive a space-efficient *batch* algorithm from any offline winnowing algorithm using the following theorem.

```
Let H be {h_1, h_2, ..., h_N}
H'_1 ← {h_1}, i ← 1
while i < N do begin
    a ← min(|H'_i|, N − i)
    compute H'_{i+a} ← (H' ∪ {h_{i+1}, h_{i+2}, ..., h_{i+a}})'
        using the offline algorithm
    i ← i + a
end
```

Figure 1: Batch algorithm for winnowing.

Theorem 2 *Fix an offline algorithm that computes* H' *using* $Nt_{off}(N)$ *time, where* $t_{off}(N)$ *is a nondecreasing function of* N, *and* $s_{off}(N)$ *space. Let* $N' = \min(2N^*, N)$, *then the algorithm of Figure 1 computes* H' *in* $(2N + N^*)t_{off}(N')$ *time and* $s_{off}(N')$ *space.*

Proof. Correctness of the algorithm follows from Lemma 1 by induction. The space requirements are obvious, as each call to the offline algorithm has input set not larger than $2N^*$. Let $a_1, \ldots, a_k$ be the values of a in consecutive runs of **while** loop. One can observe that $\sum_{j=1}^{k} a_j = N - 1$. Also, the size of the input set for the jth call to the offline is $2a_j$ for $j < k$ and at most $N^* + a_k$ for the last call. Because each of these sets has size at most N', the sum of time running times of these calls is at most $(N^* + 2\sum_{j=1}^{k} a_j)t_{off}(N') = (2N + N^*)t_{off}(N')$. □

In addition to offline and batch winnowing algorithms, we consider *online* algorithms, which can be described as follows. Let $P = \{h_1, h_2, \ldots, h_N\}$ be a permutation of H. After initializing S to $\emptyset$, hits are inspected in permutation order, and h is inserted into S if $h \not\rhd S$. The permutation order needs to be chosen so that inserting h into S does not make some $s \in S$ redundant.

An offline algorithm for M scores can be derived from an online algorithm for $M - 1$ scores; the hits with M scores are sorted by one of the scores, then winnowed online with the remaining scores. Three of the four algorithms presented in this paper have this form.

3 Winnowing off K-contained hits

In our algorithm we inspect hits in order of decreasing scores (we sort them first, as we will see below, we may assume that no two scores are equal). Hits are treated as points in $\mathbf{R}^2$ by mapping an interval $[b, e]$ into $p = (p_1, p_2) = (-b, e)$. We build a collection of points $\mathcal{P}$ by initializing it to $\emptyset$; we process each point p from the sorted sequence by changing $\mathcal{P}$ into $\mathcal{P} + p$ if $p < q$ holds for fewer than K elements of $\mathcal{P}$.

The latter condition can be checked efficiently using a data structure of McCreight [5], called a priority search tree, which maintains a set of points with insertion, deletion and a query that for given a, b, y, returns the set of points that satisfy $a \le p_1 \le b$ and $p_2 < c$. The running time of operations is $O(\log n)$ for updates and $O(\log n + s)$ for the query, where n is the current set size and s is the size of the output of the query. We can easily modify the query so in case when $s > K$ it returns the first K points only, in time $O(\log n + K)$. Because for each input point we perform one query and at most one insertion, we may conclude the following theorem (the claim about space requirements follows from Theorem 2).

Theorem 3 *If the hit intervals have one score, we can winnow off K-contained hits in time $O((K+\log N^*)N)$, using $O(N^*)$ space.*

3.1 Non-redundant hits with two score functions

In the remaining part of this section we discuss the problem of finding H' if H is a set of hits with two score functions, say s_1 and s_2, and $h \rhd H$ iff h is contained in at least K elements of H.

Let L be the length of our sequence. We map a hit $h = (b, e, s_1, s_2)$ into point

$$p = (p_1, p_2, p_3, p_4) = (1 - b/L, e/L, s_1, s_2) \in [0,1]^2 \times \mathbf{R}^2.$$

It is easy to see that h is contained in h' iff $p \leq p'$ and $p \neq p'$. In this case we say that p' dominates p and denote it $p \prec p'$. The description of the algorithm can be simplified if we assume that no two points share a value of a coordinate. This can be achieved in two ways. Mathematically, we can change the points using a sufficiently small ε to replace each point p with $p + \varepsilon(p_4, p_1, p_2, p_3) + \varepsilon^2(p_3, p_4, p_1, p_2) + \varepsilon^3(p_2, p_3, p_4, p_1)$. Algorithmically, this means that when we sort points according to one of the coordinates, we have tie-breaking rules for the cases when the respective coordinate values are equal. One can also provide similar tie-breaking rules that are appropriate for other versions of the problem, e.g. when $p \prec p'$ iff $p \leq p'$, $p_3 < p'_3$ and $p_4 < p'_4$, or, alternatively, $p \prec p'$ iff $p \leq p'$ and $(p_3, p_4) \neq (p'_3, p'_4)$.

Our data structures will be simpler if we use randomization. As a result, all running time will be given as *average* values. However, the bounds on the running time will not depend on the input, but solely on our random choices.

3.2 The problem in 3 dimensions

As we discussed in Section 2, we can reduce the number of dimensions if we convert our problem to the online version. We present the input set of points $\mathcal{P}$ as the sequence $P = (p^1, p^2, \ldots, p^N)$ in which the last coordinate is decreasing. Next, we compute, for $i = 0, \ldots, N$ the set $\mathcal{P}'_i$ according to the following rules: $\mathcal{P}'_0 = \emptyset$; for $i > 0$, if p^i is dominated by more than K points in $\mathcal{P}'_{i-1}$, then $\mathcal{P}'_i = \mathcal{P}'_{i-1}$, otherwise $\mathcal{P}'_i = \mathcal{P}'_{i-1} \cup \{p^i\}$.

Note in this computation we can ignore the last coordinate of the points: when we consider a new point, we know that it has smaller last coordinate than all the points in $\mathcal{P}'_{i-1}$, so the result depends on the first three coordinates only. Thus from now on we will view the input as a sequence of the points from $\mathbf{R}^3$.

Our approach is to decompose $\mathcal{P}'_i$ into K *shells* and the "overflow" set. For $\mathcal{S} \subset \mathbf{R}^3$, let the surface of $\mathcal{S}$ be $\{p \in \mathcal{S} : \forall q \in \mathcal{S}\ p \not\prec q\}$. We define the shells as follows: Sh_0 is the surface of $\mathcal{P}'_i$, for $0 < j < K$ Sh_j is the surface of $\mathcal{P}'_i$ with all the previous shells removed, and Sh_K, the overflow set, contains the remaining points of $\mathcal{P}'_i$. To update a shell Sh, we use the following two operations:

CODE(Sh, p, a), the count of dominating elements, returns the minimum of $|\{q \in Sh : p \prec q\}|$ and a;

PLUG_IN$(Sh, p, \mathcal{S})$, $\mathcal{S}$ becomes $\{q \in Sh : q \prec p\}$, Sh becomes $Sh - \mathcal{S} \cup \{p\}$.

We define the *size* of shell operations using a nonnegative potential function $\pi(Sh)$ such that $\pi(\emptyset) = 0$. The size of a call to CODE is 1 plus the number that is returned; while the size of a call to PLUG_IN is 3 plus the resulting $|\mathcal{S}|$, minus the increase of $\pi(Sh)$ (or plus the decrease).

```
a ← 0; j ← −1;
do
    j ← j + 1; b ←CODE(Sh_j, p, K − a); a ← a + b;
until b = 0;
if a = K then exit;
S ← {p};
while j < K do begin
    R ← ∅;
    for every q ∈ S do
        PLUG_IN(Sh_j, q, Q), R ← R ∪ Q;
    S ← R; j ← j + 1
end ;
insert S into Sh_K
```

Figure 2: High level description of the algorithm.

Lemma 4 *Assume that we have an implementation of shell data structure which uses memory linear in $O(|Sh|)$, and executes an operation of size s in time $O(s \log^2 |Sh|)$. Then we can compute $\mathcal{P}'_N$ in time $O(KN \log^2 N^*)$ using $O(N^*)$ memory.*

Sketch of proof. We initialize $Sh_i = \emptyset$ for $i = 0, \ldots, K$ and we process the sequence by executing, for each input point, the algorithm from Fig. 2. The task of the **do until** loop is to compute the count of points of $\mathcal{P}'$, defined as the shells plus the overflow set, that dominate the new point (or K, if this count is larger). Suppose that p is not dominated by any point of Sh_0, the surface of $\mathcal{P}'$. Then the first run of **do** loop results in $a = b = 0$ and the loop terminates. One can see that $a = 0$ is the correct count: if p is dominated by a point $q \in \mathcal{P}'$ that is not from the surface, then q is also dominated by some r from the surface, and such an r would be counted by a. Now suppose that p is dominated by some $a > 0$ points of this surface. Then we need to find the number a' of points from $\mathcal{P}'_i$, but not from the surface, that dominate p and then return $\min(a + a', K)$. One can see that it suffices to find $\min(a', K - a)$ and one can see that this is accomplished by the remaining runs of **do until** . An extreme case occurs if $b = 1$ for runs with $j = 0, \ldots, K-1$; then the algorithm enters the next run with $a = j = K$, and the call CODE$(Sh_K, p, 0)$ returns 0. It is easy to see that the sum of sizes of the calls of CODE that are executed for a single point p is at most $2K + 1$, and thus the running time devoted to this task is $O(KN \log^2 n)$.

The second task is to change the shells (and the overflow set), provided that the computed value of a is lower than K. The correctness of the procedure follows directly from the specification of PLUG_IN. The bound on the running time follows from the fact that the sum of sizes of all calls of PLUG_IN is bounded by the following expression:

$$\sum_{j=0}^{K} (3(j+1)|Sh_j| - \pi(Sh_j)) \leq 3(K+1)N^* \qquad \square$$

3.3 The problem in 2 dimensions

We will reduce the problem of maintaining a shell to a dynamic point location in $\mathbf{R}^2$ with an appropriate set of operations.

Consider a shell $Sh \subset [0,1]^2 \times (-L, L-1)$ where L is a sufficiently large number. For convenience, empty Sh is initialized with three special points: $(1, 1, -L)$, $(0, 1, L-1)$

and $(1,0,L)$. The polytope $P(Sh)$ consists of points from $[0,1]^2 \times [-L,L]$ that are dominated by at least one point in Sh. Because each point p of Sh belong to its *surface*, it also belongs to the boundary of $P(Sh)$. The *roof tile* of p consists of the boundary points that share the third coordinate with p. Because no two input points have the same height, each roof tile contains exactly one such point. By projecting each point p in roof tile of p with $\iota(p) = (p_1, p_2)$ we obtain tile(p). These tiles overlap only on their perimeters, and because $(1,1,-L) \in Sh$, they cover the entire unit square $[0,1]^2$.

For p^i that maximizes p^i_3, tile(p^i) is the oriented rectangle with corners (0,0) and (p^i_1, p^i_2). Other tiles have rectangular cutouts, so in general they have the following shape: straight sides on the top and on the right, closed with a descending staircase line, where the concave corners are the projections of the input points.

The partition of $[0,1]^2$ by the tiles defines planar graph G_0, where nodes are the tile corners and faces are tiles. For the special tiles we need only these lists: the left one for $(1,1,-L)$, the right one for $(0,1,L)$ and the top one for $(1,0,L-1)$, because the other ones contain exactly one predictable element.

Our data structure will contain one record for each point, and this belongs simultaneously to three skip lists and some other data structures that will be needed for the point location. When we say that a list or a tree contains a point, we mean that it contains the record of this point. Our representation of a tile(p) consists of three lists of points:

left(p) contains the points that define the concave corners, sorted by the first coordinate; p is added as the head of the list;

top(p) contains the points that define the nodes on the top side, sorted by the first coordinate, starting from p but excluding the top left corner;

right(p) is defined like top(p), but for the right side.

Note that we can compute (q_1, p_2), the upper left corner of tile(p) by finding the first point q of the right list where p belongs (and similarly for the lower right corner). Before we explain other details of our data structure, we can show how we can count points of Sh that dominate a new given point.

Lemma 5 *Assume that $p \notin Sh$ and we know $q \in Sh$ such that $\iota(p) \in$ tile(q). Then we can compute $c =$ CODE($Sh, p, K-a$) in time $O((c+1)\log n)$.*

Sketch of proof. If $p_3 > q_3$, CODE$(Sh, p, K-a) = 0$ (regardless of the value of a). In the remaining case we can view Sh as the following directed acyclic graph G_{Sh}: (r,s) is an arc if s belongs to top(r), or s belongs to right(r) or r belongs to left(s); the elements of Sh that dominate p form a subgraph $D_{Sh}(p)$ where q is the only source node. One can make the following three observations. (1) q is the only source of D_{Sh}; (2) $D_{Sh}(p)$ is planar; (3) given $r \in D_{Sh}(p)$, then endpoints of arcs (r,s) where $s \in DSh(p)$ form contiguous parts of the lists top(r) and right(p). Observation (1) allows us to compute CODE by performing the traversal of $D_{Sh}(p)$, where this traversal is interrupted if we encounter $K-a$ nodes. Observation (2) shows that the number of edges in $D_{Sh}(p)$ is linear in the number of nodes, so it suffices if we use $O(\log n)$ time per traversed edge and node. Observation (3) shows an efficient way of finding all arcs of $D_{Sh}(p)$ that start at some node r: in $O(\log n)$ we can find the head of the left list to which r belongs and check if it dominates p, in similar time we can find if top(r) contains any points that dominate p and point to the first such, if they exist, and the same can be done with right(r). □

To show that we can compute CODE in the required time we only need to show that we can perform point location (finding q from the claim of Lemma 7) in time $O(\log^2 n)$. To show a similar result for PLUG_IN, we will also need to show that we can efficiently updated our data structure.

Our method of point location is an adaptation of the technique of Goodrich and Tamassia [3]. We do not merely invoke their results for two reasons: first, the PLUG_IN operation does not directly correspond to any of the operations described in that paper and, second, the partition of the unit squares into tiles is much simpler than the general case of partition into monotone polygons, which allows us to use a much simpler data structure.

3.4 Dual pair of trees and cuts based on pre-order

As the first step, we "slice" each tile(p) into rectangles by extending horizontal line segments from the nodes on the list left(p) (the concave corners of tile(p)). Let G_1 be the resulting planar graph. Note that now the right side of a tile has two sets of nodes: nodes from right(p) and nodes of the form (p_1, q_2) where q is a node on left(p). We will use the following notation for the resulting rectangles: $\mathbf{l}(p)$ is the lowest rectangle in tile(p), and if $q \in$ left(p), $\mathbf{u}(q)$ is the rectangle in tile(p) that has $\iota(q)$ on its lower side. One can see that the special points of Sh together define only one rectangle, namely $\mathbf{l}((1,1,-m))$, and each input point defines exactly two.

As the second step, we define a spanning tree T_1 of G_1 and its dual tree T. Let $(-x_2,-x_1)$ path of u be a path in G_1 that starts from u and where a subsequent edge is chosen, if possible, to decrease the second coordinate, and otherwise to decrease the first one; such a path always terminates at the point (0,0) (later we will also use $(+x_2,+x_1)$ paths, which are defined similarly). It is easy to see that the union of all $(-x_2,-x_1)$ paths forms a spanning tree T_1 with root (0,0).

The dual tree T is a graph where the nodes are the faces of G_1, i.e. the rectangles that partitioned the tiles. An arc of T is a pair of rectangles that have a non-T_1 edge e in common. We will say that this arc crosses e. We convert T to a dag: an arc that crosses a horizontal edge is directed down.

Lemma 6 *(1) No arc of T crosses a vertical edge; (2) T is a tree, where each node of the form $\mathbf{u}(p)$ has two children, and each node of the form $\mathbf{l}(p)$ is a leaf.*

Sketch of proof. (1) Consider a vertical edge e. It is clear that it is a part of $(-x_2,-x_1)$ of its upper end. Therefore $e \in T_1$ and e cannot be crossed by an arc of T. Note that now the direction of every arc of T is properly defined.

(2) Let p_{high} be the input node in Sh with the highest value of p_3. We will show that $(2i)$ no arc starts at $\mathbf{u}(p_{\text{high}})$, $(2ii)$ exactly one arc starts at any other rectangle, $(2iii)$ no arc ends at $\mathbf{l}(p)$ and $(2iv)$ two arcs end at $\mathbf{u}(p)$. Assume that we consider the rectangle $\mathbf{r}$ shown in Fig. 3. $(2i, ii)$ Let (y,v) be the rightmost edge on the top side of $\mathbf{r}$. If $y \neq w$, the path (y,w) begins to the $(-x_2,-x_1)$ path of y, thus on the top side of $\mathbf{r}$ only the edge (y,v) does not belong to T_1 and can be crossed by an arc of T. If $\mathbf{r} = \mathbf{u}(p_{\text{high}})$, this edge coincides with the top side of $[0,1]^2$, so

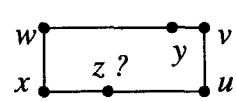

Figure 3: Example of a face of G_1.

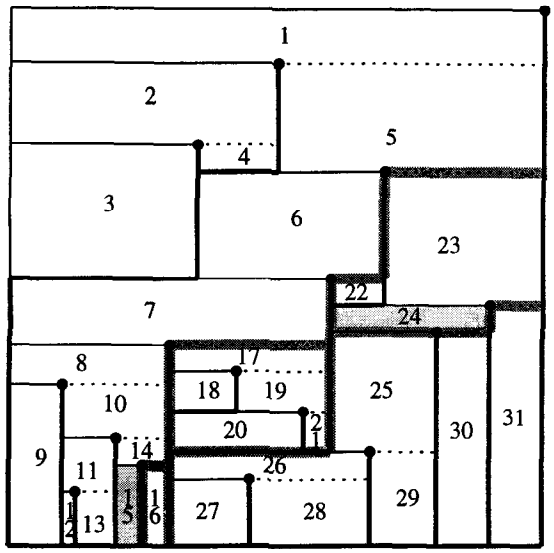

Figure 4: Dots mark the projections of the input points. Solid lines partition the square into tiles, and dashed lines partition tiles into rectangles. Thick lines show tree T_1. Each rectangle contains its preorder number. Wide gray lines show the cuts defined by rectangles 15 and 24.

there is no neighbor on the other side; in every other case the respective arc exists. (*2iii*) If $\mathbf{r} = \mathbf{l}(p)$ for some point p, then (u, x) is the first edge of $(-x_2, -x_1)$ path of u, and thus no arc crosses the bottom side of $\mathbf{r}$. (*2iv*) If $\mathbf{r} = \mathbf{u}(p)$, then the bottom side of $\mathbf{r}$ contains $z = \iota(p)$ as the only internal node. One can see that $(-x_2, -x_1)$ paths of u and z both start with vertical segments (right sides of tiles extend down from both nodes), and thus neither (x, z) nor (z, u) belong to T_1. Consequently, both edges are crossed by arcs of T. □

Because T is such a nice binary tree, a point locating structure can be defined simpler than in [3], where Goodrich and Tamassia use link-cut tree data structure of Sleator and Tarjan [8] to maintain dynamically a hierarchical partition of their tree (of triangles). Instead, we can base our structure on a simple one-dimensional binary search.

We define this structure in three stages: in terms of the pre-order listing of T (where we can use one-dimensional binary search access), in terms of T, and finally in terms of paths in G_1 that partition the unit square. A rectangle $\mathbf{r}$ splits the pre-order listing into two parts: the left part, from the beginning up to $\mathbf{r}$ (inclusive), and the right part. To obtain a balanced system of partitions of the set of faces of G_1 it suffices to choose any balanced binary search tree for the preorder listing.

The partition of T corresponding to the split according to $\mathbf{r}$ is the following: all proper descendants $\mathbf{r}$ (if any) are placed in the right part, and $\mathbf{r}$ in the left part. Each proper ancestor $\mathbf{s}$ of $\mathbf{r}$ is placed in the left part; if the left (right) subtree of $\mathbf{s}$ does not contains $\mathbf{r}$ then it is placed in the left (right) part, otherwise its split is already defined.

The partition of the unit square corresponding to this split can be described as the *cut of* $\mathbf{r}$, which consists of two directed paths, $D(\mathbf{r})$ and $U(\mathbf{r})$. Suppose that rectangle $\mathbf{r}$ is shown on Fig. 3. $D(\mathbf{r})$ contains the $(-x_2, -x_1)$ path of x, and $U(r)$ contains the $(+x_2, +x_1)$ of u. If $\mathbf{r}$ is a leaf of T, we include the bottom side of $\mathbf{r}$ in $D(\mathbf{r})$, otherwise we include it in $U(\mathbf{r})$.

If we use the cut of $\mathbf{r}$ to split an area between the cuts of, say, $\mathbf{s}$ and $\mathbf{t}$, then we truncate $D(\mathbf{r})$ ($U(\mathbf{r})$) by removing the portion which would otherwise be shared with $D(\mathbf{s})$ or $D(\mathbf{t})$ (with $U(\mathbf{s})$ or $U(\mathbf{t})$). It will be convenient to define a directed graph G_2 in which we can find every possible path of the form $U(\mathbf{r})$ or $D(\mathbf{r})$. The horizontal edges of G_2 are those of G_1, directed to the left if they belong to T, and to the right otherwise. The vertical edges are defined by the rectangles: the right sides form the upward edges, and the left sides of the left children the downward ones.

It is easy to see that G_2 has $O(n)$ edges; the fact each of these edges belongs to one cut only allows us to represent all the cuts needed for point location using $O(n)$ space.

3.5 Point location

If we apply a standard balance tree to the preorder listing of rectangles, the analysis is rather complicated because a single rotation may redefine some long cuts, an expensive operation. Therefore we adapt the idea of skip lists of Pugh [10]. To each rectangle we will assign a random number, and a rectangle with the largest number will be the root of the binary search; this idea is applied recursively to the resulting split. By assigning the random numbers to input points, we will be able to use the skip list approach to all our data structures.

When we receive a point p, we give it a *horoscope* $h(p)$, an independent random integer such that $\Pr(h(p) = i) = 2^{-i}$ for $i > 0$. Later we assign $h(p)$ also to rectangles $\mathbf{u}(p)$ and $\mathbf{l}(p)$. The hierarchical partition of the unit square defined by the random function h is defined as follows: given an area A (initially, the entire square) where rectangle $\mathbf{s}$ has the largest pre-order number, we pick $\mathbf{r} \subset A - \mathbf{s}$ that has the maximal $h(\mathbf{r})$ (breaking ties arbitrarily) and then we partition A using the cut defined by $\mathbf{r}$.

Given a new point p, we can query whether $\iota(p)$ is on the left or on the right side of a given cut. One can use the analysis contained in [10] to show that the average number of queries is at most $2 \log_2 n$.

To compare a new point p with a cut, we first perform a search for the vertical segment of the cut, $[(r_1, r_2), (r_1, s_2)]$ such that $r_2 < p_2 < s_2$. Then to decide whether p is in the left or on the right side of the cut, we compare p_1 with r_1. Our representation of a cut allows us to find such vertical segment in time $O(\log n)$, on the average.

We will describe in detail the representation of $D(\mathbf{r})$ when $\mathbf{r} = \mathbf{l}(q)$, and the way it can be used; the representation of $U(\mathbf{r})$ is simpler and the case of $\mathbf{r} = \mathbf{l}(q)$ is very similar. Both parts of the cut, $D(\mathbf{r})$ and $U(\mathbf{r})$, at the lower right corner of $\mathbf{r}$, say (q_1, r_2) (assuming that q belongs to top(r)). The course of the downward path, $D(\mathbf{r})$, can be described as follows (we will identify the lists of tiles with the respective sides). $D(\mathbf{r})$ enters top$(q = q^0)$, then it leaves it to enter some right(q^1), then it leaves it to enter some top(q^2) etc. Each time $D(\mathbf{r})$ enters a side of a tile, it leaves it only if it is the highest value of $h(\mathbf{r})$ (and wins the tiebreaks) among the downward paths that entered that side. We represent $D(\mathbf{r})$ as a skip lists of given points $(q^0, q^1, q^2, \ldots)$ that define the sides that this path has left, i.e. we do not represent explicitly the side in which $D(\mathbf{r})$ terminated. Given a new point p, we first compare p_2 with q_2^0; if it is lower, then $D(\mathbf{r})$ decides on which side of the cut of $\mathbf{r}$ the point (p_1, p_2) is located. We first look for i such that $q_2^{2i} < p_2 < q_2^{2i+2}$, if we can find it, we compare p_1 with q_1^{2i+1}. If we cannot find it, we have two cases: (i) the last entry in the representation of $D(\mathbf{r})$ is q^{2i} and $p_2 < q_2^{2i}$; in this case we can find q^{2i+1} knowing that

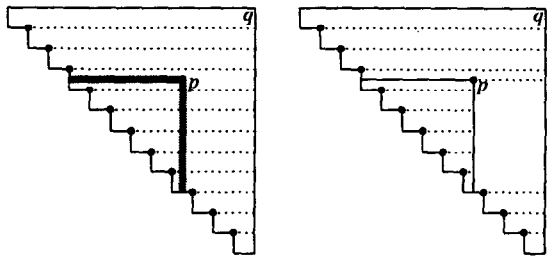

Figure 5: Example of a simple insertion.

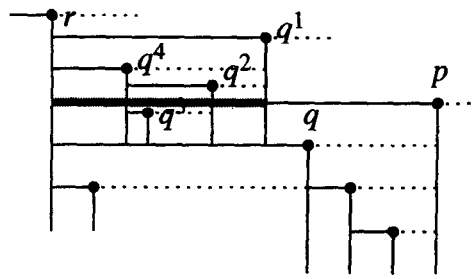

Figure 6: Extending the top rectangle of the new tile.

$q^{2i} \in \text{right}(q^{2i+1})$ and to compare p_1 with q_1^{2i+1}. (ii) the last entries are q^{2i} and q^{2i+1}. In this case we can find q^{2i+2} because $q^{2i+1} \in \text{top}(q^{2i+2})$. If $q_2^{2i} > p_2 > q_2^{2i+2}$, we compare p_1 with q_1^{2i+1}, and if $p_2 < q_2^{2i+2}$, $\iota(p)$ belongs to the right side.

The above discussion, together with Lemma 5, allows us to conclude the following:

Lemma 7 *Given a new point p, we can (1) find in expected time $O(\log^2 n)$ the point $q \in Sh$ satisfying $\iota(p) \in \text{tile}(q)$, and (2) compute $c = CODE(Sh, p, K-a)$ in time $O((c+1)\log^2 n)$.*

3.6 Inserting a new point into a shell

If for some $q \in Sh$ the new point p satisfies $\iota(p) \in \text{tile}(q)$ and $p_3 > q_3$, we have to insert p and move points that are dominated by p to the next shell (operation PLUG_IN). We have three difficulties. First, unlike the points of the shell that dominate the new point, the dominated points (more precisely, the union of their tiles) do not form a contiguous figure. Second, moreover, as we can see in Fig. 5, a single insertion of a point can change an arbitrarily large number of rectangles in a shell, and because of that, an arbitrarily large number of cuts. Both difficulties can be overcome by augmenting the data structure that we use to represent a shell. Lastly, as we can see in Fig. 6, we may be forced to change an arbitrarily large number of tiles of points that are not being removed. To analyze the running time needed to handle such changes, we must introduce the amortization function π.

The process of insertion, construction of the new tile and the removal of dominated points is performed in stages, and each stage takes $O(\log^2 n)$ steps. To "pay" for a stage, we will either perform the initial insertion, or we will delete a point dominated by the new one, or we will decrease the value of $\pi(Sh)$.

Assume that we insert the new point p and $\iota(p) \in \text{tile}(q)$. Initially we perform this insertion as if p_3 was smaller than the r_3 for every $r \in \text{left}(q)$. It is easy to see that $\text{top}(q)$ and $\text{right}(q)$ remain unchanged, and that we replace a fragment of $\text{left}(q)$ with the single node p. In turn, the removed fragment of $\text{left}(q)$ becomes the entire $\text{left}(p)$. Both $\text{top}(p)$ and $\text{right}(p)$ contain only p and one neighbor of p in $\text{left}(q)$. We also need to insert p to one top and one right list. This constant number of operations on skip lists can be performed in $O(\log n)$ time.

The update of cuts is relatively simple. Ignore at first the fact that we need to create two new cuts. $D(\mathbf{r})$ paths of cuts remain unchanged after this operation, but a number of $U(\mathbf{r})$'s are affected. However, only one of $U(\mathbf{r})$s will have altered representation, namely the one that will be leaving $\text{top}(p)$. We can find this path in time $O(\log n)$ if we store an extra piece of information for every point on a left list; it will allow us to recompute in logarithmic time the "winner" $U(\mathbf{r})$ among paths that enter a given $\text{right}(r)$ from the left (i.e. via a node that is implicitly represented on $\text{left}(r)$ list). More concretely, after we have spliced $\text{left}(q)$ into new $\text{left}(p)$ and the remaining $\text{left}(q)$, we need to calculate the "winner" of the new list.

Finally, we need to build the cut paths for the two new rectangles, $\mathbf{l}(p)$ and $\mathbf{u}(p)$. We build representations of these paths step by step. To add another segment to the representation, we need to make a winner computation, delete this segment from the path of the previous winner and to update the "pieces of information" that allow us to compute the "winners". Together, this is $O(\log n)$ steps. Moreover, the probability that a new path has K segments is $O(1/K)$, because for that to happen the new node must obtain a horoscope value matching that of at least K other points. Therefore the average length of new the paths is $O(\log n)$, and so the cost of creating the new cuts is $O(\log^2 n)$.

After the simple insertion, we must check if for any $r \in \text{left}(p)$ we have $r_3 < p_3$, in which case r has to be removed. To find such points efficiently, the list $\text{left}(p)$ must have the capabilities of a priority tree, where the third coordinate is the priority (with the least value on top). In turn, we can delete them one at the time, in order of increasing r_3. One can see that the deletion of a node that has smaller third coordinate than its neighbors on the left list of p is essentially the inverse of the simple insertion, and as such has a very similar cost.

Besides deletions of concave corners, we may need to expand the top and bottom rectangles in the tile of p. Fig. 6 shows such a situation. In this figure, we assume that p is the new point, q is the top concave corner of $\text{tile}(p)$ and that $q_3^1 < p_3$. Because we did not delete q, we know that $q_3 > p_3 > q_3^1$, and thus $q^1 \in \text{top}(q)$. Because the third coordinates of the points on this list have increasing third coordinates, we can find in logarithmic time the last point that has smaller third coordinate than p_3. We have several cases.

In the simplest, the expansion of $\text{tile}(p)$ does not reach to the right side of $\text{tile}(r)$, and it does not delete any points. In this case, for $i = 1, \ldots, j-1$ we have $q_3^i < p_3$ and $q_2^i > p_2$, while $q_3^j > p_3$ and $q_2^j > p_2$. Then we extend the top rectangle of p up to the right side of $\text{tile}(q^j)$ which requires a finite number of changes in the lists of our data structure, e.g. a part of $\text{top}(q)$ is transferred to $\text{top}(p)$. If for some $a < j$ we have $q_2^a < p_2$, then we need to remove q^a. To find efficiently nodes that have to be removed, the list $\text{top}(q)$ must have the capabilities of a priority tree where the second coordinate is the priority (while the list is sorted according to the first and third coordinate). Because each deletion increases the size of the PLUG_IN operation, we can perform it in $O(\log^2 n)$

steps.

The second case is when we have to transfer entire top(q) to top(p).

In that case, we expand the top rectangle of p up to the right side of the tile of r that is adjacent to the tile of q from the left. It is clear that $r_3 > q_3 > p_3$, so if $r_2 > p_2$, we cannot extend tile(p) any further. However, it may happen that $r_2 < p_2$. In this case, the top rectangle of the tile of p is split horizontally and r becomes a concave corner of the tile of p. Now the process can begin again, with r having the same role as q before. This may repeat many times. To amortize the cost of such repetition, we introduce the potential function $\pi(Sh)$, defined as the number of top and right lists that contains more points than just the header. If the above process is not repeated, π decreases by 1 (top(q) looses all points except the header) or remains unchanged (top(p) could start contributing to π). However, each subsequent repetition surely decreases π. On the other hand, the only situation when π may increase occurs during a simple insertion, because the new point becomes a member of one right list and one top list; for this reason the size of a simple insertion is defined as 3 (so 3-2=1).

The cases related to the expansion of the bottom slice of tile(p) downward are totally symmetric. Note that to efficiently handle the cases related to collateral point deletions, we need to have extra data items on right(q) lists that are symmetric with those on top(q) lists.

We can conclude

Lemma 8 *We can compute a call to PLUG_IN in time proportinal to its size, times* $\log^2 N^*$.

Together with Lemmas 4, 7 and Theorem 2 this implies

Theorem 9 *If the hit intervals have two scores, we can winnow off K-contained hits in time $O((K \log^2 N^*)N)$, using $O(N^*)$ space.*

4 A Weaker Redundancy Condition

While the definition of redundancy discussed in the previous section seems rather natural, it is not the only definition worthy of consideration. There are a number of criteria for choosing an approach to winnowing, including naturalness of the basic concepts, appropriateness of experimental results, ease of implementation, and efficiency with computational resources. Other approaches to winnowing might be superior on some or even all of these criteria.

This section describes algorithms based on an alternative definition of redundancy. Here, a hit is considered redundant if each position in the hit-interval is contained in at least K hits of better score. We begin with an offline algorithm for one score ($M = 1$), which runs in $O(N \log N)$ time and $O(N)$ space. The general result of Section 2 then provides a batch algorithm with running time $O(N \log N^*)$ using $O(N^*)$ space. We then present an online algorithm using $O(KN \log N)$ time and $O(KN)$ space. As discussed in Section 2, this leads to an algorithm for the case $M = 2$. For our initial descriptions of these algorithms, we assume all the hits have different scores. Ultimately we show how to accommodate identical scores.

4.1 Offline Algorithm when M = 1

As with the earlier offline algorithm, we inspect the hits by non-increasing score. Each time a hit is inspected, we check

```
1.  T ← ∅
2.  for each hit h by decreasing order of score do
3.      x ← max{t ∈ T : t ≤ h.left}
4.      y ← min{t ∈ T : t ≥ h.right}
5.      if min{C(u) : u ∈ No((x, y))} ≥ K then
6.          discard h
7.      else
8.          insert h.left and h.right into T
                if they are not already there
9.          update D(u) for u ∈ No(h)
                and nodes on the search paths
```

Figure 7: Offline Algorithm

if every point of the hit is covered by more than K earlier hits.

To do this in $O(N \log N)$ time, we maintain a balanced binary search tree whose leaves are the endpoints, *h.left* and *h.right*, of non-redundant hits h inspected so far. These endpoints partition the "line" covering query-sequence positions into intervals $(0, x_1], (x_1, x_2], \ldots, (x_{K-1}, x_K]$, where $(x, y]$ denotes the points z satisfying $x < z \leq y$. We store those intervals at leaves of the tree using *h.right* as search key. Each internal node u represents the union of all the intervals in the leaves of the subtree rooted at u, which is an interval denoted $I(u)$.

For each hit, define $No(h)$ to be the set of nodes u such that (1) h contains $I(u)$ and (2) either u is the root or h does not contain $I(p(u))$, where $p(u)$ denotes the parent of u. Let $H(u) = \{h : u \in No(h)$. At each node u we keep a number $D(u)$, which is the minimal coverage of each point in $I(u)$ by hits that do not contain $I(u)$. Given the values of D at the children v and w of node u, $D(u)$ can be computed by:

$$D(u) = \min(D(v) + |H(v)|, D(w) + |H(w)|)$$

The minimal coverage $C(u)$ of any point in $I(u)$ can be computed by adding $D(u)$ and the sizes of H-sets of u and its ancestors. Each node u stores the two numbers $D(u)$ and $|H(u)|$; $C(u)$ is computed on the fly.

We present the offline algorithm in Figure 7. Each time a new hit h is inspected, we find the tree entries immediately to the left (x) and right (y) of h, then determine $No((x, y))$. Since all the nodes in $No((x, y))$ are either on the search paths for x and y or are children of nodes on those paths (see Figure 8), the size of $No((x, y))$ is at most $O(\log N)$, and in $O(\log N)$ time we can use the $C()$ to determine the minimal coverage of h. The endpoints of h are inserted in T if the minimal coverage is less than K. After an insertion, only the D values at nodes along the search paths and in $No(h)$ will be changed, and the insertion and balancing can be done in $O(\log N)$ time. This completes the description of the offline algorithm, from which the batch algorithm follows as discussed in Section 2.

4.2 Online Algorithm

Now we give an online algorithm based on the above offline version; it has a running time of $O(KN \log N^*)$ using $O(KN^*)$ space. This will lead to an efficient batch algorithm when $M = 2$, as discussed in Section 2.

We use the same balanced search tree structure to represent intervals. The two differences are that at each node of the tree we need more information, and that we need to deal with deletions of hits.

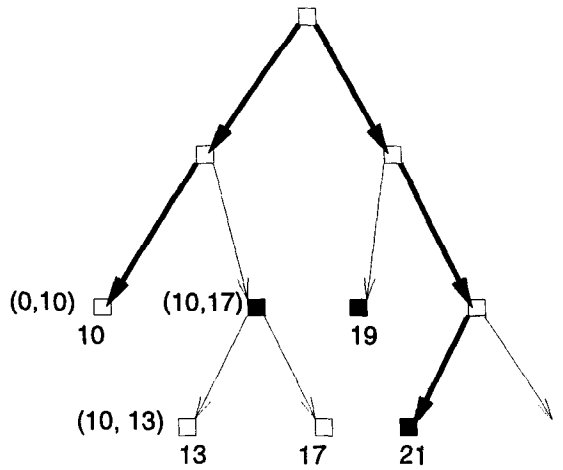

Figure 8: When the hit $(10, 21)$ is inserted, it is be divided into subintervals represented by the filled nodes. The dark lines are the search paths for the two endpoints. The search key is the right endpoint of the interval represented at the leaves. Thus $No((10, 21))$ includes nodes to the right of the left path, nodes to the left of the right path, and the node labeled "21".

At each node u we store a list, ordered by score, of at most K items $a_i^u = (x_i, y_i)$ for $1 \leq i \leq K$, where each position of $I(u)$ is covered by at least i hits that do not contain $I(p(u))$ and with score at least x_i. Also, y_j satisfies the following: it is either (1) a hit $h \in H(u)$, such that x_j equals the score of h or (2) it is a double pointer pointing to a_i^v and a_i^w for some i where v and w are children of u, and x_j is the smaller of the scores of a_i^v and a_i^w. Henceforth, when we refer to a list, we treat it as a list of score, but the reader should note that the implicit information is available and, in fact, used.

A key observation behind the online algorithm is that from a^v and a^w, where v and w are children of u, and $H(u)$, we can obtain a^u in $O(K)$ time. This is done by merging $H(u)$ and the list $\{\min(a_i^v, a_i^w)\}_{i=1}^K$.

Note the similarity between a^u and $D(u)$. From the observation in the previous paragraph, it is clear that the algorithm in Figure 7 solves the online problem, except now at line 2 we inspect the hits by the order of input. Also, at lines 5 and 9 we deal with lists of size $O(K)$, so there is an extra factor of K in the running time. Thus, each insertion takes $O(K \log N^*)$ time. Now we need to maintain $H(u)$, not merely its size. (We call each copy of the hit h in $H(u)$ a *subhit* of h.) While the online algorithm does not delete hits from the intermediate result, to guarantee that $|H(u)| = O(K)$ we may need to delete one subhit for each insertion into an $H(u)$. This does not affect the running time.

Thus, for $M = 2$, we have an offline algorithm that finds H' in $O(N \log N + KN \log |H'|)$ time using $O(N + K|H'|)$ space. This gives a batch algorithm that runs in $O(KN \log N^*)$ time using $O(KN^*)$ space.

4.3 Duplicate scores

Here we will mention how to solve the online problem when different hits can have the same score. The solution for the offline problem is similar and easier.

The method is to change the structure of the lists. At a^u for each u, instead of the list of scores, we have a list of scores and counters. For each item in the list, we include a count of the number of times the score appears in the original list. This guarantees that the length of the new list is at most K at each node. We do the same for each $H(u)$. All the operations keep the same asymptotic running time, and the space requirement is also unchanged.

	gi no.	score	start	end	comment
1	2127935	1042	1	510	query
2	2649642	601	4	510	ortholog from *A. fulgidus*
3	2621238	368	15	509	ortholog from *M. therm.*
4	2621136	322	19	480	ortholog from *M. therm.*
5	1945287	225	183	505	glutamate synthase domain
6	78469	220	181	491	glutamate synthase domain
...					
50	2129203	48	12	67	ferredoxin domain
...					

Figure 9: Blast hits to the α-subunit of *M. jannaschii* glutamate synthase.

5 Examples

Two criteria for redundancy have been considered, namely when a hit-interval is contained in K higher-scoring intervals (Section 3) and when each of the interval's points is contained in K higher-scoring intervals (Section 4). We have experimented with both approaches to winnowing in the Blast program [1, 2] for searching protein-sequence databases. Currently, Blast provides winnowing based on the first approach, i.e., an interval is not reported if it is contained in K intervals of higher score. The following examples illustrate the effectiveness of this winnowing.

The α-subunit of *Methanococcus jannaschii* glutamate synthase, GenPept gene identification (gi) number 2127935, is a protein of 510 amino acids. The C-terminus is a large glutamate synthase domain, while the N-terminus contains a ferredoxin domain. Blast search without winnowing produces 108 hits with e-value ≤ 0.1, namely the query sequence itself, then three full-sized orthologs, followed by a number of sequences containing a glutamate synthase domain or, starting with the 50th hit, a ferredoxin domain. See Figure 9. (No sequence after the first four hits contained both types of domain.) When winnowing is turned on with $K = 5$, most of the glutamate synthase hits are culled out, so the first ferredoxin hit rises from 50th to 13th place, where it is more likely to be noticed.

A similar situation holds with the human Werner syndrome protein (gi number 1280208; 1432 amino acids), which contains a large helicase domain and a much smaller exonuclease domain [7, 6]. Regular Blast search puts the first exonuclease domain hit (*Drosophila melanogaster* egl protein; gi no. 1899075) at the 136th place, while winnowing with $K = 10$ moves it to 35th. Thus, like the first example, winnowing significantly improves the rank of an interesting hit and accordingly facilitates the identification of the pertinent relationship.

6 Final Remarks

After this paper was finished, we learned that the techniques of Gupta *et. al.* [4] can be used for the case with $k > 1$ scores to winnow 1-contained hits in time $O(N \log N^* \log^k \log N^*)$. Therefore a very natural question to invetigate is whether this approach can be extended to the case of K-contained

hits, where $K > 1$, and whether we can achive this running time with memory resticted to $O(N^*)$.

7 Acknowledgments

P.B was supported by NSF grant CCR 9700053. Z.Z. and W.M were supported by grant LM05110 from the National Library of Medicine. P.B. would also like to thank Department of Mathematics and Computer Science of Warsaw University for its hospitality.

References

[1] Altschul, S., W. Gish, E. Myers, W. Miller and D. Lipman (1990) Basic local alignment search tool. *J. Mol. Biol.* **215**, 403–410.

[2] Altschul, S., T. Madden, A. Schäffer, J. Zhang, Z. Zhang, W. Miller and D. Lipman (1997) Gapped BLAST and PSI-BLAST — a new generation of protein database search programs. *Nucleic Acids Res.* **25**, 3389–3402.

[3] Goodrich, M.T., and R. Tamassia (1991), Dynamic trees and dynamic point location, *Proc. 23rd STOC*, 523–533.

[4] Gupta, P., R. Janardan, M. Smid and B. DasGupta (1997) The rectangle enclosure and point-dominance problems revisited. *International Journal of Computational Geometry & Applications* **7**, 437–455.

[5] McCreight, E. M. (1982) Priority search tree. *SIAM J. Comput.* **11**, 149–165.

[6] Mian, I. S. (1997) Comparative sequence analysis of ribonucleases HII, III, II PH and D. *Nucleic Acids Res.* **25**, 3187–3195

[7] Mushegian, A. R., D. E. Bassett Jr, M. D. Boguski, P. Bork and E. V. Koonin (1997) Positionally cloned human disease genes: patterns of evolutionary conservation and functional motifs. *Proc. Natl. Acad. Sci. USA* **94**, 5831–5836

[8] Sleator, D.D and R.E. Tarjan (1983), A data structure for dynamic trees, *JCSS* **26**, 362–391.

[9] Sohnhammer, E.L.L., and R. Durbin (1994) A workbench for large-scale sequence homology analysis. *CABIOS* **10**, 301–307.

[10] Pugh, W. (1990), Skip lists: a probabilistic alternative to balanced trees, *Communications of ACM* **33**, 668–676.