GOING AGAINST THE GRAIN

Gene Myers[1,2] Mudita Jain[1,2]

[1] Dept. of Computer Science, University of Arizona, Tucson, AZ 85721
[2] Partially supported by NLM Grant LM-04960.

Abstract. We review a general, space and time efficient technique for delivering a sequence of values computable by a recurrence relation, in the order *opposing* the data-dependencies of the recurrence. The technique provides a series of time/space tradeoffs we characterize by parameter $K > 0$. Namely, N values can be delivered against the grain in $O(KN)$ time and $O(KN^{1/K})$ space. This basic idea is not new, but here we present it in a framework exposing its essential nature and we give a concise yet easily understood explanation of it in terms of counting in a radix-$N^{1/K}$ number system. We then show how to apply this paradigm to a couple of problems in sequence comparison, a domain where it has here to fore not been used. We show that in the limiting case of $K = \log_2 N$, the method coincides with the well-known divide-and-conquer algorithm of Hirschberg. Thus, our observation provides a continuum of time/space tradeoffs for all comparison problems that have appealed to the Hirschberg paradigm for greater space efficiency, e.g., sequence comparison with concave weights, approximate matching of regular expressions, etc. In particular, an $O(MN+(M+N)W\log(M+N))$ algorithm for computing a representation of all suboptimal alignments is a corollary of our technique with the important property that only $O(M + N + W\log(M + N))$ working storage is required versus the $O(M + N + F)$ space of an earlier result by Chao. Here F is the size of a graph modeling all of the suboptimal alignments, and W is its width. Since F may be $O(MN)$ this savings can be important in contexts where only summaries of the suboptimal region is required, e.g., the number of suboptimal paths with score greater than some threshold. Another application to computing locally optimal alignments as defined by Sellers illustrates the broad utility of the technique.

1 The Paradigm

1.1 Introduction

Consider a one-variable recurrence:

$$f_n = \begin{cases} h(f_{n-1}, n) & \text{if } n > 0 \\ d_0 & \text{otherwise} \end{cases}$$

where the values f_n are from domain $\mathcal{D}$, d_0 is a value in $\mathcal{D}$, and $h : \mathcal{D} \times \aleph \mapsto \mathcal{D}$ is a computable function. Further assume that $O(S)$ memory is required to store a value from the domain $\mathcal{D}$ and that $h(d, n)$ is computable in $O(T)$ time for any

value d and integer n. For such a recurrence, the sequence $f_0, f_1, \cdots f_{N-1}$ for some $N > 0$ can be easily computed in $O(TN)$ time and delivered in that order in $O(S)$ working storage. In this case the sequence of values to be delivered follows the "grain" or data-dependency of the recurrence. The problem we consider is one of "going against the grain" of the recurrence, that is, of delivering the sequence of values $f_{N-1}, f_{N-2}, \cdots f_0$ in reverse order.

An obvious approach is to simply compute *and store* the entire sequence of values with the grain in $O(TN)$ time and $O(SN)$ space, and then deliver the values in the reverse order. At the other extreme, one can deliver each value in reverse order, computing each from scratch with the grain in only $O(S)$ working storage. The obvious drawback is that the total time taken is $O(TN^2)$. We present an approach that produces the N values against the grain in $O(TKN)$ time and $O(SKN^{1/K})$ space, where $K > 0$. Thus our algorithm provides a series of time/space tradeoffs between the two extremes of $O(SN)$ space versus $O(TN^2)$ time. Starting at $K = 1$ we get the $O(TN)$ time, $O(SN)$ space algorithm, followed by a series of algorithms that end with an algorithm of complexity $O(TN \log N)$ time, $O(S \log N)$ for $K = \log_2 N$. This last algorithm in the series has the best possible space-time product. Note that as the series does not continue beyond $K = \log_2 N$ we do not meet the $O(TN^2)$ time, $O(S)$ approach at the other extreme of the spectrum.

One might characterize the approach as a K-level geometric partitioning of a problem. This paradigm has appeared several times in the context of specific problem domains. For example, we believe it was first used by Bentley and Maurer in "dynamizing" static search problems [1], later by Myers in realizing persistent arrays [7], and by Kannan and Myers in finding twins [5]. However the paradigm has never been presented as one of *going against the grain* and has not been applied in a general way to sequence comparison. Moreover, in the next two paragraphs we concisely and cleanly describe the idea by appealing to the idea of counting in a radix-$N^{1/K}$ number system. Typically, this idea is described as a series of results, first for $K = 2$, then $K = 3$, and then the inductive case, in lengthy expositions that are further complicated by the details of a specific application.

1.2 The Algorithm

Our approach counts down from $N - 1$ to 0 delivering the values in sequence. At the time a given value f_n is delivered, a specific subset of f-values has been computed and cached to assist in the efficient delivery of subsequent values. The simplest way to describe the set of cached values is to consider representing the integers from 0 to $N - 1$ in a radix $R = N^{1/K}$ number system. Let $(i_{K-1} i_{K-2} \cdots i_0)_R$ be a radix-R number. Recall that each digit i_k is a number between 0 and $R-1$ and the integer represented is $\sum_{k=0}^{K-1} i_k R^k$. For example, if $N = 625$ and $K = 4$, then $R = 5$ and $(2431)_5$ denotes the integer 366. In our algorithm, upon delivery of the 366^{th} value, the cache contains the values f_c where $c \in \{(x000)_5, (2x00)_5, (24x0)_5, (243x)_5\}_{x=1}^{4}$. In general, when f_n has just been delivered for $n \equiv (i_{K-1} i_{K-2} \cdots i_0)_R$, then the algorithm has currently cached the

values f_c where $c \in \{(x0^{K-1})_R, (i_{K-1}x0^{K-2})_R, \cdots, (i_{K-1}i_{K-2}\cdots i_1 x)_R\}_{x=1}^{R-1}$. Simple counting reveals that exactly $K(N^{1/K}-1)$ values are cached at any one time and so our space complexity claim follows. Since f_n is in the set of cached values it can certainly be delivered once the cache has been updated.

So it only remains to investigate the work involved in updating the set of cached values when n is decremented. Suppose $n \equiv (\alpha t 0^k)_R$ where t is the rightmost non-zero digit and α represents the sequence of the leftmost $K-k-1$ digits. Then $n-1 \equiv (\alpha(t-1)(R-1)^k)_R$. Thus updating the cache requires discarding the cached values $\{(\alpha t 0^{k-j-1}x0^j)_R\}_{x=1}^{R-1}$ for each $j<k$ and replacing them with $\{(\alpha(t-1)(R-1)^{k-j-1}x0^j)_R\}_{x=1}^{R-1}$. For each choice of j the new cache values can be computed with $O(S)$ working storage by computing the "panel" f_c to $f_{c+R^{j+1}-1}$ where $c \equiv (\alpha(t-1)(R-1)^{k-j-1}0^{j+1})_R$ in forward order in $O(TR^{j+1})$ time and saving the needed values. For a given level $j<K$, such an update occurs only when the j^{th} digit transits from 0 to $R-1$, so the values computed in a level-j panel are disjoint. Thus for each level a total of $O(TN)$ time is taken as n is counted down from $N-1$ to 0. Thus the total time is $O(TKN)$ over all K levels. Appendix A gives a detailed pseudo-code for the algorithm.

1.3 Practical Considerations

With regard to choosing K, note that in asymptotic terms the time taken is $O(TN)$ for fixed small choices of K, e.g., 2, 3, or 4, whereas space complexity improves, e.g., $O(SN^{\frac{1}{2}})$, $O(SN^{\frac{1}{3}})$, or $O(SN^{\frac{1}{4}})$. The best possible space consumption, $O(S\log N)$, occurs when $K=O(\log N)$ for which choice the time is $O(TN\log N)$. Another interesting choice of K is $O(\log N/\log N\log N)$, giving $O(TN\log N/\log N\log N)$ time and $O(S\log^2 N/\log\log N)$ space.

In practice our result is best used when the amount of memory, say M, available for the task is known *a priori*. Then for a given N choose the smallest value of K such that the number of cached values will not be greater than M/S. In this way one gets the best possible time performance within a given space limitation. Table 1 gives an example of the largest N possible for each value of K assuming $M=1,000,000$ and $S=1,000$, $S=10,000$, and $S=N$, respectively. When $S=1,000$, 1,000 values can be cached and when $S=10,000$, only 100 can be cached. The final column is relevant to one of our forthcoming applications where two strings of length N are being compared. In this case $S=N$, i.e., as the length of the sequence to deliver in reverse order grows, the size of each value grows proportionally. The apparent anomaly at $K=9$ is due to rounding factors, namely R must in practice be chosen to be $\lceil N^{1/K}\rceil$ and this creates such non-monotone progressions as K approaches $\log_2 N$.

K	$S = 1,000$	$S = 10,000$	$S = N$
1	1,001	101	1,000
2	251,001	2,601	6,329
3	37,259,704	39,304	13,888
4	$> 10^9$	456,975	20,833
5	$> 10^9$	4,084,101	28,571
6	$> 10^9$	24,137,569	33,333
7	$> 10^9$	170,859,375	35,714
8	$> 10^9$	$> 10^9$	41,666
9	$> 10^9$	$> 10^9$	37,037
10	$> 10^9$	$> 10^9$	50,000

Table 1. Maximum problem sizes solvable with $M = 1,000,000$.

2 Applications to Sequence Comparison

2.1 Sub-Optimal Alignments

Consider the comparison of sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$ over alphabet Σ with respect to scoring scheme $\delta : (\Sigma + \varepsilon)^2 \mapsto \Re$. Polymorphically let $\delta(A, B)$ denote the score of the best alignment between A and B. Let A_i be the prefix $a_1 a_2 \cdots a_i$ and let A^i be the suffix $a_{i+1} a_{i+2} \cdots a_M$. Let $P(i, j) = \delta(A_i, B_j)$ be the score of aligning the prefixes A_i and B_j, and let $S(i, j) = \delta(A^i, B^j)$ be the score of aligning the suffixes A^i and B^j. The now classic dynamic programming algorithm for sequence comparison computes $\delta(A, B) = P(M, N) = S(0, 0)$ by computing an $(M + 1) \times (N + 1)$ table of one or the other of P or S using the well-known recurrences:

$$P(i,j) = \min \begin{pmatrix} P(i-1,j-1) + \delta(a_i, b_j) \\ P(i-1,j) + \delta(a_i, \varepsilon) \\ P(i,j-1) + \delta(\varepsilon, b_j) \end{pmatrix}$$

$$S(i,j) = \min \begin{pmatrix} S(i+1,j+1) + \delta(a_{i+1}, b_{j+1}) \\ S(i+1,j) + \delta(a_{i+1}, \varepsilon) \\ S(i,j+1) + \delta(\varepsilon, b_{j+1}) \end{pmatrix}$$

We assume the reader is familiar with the "edit graph" model of sequence comparison. Briefly, the graph has $(M + 1)(N + 1)$ vertices labeled $(i, j) \in [0, M] \times [0, N]$ arranged in a rectangular grid. There are alignment edges $(i, j) \to (i + 1, j + 1)$ of weight $\delta(a_i, b_j)$ modeling the alignment of a_i and b_j. Also, there are indel edges $(i, j) \to (i + 1, j)$ and $(i, j) \to (i, j + 1)$ of weights $\delta(a_i, \varepsilon)$ and $\delta(\varepsilon, b_j)$ that model leaving a_i and b_j unaligned, respectively. Within this framework, a best alignment between A and B is modeled by a least cost path from $(0, 0)$ to (M, N) and the score of such a path is $\delta(A, B)$. Moreover, $P(i, j)$ is the cost of a best path from $(0, 0)$ to (i, j), $S(i, j)$ is the cost of a best path from (i, j) to (M, N), and note carefully that $P(i, j) + S(i, j)$ is the cost of a best path from $(0, 0)$ to (M, N) that passes through (i, j).

A series of results in the literature concern enumerating all suboptimal paths that score within some threshold τ, typically near the optimum [13, 9, 2]. Since the number of such alignments grows very rapidly as τ moves away from the optimum, investigators have found it superior to depict the subgraph of the edit graph containing the paths representing the alignments [9]. Such depictions reveal invariant aligned segments common to all or many of the subpaths and give an interesting characterization of the similarity between two sequences. The subgraph G_τ containing all suboptimal paths of score not greater than τ consists of the vertices v such that $P(v) + S(v) \leq \tau$ and the edges $v \to w$ such that $P(v) + \delta(v \to w) + S(w)$ where $\delta(v \to w)$ is the weight of the edge.

Let $\vec{P}_j$ be the $(M+1)$-vector $[P(0,j), P(1,j), \cdots P(M,j)]$ and similarly define $\vec{S}_j$. Next observe that these values satisfy the recurrence relations for going against the grain. For example, for the $\vec{S}$ values/vectors:

$$\vec{S}_j \;=\; \begin{cases} h_S(\vec{S}_{j+1}, j) & \text{if } j < N \\ \vec{S}_N & \text{otherwise} \end{cases}$$

where $\vec{S}_N = [\,\sum_{k=i+1}^{M} \delta(a_k, \varepsilon)\,]_{i=0}^{M}$ and h_S computes the next column from the previous one using the fundamental recurrence above. Note that the sequences $\vec{S}_N, \vec{S}_{N-1}, \cdots \vec{S}_0$ and $\vec{P}_0, \vec{P}_1, \cdots \vec{P}_N$ are with the grain, i.e., the grain directions of S- and P-vectors oppose each other.

To deliver the vertices and edges of G_τ in column increasing order it suffices to compute the sequence $\vec{P}_0 + \vec{S}_0, \vec{P}_1 + \vec{S}_1, \cdots \vec{P}_N + \vec{S}_N$, i.e. the P-vectors with their grain and the S-vectors against theirs. Using our central idea we have an $O(KMN)$ time, $O(KMN^{1/K})$ space algorithm for doing so for any choice of $K > 0$. Note that G_τ does not need to be saved, it can be sent to a graphics device to be drawn or statistics about it can be gathered as the algorithm proceeds. Especially note that since G_τ is delivered in column order one can accumulate order-dependent information such as the number of suboptimal paths, the invariant alignment segments, or suboptimal alignments matching a pattern as in [2]. Since the size of G_τ, call it F, may be $O(MN)$ this feature of our algorithm is an advantage not possessed by the earlier $O(M + N + F)$ space algorithm by Chao which while achieving $O(MN + F \log\log W)$ time cannot deliver G_τ in column major order. By contrast the grain-based algorithm delivers columns in order and requires only $O(M \log N)$ space when K is chosen to be $\log_2 N$.

At this juncture we digress to make the observation that our approach generalizes the basic divide and conquer algorithm of Hirschberg [4]. To see this observe that in the case that $K = \log_2 N$, the set of S-vectors cached at any moment is exactly the set of mid-point vectors that would be on the recursion stack of the divide-and-conquer algorithm. So for this choice of K, going-against the grain can alternately be seen as basically an iterative version of the divide-and-conquer algorithm. So what's important about the grain-based algorithm, is that for different choices of K one gets a tradeoff in space and time, thus generalizing the divide-and-conquer approach. Indeed, in a recent independent discovery Grice *et al.* [3] also arrived at an algorithm like the recursive formulation given in Figure 1.

```
Procedure D_and_C(start,end: integer, p,s: array [0 ··· M] of real)
{   Var P, S: array [0 ··· N^{1/K}][0 ··· M] of real

        # p ≡ P⃗_start and s ≡ S⃗_end #

    If start + 1 = end then
      { Process column p + h_S(s, start)
        return
      }

    q ← (end − start)/N^{1/K}
    P[0] ← p
    S[N^{1/K}] ← s
    For j ← 1 to N^{1/K} − 1 do
      { For k ← (j − 1) · q + 1 to j · q do
          { p ← h_P(p, start + k)
            s ← h_S(s, end − k)
          }
        P[j] ← p
        S[N^{1/K} − j] ← s
      }

    For j ← 1 to N^{1/K} do
        D_and_C(start + (j − 1) · q, start + j · q, P[j − 1], S[j])
}

D_and_C(0, N, P⃗_0, S⃗_N)
```

Fig. 1. Recursive Version of Going-Against-The-Grain.

For example, when $K = 2$, the recursive formulation computes and saves the $\sqrt{N}$ column vectors $P[j] \equiv \vec{P}_{j \cdot \sqrt{N}}$ and $S[j] \equiv \vec{S}_{j \cdot \sqrt{N}}$ and then recursively computes the P- and S-vectors for columns between each pair $\vec{P}_{j \cdot \sqrt{N}}$ and $\vec{S}_{(j+1) \cdot \sqrt{N}}$. One can show that at the time a particular column, say $\vec{P}_j + \vec{S}_j$, is delivered, the recursion stack contains exactly the vectors that would be cached by the grain-based algorithm if one were going against the grain of both P and S. The recursive formulation has the advantage of permitting one to manipulate each vector based on the values of the columns at the division points *before* further recursive division because *both* P and S are effectively being computed against the grain. The grain-based algorithm has the advantage that it stores half as many vectors and one can easily extend it to be finger-based so that one may efficiently deliver a set of columns in any order desired. Since many sequence comparison results appeal to variations of Hirschberg's divide-and-conquer algorithm to deliver alignment in an efficient amount of space (e.g., [6, 8]), it follows that our central idea is immediately applicable and gives one the flexibility of a series of time/space tradeoffs parameterized by K.

Returning to the suboptimal paths problem, we now show how to refine our

approach to make its complexity sensitive to the width of G_τ. The source of inefficiency is that we compute all of $\vec{P}_j + \vec{S}_j$ for a given j, whereas all we need to know are the entries not greater than τ. In order to do so, we must move from computing column-vectors of values to anti-diagonals of values. That is, for $a \in [0, M+N]$ let $\vec{P}_a = [P(l, a-l), P(l-1, a-l-1), \cdots P(a-r, r)]$ where $l = \min(a, M)$ and $r = \min(a, N)$. Let $\mathit{left}(a) = \max_i \{P(i, a-i) + S(i, a-i) \leq \tau\}$ and let $\mathit{right}(a) = \min_i \{P(i, a-i) + S(i, a-i) \leq \tau\}$. We will now maintain only the portion of the vectors $\vec{P}_a$ and $\vec{S}_a$ between $\mathit{left}(a)$ and $\mathit{right}(a)$, i.e., let $\vec{P}^\star_a = [P(\mathit{left}(a), a - \mathit{left}(a)), P(\mathit{left}(a)+1, a - \mathit{left}(a) - 1), \cdots P(\mathit{right}(a), a - \mathit{right}(a))]$. It suffices to maintain just these portions of the vectors as all suboptimal path of score not greater than τ must pass through a vertex of the retained portion.

Consider a K-level version of the recursive algorithm. Its goal is to compute just the $\vec{P}^\star$ and $\vec{S}^\star$ portions of the P- and S-vectors. In a given call suppose $p \equiv \vec{P}^\star_a$ and $s \equiv \vec{S}^\star_b$ where $a < b$. It follows from the basic structure of the edit graph that for any $d \in [a, b]$:

$$\mathit{left}(d) \in [\max(\mathit{left}(a), \mathit{left}(b) + (d-b)), \min(\mathit{left}(a) + (d-a), \mathit{left}(b))]$$

and that the statement also holds if *left* is replaced by *right*. The basic structure being appealed to is that the paths in G_τ are connected and so from a given vertex such a path can proceed, in the most extreme case, either horizontally or vertically. The critical point is that these bounds on the range of the limits of $\vec{P}^\star_d$ and $\vec{S}^\star_d$ imply that they may be found by computing an area of the graph between $\vec{P}^\star_a$ and $\vec{S}^\star_b$ that is of size $O(W(b-a) + (b-a)^2)$ where W is the width of the largest starred vector, i.e., $W = \max_a(\mathit{left}(a) - \mathit{right}(a))$. It then follows that the algorithm takes $O(MN + K(M+N)W)$ time and $O(M+N+KW(M+N)^{1/K})$ space. Thus when $K = \log_2 N$ our sparse variation takes $O(MN + W(M+N)\log(M+N))$ time and $O(M+N+W\log(M+N))$ space.

We compare this with the $O(MN+F\log\log W)$ time and $O(M+N+F)$ space result of Chao. In worst case terms these algorithms are basically incomparable. However, in practice one expects $F = O(NW)$ and usually $W = o(M+N)$. Under these circumstances our algorithm takes $O(MN\log(M+N))$ time and $O(M+N)$ space, where as Chao's takes $O(MN)$ time and $O((M+N)W)$ space. While less time efficient, recall that our algorithm can deliver order-dependent statistics without storing G_τ and in practice we can pick the smallest K fulfilling a preset space requirement, for $O(MN)$ time performance in practice.

2.2 Seller's Locally Optimal Alignments

In the context of comparing protein sequences, one is frequently interested in whether or not there are substrings of each sequence that are unusually similar. In terms of the edit graph, this local alignments problem requires finding paths in the graph of unusually high score. In the basic approach of Smith and Waterman [12], the underlying scoring scheme is assumed to be negatively biased so that the expected value of any path is negative. One then asks for subpaths of maximum

(positive) score. This is easily accomplished by adding a 0 term to the recurrence for either $P(i,j)$ and/or $S(i,j)$.

The one potential difficulty of such an approach is that there may be many "interesting" local alignments. In the approach of Waterman and Eggert [14], this problem is resolved by finding the highest scoring path, removing it, finding the next highest path, removing it, and so on. While conceptually simple, this approach does have the disadvantages of requiring repeated recomputation of regions of the dynamic programming matrix and of requiring the space to store all previously reported local alignments.

Another unpublished proposal was made by Sellers in 1987 [10, 11] that is interesting and involves only a single pass. Let G_P be the graph whose edges give rise to the P-values at each vertex, i.e., $v \to w \in G_P$ iff $P(v)+\delta(v \to w) = P(w)$. Similarly, let G_S be the edges for which $S(w) + \delta(v \to w) = S(v)$. Sellers defines a path p as locally optimal if it is maximal with respect to the following properties: (1) all its edges are in $G_P \cap G_S$, (2) there does not exist a path in G_P ending at a vertex of p such that one of its edges is not also in G_S, and (3) there does not exist a path in G_S starting at a vertex of p such that one of its edges is not also in G_P. Sellers showed that the implications of this definition are that p is prefix- and suffix-positive and that any other path intersecting p has lesser or equal score. A path is prefix-positive if every prefix of the path has positive score. Clearly one can compute G_S and G_P in $O(MN)$ time and space, and it then follows that one can compute *all* locally optimal paths in a single $O(MN)$ time and space computation. Using our grain-based algorithm we will develop an $O(K^2MN)$ time algorithm that delivers Sellers locally optimal paths in column-order with only $O(KMN^{1/K})$ space.

We will say that a vertex v is *P-free* if it satisfies condition (2) above, i.e., there does not exist a path in G_P to the vertex such that one of its edges is not also in G_S. It then follows by induction that v is P-free iff for all edges $e \equiv w \to v$, $e \notin G_P$ or $e \in G_P \cap G_S$ and w is P-free. A similar recurrence (with opposing grain) holds for the definition of an *S-free* vertex. It follows next that an edge $e \equiv w \to v$ is on a locally optimal path if and only if v and w are both S- and P-free and $e \in G_P \cap G_S$. It remains to lever these recurrences in a computation.

Let $\vec{P}_j$ and $\vec{S}_j$ be column vectors of the P- and S-values as defined in the previous subsections, save that the recurrence is slightly altered by the introduction of a 0-term. Also let $\vec{I}_j^P$ be the $(M+1)$-vector $[$*Is_P-free*$(0,j)$, *Is_P-free*$(1,j), \cdots$ *Is_P-free*$(M,j)]$ of boolean values where *Is_P-free*(w) is true iff w is P-free. Similarly define $\vec{I}_j^S$. Now consider the following demand-driven chain of computations. To deliver the edges of optimal paths whose head vertex is in column j, it suffices to know $\vec{P}_{j-1}$, $\vec{S}_{j-1}$, $\vec{I}_{j-1}^P$, $\vec{I}_{j-1}^S$, $\vec{P}_j$, $\vec{S}_j$, $\vec{I}_j^P$, and $\vec{I}_j^S$. The P- and S-vectors permit one to determine which edges are in G_P and G_S, respectively. To deliver the optimal edges in column major order thus implies that we must deliver the P-, S-, I^P-, and I^S-vectors in order of increasing j. This is with the grain of both the P- and I^P-vectors and against that of the S- and I^S-vectors. We can deliver S-vectors in sequence against their grain as previously discussed.

Now note that to compute $\vec{I}_j^P$ requires knowing $\vec{I}_{j-1}^P$, $\vec{P}_{j-1}$, $\vec{S}_{j-1}$, $\vec{P}_j$, and $\vec{S}_j$. Thus to deliver I^P-vectors in increasing order of j, requires delivering P- and S-vectors in increasing order of j. Thus another thread of S-vectors, separate from the one needed to compute local path edges, must be computed against the grain. Finally, to compute $\vec{I}_j^S$ vectors against their grain involves with-the-grain computations to cache values. But to deliver $\vec{I}_j^S$ in decreasing order of j requires knowing $\vec{I}_{j+1}^S$, $\vec{P}_{j+1}$, $\vec{S}_{j+1}$, $\vec{P}_j$, and $\vec{S}_j$. Now S-values are with the given grain but the P-values are not, and so one must *recursively* run a grain-based subalgorithm for P-vectors within the grain-based algorithm for computing I^S-vectors. A careful analysis shows that the subalgorithm consumes $O(K^2MN)$ time in the process of carrying out the $O(KMN)$ steps of the I^S grain-based algorithm.

The analysis just completed has the interesting feature that the grain of optimal edge delivery opposes the grain of I^S-vectors which opposes the grain of P-vectors, and each depends in turn on the other. Thus we have doubly nested *grain-conflicts*. In general, one may show that for a multi-recurrence problem with at most a C-level nesting of grain-conflicts, the grain-based approach yields an $O(K^C NT)$ time, and $O(CKN^{1/K}S)$ space algorithm for delivering the outermost entity in the desired order. In the case of Seller's local alignment problem, $C = 2$ and we have an $O(K^2MN)$ time, $O(KMN^{1/K})$ space algorithm. For any fixed value of K, time is $O(MN)$ and space is $o(MN)$. For the limiting choice of $K = \log_2 N$, the result is an $O(MN \log^2 N)$ time, $O(M \log N)$ space algorithm.

This last example, of a two-tiered grain-based approach hopefully illustrates the utility of the going-against-the-grain framework. It is very awkward to express the demand driven computation above within a recursive divide-and-conquer framework and we doubt that one would have easily seen the optimization opportunity from such a conceptual vantage point.

References

1. Bentley, J., and H. Maurer, "Efficient worst-case data structures for range searching," *Acta Informatica* 13 (1980), 155-168.

2. Chao, K.M., "Computing all suboptimal alignments in linear space," *Proc. 5th Symp. on Combinatorial Pattern Matching* (Asilomar, CA 1994), 31-42.

3. Grice, J.A., Hughey, R. and D. Speck, "Parallel Sequence Alignment in Limited Space," *Proc. 3rd Conf. on Intelligent Systems for Molecular Biology* (Cambridge, England 1995), 145-153.

4. Hirschberg, D.S., "A linear space algorithm for computing longest common subsequences," *Comm. Assoc. Comput. Mach.* 18 (1975), 341-343.

5. Kannan, S. and E. Myers, "An algorithm for locating non-overlapping regions of maximum alignment score," *Proc. 4th Symp. on Combinatorial Pattern Matching* (Padova, Italy 1994), 74-86. Also to appear in *SIAM J. on Computing.*

6. Miller, W. and E. Myers, "Sequence Comparison with Concave Weighting Functions," *Bull. of Mathematical Biology* 50 (1988), 97-120.

7. Myers, E., "Efficient applicative data types," *Proc. 11th ACM Symp. on Principles of Prog. Lang.* (1984), 66-75.

8. Myers, E., and W. Miller, "Approximate Matching of Regular Expressions," *Bull. of Mathematical Biology*P 51 (1989), 5-37.

9. Naor, Dalit and D. Brutlag, "On suboptimal alignments of biological sequences," *Proc. 4th Symp. on Combinatorial Pattern Matching*(Padova, Italy 1994), 179-196.

10. Sellers, P.H., "Pattern recognition in genetic sequences by mismatch density," *Bull. Math. Biol.* 46 (1984), 501-514.

11. Sellers, P.H., personal communication.

12. Smith, T.F. and M.S. Waterman, "Identification of common molecular sequences," *J. Mol. Biol* 147 (1981), 195-197.

13. Waterman, M., and T. Byers, "A dynamic programming algorithm to find all solutions in a neighborhood of the optimum," *Math. Biosciences* 77 (1985), 179-185.

14. Waterman, M.S. and M. Eggert, "A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons," *J. Mol. Biol.* 197 (1987), 723-728.

APPENDIX A

Figure 2 below details the general algorithm for going against the grain. The generation of values in reverse order is begun by setting a global variable *cntr* to N. Thereafter, the sequence $f_{N-1}, f_{N-2}, \cdots f_0$ of values are delivered with each successive call to the function *getnext*. The logic of the algorithm can be simplified by roughly half if one caches $KN^{1/K}$ values by including $(\alpha 00^j)_R$ in the level-j cache $\{(\alpha x 0^j)_R\}_{x=1}^{R-1}$. However, when $k = \log_2 N$, $N^{1/K} = 2$ so that in this important limiting case our more complex algorithm caches half as many values as the simplified version, i.e., $\log_2 N$ versus $2\log_2 N$. For this reason we presented the somewhat more complex alternative.

```
Var C: array [0 ··· K − 1][1 ··· R − 1] of value
    cntr: integer

Function cache(base: value; start,step: integer): array [1 ··· R − 1] of value
{   For j ← 1 to R − 1 do
      {For k ← 1 to step do
         {start ← start + 1
          base ← h(base, start)
         }
       cache[j] ← base
      }
}

Function getnext: value
{   cntr ← cntr − 1
    k ← 0
    While k ≤ K and cntr modR^k = R − 1 do
       k ← k + 1
    j ← k
    While j ≤ K and (cntr divR^j)modR = 0 do
       j ← j + 1
    If j > K then
       base ← d_0
    Else
       base ← C[j][(cntr divR^j)modR]
    For j ← k − 1 downto 0 do
      {C[j] ← cache(base, (cntr divR^{j+1}) * R^{j+1}, R^j)
       base ← C[j][R − 1]
      }
    k ← 0
    While k ≤ K and cntr modR^k = 0 do
       k ← k + 1
    If k > K then
       getnext ← d_0
    Else
       getnext ← C[k][(cntr divR^k)modR]
}

cntr ← N
For j ← N − 1 downto 0 do
    f ← getnext        # delivers f_j #
```

Fig. 2. Iterative Going-Against-The-Grain Algorithm.

This article was processed using the LaTeX 2_ε macro package with CUP_CS class