Optimizing reduced-space sequence analysis

2 authors, including:

Richard Hughey
University of California, Santa Cruz
82 PUBLICATIONS   **6,729** CITATIONS

SEE PROFILE

Optimizing reduced-space sequence analysis

Raymond Wheeler[1,2] and Richard Hughey[1,3]

[1]Department of Computer Engineering, Jack Baskin School of Engineering, University of California, Santa Cruz, CA 95064, USA

Abstract

Motivation: *Dynamic programming is the core algorithm of sequence comparison, alignment and linear hidden Markov model (HMM) training. For a pair of sequence lengths m and n, the problem can be solved readily in $O(mn)$ time and $O(mn)$ space. The checkpoint algorithm introduced by Grice* et al. *(CABIOS, **13**, 45–53, 1997) runs in $O(Lmn)$ time and $O(Lm\sqrt[L]{n})$ space, where L is a positive integer determined by m, n, and the amount of available workspace. The algorithm is appropriate for many string comparison problems, including all-paths and single-best-path hidden Markov model training, and is readily parallelizable. The checkpoint algorithm has a diagonal version that can solve the single-best-path alignment problem in $O(mn)$ time and $O(m + n)$ space.*
Results: *In this work, we improve performance by analyzing optimal checkpoint placement. The improved row checkpoint algorithm performs up to one half the computation of the original algorithm. The improved diagonal checkpoint algorithm performs up to 35% fewer computational steps than the original.*

We modified the SAM hidden Markov modeling package to use the improved row checkpoint algorithm. For a fixed sequence length, the new version is up to 33% faster for all-paths and 56% faster for single-best-path HMM training, depending on sequence length and allocated memory. Over a typical set of protein sequence lengths, the improvement is ~10%.
Availability: *The SAM hidden Markov modeling package is freely available for academic use from http://www.cse.ucsc.edu/research/compbio/sam.html. The C++ code used to find optimal checkpoint placements is available from http://www.cse.ucsc.edu/research/kestrel.*
Contact: *rph@cse.ucsc.edu*

Introduction

Dynamic programming for sequence comparison and alignment forms the core of many computational biology and other sequence analysis methods. The simpler

methods consist of a series of recurrence equations with values added along an optimal path selected by best score (Smith and Waterman, 1981; Gotoh, 1982). Probabilistic methods used with hidden Markov models (HMMs) sum values over all possible paths up to a given point in the dynamic programming matrix, and multiply values along the path (Krogh *et al.*, 1994).

In forming a matrix based on a cost function and two sequences *a* and *b* (or a sequence and an HMM) of length *n* and *m*, an $m \times n$ table is formed with one or more entries per index point. For example, perhaps the simplest form of the dynamic programming equation is that of edit distance (Wagner and Fischer, 1974; Sellers, 1974):

$$c_{i,j} = \min \begin{cases} c_{i-1,j-1} & + & \mathrm{dist}(a_i, b_j) & \text{match} \\ c_{i-1,j} & + & \mathrm{dist}(a_i, \phi) & \text{insert} \\ c_{i,j-1} & + & \mathrm{dist}(\phi, b_j) & \text{delete} \end{cases}$$

If during the $O(mn)$ calculation of $c_{m,n}$ the mn choices that were made in the minimizations are saved, sequence alignment can be performed. The optimal alignment between the two sequences is constructed by tracing the best path back from $c_{m,n}$ to $c_{1,1}$ in $O(n + m)$ time but, using the obvious method, traceback requires $O(mn)$ space.

Other sequence analysis methods use multiple recursive variables but still maintain the general form of

$$\vec{c}_{i,j} = \mathcal{F}(\vec{c}_{i-1,j-1}, \vec{c}_{i-1,j}, \vec{c}_{i,j-1}, a_i, b_j). \tag{1}$$

There are many variations on this equation (Durbin *et al.*, 1998). One method of comparing a protein sequence to an amino acid sequence requires looking at the previous three indices in the nucleic acid sequence. In general, the methods discussed in this paper will work with any uniform recurrence equation; that is, the index points of $\vec{c}$ that $\vec{c}_{i,j}$ depend upon are all of the form $\vec{c}_{i-g,j-h}$ for integer constants $g \geq 0$, $h \geq 0$, $g + h > 0$.

For finding an alignment, the function $\mathcal{F}$ must effectively choose one of the three index points as being the best, corresponding to matching characters, matching a_i to a gap, and matching b_j to a gap, respectively. In the case of HMM training or the calculation of the probability that a_i

was generated by HMM state b_j, $\mathcal{F}$ is essentially additive across the three different choices, so no single best path is produced. In this case, another recurrence similar to Equation (1) is used to accumulate the probabilities across all possible paths, and traceback requires $O(mn)$ time, independent of checkpointing.

Because sequences can be very long, the problem of computing sequence alignments requiring less than $O(nm)$ space has been actively studied.

Related work

The first linear-space algorithm for sequence alignment was presented by Hirschberg (1975) and extended and popularized by Myers and Miller (1988). The algorithm relies on a divide-and-conquer (D&C) strategy to reduce the space from $O(mn)$ to $O(n)$, with an additional slowdown factor (1.8 for Myers and Miller) to the $O(mn)$ time algorithm. In the D&C algorithm, an optimal middle row of the $m \times n$ matrix is found using a forward calculation from the beginning of the sequences and a backwards calculation from the end of the sequences. The forward and backward calculations meet in the center to produce a row of matrix cells, each containing the cost of the best alignment through that point. The optimal alignment for the two sequences passes through the cell with the minimum cost over the entire row. This allows the alignment problem to be divided into two subproblems of combined size $m \times n/2$, which can be solved recursively.

Grice, Hughey, and Speck introduced a family of reduced-space algorithms that can be used not only for sequence alignment (Grice *et al.*, 1997), but also for hidden Markov model (HMM) training (Tarnas and Hughey, 1998). The algorithms are called 'checkpoint' algorithms because they preserve only certain rows (checkpoints) of the dynamic programming (DP) matrix. The members of the family are distinguished by their 'level', a positive integer, which also determines their space and time complexity. A checkpoint algorithm of level L can perform HMM training in space bounded by $O(Lm \sqrt[L]{n})$, and runs in $O(Lmn)$ time. The work also introduced a diagonal checkpoint algorithm that can solve the single-best-path alignment problem in $O(m + n)$ space and $O(mn)$ time.

Chao (1998) extended the D&C method to calculating the set of index points present in all sub-optimal alignments. In the most restrictive case, this is similar to calculating the single best alignment, while in the least restrictive case, this is similar to calculating all possible paths through the dynamic programming matrix. Additional efficiency is gained by splitting the problem into multiple components (rather than just two) at each level of the recursion and in both dimensions (as informally proposed for a single level in one dimension by Edmiston *et al.* (1988)). The divide-and-conquer method that splits a single dimension into multiple components results in a calculation that is similar to the multiple checkpoints of our checkpointing algorithm. We do not analyze the possibility of checkpointing in both dimensions in this paper.

Powell *et al.* (1999) also note the flexibility of the checkpointing technique with respect to L, and apply checkpointing to Ukkonen's algorithm for sequence alignment to reduce its space bound in the calculation of added cost. In fact, checkpoint algorithms are appropriate for any dynamic programming problem that can be described by uniform recurrence equations, in which the amount of data necessary to store the checkpoint is defined by the maximum distance of a recurrence dependency over the index set.

This paper further explores the checkpoint paradigm, in particular the placement of checkpoints. The cost of different checkpoint placement strategies can be calculated in terms of the number of dynamic programming cell calculations (Equation (1)). As long as the algorithm has sufficiently low memory requirements to avoid the perils of virtual memory swapping data to disk, dynamic programing cell calculation dominates the differences between these schemes. Because these algorithms step through successive dynamic programming cells in a regular manner, ample use is made of typical long cache lines, so that the difference between fitting in memory and fitting in cache can be slight.

Checkpoint algorithm

In aligning (single-best or all-paths) two sequences, let n be the length of the longer sequence and m the length of the shorter. If M is the number of cells of the dynamic programming (DP) matrix that can be stored in memory, then let $R = M/m$ be the number of DP matrix rows that can be stored.

Fundamentally, the checkpoint algorithm is identical to the dynamic programming algorithm. The significant difference is that the checkpoint algorithm does not store the entire matrix, but instead stores selected rows of the matrix. The missing portions of the matrix, the parts between the stored rows, are recalculated during traceback, as needed. The rows that are saved are determined in such a way that the space-saving factor is polynomially greater than the slowdown factor for recalculation.

The checkpoint algorithm represents a family of algorithms, each identified by its level. For reasons that will become clear later, the regular dynamic programming algorithm is considered a 1-level checkpoint algorithm. Of the higher level checkpoint algorithms, the 2-level algorithm is the easiest to describe.

2-level algorithm

The 2-level algorithm starts by calculating and storing successive rows of the DP matrix until the memory has

been filled. At this point, the most recently calculated row (now called the checkpoint) is saved, the previous rows are erased, and the next section of the DP matrix is calculated using the space now available. This procedure continues until the final row is calculated, and traceback begins. Obviously, the penultimate rows are not purged like the others, but kept for traceback. During traceback, sections of the DP matrix are recalculated forward from the most recent checkpoint. Traceback continues through each section until the checkpoint for that section is reached. As the traceback procedure reaches the end of a section, the section is cleared from memory, and the next section is recalculated forward from its prior checkpoint, using the now available memory.

For example, let the number of rows that can be stored in memory $R = 50$. The algorithm starts by calculating the first 50 rows of the DP matrix, and then saving the 50th row (the checkpoint). The previous 49 rows are then deleted, and the next 49 rows are calculated. The 49th row is then saved (equivalent to the 99th row of the DP matrix), and so on. The 2-level algorithm can calculate a DP matrix with $\sum_{i=1}^{R} i = R(R + 1)/2$ rows. So, for a sequence alignment problem with $n = R(R + 1)/2$, then the algorithm uses in $O(m\sqrt{n})$ space and takes time bounded by $2mn + m + n$, since each cell is calculated at most twice.

Multilevel algorithm

For $n > R(R + 1)/2$, a higher-level checkpoint algorithm can be used. An L-level checkpoint algorithm (where L is a positive integer) picks up where the $(L - 1)$-level algorithm stops. In other words, the first row to be checkpointed by the L-level algorithm is the *last* row checkpointed by an $(L - 1)$-level algorithm. So, once the memory has been filled with $(L - 1)$-level checkpoints, the last checkpoint is saved, the other rows are cleared, and the memory is again filled recursively with $(L - 1)$-level checkpoints. Therefore, if R rows can be stored in memory, the first checkpoint for a 3-level algorithm is at row $R(R + 1)/2$ (the last 2-level row) of the DP matrix. The second checkpoint for the 3-level algorithm is $(R - 1)((R - 1) + 1)/2$ rows later, and so on. During the traceback phase, the algorithm calculates forward from the previous checkpoint, just as before. The missing rows are recalculated and discarded as they are no longer needed for traceback.

The total number of rows that can be calculated by a 3-level algorithm (Figure 1) is $\sum_{i=1}^{R} i(i + 1)/2 = (R(R + 1)(R + 2)/6)$. If $n = R(R + 1)(R + 2)/6$ then the algorithm uses $O(m\sqrt[3]{n})$ space and takes time bounded by $3mn + m + n$.

Row#	Status	# Times calculated
1	○	3
2	○	3
3	⋯	2
4	○	3
5	⋯	2
6	——	1
7	○	3
8	⋯	2
9	——	1
10	——	1

——	=	3-level checkpoint
⋯	=	2-level checkpoint
○	=	1-level checkpoint

Fig. 1. Checkpointing scheme for memory rows $R = 3$, level $L = 3$, and sequence length $n = 10$. Total number of rows calculated is 21. Regular dynamic programming would require 10 rows of memory and do only 10 rows of calculation.

For an arbitrary level L, the maximum number of rows that can be calculated r_L is

$$r_L(R) = \sum_{i=1}^{R} r_{L-1}(i) = \binom{R + L - 1}{L}$$

since $r_1(R) = R$.

When $n = \binom{R+L-1}{L}$, there are $\binom{R+L-2}{L}$ 1-level checkpoints, each computed L times, leaving $\binom{R+L-2}{L-1}$ checkpoints at higher levels, which can be similarly divided according to Yang-Hui's (Pascal's) triangle. Considering the diminishing number of recalculations at each i-level checkpoint as i increases to L, the total amount of row calculation is (adding a 1 to correct the $i = 1$ term):

$$t_L(R) = 1 + \sum_{i=1}^{L} i \times \binom{R + i - 2}{i},$$

or

$$t_L(R) = 1 + \frac{L(L + 1)}{R} \binom{R + L - 1}{L + 1}.$$

For R, L, and $n = \binom{R+L-1}{L}$, the algorithm uses $O(m\sqrt[L]{n})$ space and takes time bounded by $Lmn + m + n$ for the single best path. The base case, $L = 1$, is equivalent to the traditional dynamic programming algorithm. If $L = n$, on the other hand, then only 2 rows are needed, and the time is bounded by $O(n^2 m)$. Generally, it is best to make use of all the available memory, and to find the lowest level of checkpointing that will accommodate the longer sequence length n. Consequently, the optimum level is when

$$L = \min \left\{ i : n \le \binom{R + i - 1}{i} \right\}. \qquad (2)$$

```
R = number of rows of available memory
L = checkpoint level
n = number of rows in the DP matrix
curr_row (global) = most recently calculated row number
ending_mem_loc (global) is used to initialize traceback

F (R, L, n){
    if (L = 1)
        bound ← min{(curr_row + R), n}
        for k ← curr_row to bound
            Compute&Save (row[k], Mem[curr_row + 1 + R − k]);
        if (bound = n)
            ending_mem_loc ← curr_row + 1 + R − n;
        curr_row ← bound;
    else
        for i ← R to 1
            F (i, L − 1, n);
            if (curr_row = n)
                break ;
            else
                Mem[i] ← Mem[1];
                SetLevelAndRow (Mem[i], L, RowNum(Mem[1]));
}
```

Fig. 2. Forward calculation function for basic checkpoint algorithm. The subroutine calls, respectively, compute the recurrence saving salient information, set the level and row number for row at given memory location, and return the row number for row at given memory location.

Since $\binom{R+L-2}{L-1} \leq n$, then $2^{L-1} \leq n$ and $L \leq \log n + 1$. Using exponential search, L can be found in $O(\log \log n)$ time.

In the forward calculation (Figure 2) for the basic checkpoint algorithm, memory is assumed to be an array, with row[0] permanently stored at Mem[0]. The function $F(R, L, n)$ causes the first R rows of memory to be filled with L-level checkpoints. The first L-level checkpoint is at Mem[R], and the last is at Mem[1]. When each row is computed and stored, its row number and checkpoint level are also saved with it. This information is necessary for the traceback phase. The extra space required for this data is $O(\log n)$ for the row number, and $O(\log \log n)$ for the level. The call stack also requires $O(\log n)$ memory.

The backwards calculation function $T(R, n)$ (Figure 3) traces back one row at a time until a gap in the data is reached. At this point, it uses function $F(R, L, n)$ to calculate forward from the most recent checkpoint. Function $T(R, n)$ relies on the fact that higher-level checkpoints always come after lower-level checkpoints in memory. Traceback continues using recursive forward re-calculations until the first row has been reached.

Improved checkpoints

Although the above checkpoint placement method is optimal for $n = \binom{R+L-1}{L}$, it is suboptimal for other values

```
T (R, n){
    j ← ending_mem_loc;
    do
        // Traceback as much as possible
        while (j ≠ R &&
                RowNum(Mem[j + 1]) = RowNum(Mem[j]) - 1)
            Traceback(Mem[j], Mem[j + 1]);
            j++;
        // Stopping condition
        if (RowNum(Mem[R]) = 1)
            break ;
        // Calculate forward from last checkpoint
        if (j = R)
            curr_row ← 0;
        else
            curr_row ← RowNum(Mem[j + 1]);
        F (j, Level(Mem[j]) - 1, n);
        j ← 1;
    while (TRUE)
}

Main(R, n){
    curr_row ← 0;
    ending_mem_loc ← 1;
    L ← CalculateLevel(R, n);
    F (R, L, n);
    T (R, n);
}
```

Fig. 3. Traceback management function and sample call function for basic checkpoint algorithm. The subroutine call Traceback determines the optimal path, and CalculateLevel determines the optimal checkpoint level.

of n. Consider the following example (Figure 4) with 4 rows of memory, or $R = 4$.

In the first column ($L = 2$), at most $\binom{4+2-1}{2} = 10$ rows can be calculated. The 10th row is therefore the first row to be checkpointed in the 3-level algorithm, shown in the second column. This is not necessarily the best strategy. The third column shows a checkpoint scheme with significantly fewer row calculations than the basic 3-level algorithm. The problem in this situation is that with $n = 11$ and $R = 4$, the 2-level checkpoint algorithm falls short, only able to handle sequence lengths up to $n = 10$, while the 3-level algorithm will waste time because it is suited for $n = 20$.

The inefficiency of the original L-level algorithms is high when $n = r_{L-1}(R) + 1$. Here, the original L-level algorithm will calculate the first $n-2$ rows without saving, calculate the final two rows at level L, and then recalculate the $n - 2$ rows from the initial L-level checkpoint and complete traceback using the $(L - 1)$-level algorithm. Thus, when $n = r_{L-1}(R)$, each row is calculated up to $L - 1$ times, while for a sequence one character longer, most rows are calculated L times. This discontinuity in performance results in a factor of $\sim L/(L-1)$ slowdown,

Row #	$n = 10$	$n = 11$	$n = 11$	$n = 11$
	$L = 2$	$L = 3$	$L = 2, 3$	$L = 1^\star$
1	○	○	○	○
2	○	○	○	○
3	○	○	○	○
4	—	⋯	—	○
5	○	○	○	○
6	○	○	○	○
7	—	⋯	—	○
8	○	○	○	○
9	—	⋯	⋯	○
10	—	—	—	○
11		—	—	○
# Rows Calculated	16	26	19	11

—	=	L-level checkpoint
⋯	=	$(L-1)$-level checkpoint
○	=	1-level checkpoint

Fig. 4. Checkpoint schemes for $R = 4$ memory rows, and sequence lengths $n = 10, 11$. *Regular dynamic programming: $R = 11$.

Fig. 5. Row computations of the 2-level and 1, 2-level algorithms for $n = 1000$ and a range of memory sizes according to the analysis in the text.
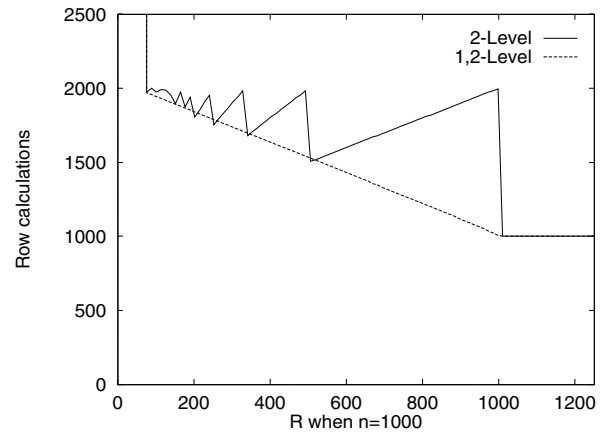
or a factor of two for the 2-level algorithm.

A simple change to the algorithm, which we call the improved, or $(L - 1, L)$-level checkpoint algorithm, can greatly improve the situation. The change is to perform the $(L - 1)$-level algorithm for the first ρ rows of the dynamic programming, until the remaining rows just fit in the remaining memory using the L-level algorithm.

For example, in R rows of memory, the original 2-level algorithm will have l checkpoints until $\sum_{i=0}^{l}(R - i) > n$, or $l \sim R - \sqrt{R^2 + R - 2n}$. All calculations will be repeated except for the final group and for the $l - 1$ checkpoints, so $t_2(R, n) \sim 2n - \left(n - \sum_{i=0}^{l-1}(R-i) - l + 1\right)$. This function is highly dependent on R and n when l is required to be an integer (Figure 5).

Recall that the 1-level algorithm can calculate $r_1(R) = R$ rows with $t_1(n) = n$ row calculations, while for the 2-level algorithm, $r_2(R) = (R)(R + 1)/2$ and $t_2(n) = 2n$, both when $n = r_L(R)$. With the 1, 2-level algorithm, for $r_1(R) < n < r_2(R)$, ρ iterations of the 1-level algorithm are performed, until $n - \rho = r_2(R - r_1(\rho))$, or $\rho \sim R - \sqrt{2n - 2R}$. The number of row calculations is $t_{1,2} = t_1(\rho) + t_2(R - \rho)$, or $t_{1,2}(R, n) \sim 2n - R - \sqrt{2n - 2R}$. In the example shown in Figure 4, $\rho = 2$ so that 2 rows are checkpointed at level 2 and the remaining 2 are checkpointed at level 3. As can be seen from the bottom curve of Figure 5, this algorithm has greatly improved performance, and continuity, between $r_2^{-1}(n)$ and $r_1^{-1}(n)$.

Similar analysis can be performed on the other improved algorithms, with resulting higher-degree polynomials. For implementation, the idea is quite simple: use the $(L - 1)$-level algorithm for the first ρ rows of memory such that $r_{L-1}(\rho) + r_L(R - \rho) > n$. The optimal choice of ρ is the maximum number of rows that can be checkpointed at level $(L-1)$, while still leaving enough rows to finish the matrix at level L. Using an exponential search algorithm, ρ can be found in $O(\log R \times \log n)$ time.

The major difference in the code for the improved checkpoint algorithm and the basic algorithm is the inclusion of a 'wrapper' function around the forward calculating function, $F(R, L, n)$ (Figure 6). The wrapper function $\text{Impr}F(R, L, \rho, n)$ requires the parameter ρ, which equals the number of rows (possibly zero) that will be checkpointed at level $L - 1$. Function $\text{Impr}F(R, L, \rho, n)$ proceeds by filling the last ρ indexes of memory with $(L-1)$-level checkpoints, using function $F(R, L, n)$ as a subroutine. The remaining initial $R - \rho$ indexes are filled at level L, again using function $F(R, L, n)$.

Optimal checkpoints

The $(L - 1, L)$-level algorithms are not optimal. The optimal algorithms freely switch between levels to make optimal use of available memory for any subproblem. For example, in the 2, 3-level algorithm, the 2-level algorithm is used first to fill the first ρ rows, but we already known that the 1, 2-level algorithm is more efficient.

To study optimal checkpoint placement, we used an $O(n^3)$ dynamic programming algorithm. Let $C(R, n)$ be the minimum number of row calculations necessary, given the longer sequence length n, and using only R rows of

```
ImprF (R, L, ρ, n){
    if (ρ = 0)                // e.g., if L=0
        F (R, L, n);
    else
        // checkpoint at level L − 1 for ρ rows
        for i ← R to (R − ρ + 1)
            F (i, L − 2, n);
            Mem[i] ← Mem[1];
            SetLevelAndRow(Mem[i], L − 1, RowNum(Mem[1]));
        // checkpoint remaining rows at level L
        F (R − ρ, L, n);
}

Main(R, n){
    curr_row ← 0;
    ending_mem_loc ← 1;
    L ← CalculateLevel(R, n);
    ρ ← NumRowsAtLowerLevel(R, L, n);
    ImprF (R, L, ρ, n);
    T (R, n);
}
```

Fig. 6. Forward calculation 'wrapper' function for $F(R,L,n)$, and sample Main for improved checkpoint algorithm. The subroutine call NumRowsAtLowerLevel determines number of rows to be checkpointed at level L-1.

memory. Letting $C(R, n) = n$ when $n \leq R$,

$$C(R, n) = \min_{0 < i \leq n} \{C(R − 1, n − i) + C(R, i) − 1 + i\}.$$

The last term, i, is the cost of calculating up to row i. The term $C(R, i) − 1$ is the cost of backtracking from the beginning of the sequence up to row i, and the term $C(R − 1, n − i)$ is the cost of the checkpoint algorithm calculating the remainder of the sequence after row i. When $n = \binom{R+L-1}{L}$, $C(R, n) = t_L(R)$. When $\binom{R+L-1}{L} < n \leq \binom{R+L}{L+1}$, $C(R, n) < t_{L+1}(R) = O((L + 1)n) = O(Ln)$, but we have not found a more satisfying bound on $C(R, n)$.

Table 1 shows the worst case performance of the basic checkpoint algorithm, relative to the optimal checkpoint algorithm, for small values of R. For each level and value of R, the first number is the number of row-calculations performed by the checkpoint algorithm divided by the optimal number of row-calculations, and the second number is the maximum value that n can attain for the given R and L. The worst case was determined by comparing the algorithm's performance to optimal performance for each sequence length $\binom{R+L-2}{L-1} < n \leq \binom{R+L-1}{L}$. For example, if $R = 8$, then the 2-level algorithm would be used for sequence lengths $9 \leq n \leq 36$, the 3-level $37 \leq n \leq 120$, and the 4-level $121 \leq n \leq 330$, and so on.

As can be seen, the $L/(L − 1)$ slowdown is not yet

Table 1. Worst case performance of basic checkpoint algorithm for small R, expressed as a ratio of row-calculations with respect to the optimal checkpoint scheme determined by dynamic programming on $C(R, n)$

R	2-level		3-level		4-level		5-level	
2	1.000	3	1.000	4	1.000	5	1.000	6
3	1.200	6	1.250	10	1.241	15	1.217	21
4	1.333	10	1.368	20	1.320	35	1.270	56
5	1.429	15	1.429	35	1.348	70	1.283	126
6	1.500	21	1.462	56	1.359	126	1.284	252
7	1.556	28	1.481	84	1.362	210	1.282	462
8	1.600	36	1.493	120	1.362	330	1.280	792
9	1.636	45	1.500	165	1.362	495	1.277	1287
10	1.667	55	1.505	220	1.361	715	1.275	2002

reached in the 2-level column (1.975 is reached at $n \sim$ 5000), but is exceeded in the other columns. The optimal algorithm avoided cases where the initial calculation of $ρ$ rows could be better performed with a mixture of other levels. However, the additional performance gain is small. For example, the 2, 3-level algorithm calculates fewer than $\sqrt{n}$ initial rows using the 2-level algorithm (otherwise, the 2-level algorithm could be used for the entire problem). If this initial calculation could be sped by a factor of two using the 1, 2-level algorithm rather than the 2-level algorithm, the comparative times would be $\sim 3n$ for the 3-level algorithm, $\sim 2n$ for the 2, 3-level algorithm, and $\sim 2n − \sqrt{n}$ for the 1, 2, 3-level algorithm.

Asymptotically, the L-level algorithm performs Ln row calculations, the $(L, L − 1)$ algorithm performs at least $(L − 1)n + 1$ calculations, and the optimal $(L, L − 1, L − 2, \ldots)$ improves at most $\sqrt[L-1]{n}$ initial rows by using (at best) the 1-level algorithm rather than the $(L − 1)$-level algorithm. Thus, the $(L, L − 1, L − 2, \ldots)$ algorithm performs at least $(L−1)n−(L−2)\sqrt[L-1]{n}$ row calculations and the performance ratio between L-level and optimal is at most

$$\frac{Ln}{(L − 1)n − (L − 2)\sqrt[L-1]{n}} = O\left(\frac{L}{L − 1}\right).$$

The $(L, L − 1)$ algorithm and the optimal algorithm both range between Ln and $(L − 1)n$ calculations with a maximum difference of $(L − 2)\sqrt[L-1]{n}$ row calculations, giving a maximum ratio of

$$\frac{(L − 1)n}{(L − 1)n − (L − 2)\sqrt[L-1]{n}} = O(1).$$

In practice (Table 2), the improved algorithm is at most 10% slower than the optimal for worst-case values of $n \geq 150$ and a given R.

Table 2. Worst case performance of improved checkpoint algorithm for R rows of memory, measured as a ratio of row-calculations with respect to the optimal checkpoint algorithm (as determined by dynamic programming)

R	1, 2-level		2, 3-level		3, 4-level		4, 5-level	
2	1.000	3	1.000	4	1.000	5	1.000	6
3	1.143	6	1.133	10	1.138	15	1.137	21
4	1.125	10	1.147	20	1.129	35	1.099	56
5	1.111	15	1.121	35	1.115	70	1.085	126
6	1.100	21	1.101	56	1.109	126	1.089	252
7	1.095	28	1.089	84	1.086	210	1.089	462
8	1.091	36	1.081	120	1.067	330	1.068	792
9	1.087	45	1.074	165	1.055	495	1.051	1287
10	1.083	55	1.067	220	1.048	715	1.049	2002
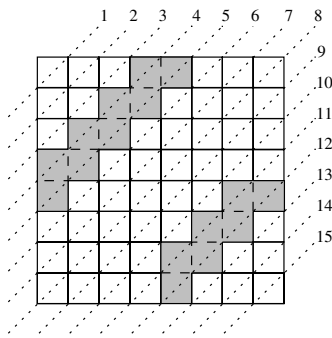
**Fig. 7.** Diagonal checkpoints at rows 5 and 12 of an 8×8 DP matrix.

Single best path

The row checkpoint algorithms find the single best path or all paths in $O(Lmn)$ time and $O(Lm \sqrt[L]{n})$ space. A diagonal version of the checkpoint algorithm can find the single best path using less time and memory because the best path will only pass through a limited region of the DP matrix between any two diagonals.

The basic diagonal checkpoint algorithm operates on the same principle as its row-checkpointing counterpart. The cells of the DP matrix are calculated in a diagonal 'sweep' across the matrix, from the upper-left to the lower-right corner. Once the memory is full, the most recently completed full diagonal is saved, and the remaining portion of the memory is used for continuing the DP calculation. The process is repeated across the rest of the DP matrix.

Our approach to improving diagonal checkpoint performance was similar: use dynamic programming to find the optimal checkpoint algorithm, and then attempt to discover a simpler, but still improved, algorithm for general use.

Solving the optimal diagonal checkpoint placement
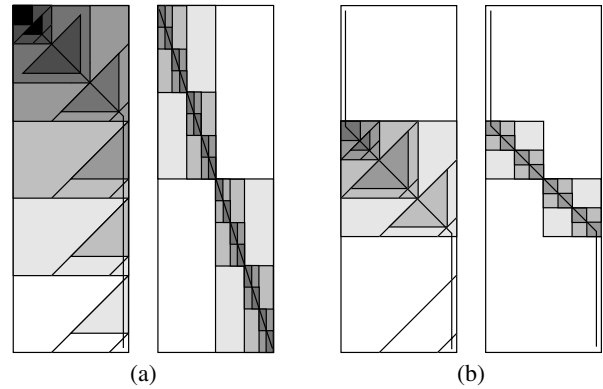
Fig. 8. (a) Respective worst case paths for diagonal checkpoint algorithm and D&C algorithm, with $n \gg m$. (b) Typical paths in the Smith and Waterman algorithm.
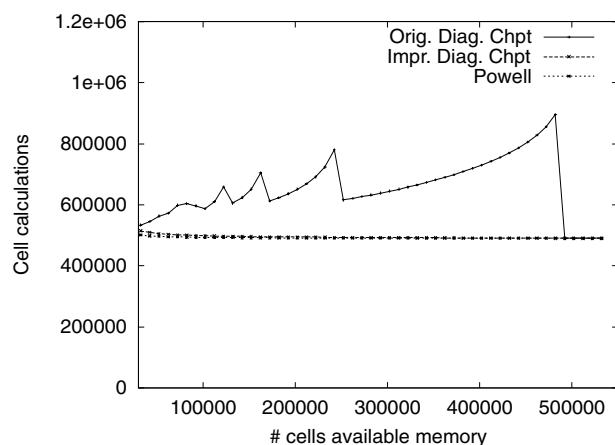
problem requires an $O(n^4)$, 3-dimensional dynamic programming algorithm. The checkpoint placement scheme provided by this dynamic programming algorithm was extremely complicated and difficult to model. Instead, we simplified the problem by only considering triangular regions of the dynamic programming matrix to produce an $O(n^3)$ dynamic programming problem. Analysis of these results gave us a means of improving the checkpoint algorithm for the single-best path problem.

The analysis was unsuccessful from the point of view of being unable to create an algorithm better than its competitors. For sequences of approximately equal length, the improved diagonal checkpoint algorithm performed slightly fewer cell calculations then the basic divide and conquer algorithm, each using 2 rows of memory. However, when the sequences are of differing lengths, the divide and conquer algorithm is clearly superior (Figure 8a) because the diagonal checkpoint algorithm always recalculates from the first column.

The situation is improved when, following Powell *et al.* (1999), the algorithm maintains two additional dynamic programming variables that respectively keep track of the row at which each path departs from the zero column and enters the last column. During traceback, one can skip calculation to the row at which the path entered the last row. Likewise, the initial portion of the matrix would not have to be recalculated, since one would already know the row at which the optimal path departs from the zero column. Although the diagonal checkpoint algorithm shows greater benefit from these techniques, it still performs more computation than the D&C algorithm (Figure 8b).

When more than 2 rows of memory are available, the diagonal checkpoint algorithm is better suited to exploiting the extra resource, and performs significantly

Fig. 9. Diagonal checkpoints and Powell's algorithm for a 700×700 alignment.

Fig. 10. Execution time (minimum of 3 runs) for 2-Level and 1, 2-Level SAM all-paths (4 sequences) and single-best-path (12 sequences) HMM training with $n = 1000$, $m = 1000$, and various DP memory sizes.
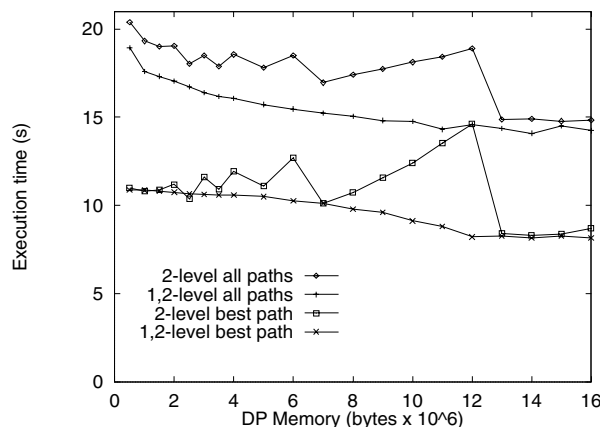
better than D&C in these situations. However, one can do even better by implementing Powell's algorithm rather than just borrowing its techniques. Figure 9 shows a comparison of the 2-level diagonal checkpoint algorithm (original and improved), and Powell's algorithm (the basic divide and conquer algorithm is not shown). As with the row checkpoint case, the original diagonal checkpoint algorithm calculates as much as it can during any given step, inducing a sawtooth function. In this memory range, Powell's algorithm performs up to 2.5% fewer cell calculations than our improved diagonal checkpoint algorithm, although each cell calculation does have the extra recurrence variable.

In summary, when only the single best path is desired, Powell's algorithm should be used to provide full flexibility over a range of memory sizes and sequence lengths. When an all-paths calculation is to be performed, or if the same code is to perform both all-paths and single-best-path, row or diagonal checkpoints can be used.

HMM implementation

We modified the SAM hidden Markov model package (Hughey and Krogh, 1996) from using the 2-level row checkpoint algorithm to the 1, 2-level algorithm. With 64 MB, these algorithms enable the evaluation of an 80 000-character sequence and an 80 000-node HMM, or a 1M-character sequence and 5000-node HMM, but the expensive all-paths cell calculation makes such tasks infeasible. Thus, in this case, we do not need higher-level checkpointing. We evaluated algorithms on a low-memory (64 MB) Sun UltraSparc 1.

For a single sequence, the original algorithm has higher variation in execution time than the 1, 2-level algorithm for both all-paths and single-best-path HMM training

(Figure 10, with $m = n = 1000$). At high memory use, both algorithms become the 1-level algorithm, and perform similarly. At minimal memory use, both algorithms also behave similarly. In the intermediate range, the 1, 2-level algorithm provides a smooth decrease in runtime as the amount of recalculation diminishes. On the other hand, the 2-level algorithm is sensitive to the precise amount of memory allocated for the DP matrix, as expected from Figure 5. The 33% (all-paths, with its costly traceback) and 56% (best-path) slowdowns at 12×10^6 bytes occur when there is not quite enough memory to perform the 1-level algorithm.

Single-path memory use could be further improved because the traceback information requires less memory than the checkpoint information (less than three bytes per index point, rather than three 4-byte words). In SAM, the desire to maintain consistency between the algorithms is deemed more important than the small improvement that would result.

To measure average speedup, we used a set of 200 protein sequences (minimum length 35, average 329, maximum 2027) randomly chosen from the SwissProt 38 database (minimum length 3, average 363, maximum 6486) (Bairoch and Apweiler, 1998), and three protein model sizes (250, 1058, and 5000). SAM's previous default behavior was to use the minimum amount of memory required by the longest sequence, or in this case, 63 dynamic programming rows, or ~ 200 KB, 1 MB, and 5 MB for the three model sizes. We varied the DP memory up to 60 MB (1000 rows for the 5000-node model).

Over the range of DP memory sizes, regardless of model length, the new all-paths algorithm is 3–12% faster. The typical improvement is $\sim 10\%$ when 10–20 MB

is allocated the problem. This advantage decreases with increasing memory allocation as more of the sequences can use the simple 1-level algorithm with no extra work. The absolute running times of both algorithms decrease 15–20% as allowed memory is increased within the bounds available for the machine. This allows one to optimize the dynamic programming calculation to the specifics of a given machine. For the 5000-length model, both algorithms begin paging at 60 MB with a factor of 1.9 slowdown.

Discussion

This paper presented an improvement on the original checkpoint algorithms. The improved row-checkpoint algorithm performs up to one half as many computations than the original checkpoint algorithm, and the improved diagonal-checkpoint algorithm performs up to 35% fewer than the original diagonal checkpoint algorithm.

When only the single-best-path is required, Powell's algorithm is an appropriate choice as an alternative to the basic divide-and-conquer or diagonal checkpoint algorithms.

In cases where all possible paths are to be evaluated, such as HMM training and posterior-decoded alignment (Holmes and Durbin, 1998), the checkpoint algorithm's ability to easily and efficiently process all possible paths makes it the only choice.

Optimizing the SAM HMM package's checkpoint algorithm resulted in a $\sim 10\%$ speedup over a typical range of sequence lengths.

In the future, we expect to modify SAM to use optimized reduced space dynamic programming for its posterior-decoded alignment calculations. In creating these alignments, first an $m \times n$ matrix of posterior probabilities is generated using an all-paths forward and backward calculation. Then, a single-best-path dynamic programming algorithm, forward and traceback, on the posterior matrix is performed to find the alignment. SAM's current implementation uses the checkpoint algorithm for the first part, but stores the complete $m \times n$ posterior matrix, making this a memory-expensive operation that causes paging for long sequences. To turn this into a reduced memory algorithm, both the forward and the backward portions of the posterior calculation must be checkpointed.

During the second phase, the checkpoint pairs of the first phase are used to compute the posterior values in each region. This allows the second (single-best-path) forward recurrence to be evaluated with its own checkpointing. During final traceback, areas are again recalculated, and the minimization information is saved,

enabling construction of the single best path. Without checkpointing, 2 cell calculations per index point are performed in the first phase (forward and backward), and 1 in the second phase (forward, since traceback is $O(n + m)$). The interwoven 2-level checkpoint algorithm will double the number of cell calculations to 6 per index point while requiring $O(m\sqrt{n})$ space.

References

Bairoch,A. and Apweiler,R. (1998) The SWISS-PROT protein sequence data bank and its supplement TrEMBL in 1998. *Nucleic Acids Res.*, **26**, 38–42.

Chao,K.-M. (1998) On computing all suboptimal alignments. *J. Inform. Sci.*, **105**, 189–207.

Durbin,R., Eddy,S., Krogh,A. and Mitchison,G. (1998) *Biological Seqeunce Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Oxford University Press, UK.

Edmiston,E.W., Core,N.G., Saltz,J.H. and Smith,R.M. (1988) Parallel processing of biological sequence comparison algorithms. *Int. J. Parallel. Program.*, **17**, 259–275.

Gotoh,O. (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**, 705–708.

Grice,J.A., Hughey,R. and Speck,D. (1997) Reduced space sequence alignment. *CABIOS*, **13**, 45–53.

Hirschberg,D.S. (1975) A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, **18**, 341–343.

Holmes,I. and Durbin,R. (1998) Dynamic programming alignment accuracy. *J. Comp. Biol.*, **5**, 493–504.

Hughey,R. and Krogh,A. (1996) Hidden Markov models for sequence analysis: extension and analysis of the basic method. *CABIOS*, **12**, 95–107.

Krogh,A., Brown,M., Mian,I.S., Sjölander,K. and Haussler,D. (1994) Hidden Markov models in computational biology: applications to protein modeling. *J. Mol. Biol.*, **235**, 1501–1531.

Myers,E.W. and Miller,W. (1988) Optimal alignments in linear space. *CABIOS*, **4**, 11–17.

Powell,D., Allison,L. and Dix,T. (1999) A versatile divide and conquer technique for optimal string alignment. *Inform. Process. Lett.*, **70**, 127–39.

Sellers,P.H. (1974) On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, **26**.

Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

Tarnas,C. and Hughey,R. (1998) Reduced space hidden Markov model training. *Bioinformatics*, **14**, 401–406.

Wagner,R.A. and Fischer,M.J. (1974) The string-to-string correction problem. *J. ACM*, **21**, 168–173.