A Fast Pruning Algorithm for Optimal Sequence Alignment

Aaron Davidson
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
davidson@cs.ualberta.ca

Abstract

Sequence alignment is an important operation in computational biology. Both dynamic programming and A heuristic search algorithms for optimal sequence alignment are discussed and evaluated. Presented here are two new algorithms for optimal pairwise sequence alignment which outperform traditional methods on very large problem instances (hundreds of thousands of characters, for example). The technique combines the benefits of dynamic programming and A* heuristic search, with a minimal amount of additional overhead. The dynamic programming matrix is traversed along antidiagonals, bounding the computation to exclude portions of the matrix that cannot contain optimal paths. An admissible heuristic assists in pruning away unnecessary areas of the matrix, while preserving optimal solutions for any given scoring function. Since memory requirements are a major concern for large sequence alignment problems, it is shown how the standard algorithm (requiring quadratic space) can be reformulated as a divide and conquer algorithm (requiring only linear space, at the cost of some recomputuation).*

1 Introduction

In the sequence alignment problem a set of strings must be aligned to maximize the number of positions where the strings have matching characters. Gaps ('-') may be inserted into the strings in order to shift the remaining characters into better alignments. Alignments are evaluated by a *scoring function* that gives points for matching characters and penalties for any gaps. Our task is to find an optimal alignment for a set of strings and a given scoring function.

There are many useful applications for sequence alignment in computational biology. In biological sequence alignments for strings representing DNA or proteins, the scoring function represents the biological plausibility of various mutations occurring in the strings.

Suppose a gene has been discovered in a model organism, and the function it plays in cell metabolism is well known. It is plausible that a similar, mutated gene exists in humans. The human genome database contains millions of sequence fragments. By taking the sequence of a gene whose function is known, we can align it against all of the sequences in the human genome database [7]. If a sequence aligns very well with our probe, it is likely to have a similar function. These matches are called homologies. Once many homologous genes have been discovered in several different species, an optimal alignment between all species can help show the evolutionary history of a gene as the species diverged from a common ancestor over time.

Sequence alignment is also used to discover the causes of genetic disease. A geneticist can align sequences of several healthy subjects together with several afflicted subjects. If the afflicted subjects all share a common mutation that none of the healthy subjects have, it is strong evidence that this mutation is a cause of the ailment. There are further uses for biological sequence alignment, but they are beyond the scope of this discussion.

Section 2 discusses the well known dynamic programming algorithms for sequence alignment, as well as A* heuristic search methods. In Section 3 we present an algorithm that combines the selective computation of a heuristic search with the efficiency of dynamic programming. This hybrid approach bounds the traversal of the dynamic programming matrix so that unneeded portions of the matrix are

```
AGGTATTA-   2*(6 matches) -2*(3 gaps)
A--TATTAG      score = 6

A-GGTATTA-  2*(4 matches) -1*(1 mismatch) -2*(5 gaps)
AT-AT-T-AG     score = -3
```

Figure 1. Two example alignments of 'AGGTATTA' against 'ATATTAG'.

A	136			
G	-37	136		
C	-160	-160	136	
T	-160	-160	-37	136
	A	G	C	T

Table 1. A nucleic acid scoring matrix.

not visited. This approach does especially well on large alignments. For instance on an alignment of two DNA strings over two hundred-thousand base-pairs in length, this method was up to twice as fast as previous methods for optimal sequence alignment. These results are detailed in Section 4. Section 5 gives future work and conclusions.

2 Sequence Alignment

2.1 Scoring Functions

To facilitate simple examples, we will use a very basic scoring function. Matching characters will be worth 2 points, a mismatch will be worth -1 point, and each gap will be worth -2 points. For example, the two strings ‘AGGTATTA’ and ‘ATATTAG’ could be aligned in many ways. Two possible alignments are shown in Figure 1.

In real biological applications the scoring function is far more complicated. There can be many different values for each type of match and mismatch. Typically a scoring matrix such as shown in Table 1 is used. Furthermore, the cost of each gap inserted depends on the context as well. For instance, starting and trailing gaps have no penalty, and the cost of opening a gap is much higher than extending an existing gap. These are called *affine* gap penalties. The scores for gaps, matches and mismatches are chosen to correspond to the likelihood of the mutations occurring in nature. A few long gaps are much more common in real biological sequences than are many small gaps, so the affine gap cost favors alignments that have the former characteristic. The scoring function used can drastically affect the performance of certain alignment algorithms. Scoring functions are discussed more thoroughly in [3] and [7].

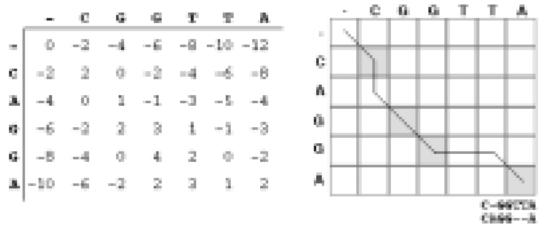
2.2 Sequence Alignment As Path Finding

Sequence alignment can be formalized as a shortest-path problem through a d-dimensional lattice [2], where d is the number of sequences being aligned. The alignment state-space consists of nodes where either the next character from a sequence is accepted or a gap is entered. We start from an empty state and move towards the goal state by accepting characters and inserting gaps. A goal state is reached when all characters have been aligned.

The distinction between this type of problem and traditional path finding problems is that in sequence alignment

```
Matrix computeMatrix(Sequence A, Sequence B) {
   int x,y,max;
   Matrix m = newMatrix(length(A), length(B));
   for (x=0; x < length(B); x++)
      m[x][0] = x * GAP_PENALTY;
   for (y=0; y < length(B); y++) {
      m[0][y] = y * GAP_PENALTY;
      for (x=1; x < length(B); x++) {
         v1 = m[x-1][y-1] + SCORE(A[x], B[y]);
         v2 = m[x-1][y] + GAP_PENALTY;
         v3 = m[x][y-1] + GAP_PENALTY;
         m[x][y] = max(v1, v2, v3);
      }
   }
   return m;
}
```

Figure 2. Sample C-like code for computing the matrix.



	-	C	G	G	T	T	A
-	0	-2	-4	-6	-8	-10	-12
C	-2	2	0	-2	-4	-6	-8
A	-4	0	1	-1	-3	-5	-4
G	-6	-2	2	3	1	-1	-3
G	-8	-4	0	4	2	0	-2
A	-10	-6	-2	2	3	1	2

Figure 3. A sample matrix generated for a small problem. The right shows an optimal path traced through the matrix.

there is a large branching factor ($2d - 1$), and a large number of alternate paths between nodes in the graph. This gives the domain a unique, regular structure that still challenges traditional search algorithms such as A* and IDA* [9].

2.3 Dynamic Programming

Needleman and Wunsch developed a dynamic programming method that places each sequence along the edge of a matrix [11]. We will refer to this method as the *full-matrix* algorithm. In two dimensions, the algorithm starts in the top left corner, and computes the cells left to right, and top to bottom. An alignment of two sequences corresponds to a path through the matrix (from the top-left corner, to the bottom-right).

Moving horizontally or vertically through the matrix represents inserting a gap in the opposing sequence. Moving diagonally represents aligning the two values for that cell. The value of each cell is computed by taking the move that is best from the three neighbor cells to its top, left, and top-left.

When the matrix is complete, the value in the bottom-right corner is the optimal score. To reconstruct the paths

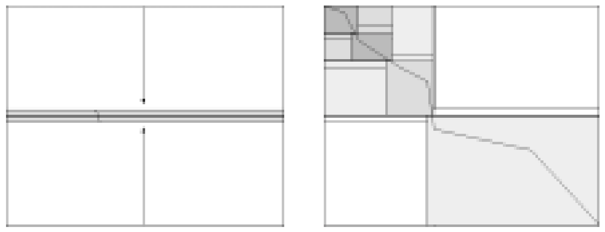


Figure 4. Hirschberg's Divide and Conquer Method.

through the matrix that yield the optimal score, we simply need to follow the matrix from the bottom-right cell, all the way back to the start, by moving back into cells that are on the optimal pathway. Figure 2 gives some simple pseudo-code for matrix computation (with a linear gap penalty). Figure 3 shows an example of a small matrix after being computed, and a traced optimal path.

Obviously this method is $O(n^2)$ in both time and space. The computations needed at each cell are extremely light, so even with large sequences, the computation time is tractable. However, the size of the sequences do not have to be very large before the matrix is too large to be stored in main memory.

If we generalize this algorithm to multiple sequence alignment, we add a new dimension to the matrix for each sequence. The time and space complexity becomes $O(n^d)$, where n is the sequence length, and d is the number of sequences. With even a few sequences of moderate length, the memory of current computers is quickly exhausted with this approach.

2.4 Reducing the Memory Requirements

Hirschberg developed a similar algorithm that changes the space requirements to a more respectable $O(n)$ in two dimensions ($O(n^{d-1})$ in general), at the expense of requiring extra recomputation of lost matrix values [6].

For the two dimensional case, the algorithm stores only two rows of the matrix. It starts at the top and computes the matrix one row at a time, overwriting the values in the array as it works. When it reaches the middle row of the matrix it stops and uses another array to compute from the bottom, up to just below the other saved row in the middle.

Using these two rows from the middle, we can find the maximum sum of the transitions between the two rows. This will give us the optimal score, and the point along the diagonal where the optimal path crosses. The algorithm then recursively solves the two sub problems from the top-left to the crossing point, and from the crossing point to the bottom-right (Figure 4). At some point, the problems can become small enough that a full matrix can be used to solve it. This algorithm typically does more than $2n^2$ cell computations.

The FastLSA algorithm [4] generalizes Hirschberg's algorithm by adding extra rows and columns, allowing the use of more memory to reduce the number of recomputations. Both rows and columns are stored, and the number of extra rows and columns is a scalable parameter for the trade-off between extra memory and fewer recomputations.

The surprising fact about these two linear-space algorithms is that for many problems they require less time to execute than the full-matrix versions. In fact, the predictions made by classic time-complexity analysis do not reflect what is observed empirically. This is due to the effects of cache-memory. Once the full matrix can no longer fit in cache, the cost of constantly going to main-memory is extremely high (typically almost a factor of ten times slower). This cost is far higher than the cost of the extra calculations.

2.5 A* Heuristic Search Methods

A* (read 'A-star') is a classic best-first search algorithm, well known in the artificial intelligence community, and is applicable to a wide variety of domains [5]. When combined with a good heuristic evaluation function, it is often the most efficient method known for finding optimal solutions.

Since the state-space representation defines sequence alignment as a path finding problem, A* would appear to be a natural candidate. Unfortunately, the memory constraints of the basic A* algorithm allow it to work only for relatively small problems.

The basic strategy of A* is to search forward from a start state, always expanding the most promising node first. All nodes on the frontier of the search are stored in an *open list* (usually a priority queue), and all the children of a node are added to the frontier when it is expanded.

To begin, the start node is placed into the open list. The cost to reach a node is called g, while the heuristic estimate of the cost from the node to the goal node is called h, and the sum of these two values is f.

A* is guaranteed to return an optimal path if the heuristic evaluation function is *admissible*. A heuristic is admissible if it never overestimates the cost of the optimal path from any node to the goal node. [1]

Nodes are sorted in the open list by their f values. The search itself is a simple loop. While there remain nodes in the open list, pop the highest ranked node n from the list. If

[1]The A* literature usually considers admissible heuristics which are lower bounds on the cost of an optimal path. Since sequence alignment works with a score function instead of a cost function, we want to maximize the score rather than minimize the cost. As a result, our discussion in the next section reverses the meaning of upper and lower bounds with respect to traditional A* literature.

n has been visited before (we also maintain a closed list of nodes visited in the past), by some other path, we compare the score of the current path to the previous path, and only expand the node if it is better. If this node is a goal node, we remember it if it has a lower cost than any previously found path to a goal.

To expand a node, we place the children of n into the open list. If any children have an f value higher than the lowest cost found so far, they can be pruned by not placing them into the open list. Finally, node n is placed onto the closed list. When there are no longer nodes left on the open list, the best path found to the goal will be the optimal path.

2.5.1 IDA*

A major problem with A* is that it may require a large amount of memory in order to store all of the nodes in the open and closed lists. IDA* (Iterative Deepening A*) [8], removes the memory constraint, but fails at sequence alignment problems due to the combinatorially explosive number of equivalent paths through the grid. Since IDA* has no memory of past search, it explores all paths to each node. Korf reported that in his experiments, IDA* was unable to solve sequence alignment problems larger than 10x10 [9].

2.5.2 Divide and Conquer Frontier Search

Korf introduced a search algorithm for sequence alignment called divide-and-conquer frontier A* (DCFA*) [10]. This algorithm behaves like A*, but stores only the open list. Since the closed list is not stored, the path cannot be reconstructed once a goal node is found, as it cannot retrace its steps. Instead, it uses a divide and conquer method. When the search crosses a designated boundary at the center of the search-space, each node afterward remembers, via propagation, which node from its path went through the boundary. When the goal node is reached we can simply check which boundary node was on its path (an optimal one) and use that as the fulcrum of the divide-and-conquer step. By not storing the closed list, the memory requirements of A* are avoided. This is quite similar in behavior to Hirschberg's dynamic programming algorithm, and is presented as a generalization of it.

2.5.3 Partial Expansion A*

Ishida et al, have worked on a variation of A* using partial node expansion [13]. Instead of inserting all child nodes into the open list (an expensive operation), this algorithm only generates the most promising nodes. If a node has unpromising children, it is reinserted back into the open list with the value of the best unpromising child node. Since nothing is lost (the node can always be re-expanded again later), correctness is maintained. Unpromising nodes are never generated, so the memory they would otherwise require is saved (as well as the time spent adding them to the open list). This method is currently one of the most successful for aligning several sequences together at once.

3 Bounded Dynamic Programming to Approximate A*

One of the largest problems with the A* and its variants, is that even without the closed list these algorithms are still memory hungry. Furthermore, there is a high cost in managing the open list, and in estimating upper and lower bounds for pruning and ordering.

One of the nice features of the sequence alignment problem is that the search space is well structured and regular. We know the neighbors of any node, and all paths are monotonically increasing towards the goal node (not in cost, but in the number of nodes remaining to reach the goal node).

Presented here is an algorithm for exploiting the best traits of both Korf's DCFA* and the dynamic programming methods. It attempts to use the low overhead of dynamic programming coupled with the pruning abilities of A* to do less overall work than other algorithms. The algorithm is called *Linear Bounded Diagonal Alignment* or LBD-Align, for reasons that will be discussed below.

The major modification to the dynamic programming algorithm is to progress along the antidiagonals (diagonals running from upper-right to lower-left) of the matrix, rather than row by row. This may seem strange but, as we will see shortly, it allows for an efficient method for pruning away useless portions of the matrix.

The algorithm works as shown in the left of Figure 5. The dynamic programming matrix is computed one diagonal at a time, starting in the top-left corner, and working down towards the bottom-right. Changing to a diagonal traversal does not compromise the correctness, since it preserves the proper ordering. For each cell, the values of the three parents that a cell requires (to the top, left, and top-

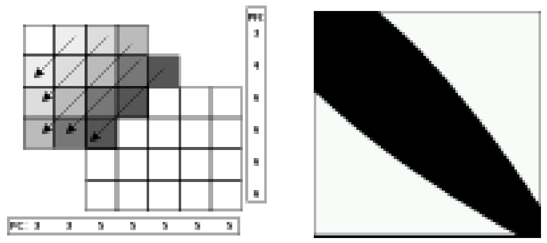


Figure 5. A small LBD-Align matrix (left), and a sample 5000x5000 matrix showing the computed region (right).

left) will already have been computed by the two previous diagonals.

All that has been done so far is to change the order in which cells are computed in the matrix. This change allows us to prune unnecessary regions of the matrix at a very low cost. A pruning test, comparing upper and lower bound estimates, is done to determine if a portion of the search space can be logically eliminated from consideration (this is discussed in more detail in Section 3.1). A* would perform this test every time a node is expanded, resulting in a higher overhead cost. LBD-Align only does two pruning tests per diagonal – a linear amount in the size of the problem.

Only the first and last cells on each diagonal (we will call this the pruning frontier) are tested for pruneability. For instance, if the top cell on a diagonal can be pruned, then all remaining cells to the right of that cell can also be pruned since any path through them must come through that pruned cell. Since the top cell on the next diagonal is to the right of the pruned cell, we can move our pruning frontier down one row and the size of the next diagonal will shrink. Similarly, the remainder of columns can be pruned from the bottom frontier. We continue shifting the frontier with each successful pruning, effectively shrinking the size of the matrix. The image on the right of Figure 5 is generated from a 5000x5000 matrix. The black area shows the computed region and the white space shows the pruned, uncomputed region of the matrix.

We can keep track of what has been pruned with a list of numbers for the extent of each row and column. PR is the pruned-row list. In Figure 5, since row zero has been pruned after the cell at the third position, we store a three at PR[0]. Likewise for PC, the pruned-column list. For rows and columns that have not been pruned yet, the value of their length is stored instead, indicating that all cells are still valid. [2]

Since this algorithm only does a linear number of bounds checks within an $O(n^2)$ algorithm, the overhead of this extra work stays at a minimum (in fact, it is nearly free). Since the checks are always on the frontier, it focuses the extra work of pruning on the areas that are most likely to benefit. Trying to prune cells near the optimal path will usually be futile. A* will often manage to prune more of the search space than LBD-Align, but at a higher cost per node.

If the initial lower bound estimation of the optimal score is poor, the pruning will not be as dramatic as it could be. One technique to improve the pruning is to occasionally re-estimate the lower-bound. If the algorithm detects that the score has improved locally at a cell, it can choose (at most once per diagonal) to re-estimate the lower bound at the promising cell. Re-estimation should not be done too often, otherwise the extra overhead may not be worthwhile.

[2]The PR and PC arrays are not actually required to implement this algorithm, but are included to simplify the discussion. Instead, a single value for each dimension can store the previous row or column's cut-off.

Cache usage is an important consideration for the actual performance of this algorithm on modern computers. When a memory location is accessed the cache line is filled with a sequential chunk of memory near the accessed location. A traditional raster scan of a matrix thus uses the cache effectively. Traversing along a matrix diagonally has poor locality of reference, and cache reuse will be extremely low as a result. To battle this problem, we store the matrix at a forty-five degree angle, so that each antidiagonal is sequentially located in memory.

3.1 Heuristics for Upper and Lower Bounds

To prove that a cell is not a part of any optimal path we need to have good estimations of the optimal score. The lower bound is a score less than or equal to the optimal score. Likewise, the upper bound is always greater than or equal to the optimal score. If a cell does not lie within this window, we can safely prune it. The closer our high and low estimations are to the optimal score, the more cells will fall outside of the window. Getting accurate bounds is crucial to successful pruning.

3.1.1 Lower Bounds

One quick and easy way to estimate a fairly inaccurate, but admissible lower bound on the optimal score, is to take the score of an arbitrary path through the matrix. For example, we can take the score of aligning the strings directly without inserting any gaps. This is equivalent to taking the path through the main diagonal of the matrix (we will refer to it as the *diagonal heuristic*). Not surprisingly, this gives a weak bound with little heuristic power.

A greedy, local alignment technique can also generate a more reasonable estimate. This greedy method chains the results of several small dynamic programming matrices to compute a reasonable path (Figure 6). Instead of computing the full rectangle, only the upper-triangular half is computed, one antidiagonal at a time. Since we do not need to reconstruct the path in this case (we are only interested in the score), only the last two diagonals are needed in memory to compute the values. Once the upper-triangular matrix is computed, the maximum score on the frontier is found and that point is used as the starting point for another small local search. This process continues until the bottom-right corner is reached, where the final score is then returned.

This greedy algorithm is far more accurate than the diagonal heuristic. However, it is only useful for the initial lower bound guess because it is far more expensive to compute than the diagonal heuristic. For re-estimations within the search, the diagonal heuristic is still used.

Much better lower bounds can be quickly computed from linear time algorithms like BLAST [1] or FASTA [12].

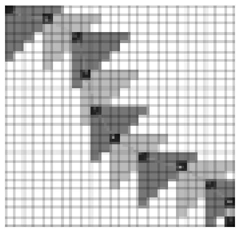


Figure 6. A greedy 'headlight' search of the matrix for lower bounds.

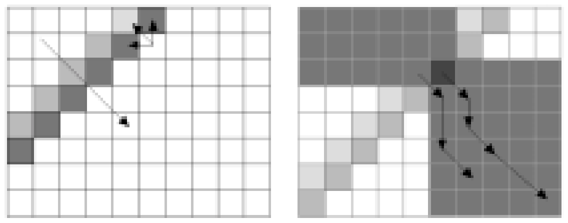


Figure 7. Divide and Conquer Linear Bounded Diagonal Alignment.

While the scores generated from an algorithm like BLAST are not comparable to our scoring method, the alignment produced can then be scored using our own scoring function.

3.1.2 Upper Bounds

For a an upper bound heuristic, we can use a technique similar to Manhattan distance [8]. The estimate of cost is calculated by assuming we match the remaining characters perfectly and then take any gap penalties (which are of zero cost in a real biological scoring function) if the remaining strings are not of equal length. This is an admissible heuristic since it is overly-optimistic, always overestimating the optimal score.

Using these lower and upper bounds, when we reach a cell whose score plus the heuristic estimate of the remaining score, is worse than the lower bound, then the optimal path cannot go through the cell in question. We may prune that cell, and cells that can only be reached by a path through it.

3.2 Linear-Space LBD-Align

The bounded dynamic programming algorithm given above can reduce the number of computations substantially, especially when good bounds are given. It pays a very low overhead for doing so, but since a full matrix is used it is still constrained by memory.

Fortunately we can take the the ideas from Hirschberg's algorithm and Korf's DCFA*, and apply it to this method as well. Instead of storing the entire matrix, we can just store a few diagonals from the matrix. In fact, we can compute the entire matrix using just two diagonals, overwriting one of them as it sweeps across the matrix. Figure 7 shows a diagram of the diagonal matrix computation using just two diagonals (left). On the right, the matrix has been computed and a pivot on the optimal path has been determined. The shaded areas represent the two new sub problems. Just as Korf's DCFA* stores the point at which a node's path crosses the middle of the matrix, we can also store this information using two more lists, and propagate the values to the goal. When the bottom-right cell is finally computed, we can look up which cell its path crossed the middle diagonals at, and use that point to break the problem into two smaller problems.

As an added bonus during the recomputation stage, if we save the middle diagonals we now have the optimal scores for the two sub problems. When we solve the sub problems with the optimal score given as the lower bound, we get the best possible pruning during the recomputations.

In total, this method needs only about $6n$ integers for its memory storage requirements: two diagonals for computing the matrix, two saved middle diagonals, and two more to store the propagated crossing points.

This linear memory version of LBD-Align will be referred to as DCLBD-Align, for *Divide and Conquer LBD-Align*.

4 Experiments

Several different sequence alignment programs were implemented (in C) for comparison to one another. For a basic reference, the Needleman-Wunsch dynamic programming, or 'full-matrix' algorithm was implemented. A linear-space program using Hirschberg's method was also written. Finally, an implementation of an A* program for sequence alignment was also built. [3]

For all the programs, a real scoring matrix was used (the scoring matrix shown in Table 1, with a linear gap penalty of -150). Experiments with affine gaps will be detailed later. Benchmarks were taken on a 933 mHz P-III Linux PC with 512 Mb of RAM, and a 256k cache. A wide range of real DNA sequences were used. Some sequences were highly similar to each other, while others were extremely dissimilar.

For small alignment problems there is little advantage in using LBD-Align. Any gains in pruning cells from an

[3]The A* implementation could likely be improved, but not significantly – many optimizations beyond the basic A* algorithm were implemented to improve the search performance.

n	Full Matrix	LBDA	DCLBDA	Hirschberg	A*
3000	0.46	0.33	0.37	0.60	6.04
5000	1.27	0.90	1.03	1.70	20.60
8000	3.16	1.70	2.18	4.08	*178.26*
10000	5.64	3.03	3.97	7.26	-
15000	*71.19*	*52.50*	11.81	13.32	-
40000	-	-	48.73	114.24	-
100000	-	-	384.84	921.96	-
230000	-	-	2454.64	5565.12	-

Table 2. Best times (in Seconds) for each method on DNA alignment problems of varying sizes.

already small matrix, are generally not worth the overhead. A simple full-matrix algorithm will usually execute fastest when aligning small sequences.

For all large alignments, the LBD-Align methods were substantially faster than the others. With extremely large problems, the DCLBD-Align version pulls far ahead of Hirschberg's algorithm, since the benefits of pruning become much greater. Results are given in Table 2. The greedy headlight search described earlier was used to obtain lower bound estimates. Italicized values represent the point at which the algorithm ran out of main memory and was required to swap to disk.

4.1 Pruning

Profiling of the programs revealed that in the LBD-Align programs, bounds checking accounted for less than two percent of the total execution time, most of which is incurred during the initial lower bound calculation.

The scoring function has a huge effect on pruning. For instance, if gaps are penalized strictly, more pruning is done than would be with a less expensive gap cost – the horizontal and vertical moves that occur on the pruning frontier will drop faster and as a result be pruned sooner. For instance, reducing the gap penalty from -150 to -100 in the test cases above causes the pruned amounts to drop by about 15%.

The effect of having a good lower bound is shown in Table 3 on an example 10000x10000 alignment. The first column shows the initial lower bound estimate for the alignment. The second column gives the percentage of the matrix that was computed, and the third column gives the total execution time on our test machine. The diagonal heuristic for this example estimates the score to be -642420, while a greedy search estimates it at 80349. The actual optimal score is 139536. The amount of the matrix that must be computed depends heavily on the quality of the estimate. With the worst lower bound, 51.35% of the matrix was computed. When the optimal score was used as a bound, only 35% of the matrix was calculated. Figure 8 shows a graphical representation of the difference between the diagonal heuristic and the greedy search. The black region is the portion of the matrix that was computed when the greedy heuristic was used and the white regions are portions that were pruned, and thus never visited. The grey shows the amount computed when the diagonal heuristic was used.

Lower Bound	Computed	Time (sec.)
-642420	51.35%	5.22
0	42.51%	4.40
80349	38.52%	3.97
139536	35.54%	3.62

Table 3. Effect of lower bounds on pruning.

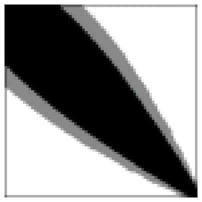


Figure 8. A 10000x10000 alignment matrix showing regions computed using different lower bounds.

4.2 Leading, Trailing, and Affine Gap Penalties

To handle affine gaps, the implementation details become more complicated. Extra state information is needed at each cell, to keep track of whether the gaps are being opened or extended. Since leading and trailing gaps are also free, aligning two sequences of significantly different lengths hinders pruning – the smaller sequence can slide all the way towards the far end of the longer sequence at zero cost, potentially aligning with a high score in that region. Pruning is deferred until that region is reached. Figure 9 demonstrates this problem clearly. The 650x650 alignment matrix shown on the left was computed with a linear gap cost, while the matrix shown on the right was calculated with an affine gap cost (including free leading and trailing gaps). The affine cost greatly reduces the pruning. However, when affine gaps are implemented, more computation and more memory is required for each cell in the matrix. Thus the benefits of pruning are larger. Results are pending.

5 Conclusions and Future Work

There are two important directions to take this bounded dynamic programming method further. In a full-scale sys-

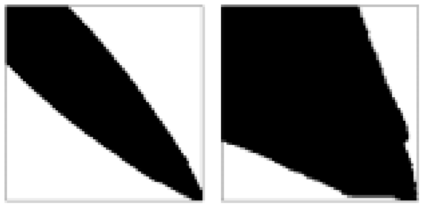


Figure 9. The area of a 650x650 alignment matrix computed with a linear gap cost (left), and an affine gap cost (right).

tem the initial lower bound estimate should be much better, coming from a quick local alignment method such as BLAST or FASTA.

Secondly, LBD-Align can be extended for multiple sequence alignment. Implementing the algorithm for an arbitrary dimensional matrix will be much more difficult than the two-dimensional case. However, this method uses less memory than Korf's DCFA*, has far less computational overhead, but can still prune away large portions of the search-space.

These results also need to be compared to FastLSA, one of the best sequence alignment algorithms. Some of the ideas from FastLSA could potentially be used in this approach as well. For instance, several sets of diagonals could be stored, in both directions, helping to further reduce recomputation.

The LBD-Align algorithm has very clear strengths and weaknesses. Several factors affect its ability to prune away portions of the matrix. If the sequences being aligned are extremely dissimilar, or of very different lengths, substantial pruning will be almost impossible. Similarly, if the initial lower bound estimate is far lower than the optimal score, the pruned region will be small.

LBD-Align would not perform well as a first-order sequence database search tool, since the majority of sequences would be very dissimilar to the probe sequence. However, it would work extremely well in conjunction with a faster, non-optimal alignment algorithm such as gapped BLAST. A BLAST search would filter out extremely dissimilar sequences, and could then pass promising sequences to LBD-Align for optimal alignments. The alignments from the BLAST searches could be used as an extremely good lower bound estimate for LBD-Align, ensuring excellent cut-offs.

A* can prune more effectively, but its high overhead makes it too slow for aligning only two sequences. LBD-Align can be considered a 'cheap A*' for sequence alignment. In nearly any situation where two sequences are known to be similar prior to alignment, LBD-Align should significantly beat any other well-known algorithm for optimal sequence alignment.

5.1 Acknowledgements

This research was funded by iCORE. I'd like to thank Jonathan Schaeffer, Darse Billings, Duane Szafron, Deepak Kamnasaran, and Martin Müller for extensive discussions and proofreading.

References

[1] S. Altschul, W. Gish, W. Miller, W. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Mathematics*, 48(5):1073–1082, 1988.

[3] K. Charter, A. Driga, P. Lu, J. Schaeffer, D. Szafron, and I. Parsons. Fastlsa – a fast linear-space algorithm for sequence alignment. *To be published*, 2001.

[4] K. Charter, J. Schaeffer, and D. Szafron. Sequence alignment using fastlsa. In *Proceedings of The 2000 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS'2000)*, pages 239–245, 2000.

[5] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107, 1968.

[6] D. S. Hirschberg. A linear-space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[7] H. N. Jr, D. D. II, and A. Ropelewski. Strategies for searching sequence databases. *Biotechniques*, 28(6), 2000.

[8] R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[9] R. Korf. Divide-and-conquer bidirectional search: First results. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1184–1189, 1999.

[10] R. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2000)*, pages 910–916, 2000.

[11] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[12] W. Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in Enzymology*, 183:63–98, 1990.

[13] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2000)*, pages 923–929, 2000.