Divide-and-Conquer Frontier Search Applied to Optimal Sequence Alignment

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Weixiong Zhang
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
zhang@isi.edu

Abstract

We present a new algorithm that reduces the space complexity of heuristic search. It is most effective for problem spaces that grow polynomially with problem size, but contain large numbers of short cycles. For example, the problem of finding an optimal global alignment of several DNA or amino-acid sequences can be solved by finding a lowest-cost corner-to-corner path in a d-dimensional grid. A previous algorithm, called divide-and-conquer bidirectional search (Korf 1999), saves memory by storing only the Open lists and not the Closed lists. We show that this idea can be applied in a unidirectional search as well. This extends the technique to problems where bidirectional search is not applicable, and is more efficient in both time and space than the bidirectional version. If n is the length of the strings, and d is the number of strings, this algorithm can reduce the memory requirement from $O(n^d)$ to $O(n^{d-1})$. While our current implementation of DCFS is somewhat slower than existing dynamic programming approaches for optimal alignment of multiple gene sequences, DCFS is a more general algorithm [1]

Introduction: Sequence Alignment

While we present a completely general heuristic search algorithm, it was motivated by a problem in computational biology, known as sequence alignment. Consider the two DNA sequences ACGTACGTACGT and ATGTCGTCACGT. The problem is to align these sequences, by inserting gaps in each one, so that the number of matches between corresponding positions is maximized. For example, if we insert gaps as follows: ACGTACGT_ACGT and ATGT_CGTCACGT, then all the letters in corresponding positions are the same, except for the substitution of T for C in the second position.

The optimal solution to this problem is defined by a cost function. For example, we may charge a penalty of one unit for a mismatch or substitution between characters, and two units for a gap in either string. The cost of an alignment then is the sum of the individual substitution and gap costs. With this cost function, the alignment above has a cost of five, since there is one substitution and two gaps. The optimal alignment of a pair of strings is the alignment with the lowest cost, and the above alignment is optimal. Alignments are important for determining the structural similarity between different genes, and identifying subsequences that are conserved between them.

This problem can be mapped to the problem of finding a lowest-cost path from corner to corner in a two-dimensional grid (Needleman and Wunsch, 1970). One sequence is placed on the horizontal axis from left to right, and the other sequence on the vertical axis, from top to bottom. An alignment is represented by a path from the upper-left corner of the grid to the lower-right corner. Figure 1 shows the path that represents our example alignment. If there is no gap in either string at a given position, the path moves diagonally down and right, since this consumes both characters. The cost of such a move is zero if the corresponding characters match, or the substitution penalty if they differ. A gap in the vertical string is represented by a horizontal move right, since that consumes a character in the horizontal string, but leaves the position in the vertical string unchanged. Similarly, a gap in the horizontal string is represented by a vertical move down. Horizontal and vertical moves are charged the gap penalty. Given this mapping, the problem of finding an optimal sequence alignment corresponds to finding a lowest-cost path from the upper-left corner to the lower-right corner in the grid, where the legal moves at each point are right, down, and diagonally down and right.

This problem readily generalizes to aligning multiple strings simultaneously. For example, to align three strings, we find a lowest-cost path in a cube from one corner to the opposite corner. The cost of a multiple alignment is often computed as the sum-of-pairs cost, or the sum of each of the different pairwise alignments (Setubal and Meidanis, 1997). Equivalently, we can

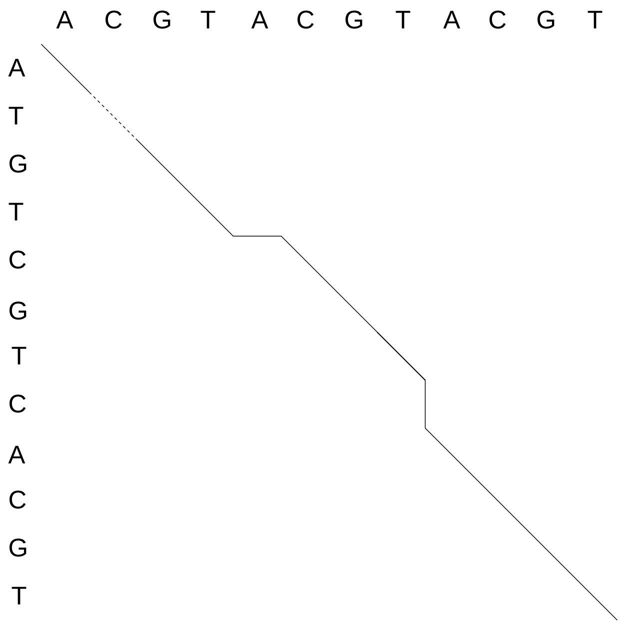

Figure 1: Sequence alignment as path-finding in a grid

score each character position by summing the cost of each of the character pairs. For example, if we have a C, a G, and a gap at one position, the cost at that position is two gap penalties plus a substitution penalty. If we have two gaps and a character at one position, the cost is two gap penalties, since no cost is charged for two gaps in the pair of strings that contain them.

Overview

We first discuss existing techniques for this problem. These include a dynamic programming algorithm (Hirschberg, 1975), a bounded dynamic programming algorithm (Spouge, 1989), and our previous best-first search algorithm, divide-and-conquer bidirectional search (DCBDS) (Korf, 1999). These methods save memory by only storing the frontier nodes of the search, and not the interior nodes. Our new algorithm, divide-and-conquer frontier search (DCFS), is closely related to DCBDS, but uses unidirectional rather than bidirectional search. This is more general, since bidirectional search is not always applicable, more efficient in time and space, and easier to implement. We also discuss the relative efficiencies of unidirectional and bidirectional search. For two-dimensional alignment, Hirschberg's algorithm seems to be the best. For three dimensions, however, an accurate heuristic evaluation function exists, and DCFS outperforms Hirschberg's algorithm and DCBDS, but is slower than bounded dynamic programming. All the algorithms apply to strings of unequal length, but we use strings of equal length in our experiments. Typical size problems that we can readily solve with these techniques are aligning pairs of strings of length 100,000 each, or three strings of length 6,000 each. The sizes of the corresponding grids are 10 billion nodes and 216 billion nodes, respectively.

Previous Work

For simplicity, we describe previous work in the context of aligning two strings, but all the algorithms can be generalized to multiple-string alignment as well. While most of the work in this area finds approximate alignments, we only consider here algorithms that are guaranteed to find optimal alignments.

Problems that Fit in Memory

If the grid is small enough to fit into memory, Dijkstra's single-source shortest path algorithm (Dijkstra, 1959) will solve the problem in $O(n^2)$ time and $O(n^2)$ space, for two strings of length n.

The particular problem of sequence alignment, as opposed to the general shortest-path problem, can also be solved by a much simpler dynamic programming algorithm. We scan the grid from left to right and from top to bottom, storing at each node the cost of a lowest cost path from the start node to that node. For each node, we add the gap cost to the cost of the nodes immediately to the left and immediately above, we add the substitution cost or no cost to the cost of the node diagonally above and to the left, and we store the smallest of these three sums in the current node. This also requires $O(n^2)$ time and $O(n^2)$ space.

Since both algorithms have to store the whole grid in memory, space is the main constraint. For example, if we assume that we can store 100 million nodes in memory, this limits us to aligning two strings of length 10,000 each, or three strings of length 464 each.

Problems that Don't Fit in Memory

The interesting case is when the grid doesn't fit in memory. One approach is to use a heuristic search, such as A* (Hart, Nilsson, and Raphael, 1968), to reduce the size of the search. This requires efficiently computing a lower bound on the cost of a given alignment, and has been applied to sequence alignment by (Ikeda and Imai, 1999). Unfortunately, A* stores every node it generates, and is still memory limited.

The memory limitation of best-first search algorithms like Dijkstra's and A* has been addressed (Korf, 1995). Many algorithms, such as iterative-deepening-A* (IDA*) (Korf, 1985), rely on depth-first search to avoid this memory limitation. The key idea is that a depth-first search only has to store the path of nodes from the start to the current node, and hence only requires space that is linear in the maximum search depth.

While depth-first search is very effective on problem spaces that are trees, or only contain a small number of cycles, it is hopeless on problem spaces with a large number of short cycles, such as a grid. The reason is that a depth-first search must generate every distinct path to a given node. In an $n \times m$ grid, the number of shortest paths from one corner to the opposite corner, ignoring diagonal moves, is $(n+m)!/(n! \cdot m!)$. For example, a 10×10 grid, which contains only 100 nodes, has $184,756$ different corner-to-corner paths, and a 25×25 grid, with only 625 nodes, has over 10^{14} such paths. Adding diagonal moves increases these numbers further. As a result, depth-first searches are completely hopeless on this problem.

Other techniques, such as caching some nodes that are generated, have been applied to sequence alignment (Miura and Ishida, 1998). The difficulty with these techniques is that they can only cache a small fraction of the total nodes generated on a large problem, and can only solve relatively easy problems.

Hirschberg's Algorithm

(Hirschberg, 1975) presented an algorithm for computing a maximal common subsequence of two character strings in linear space, based on a two-dimensional grid, with each of the strings placed along one axis. A node of the grid corresponds to a pair of initial substrings of the original stings, and contains the length of a maximal common subsequence of the substrings.

The standard dynamic programming algorithm for this problem requires $O(n^2)$ time and $O(n^2)$ space, for two strings of length n. To compute an element of the grid, however, we only need the value immediately to its left and the value above it. Thus, we can solve the problem by only storing two rows at a time, deleting each row as soon as the next row is completed. In fact, only one row is needed, since we can replace elements of the row as soon as they are used. Unfortunately, this only yields the length of a maximal common subsequence, and not the subsequence itself.

Hirschberg's algorithm computes the first half of this grid from the top down, and the second-half from the bottom up, storing only one row at a time. Then, given the two versions of the middle row, one from each direction, it finds a column for which the sum of the two corresponding elements from each direction is a minimum. This point splits both original strings in two parts, and the algorithm is then called recursively on the initial substrings, and on the final substrings. Hirschberg's algorithm is easily generalized to solve the sequence alignment problem. It can also be generalized to more than two dimensions. These generalizations reduce the space complexity of the d-dimensional alignment problem from $O(n^d)$ to $O(n^{d-1})$, a very significant reduction. The additional cost in time is only a constant factor of two in two dimensions, and even smaller in higher dimensions.

Bounded Dynamic Programming

This algorithm can be improved by using upper and lower bounds on the cost of an optimal solution (Spouge, 1989). For pairwise alignments, a lower bound on the cost to reach the lower right corner of the grid is the gap penalty times the number of gaps needed to reach the corner. For multiple sequence alignments, a much more effective lower bound is available, which will be described in the section on experimental results. An upper bound on the cost of an optimal alignment is the cost of aligning the strings directly, with no gaps in either one. Given an upper bound on the optimal alignment cost, and a lower bound on the cost of aligning any pair of substrings, we can limit the dynamic programming algorithm to that region around the main diagonal in which the cost to reach each node, plus the estimated cost to reach the goal, is no greater that the upper bound on solution cost. For the top-level search, we start with an upper bound equal to the lower bound, and run a series of iterations, incrementally increasing the upper bound until it equals or exceeds the actual optimal alignment cost, and the iteration aligns the entire strings (Ukkonen, 1985). After each recursive search completes, we know the optimal solution costs of the next recursive searches, and use those values for the upper bounds. We refer to this algorithm as iterative-deepening bounded dynamic programming (IDBDP). These ideas are also used in MSA (Gupta, Kececioglu, and Schaffer, 1985), one of the best programs for optimal multiple sequence alignment.

Divide-and-Conquer Bidirectional Search

Divide-and-conquer bidirectional search (DCBDS) (Korf, 1999) generalizes Hirschberg's dynamic programming algorithm to arbitrary path-finding problems. To apply dynamic programming, we have to know in advance which neighbors of a node are its ancestors and which are its descendents, in order to evaluate the ancestors of a node before the node itself. We can do this in the sequence alignment problem because only moves down, right, and diagonally down and right are allowed, making the nodes above, to the left, and diagonally up and left the ancestors. In the general case, where moves to any neighboring nodes are allowed, we can't apply dynamic programming, but rather must apply a best-first search such as Dijkstra's algorithm. DCBDS achieves the same memory savings of Hirschberg's algorithm, but for the general shortest-path problem in any graph.

A best-first search, such as Dijkstra's or A*, stores both a Closed list of nodes that have been expanded, and an Open list of nodes that have been generated, but not yet expanded. The Open list corresponds to the frontier of the search, while the Closed list corresponds to the interior region. In the A* cost function, $f(x) = g(x) + h(x)$, $g(x)$ is the cost from the initial state to node x, and $h(x)$ is a heuristic estimate of the cost from node x to a goal. If h has the property that for all nodes x and their neighbors x', $h(x) \leq c(x, x') + h(x')$, where $c(x, x')$ is the cost from node x to its neighbor x', we say that h is *consistent.* Since consistency is similar to the triangle inequality of all metrics, almost all naturally occurring heuristic functions are consistent. If the heuristic function is consistent, or in the absence of a heuristic function, once an Open node is expanded, an optimal path has been found to it, and it never is expanded again. In that case, we can execute a best-first search without storing the Closed list at all.

In an exponential problem space with a branching factor of two or more, the Open list is larger than the Closed list, and not storing the Closed list doesn't save much. In a polynomial space, however, the dimension of the frontier is one less than that of the interior, resulting in significant memory savings. For example, in a two-dimensional grid, the Closed list is quadratic in size, while the size of the Open list is only linear.

There are two challenges with this approach. The first is that duplicate node expansions are normally eliminated by checking new nodes against the Open and Closed lists. Without the Closed list, to prevent the search from "leaking" back into the closed region, DCBDS stores with each Open node a list of forbidden operators that lead to closed nodes. For each node, this is initially just the operator that leads to its parent. As each node is generated, it is compared against the nodes on the Open list, and if it already appears on Open, only the copy arrived at via a lowest-cost path is saved. When this happens, the new list of forbidden operators for the node becomes the union of the forbidden operators of each copy.

In some problem spaces the operators or edges are directed. For example, in the two-dimensional sequence alignment problem, the only legal operators from a node are to move down, right, or diagonally down and right. After expanding a given node and removing it from Open, if the node immediately above it has not yet been expanded for example, when it is expanded it will regenerate the given node, and place it back on Open. This will cause the search to leak back into the closed region, eliminating the space savings. One solution to this problem is that when a node is expanded, all of its neighboring nodes are generated, including nodes generated by an edge going the wrong way, such as the nodes above, to the left, and diagonally up and left in this case. These latter nodes are also placed on Open, but their cost is set to infinity, indicating that they haven't been arrived at via a legal path yet. By placing all the neighbors of a node on Open when it is expanded, we ensure that a closed node can't be regenerated, and prevent the search from leaking back into the closed region.

Saving only the Open list can be used to speed up the standard Dijkstra's and A* algorithms as well. It is faster not to generate a node at all, than to generate it and then search for it in the Open and Closed lists. On a two-dimensional grid, this technique alone speeds up Dijkstra's algorithm by over 25%.

The main value of this technique, however, is that it executes a best-first search without a Closed list, and never expands a state more than once. When the algorithm completes, it has the cost of an optimal path to a goal, but unfortunately not the path itself. If the path to each node is stored with the node, each node will require space linear in its path length, eliminating all of the space savings. In fact, this approach requires more space than the standard method of storing the paths via pointers through the Closed list, since it doesn't allow the sharing of common subpaths.

One way to construct the path is the following. Perform a bidirectional search from both initial and goal states simultaneously, until the two search frontiers meet, at which point a node on a solution path has been found. Its cost is the sum of the path costs from each direction. Continue the search, saving the middle node on the best path found so far, until the best solution cost is less than or equal to the sum of the lowest-cost nodes on each search frontier. At this point we have a node on a lowest-cost solution path. Save this node in a solution vector. Then, recursively apply the same algorithm to find an optimal path from the initial state to the middle node, and from the middle node to the goal state. Each of these searches adds another node to the final solution path, and generates two more recursive subproblems, etc, until the entire solution is reconstructed.

Divide-and-Conquer Frontier Search

Our new algorithm, divide-and-conquer frontier search (DCFS), also saves memory by storing only the Open list and not the Closed list, but using unidirectional rather than bidirectional search.

Consider our problem of finding an optimal corner-to-corner path in a two-dimensional grid. DCFS begins with a single search from the initial state to the goal state, saving only the Open list, in the same manner as described for DCBDS. When the search encounters a node on a horizontal line that splits the grid in half, each of the children of that node store the coordinates of the node. For every Open node that is past this halfway line, we save the coordinates of the node on the halfway line that is on the current path from the initial state to the given Open node. Once the search reaches the goal state via an optimal path, the corresponding node on the halfway line is an intermediate node roughly halfway along this optimal path. We then recursively solve two subproblems using the same algorithm: find an optimal path from the initial state to this middle node, and find an optimal path from the middle node to the goal node. If the original grid, or one defined by a recursive subproblem, is wider than it is tall, we choose a vertical line to represent the set of possible halfway nodes instead of a horizontal line.

In a grid problem space, we can easily identify a set of nodes that will contain a node roughly halfway along the optimal solution. In a three-dimensional grid, for example, the halfway line becomes a halfway plane, cutting the cube in half either horizontally or vertically. In a general problem-space graph however, identifying such nodes is only slightly more difficult. If we have a heuristic evaluation function, we can choose as a midpoint node one for which $g(x)$, the cost from the initial state to node x, equals $h(x)$, the heuristic estimate from node x to a goal node. If we don't have a heuristic function, but have an estimate of the total solution cost c, we can use as a halfway node any node x for which $g(x)$ is approximately $c/2$.

There are several advantages of unidirectional DCFS over bidirectional DCBDS. One is that unidirectional search is more general, and can be applied to problems that bidirectional search cannot be applied to. For example, if we only have a test for a goal, and don't have an explicit goal state in advance, we may not be able to apply bidirectional search. Secondly, unidirectional search may be more efficient, as we will see below. Finally, unidirectional search is simpler and easier to implement correctly.

Bidirectional vs. Unidirectional Search

Which is more efficient, bidirectional or unidirectional search? The answer is different for brute-force and heuristic searches. A common unidirectional brute-force algorithm is Dijkstra's algorithm or uniform-cost search, a best-first search using the cost function $f(x) = g(x)$. The algorithm terminates when a goal node is chosen for expansion, or the cost of a goal node is less than or equal to the cost of all Open nodes.

To guarantee an optimal solution, bidirectional uniform-cost search terminates when the cost of the best solution found so far is less than or equal to the sum of the minimum $f(x) = g(x)$ costs in the two search frontiers. Thus, the two search frontiers only go half the distance to the goal, instead of one frontier extending all the way to the goal. As a result, bidirectional uniform-cost search usually expands fewer nodes than unidirectional uniform-cost search, with the difference increasing with increasing branching factor of the problem space.

The situation is different for A*, however, which uses $f(x) = g(x) + h(x)$ for its cost function. To guarantee optimal solutions, bidirectional A* terminates when the best solution found so far costs no more than the minimum $f(x) = g(x) + h(x)$ cost in either direction. Since this is the same terminating condition for each of the unidirectional searches, bidirectional A* generates more nodes than unidirectional A*, by virtue of performing two such searches. We could also terminate bidirectional A* when the sum of the minimum $g(x)$ costs in the two directions exceeds the cost of the best solution so far, but with an accurate heuristic function the former terminating condition will occur first.

In addition to the number of nodes generated, we also need to consider the time per node generation. In our experiments, described below, various overheads in bidirectional search made it more expensive per node generation than the unidirectional version.

In terms of memory, unidirectional search only has to maintain a single search horizon, while bidirectional search has to maintain two. This can result in up to a factor of two difference in space complexity. In practice, the difference may be smaller, since the unidirectional search horizon will be larger than an individual bidirectional search horizon.

Finally, unidirectional search is much simpler to implement than bidirectional search. Thus, unidirectional heuristic search is often preferable to bidirectional heuristic search because it generates fewer nodes, takes less time per node generation, requires less memory, and is easier to implement. For a more thorough treatment of bidirectional heuristic search, see (Kaindl and Kainz, 1997).

Experimental Results

We tested our algorithms on random sequence alignment problems. Each triple of DNA base pairs encodes one of 20 different amino acids. For each problem instance we generated random 20-character strings, simulating amino-acid sequences, and computed an opti-

Length	DCBDA*			DCFA*			IDBDP		
	Nodes	Mbytes	Seconds	Nodes	Mbytes	Seconds	Nodes	Mbytes	Seconds
1000	4,385,088	48	4.66	2,428,388	15	1.55	1,478,166	20	2.38
2000	45,956,680	119	50.04	24,147,383	56	17.05	12,419,072	78	15.78
3000	184,459,312	248	212.01	94,778,522	117	77.99	46,080,688	173	55.66
4000	464,723,712	472	665.65	236,315,529	221	192.33	111,106,632	306	136.96
5000	979,234,880	575	1398.18	496,715,891	272	498.53	237,084,672	478	275.41
6000				883,732,606	463	896.78	466,050,656	689	556.162

Table 1: 3-Way Alignment of Random 20-Character Strings

mal alignment between them. We chose random problems to allow generating a large number of problem instances. Our cost function charges nothing for a match, one unit for a substitution, and two units for a gap.

For aligning two strings, the best lower-bound heuristic is the number of gap penalties required to reach the bottom-right corner. This allows us to use the A* cost function, $f(x) = g(x) + h(x)$, with DCBDS and DCFS. We refer to the A* versions of these algorithms as divide-and-conquer bidirectional A* (DCBDA*) and divide-and-conquer frontier A* (DCFA*), respectively. This heuristic is relatively weak, however, and both algorithms examine a large fraction of the nodes in the grid. Since Hirschberg's algorithm has significantly lower overhead per node, it is more efficient in this case. In practice, Hirschberg's algorithm will optimally align two strings of length 100,000 in about 21 minutes, on a 440 megahertz Sun Ultra 10 workstation, the machine used for all our experiments. This requires generating 20 billion nodes.

In three dimensions, corresponding to the simultaneous alignment of three strings, there is a much more effective lower-bound heuristic function available. Recall that the cost function for multiple-string alignment is the sum-of-pairs cost, meaning the sum of the costs of each of the pairwise alignments induced by the three-way alignment. To compute this heuristic function, we optimally align each pair of strings, and then use the sum of the optimal pairwise alignments as a lower bound on the cost of the best three-way alignment. This is a very accurate heuristic function, and greatly reduces the number of nodes that are expanded by the best-first algorithms. The reason this heuristic is not the same as the actual cost is that in the optimal three-way alignment, each of the pairs of strings will not be optimally aligned in general. The reason it is a lower bound is that the cost of the actual alignment of each pair induced by the three-way alignment must be at least as great as the cost of their optimal pairwise alignments. To compute the optimal pairwise alignments, we use the standard dynamic programming algorithm, since it is the most efficient for pairwise alignment. These values are precomputed and stored, instead of recomputed for each node.

For each problem instance, we generated three random strings, and optimally aligned them. Table 1 shows the average results of 100 problem instances for each case. We ran three different algorithms, all of which return optimal alignments. For each algorithm we give the average number of nodes generated, the amount of memory used in megabytes, and the average running time per problem instance in seconds.

Hirschberg's algorithm is not competitive in more than two dimensions, since it doesn't use a lower bound function, and generates every node in the space at least once. For example, for three strings of length 1000, it generates 1.3 billion nodes, and takes 405 seconds to run, compared to a few seconds for the other algorithms.

The first algorithm in the table is divide-and-conquer bidirectional A* (DCBDA*), and the second is divide-and-conquer frontier A* (DCFA*), both using the same heuristic function. DCFA* generates about half the nodes of DCBDA*, and runs more than twice as fast, due to higher overhead per node. It also uses about half the memory. 640 megabytes was not enough memory to run DCBDA* on strings of length 6000.

The last algorithm in the table is iterative-deepening bounded dynamic programming (IDBDP), described above in the previous work section. IDBDP generates fewer nodes than DCFA*, primarily because DCFA* applies operators in all directions, including the illegal backward directions, to keep the search from leaking into the closed region. IDBDP also runs faster than DCFA* on strings of length 2000 or longer. Our current implementation of IDBDP uses more memory than DCFA*, but this can be reduced.

It should be noted that the absolute performance of these algorithms is sensitive to the cost function, the size of the alphabet, and the correlation of the strings. Thus, these results can only be used for relative comparison of the different algorithms. For example, changing the alphabet to 4 characters to simulate random DNA strings degrades the performance of all three algorithms significantly. On the other hand, since real data is much more highly correlated than random strings, we expect the performance on real data to be significantly better.

Conclusions

We have generalized divide-and-conquer bidirectional search (DCBDS) to unidirectional divide-and-conquer frontier search (DCFS). DCFS is a completely general heuristic search algorithm. Like DCBDS, DCFS reduces the space complexity of finding a lowest-cost path in a d-dimensional grid from $O(n^d)$ to $O(n^{d-1})$. Unlike DCBDS, however, DCFS can be applied to problems that don't allow bidirectional search. In addition, DCFS uses less memory, runs faster, and is easier to implement than DCBDS.

We applied DCFA*, the A* version of DCFS, to find optimal alignments for three random strings of up to 6000 characters each. The performance of DCFA* is compared to that of existing dynamic programming algorithms for optimal sequence alignment of more than three strings. Our current implementation runs faster on problems of length less than 2000 characters, but slower on larger problems.

Acknowledgements

We'd like to thank Matt Ginsberg, Andrew Parks, Louis Steinberg, and Victoria Cortessis for helpful discussions on this research. This research was supported by NSF grants No. IRI-9619447 and IRI-9619554.

References

[1] Dijkstra, E.W., A note on two problems in connexion with graphs, *Numerische Mathematik*, Vol. 1, 1959, pp. 269-71.

[2] Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, Vol. SSC-4, No. 2, July 1968, pp. 100-107.

[3] Hirschberg, D.S., A linear space algorithm for computing maximal common subsequences, *Communications of the ACM*, Vol. 18, No. 6, June, 1975, pp. 341-343.

[4] Gupta, S.K., J.D. Kececioglu, and A.A. Schaffer, Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment, *Journal of Computational Biology*, Vol. 3, No. 2, 1995, pp. 459-472.

[5] Ikeda, T., and H. Imai, Enhanced A* algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases, *Theoretical Computer Science*, Vol. 210, No. 2, Jan. 1999, pp. 341-374.

[6] Kaindl, H., and G. Kainz, Bidirectional heuristic search reconsidered, *Journal of Artificial Intelligence Research*, Vol. 7, 1997, pp. 283-317.

[7] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[8] Korf, R.E., Space-efficient search algorithms, *Computing Surveys*, Vol. 27, No. 3, Sept., 1995, pp. 337-339.

[9] Korf, R.E., Divide-and-conquer bidirectional search: First results, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, August 1999, pp. 1184-1189.

[10] Miura, T., and T. Ishida, Stochastic node caching for memory-bounded search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, July, 1998, pp. 450-456.

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

[12] Setubal, J., and J. Meidanis, *Introduction to Computational Molecular Biology*, PWS Publishing, Boston, MA, 1997.

[13] Spouge, J.L., Speeding up dynamic programming algorithms for finding optimal lattice paths, *SIAM Journal of Applied Math*, Vol. 49, No. 5, 1989, pp. 1552-1566.

[14] Ukkonen, E., Algorithms for approximate string matching, *Information and Control*, Vol. 64, 1985, pp. 100-118.