Memory-Efficient A* Heuristics for Multiple Sequence Alignment

Matthew McNaughton and **Paul Lu** and **Jonathan Schaeffer** and **Duane Szafron**
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2E8
{mcnaught,paullu,jonathan,duane}@cs.ualberta.ca

Abstract

The time and space needs of an A* search are strongly influenced by the quality of the heuristic evaluation function. Usually there is a trade-off since better heuristics may require more time and/or space to evaluate. Multiple sequence alignment is an important application for single-agent search. The traditional heuristic uses multiple pairwise alignments that require relatively little space. Three-way alignments produce better heuristics, but they are not used in practice due to the large space requirements. This paper presents a memory-efficient way to represent three-way heuristics as an octree. The required portions of the octree are computed on demand. The octree-supported three-way heuristics result in such a substantial reduction to the size of the A* open list that they offset the additional space and time requirements for the three-way alignments. The resulting multiple sequence alignments are both faster and use less memory than using A* with traditional pairwise heuristics.

Introduction

The problem of aligning s DNA/protein sequences of (average) length t is one of the most important problems in computational biology today. Obtaining the optimal alignment can be expressed as a dynamic programming problem over a lattice. However, a naïve solution to this problem has time and space complexities of $O(t^s)$, which are prohibitively large for real-world alignments. There are several algorithms in the literature that can reduce the space needs to $O(t^{s-1})$ but this is still unacceptably large. The space issue has received attention from both the computing science (Hirschberg 1975; Korf 1999; Korf & Zhang 2000; Yoshizumi, Miura, & Ishida 2000) and the biology communities (Spouge 1989; Myers & Miller 1988; Chao, Hardison, & Miller 1994).

A* (and its variants) have been used for this problem. The exponential time concerns are dampened by the elimination of irrelevant portions of the search space. The need for space is determined by the size of the open list, and it varies considerably with the quality of the heuristic function (h) used by A*. Nevertheless, it is not unusual for A* to rapidly consume all available memory.

Most space-related single-agent-search research concentrates on the search algorithm. This is appealing since it can generally be done in an application-independent way. In contrast, evaluation function discussions tend to be either too general to be useful, or too specific so that it is application dependent. However, a better h results in a more focused A* search and, hence, a smaller open list. Using space to improve the heuristic evaluation function quality in exchange for a (presumably) smaller open list is an important issue. For example, the pattern database work applied to the sliding-tile puzzles used large tables of heuristic values to reduce the search tree sizes by many orders of magnitude (Culberson & Schaeffer 1998; Korf 2000).

This paper shows how the dynamic (re-)computation of heuristic information can reduce the space requirements for A*-based search. Specifically, multiple sequence alignment (MSA) is used to illustrate these ideas. The standard heuristic used is the sum of pairwise alignments (see Section 2). Using three-way alignments can improve the quality of the evaluation function, but this is usually not feasible because of the space requirements. In this paper we show how an octree can be used to dynamically calculate relevant portions of three-way (or more) alignments. By saving this information in a storage-efficient way and recomputing parts as needed, the overall space requirements for large multiple alignment problems are dramatically reduced.

Note that the memory technique presented in this paper provides space/time tradeoffs for computing A* bounds. The search algorithm—A* or one of its variants—is orthogonal. Hence, one could use one of the recent AI algorithms for solving these types of problems, such as Divide-and-Conquer Frontier Search (Korf & Zhang 2000) or Partial-Expansion A* (Yoshizumi, Miura, & Ishida 2000), to further enhance performance.

This research makes the following contributions:

1. a new, memory-efficient algorithm for computing and recomputing heuristics for lattice computations on an as-needed basis,
2. application of this algorithm to the optimal MSA problem, reducing both the time and memory requirements needed for large alignments, and
3. a case study showing that for heuristic functions which require a significant amount of space, a better evaluation

function can lead to *both* a time and space improvement.

Although this paper uses the MSA problem as a sample application, the ideas are not restricted to this problem. The message of this paper, that the investment of time and space for the generation of better quality heuristics can result in a win/win situation, has received little attention in the literature.

Section 2 motivates the multiple sequence alignment problem and the need for better quality heuristics. Section 3 introduces the octree and an algorithm for selecting a set of heuristics for the search. Section 4 analyzes the performance of our algorithm. Finally, Section 5 presents future work.

Multiple Sequence Alignment

Each DNA or protein sequence is represented by a sequence of letters over a restricted alphabet. For example, DNA uses A, T, G, and C to represent the four nucleic acids. During sequence alignment, gaps (denoted by -) can be inserted into the sequences to make more letters in the sequences align with each other. An optimal alignment is one that minimizes a scoring function that assigns scores to corresponding letters and/or gaps in the sequences. A simple scoring function assigns -2 for an exact match, +1 for a mismatch and +2 for each gap. For example, the optimal alignment of the sequences $s_1 =$ GATAGTC and $s_3 =$ AGGTGCA is:

```
-GATAGTC
AGGT-GCA
```

since this alignment has a score of +1(3 matches, 3 mismatches, and 2 gaps) (Table 1(c)) and all other alignments have a higher score.

There are efficient dynamic programming (DP) solutions to this problem. Given two sequences of length j and k, the naïve DP algorithms require an array of size $O(j \times k)$. They require time $O(j \times k)$, to compute the array score entries, and then time $O(j+k)$ to identify the minimum-score path in the matrix. Unfortunately, $O(j \times k)$ space can be limiting, especially given that a DNA sequence can be millions of characters long. Even aligning two (small) sequences of 10,000 requires a prohibitive amount of storage (i.e., 100,000,000 entries).

Hirschberg was first to report a way of aligning two sequences using linear space (Hirschberg 1975), by recomputing some values in a classic space-time tradeoff.

In multiple sequence alignment, three or more sequences must be aligned, so even with Hirschberg-style one-dimensional space reductions, DP algorithms still require too much storage. There are two approaches to solving the multiple-alignment problem. The most popular algorithms used by biologists are based on a series of progressive pairwise alignments to generate non-optimal multiple alignments (e.g., DCA (Reinert, Stoye, & Will 2000) and CLUSTAL W (Thompson, Higgins, & Gibson 1994)). In contrast, the large space requirement of the optimal multiple alignment problem is viewed as a research challenge by AI researchers interested in heuristic search (Korf 1999; Korf & Zhang 2000; Yoshizumi, Miura, & Ishida 2000).

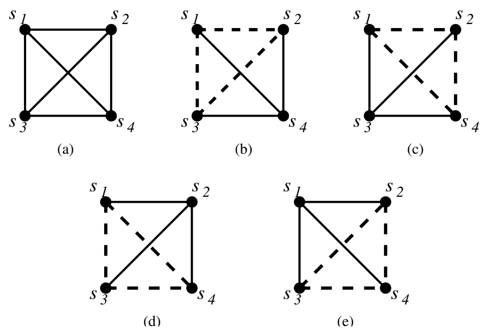

Figure 1: Covering a 4-way MSA with 2-way (solid line) and 3-way (dashed line) Alignments.

	$\overline{s_1, s_2, s_3, s_4}$	$\overline{s_1, s_2}$	$\overline{s_1, s_3}$	$\overline{s_1, s_2, s_3}$
s_1	GATAGT-C-	GATAGTC	-GATAGTC	GATAG-TC-
s_2	-ATAG-GC-	-ATAGGC		-ATAG-GC-
s_3	-A-GGTGCA		AGGT-GCA	-A-GGTGCA
s_4	-A-ATTGCA			
	(a)	(b)	(c)	(d)

Table 1: Optimal 4-way MSA, Example 2-way Alignments, and an Example 3-way MSA.

In A*, we want the lowest score for a goal node, where $f(n) = g(n)+h(n)$ of node n, $g(n)$ is the known score from the root node to n, and $h(n)$ is a heuristic score from n to the goal node. For the heuristic $h(n)$ to be admissible, for all nodes n, it must never be more than the actual value $h^*(n)$. As the search approaches a goal node, $f(n)$ approaches the actual score from below. For search efficiency, $h(n)$ should be a tight bound on $h^*(n)$.

Assuming there are s sequences to align, one technique for computing $h(n)$ is to align strict subsets of the original s sequences and combine the scores of the subsets. We call the s-way MSA the *full-sized alignment* and k-way alignments, where $k < s$, *lower-dimensional optimal alignments* (Reinert & Lermen 2000).

It is a theorem that the (locally) optimal alignment of any lower-dimensional k-way alignment of the sequences, where $k < s$, has a score that is no worse than the score attributed to the same k sequences in the full-sized s-way MSA (Carrillo & Lipman 1988). In particular, one way to construct $h(n)$ is to sum the optimal alignment scores of each pair of sequences. By the preceding, this sum is guaranteed to be an admissible value. To improve the quality of $h(n)$ and still be admissible, we may incorporate a larger, 3-way MSA.

Consider a 4-way MSA where s_1 = GATAGTC, s_2 = ATAGGC, s_3 = AGGTGCA, and s_4 = AATTGCA. We can represent the four sequences as the nodes of a graph, where the arcs are alignments between specific sequences (Figure 1). We denote the 3-way MSA of s_1, s_2, and s_3 as $\overline{s_1, s_2, s_3}$ and similarly for 2-way alignments. The standard definition for the score of a 4-way MSA is the sum of the constituent pairwise alignments. In other words, the scoring function for a 4-way MSA is composed of only 2-way alignments and does not include any 3-way alignment scoring functions. Therefore, the optimal 4-way MSA is the alignment with the lowest score when all possible pairwise alignments are considered. In particular, for a 4-way MSA, there are $\binom{4}{2} = 6$ different pairwise alignments that cover the four sequences (Figure 1(a), solid lines).

In our example, the optimal MSA $\overline{s_1, s_2, s_3, s_4}$ has a score of -12 and the alignment is shown in Table 1(a). Note how, in the context of a 4-way MSA, the alignment between s_1 and s_3 (in Table 1(a)) is different than $\overline{s_1, s_3}$ (Table 1(c)). The locally optimal $\overline{s_1, s_3}$ is not necessarily globally optimal for the 4-way MSA.

Now, suppose we want to compute the heuristic value $h(n)$ for the root node of the search tree (i.e., $g(root) = 0$) by summing the optimal alignment scores of each pair of sequences. $\overline{s_1, s_2}$ (Table 1(b)) has a score of -7, and $\overline{s_1, s_3}$ (Table 1(c)) has a score of 1. $\overline{s_1, s_4}$, $\overline{s_2, s_3}$, $\overline{s_2, s_4}$, and $\overline{s_3, s_4}$ have scores of -2, -1, -1, and -8, respectively (not shown). The sum of all the optimal pairwise alignments is -18 (Figure 1(a)): $h(n) = \sum_{1 \le i < j \le 4} \overline{s_i, s_j} = -18$. As discussed earlier, the optimal 4-way MSA has an actual score of -12, but $h(n) = -18$ is still an admissible heuristic value.

To improve the quality of $h(n)$ and still be admissible, we can incorporate a larger, 3-way MSA into our example. Therefore, instead of requiring six 2-way alignments to cover our 4-way problem (Figure 1(a)), we only need one 3-way and three 2-way alignments (Figure 1(b)). In fact, for a 4-way MSA, there are $\binom{4}{3} = 4$ possible 3-way MSAs and their corresponding 2-way alignments that cover the graph (Figure 1(b), (c), (d), (e)). Intuitively, a 3-way MSA better captures global trade-offs when aligning multiple pairs of sequences, and therefore 3-way MSAs should provide better $h(n)$ values than 2-way alignments.

Consider the optimal 3-way MSA $\overline{s_1, s_2, s_3}$ (Table 1(d)), which has has a score of -4. Using this 3-way MSA, we estimate $h(n)$ to be -15: $h(n) = \overline{s_1, s_2, s_3} + \overline{s_1, s_4} + \overline{s_2, s_4} + \overline{s_3, s_4} = -15$ (Figure 1(b)). Although -15 with a single 3-way MSA is still not ideal, it is a better $h(n)$ than the score of -18 obtained using only 2-way alignments. The closer $h(n)$ comes to the optimal value, the smaller the A* search tree becomes.

Since different choices of 3-way MSAs can result in different values of $h(n)$, the quality of the heuristic can depend on the specific choice of the 3-way MSA. For example, if we had used $\overline{s_1, s_2, s_4} = -10$, we would have obtained $h(n) = -18$ (Figure 1(c)). For $\overline{s_1, s_3, s_4} = -6$, $h(n) = -15$ (Figure 1(d)). Finally, for $\overline{s_2, s_3, s_4} = -6$, $h(n) = -18$ (Figure 1(e)). Therefore, for our example, using either $\overline{s_1, s_2, s_3}$ or $\overline{s_1, s_3, s_4}$ would give us the best $h(n) = -15$. However, choosing either of the two remaining possible 3-way MSAs results in a $h(n)$ estimate that is no better than using only 2-way alignments.

We know of only one work that attempts to use a 3-way alignment heuristic (Reinert & Lermen 2000). However, they only use part of a single 3-way alignment (due to storage constraints).

Representing Heuristics as an Octree

A sequence alignment heuristic that uses 3-way alignments is desirable but requires too much storage (t^3 for strings of length t). We use an octree to efficiently represent the three-dimensional (3D) dynamic programming tables of 3-way alignments. An octree stores only those portions of the table that are needed by the search, and efficiently regenerates omitted portions as they are needed. Octrees have been used traditionally in CAD, GIS, fluid dynamics simulations, and other 3D image processing applications to compress grid representations of space containing large uniform volumes.

An octree is a tree data structure that represents a 3D space using rectangular blocks (Figure 2). Each node in the octree represents a rectangular portion of the full dynamic programming table for three sequences. An internal node has eight children, each of which represents one octant (corner) of its parent's space in more detail. A leaf node is either 'empty' or 'full'. A 'full' leaf node contains all of the values of the dynamic programming matrix for the portion of the space it represents. An 'empty' leaf node contains only the values over the surface of the volume, so that the contents may be regenerated as needed. Obviously, the root node represents the entire table. The pseudo-code for octree construction is in Figure 3.

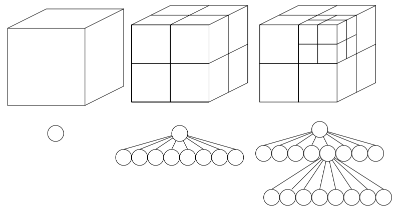

Figure 2: Octree Data Structure.

At the start of the A* search, the octree contains only internal nodes and 'empty' leaf nodes. As table values are requested by the search, 'empty' leaves containing the queried position are converted to internal nodes with new children. Children for areas beneath a threshold volume τ are created as 'full' nodes, and those above, as 'empty' nodes. 'Full' children can answer queries without further computation.

Our implementation uses $10,000$ for the leaf type threshold τ. Experiments indicate that this is a good trade-off between the fixed memory overhead for each node, the storage of unneeded table cells in excessively large 'full' nodes, and the time to traverse deep trees.

As Figure 1 shows, many 3-way alignments may be used. The program must decide which 2- and 3-way alignments to use for its heuristic function. To maintain admissibility, optimal 3-way alignments used in the heuristic are not permitted to have a pair of sequences in common, because each pair of sequences must contribute a score to the heuristic at most once (eg., in a set of five sequences, only two optimal 3-way alignments may be used). We determine which combination is best by evaluating each candidate heuristic at the root node and picking the one with the highest score. This heuristic is

```
Lookup(node,x,y,z):
  if node is Internal then
    octant o = octantFor(node.bounds,x,y,z)
    return Lookup(node.child[o],x,y,z)
  else if node is Full Leaf then
    return node.table[x][y][z]
  else if node is Empty Leaf then
    Node n = allocateInternal(node.bounds)
    addChildren(n)
    // regenerate table from node.surface
    node = n
    return Lookup(node,x,y,z)
  endif
end Lookup

addChildren(node):
  for each octant o of node.bounds
    if volume(o) > tau then
      node.child[o] = allocateEmptyLeaf(o)
    else
      node.child[o] = allocateFullLeaf(o)
    endif
  endfor
end addChildren
```

Figure 3: Pseudo-code for Octree Construction.

used for the rest of the search. Experiments indicate that for two such heuristics h_1 and h_2, there is a good correlation between the fact $h_1(r) > h_2(r)$ and $h_1(n) > h_2(n)$ for the root node r and any node n. That is, the heuristic which is best at the root node tends to be better at other nodes in the search space.

The octree conserves memory by storing a leaf as 'full' only once a table cell from that leaf's volume is demanded by the A* search. In the worst case that the search demands table values from every region of the space, the octree would be forced to store the entire table, and if carefully implemented to take advantage of the A* search's access pattern, would recompute the table once.

The octree may be made to use $O(t^2)$-bounded storage by fixing the number of 'full' leaves available and recycling them in LRU-order. For our experiments we allowed unbounded storage for the octree and never freed 'full' leaves once constructed.

Note that the octree structure can be generalized to any number of dimensions, with the two-dimensional case known as a quadtree.

The idea of doing dynamic computations to improve heuristic evaluation quality is not new. Examples include hierarchical A* (Holte 1996) and pattern search (Junghanns & Schaeffer 2001).

Experiments

Experiments were done using the Needleman-Wunsch global optimal alignment algorithm (Needleman & Wunsch 1970) using the well-known Dayhoff PAM250 (Dayhoff, Schwartz, & Orcutt 1979) scoring matrix with a linear gap cost of 8. For simplicity (and comparison with the other computing-science related work), leading/trailing blanks and affine gaps are not included in our implementation. The program uses A* without any enhancements. The program is coded in C++ and was compiled with GNU CC version 2.95.2. The experiments were run on a Sun E/420 with 4 GB of RAM and 4 CPUs running at 450 MHz. Runs were stopped if they used more than 2 GB of RAM. The octree used a parameter setting of $\tau = 10,000$.

The experiments tested the following heuristic evaluation functions: Pairwise (uses the conventional sum of all pairs of sequences heuristic score), 1 Full (uses a single 3-way alignment, storing the entire matrix in memory), 2 Full (uses two 3-way alignments, storing both matrices in memory), 1 Octree (uses a single 3-way alignment stored as an octree), and 2 Octree (uses two 3-way alignments, each stored as an octree).

A data set of 74 real biological sequences was used (Gallin 2001) with an average sequence length of 300. Random combinations of five sequences were selected to see which MSAs could be completed with the given space constraint. There were 161 5-way alignments attempted, of which 41 were solved by both pairwise and octree versions, 27 were solved only by the octree version, and 93 were not solved by any version. Note that there was no instance where a pairwise heuristic solved an alignment but the octree did not.

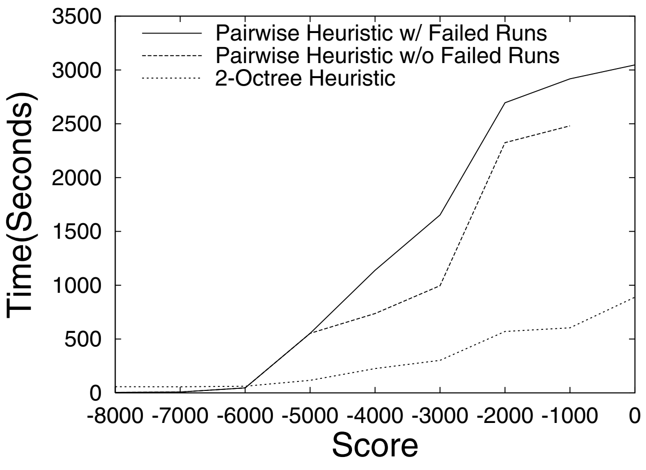

Figure 4: 5-way MSA Time.

Figure 4 shows the time used by the various A* searches as a function of the alignment score; Figure 5 shows the memory usage. The more negative the score is, the better the alignment. The data has been grouped into buckets of 1,000. For example, all multiple alignments with a score between -8000 and -7000 were averaged into a single data point at -8000.

Figure 4 shows a positive correlation between the optimal alignment score and the ease of computation. For sequences with good optimal alignments (low scores), the pairwise heuristic does a reasonable job of estimating the score and the searches are small. For these easy alignments, the pairwise time may be marginally faster than the octree because the cost of doing the 3-way alignment is not offset by the

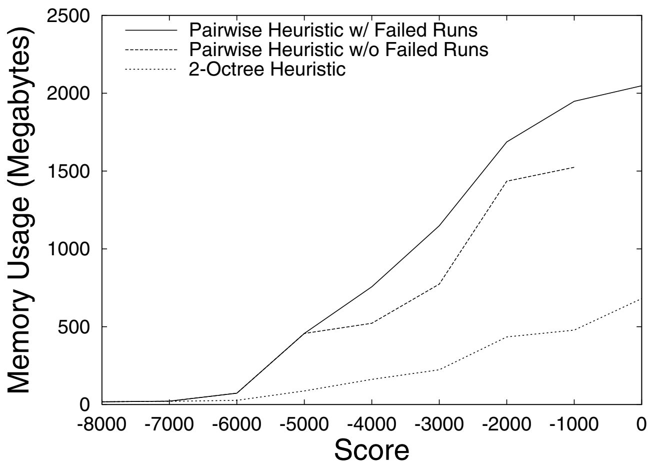

Figure 5: 5-way MSA Memory.

savings in the A* search. As the quality of the score decreases (ie., the score increases), it becomes harder to find the optimal solution. Beyond scores of -6000, the octree dominates the pairwise results, with the performance gap widening as the difficulty of the alignment increases. For the -3000 bucket, the run time is roughly three times faster for the data points where the pairwise heuristic completed. The figure also shows an estimate of the pairwise time that includes the cases where the octree finished but the pairwise did not. Here the pairwise was conservatively assumed to take exactly 2 GB and 3500 seconds (roughly the time it took to exhaust memory). Clearly this is a lower bound, and the performance gap is larger than what is portrayed in the figure.

Even though the octree has the additional preprocessing overhead of the 3-way alignments and the cost of recomputing values, it still runs faster. For large problems, the cost of the more expensive heuristic is more than offset by the much smaller A* search.

The total memory used by the octree program is significantly smaller than the memory used by the pairwise program (see Figure 5). Although the two 3-way octrees require more memory than the pairwise heuristic tables, this is offset by the substantially smaller A* open list that it produces.

As Figures 4 and 5 show, the time and memory for a 5-way sequence alignment grows with the difficulty of the alignment. Extrapolating these curves indicates that there is an exponential growth. This is expected, since the A* search grows exponentially in the length of the solution path. The octrees's better-informed heuristics dampen the exponential growth.

A more detailed study can be seen of the (-3000,-2000) bucket (the bucket with the most data points). Figure 6 shows a breakdown of the alignment times for this bucket. Figure 7 shows the number of A* nodes considered, broken down by number of open and closed nodes stored at the end of the search. The first bar expresses only open nodes because failed program runs did not report this information, and the total number of nodes given here is an estimate based on the running time. Figure 8 shows the total memory requirements (heuristics and open list). There are some interesting points to notice in these graphs:

1. The 2 octree implementation is almost 4 times faster than the pairwise, and more than 6-fold faster if one includes the pairwise data points that did not complete. These numbers are similar for the memory usage. This is a clear win/win situation for the octree. The program runs substantially faster and uses substantially less memory.
2. A full matrix is slightly faster than the octree. Computationally, the matrix should be faster since it has no recomputations. However, the more compact octree gets more favorable cache performance, almost completely offsetting the re-computation costs.
3. Predictably, the octree uses less memory than the full matrix. It is interesting to note that the total memory usage for a search employing two full matrices or two octrees is *less* than that of one of either! Clearly, they use more memory to store the heuristics, but the better quality evaluation function reduces the size of the open list, resulting in an overall reduction in space.
4. Compared to the pairwise scoring function, two 3-way alignments reduce the open list size by at least a factor of 4.

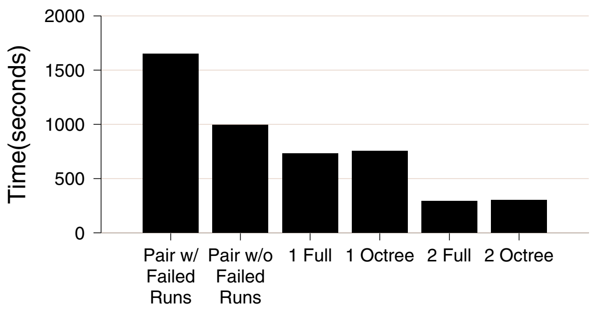

Figure 6: Time (-3000 bucket).

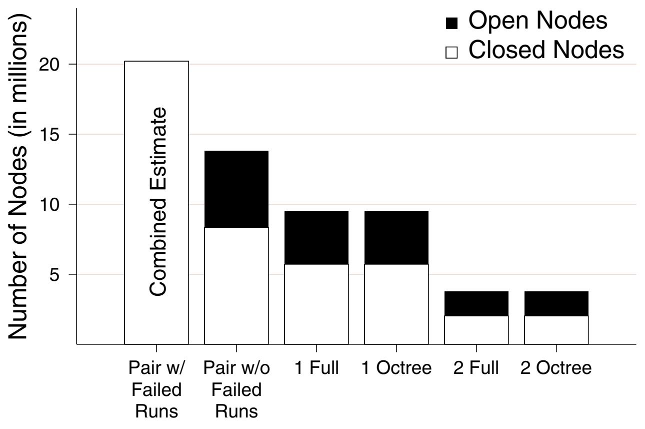

Figure 7: Search Space (-3000 bucket).

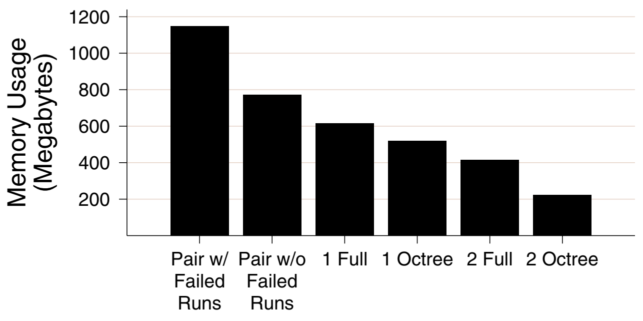

Figure 8: Memory (-3000 bucket).

The octree's space savings come from a two-fold time investment: computing the set of heuristics to use at the start of the search, and dynamically computing octree values throughout the search. For the data sets used, our implementation requires approximately 45 seconds to initially compute the two dynamic programming matrices required by the 2 octree heuristic set. It initially caches three levels of the tree, that is, down to empty leaves of side length one-quarter that of the root node.

Over 10 runs, an average of 20% of the table is recomputed by the octree, at an estimated time cost of 9 seconds. Accurate measurement is confounded by the favorable cache effects resulting from smaller dynamic programming tables that hold spatially close table values closer together in memory, and the unfavorable slowdown from the several recursive calls required to reach the table.

It is interesting to note that our implementation requires 70 seconds to compute and store the full matrix, 30% longer than the $O(t^2)$-space initial generation of the table for the octree. We posit that poor cache behavior accounts for this slower speed.

We have solved a few difficult 6-way alignments. The results are not reported here since the (predictably) large pairwise runs could not complete. For one data set, the pairwise alignment heuristic gave an average score that was 558 from the optimal score. Enhancing the heuristic evaluation to include a single 3-way alignment reduced this to 437, and two 3-way alignments lowered it to 320. For a 6-way alignment (but not a 5-way), it is possible to use three 3-way alignments in the evaluation function. Here the average score is off by only 265. In effect, the use of three 3-way alignments decreases the error in the evaluation function by more than a factor of 2. Korf's analysis of IDA* shows that halving the error in h results in halving the exponent of the search growth (Korf & Reid 1998). IDA*'s search growth is asymptotically the same as A*. Hence, the improvements in h roughly correspond to reducing the search effort to the square root in size.

Search effort is strongly tied to the sequences' degree of similarity (Figure 4) and the final alignment score: similar sequences yield alignments with low scores, few gaps and mismatches, and the search space examined is smaller than for dissimilar sequences that have optimal alignments with higher scores and many gaps and mismatches. Consequently, performance numbers for any MSA algorithm need to be qualified by the quality of the alignment. For example, this paper reports 5- and 6-way alignments. It would be easy to dismiss these results, citing previous work that reports 7- and 8-way alignments (Yoshizumi, Miura, & Ishida 2000). However, the sequences aligned by (Yoshizumi, Miura, & Ishida 2000) are in the "easy" category where the octrees are of relatively little value since the search using the pairwise heuristic can be completed before a 3-way heuristic table can be computed (we have done the experiments).

Future Work

This paper does not use the cited work of partial-expansion A* (Yoshizumi, Miura, & Ishida 2000) because it would have complicated measuring the benefits of the octree heuristic. Partial-expansion A* works by delaying putting child nodes with poor scores onto the priority queue, with the hope that the goal node will be found before they need to be re-examined. Its efficacy is dependent upon the ability of the heuristic to discriminate between nodes on the optimal path and irrelevant nodes. We implemented PEA*, and found that the space savings and time costs realized varied with the difficulty of the search and with the quality of the heuristic. On a difficult problem instance with optimal score -669, the 2-octree heuristic saved 62% of the search space when using a cutoff of 0 instead of a cutoff at infinity(normal A*), but on an easier instance with score -7605, it saved 87%. In the latter case the search took nearly 10 times as long to perform when using the cutoff at 0. In addition, we found that the 2-octree heuristic realized proportionally more savings than the pairwise heuristic – where the 2-octree heuristic saved 62% of the search space on the difficult instance and took ten times as long, the pairwise heuristic saved only 39%, and took twenty times as long. This pattern was repeated on the easy instance. This makes sense because using two three-way alignments in the heuristic should allow better discrimination between nodes. Note that both instances had 5 sequences, meaning they each had a branching factor of 31. The lesson seems to be that PEA* can realize substantial benefits with a strong heuristic, but suffers when the problem becomes difficult. However, more work is required to conclusively characterize the situations where PEA* provides a benefit. Regardless, its use would have complicated our analysis, though it seems likely that it would have portrayed the octree in an even more favourable light compared to the pairwise heuristic.

It has been difficult to compare the performance of search techniques for sequence alignment that have come out of the AI community because researchers have made up their own data sets rather than using a standard benchmark of sequences. Future results will be published against BAliBASE (Thompson, Plewniak, & Poch 1999), a benchmark set of alignments that covers a broad range of lengths and degrees of similarity of interest to biologists, with hand-verified reference alignments. Initial results indicate that the linear gap cost function used in this paper and in others from the AI community (Yoshizumi, Miura, & Ishida 2000; Korf & Zhang 2000) does not typically produce high-quality optimal alignments. Our future work will use the quasinatural affine gap model (Altschul 1989) which requires a larger search space but produces better-quality alignments according to initial tests on BAliBASE.

Conclusions

This paper illustrates that single-problem instances can benefit from computing (large) heuristic tables. The cost in time and space for the heuristics can be subsumed by the time and space savings from a smaller open list. Many single-agent search applications use generic heuristics that cover a wide range of problem instances. Generic heuristics seem best, since the cost of computing the heuristic can be amortized over multiple problems. However, as the MSA problem shows, a time and space investment in a problem-instance-specific heuristic can provide a more focused, faster, and space-efficient solution. This happens because there are two ways to use memory in an A* search: the open/closed lists, and heuristic evaluation function tables. This is not a zero-sum game. Storage can be used (and reused) to improve evaluation function quality. For the MSA application, the octree provides a nice framework for increasing the quality of a heuristic evaluation using a reasonable amount of space.

Acknowledgments

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta's Informatics Circle of Research Excellence (iCORE) and the Canadian Protein Engineering Network of Centres of Excellence (PENCE). The authors would like to thank Dr. Robert Holte for his comments.

References

Altschul, S. F. 1989. Gap costs for multiple sequence alignment. *J. of Theoretical Biology* 138:297–309.

Carrillo, H., and Lipman, D. 1988. The multiple sequence alignment problem in biology. *SIAM J. on Applied Mathematics* 48(5):1073–1082.

Chao, K.; Hardison, R.; and Miller, W. 1994. Recent developments in linear-space alignment methods: A survey. *J. of Computational Biology* 1(4):271–291.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dayhoff, M. O.; Schwartz, R. M.; and Orcutt, B. C. 1979. A model of evolutionary change in proteins. In Dayhoff, M. O., ed., *Atlas of Protein Structure*, volume 5(Suppl. 3). Silver Spring, Md.: National Biomedical Reasearch Foundataion. 345–352.

Gallin, W. 2001. A selection of potassium channel protein sequences. http://www.cs.ualberta.ca/ bioinfo/sequences/pchannel/.

Hirschberg, D. 1975. A linear space algorithm for computing maximal common subexpressions. *CACM* 18(6):341–343.

Holte, R. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. *AAAI* 530–535.

Junghanns, A., and Schaeffer, J. 2001. Enhancing single-agent search using domain knowledge. *Artificial Intelligence* 129(1-2):219–251.

Korf, R., and Reid, M. 1998. Complexity analysis of admissible heuristic search. *AAAI* 305–310.

Korf, R., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. *AAAI* 910–916.

Korf, R. 1999. Divide-and-conquer bidirectional search: First results. *IJCAI* 1184–1189.

Korf, R. 2000. Recent progress in the design and analysis of admissible heuristic functions. *AAAI* 1165–1170.

Myers, E., and Miller, W. 1988. Optimal alignments in linear space. *CABIOS* 4(1):11–17.

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

Reinert, K., and Lermen, M. 2000. The practical use of the A* algorithm for exact multiple sequence alignment. *J. of Computational Biology* 7(5):655–671.

Reinert, K.; Stoye, J.; and Will, T. 2000. An iterative method for faster sum-of-pairs multiple sequence alignment. *Bioinformatics* 16(9):808–814.

Spouge, J. 1989. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM J. of Applied Mathematics* 49(5):1552–1566.

Thompson, J.; Higgins, D.; and Gibson, T. 1994. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research* 22:4673–4680.

Thompson, J.; Plewniak, F.; and Poch, O. 1999. BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics* 15(1):87–88.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with partial expansion for large branching factor problems. *AAAI* 923–929.