Implementation of "An efficient algorithm for multiple sequence alignment based on ant colony optimization and divide-and-conquer method"

We have implemented the algorithm described in "An efficient algorithm for multiple sequence alignment based on ant colony optimization and divide-and-conquer method" by Wei Liu, Ling Chen, and Juan Chen (2007). This algorithm consists of three parts: Divide and Conquer, Ant Colony Optimization, and Assembly.

In the Divide and Conquer stage, a genetic algorithm is used to find the best way to divide the DNA sequences into more manageable chunks. First, K possible solutions are randomly generated to form a population. A simplified version of sum of pairs is used to calculate the score of each possible solution. In each of the following generations, a new solution based on the current best solution is generated. In addition, T times of mutation and crossover operations are performed to generate new possible solutions. For each new possible solution, its score is evaluated and compared with the current scores. In the population, the worst solution is always replaced by a better new solution, left with the top K solutions in each generation. When the number of generations reaches a certain number, the Divide and Conquer returns the best cutoff positions for each sequence. By applying this procedure recursively, the original set of sequences are divided into 2, 4, 8, ... subsets until the length of sequences in subsets reaches a certain threshold. The overall complexity of Divide and Conquer is $O(N^2 * Len * T * C * log\ (len))$, in which N is the number of sequences, T is the number of mutation and crossover operations, C is the number of new generations of the population, Len is the length of the longest sequence, $log\ (len)$ is the approximate number of subsets.

The next stage then aligns each of these subsequences using an Ant Colony Optimization heuristic. In this step, we loop through every nucleotide from every sequence, with an "ant" forming a path from each of these nucleotides to its best match in every other sequence. As each path is formed, the ant "pheromone" is updated, eventually leading more ants to choose the more popular (and hopefully best) paths. After every nucleotide in a sequence has a path, an alignment of all the sequences is formed from the paths, and its sum of pairs score is calculated. This is repeated for every sequence, keeping track of the best alignment. Once all sequences have formed an alignment, the process is repeated a predetermined number of times, with the best scoring alignment returned. Pheromone levels are adjusted between cycles to prevent local optimization. The computation complexity of the Ant Colony Optimization is $O(N^2 * Len * Cycle\ * log(Len))$, in which N is the number of sequences, Cycle is the number of iterations "ants" travel, and Len is the length of the longest sequence.

Finally, after each set of subsequences is aligned, they are reassembled to form one overall alignment. This is done by simply concatenating the aligned subsequences together. The time complexity is $O(N * log(len))$, where again N is the number of sequences and len is the length of the longest sequence.

To test our results, DNA sequences sets with varying numbers of sequences and varying lengths of sequences are created using Rose. We benchmarked each of these sets of sequences, determining how these changes affected runtime and memory. The Sum of Pairs scores are compared to the solutions given by Clustal Omega and Rose.

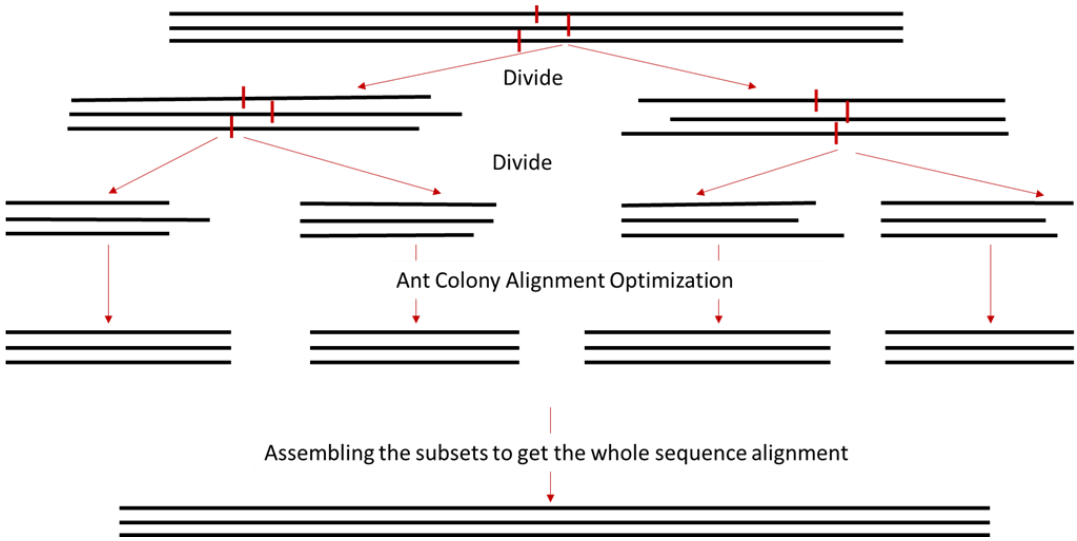
In terms of performance, both the runtime and memory usage increases quadratically as the number of DNA sequences increased. Both runtime and memory appeares to have an nlogn relationship

as the length of the DNA sequences increases. The accuracy of the Ant Colony and Divide and Conquer algorithm is compatible with Rose for short sequences and for a small number of sequences. However, when the length of sequence is large or the number of sequences increases, the accuracy decreases. e manually performed showed that our algorithm did not perform as well as Clustal Omega.

Materials and Methods

The Algorithm

The algorithm, based on Liu et al 2007, consists of three stages: Divide and Conquer, Ant Colony Optimization, and Assembly. Each of these stages is described below.



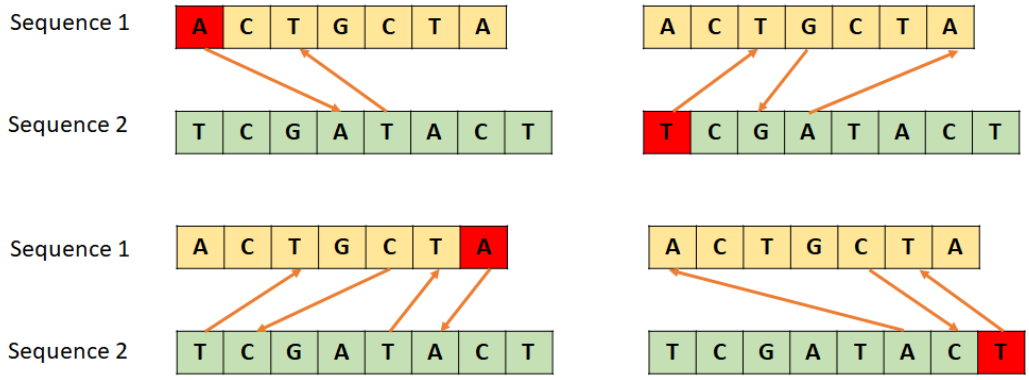
Divide and Conquer

This first stage divides the DNA sequences into more manageable subsequences. The cutoff points are optimized using a genetic algorithm.

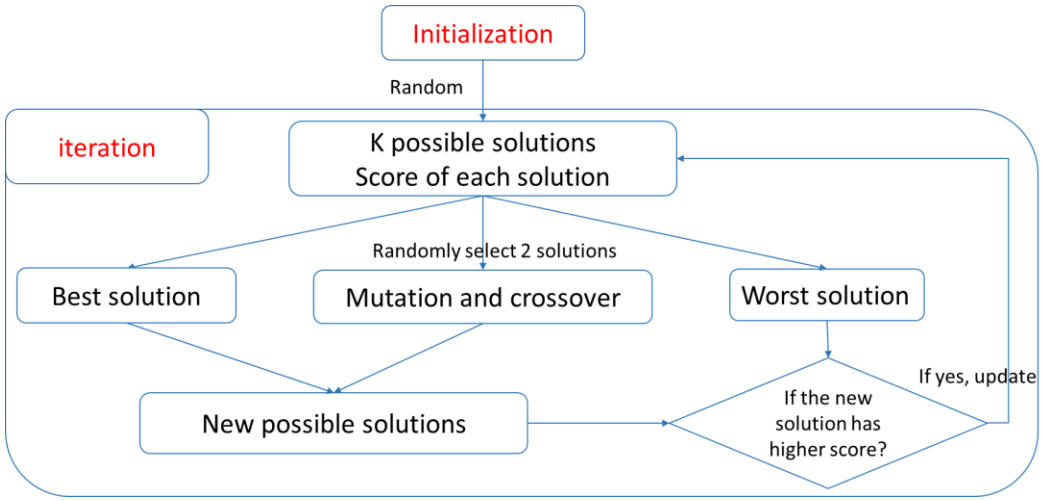
We recursively find cutoff points for each sequence to produce a set of subsequences. Starting from one set of sequences, we can produce 2, 4, 8, 16, … subsequences, until reaching the predefined subsequence length. This predefined length is a function of the length of the sequences, with longer sequences resulting in longer subsequence lengths (more on this in the Discussion section). We define d as an array of cutoff points, where each element of d represents the cutoff points for one sequence. Cutoff points will be evaluated with a score function. Because this evaluation is repeated a large number of times, a simplified version of sum of pairs is used to reduce computation time.

The simplified version of sum of pairs is implemented in four steps. First, starting from sequence 1[1], find the first matching character in sequence 2 from left. Denote this character as sequence 2[j], then find the first matching character after sequence 1[1].

Continue till the end of either sequence. The second to fourth step replicates the same procedure but starting from different positions in both sequences.



A genetic algorithm is used to optimize the cutoff points to get the highest alignment score. This genetic algorithm consists of 3 stages: initiation, mutation and crossover, and termination. In the initiation stage, we randomly generate k possible solutions in generation 0. We use the aforementioned score function to evaluate the performance of each solution. The current best score is used as a benchmark for further analysis. The next stage, mutation and crossover, creates the subsequent generations. For each generation p, we perform mutation and crossover to produce new possible solutions for generation p+1. The idea is that the new possible solution borrows information from the current best solution, with randomly generated mutations, and the crossover generates new solutions by swapping parts of two current solution vectors. We use the same score function to calculate the performance of the new solution. If the new solution is better than the worst solution in generation p, this worst solution is replaced with the new solution. Repeat this process -- finding the best and worst score, generating a new possible solution, comparing its score with the previous worst score, and replacing the worst score with the new solution -- a predetermined number of times. The k solutions with highest scores are stored in a k by s matrix, where k is the number of



possible solutions and s is the number of sequences. Finally, after a predetermined number of cycles, the termination stage outputs the best solution, which is the approximation of the global optimization of the cutoff points.

The pseudocode for the Divide and Conquer stage of our algorithm is shown below.

Input: N sequences: S1 ,S2,…… ,SN
Output: The optimal cut-off: d[best]
Begin
1 Randomly generate K cut-off, called POP
2 For population = 1 to CYCLE do
2.1 Calculating the best cut-off in POP, let its index be best
2.2 Calculating a new cut-off according to new[i]=best[i]-best[f]+f_half_length. Here, f is the index of the longest sequence.
if new is better than the worst cut-off in POP,
replace it by new
2.3 generating new cut-offs by T mutations and crossover
if mutation or crossover cutoff solution is better than the worst cut-off in POP,
replace worst cut-off by the better solution
Remaining K cut-offs of high quality
endfor-population
3. Return (d[best]), where best is the index of the best cut-off in POP
end

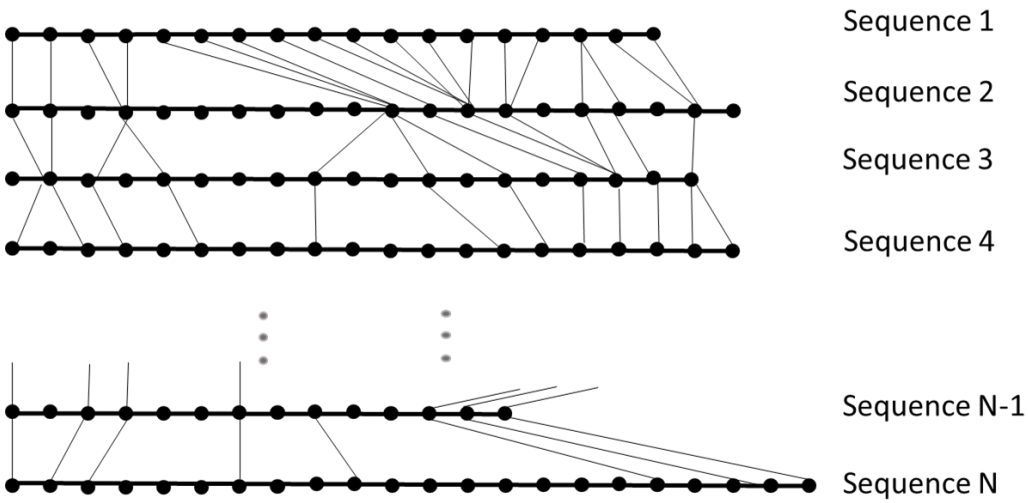
For the computation complexity of the Divide and Conquer: calculation of sum of pair(simplified version) is of O($N^2 * len$). The sum of pair function is applied for each subset log(len), each generation C, T times of mutation and crossover. The overall complexity is $O(N^2 * Len * T * C * log\ (len))$.

Ant Colony Optimization

The second stage, Ant Colony Optimization, (implemented in the AntColony function), takes the subsequences produced by the Divide and Conquer stage and aligns each set of them. This is done in two main steps that are repeated many times: forming paths and forming alignments. A path is formed when an “ant” moves from a nucleotide from one sequence to its best match in each of the other sequences. Paths from each nucleotide in a sequence form one possible alignment. This is repeated for every sequence, with the best scoring alignment remaining. Then this entire sequence is repeated a predetermined number of times to find the best approximation.

To form these paths, we cycle through each sequence of the set of sequences, S, using the index k. Within each sequence S_k, we cycle through each nucleotide with index

l, and an “ant” forms a path for that nucleotide. This is done by cycling through S once again, this time with index n. For each sequence S_n, if n != k (i.e., we’re not forming a path between a sequence and itself), we determine which nucleotide in S_n has the highest probability of matching with S_k[l], using the function P, which implements the equation in Figure **?**. This equation considers the pheromone along each possible path (the pheromone variable) with how well it matches (the mat function) and how far the location deviates from the starting point (the dev function), to produce the probability that a given nucleotide S_n[m] is the best match. If the nucleotide with highest probability matches the nucleotide S_k[l], we select it. Otherwise, we choose either a space or the next nucleotide in sequence S_n, selecting a space with a predetermined probability and the nucleotide with the remaining probability. This is done using the sample function of the Julia StatsBase library, which will select a value from an array of options using the given probabilities. Once we select a matching nucleotide, a mismatched nucleotide, or a space, we put the index, m, of the nucleotide (or -1 for a space) into the paths[l] array, at index n; this indicates that the l-th nucleotide of sequence S_k maps to the m-th nucleotide of the n-th sequence.



Once we have formed a path for every nucleotide in sequence S_k, we form an alignment. This was not described in the paper, but our implementation is done in the Form_Alignment function. We initialize a variable, alignment, as an array of char arrays; each of these char arrays is one of the aligned sequences. Next, we initialize the Int64 array previous_nucleotide_index, whose i-th element is the index of the previous nucleotide incorporated from sequence i. We once again loop through sequence S_k using index l. We define path as paths[l], the path formed by the ant starting at nucleotide S_k[l]. For this l-th nucleotide, we first determine how many characters will be added in total to keep the alignments the same length. This is done by finding the maximum of the difference between each sequence’s index in path and its previous_nucleotide_index. This is how many characters we will need to add to each alignment, whether they are

gaps, the chosen nucleotide, or skipped nucleotides. Now, we iterate over path, with each i-th element of path telling us the match_index of the nucleotide from sequence S_l that was selected, or a space if the match_index is -1. If it's -1, we simply add a gap to alignment[i]. Otherwise, we first need to determine which nucleotides were between the previously selected nucleotide and the currently selected nucleotide, and we append those to alignment[i] so that they are not skipped. Next, we look at the new current_length, and if this is less than the previously determined new_length, we pad with gaps until there is one character remaining to add. Finally, we add the selected nucleotide at match_index to alignment[i]. After every iteration, each char array in alignment should be the same length.

Once we have finished looping through the paths, on the last iteration, we need to fill in any remaining nucleotides that were after the last-selected nucleotide in each sequence so that the end of the sequences are not skipped. These remaining nucleotides are appended to their corresponding alignments. Finally, since all alignments must be the same length, we determine the maximum length of the alignments and fill in the shorter sequences with gaps.

Now that we have an alignment, we score it using the SP_Score function, which cycles through every combination of pairs and scores them depending on if they are a match, a mismatch, a nucleotide paired with a gap, or a pair of gaps. We deviated from the paper here: they used scores of +2, -1, -2, and 0 respectively, while we used scores of +1, -1, -1 and -1. If the score of this alignment is better than the best alignment so far, we keep the alignment and update the best score.

After forming this alignment based on S_k, we adjust the pheromone along the paths using Adjust_Pheromone_1 to avoid local optimizations. We then repeat the process for the next sequence, S_{k+1}, until we have formed an alignment based on every sequence.

At this point, we're done with a cycle, so we adjust a few more parameters. Pheromone is further adjusted using Adjust_Pheromone_2, only if the best score from this cycle was worse than the worst score from the previous d cycles. We also adjust a, b, and c, the importance of pheromone, matching score, and location deviation, respectively, when calculating probability. We then repeat this entire process for a predetermined number of cycles. Finally, the best alignment from all of the cycles is returned.

The pseudocode for the Ant Colony Optimization stage of our algorithm is shown below.

```
input: N sequences: S[1],S[2],……,S[N]
Output: The optimal alignment of S[1],S[2],……,S[N]
        and score of the alignment: alignsum
Begin
Initialization
for cycle = 1 to Cyclenum do # Cyclenum is the number of iterations
```

```
    for k = 1 to N do # Circulation of S[k], starting sequence
        for l= 1 to length(S[k]) do # Circulation of S[k][l]
            for n = 1 to N do # Circulation of S[n], end sequence
                if n!=k then # Select a space or a character in S[n]
                    for m = loc(k,l,n) to loc(k,l,n)+h-1 do
                        calculate P(k,l,n,m), keeping track of highest probability
                    end for m
                    m = value of m that maximized P(k,l,n,m)
                    if (S[n][m]) = S[k][l]) then
                        select Sn(m)
                    else
                        select a space based on a probability
                        select S[n][loc(k,l,n)) based on the remaining probability
                    end if
                end if
            end for n
        end for l
        form a whole alignment (see pseudocode below)
        calculate the score of the alignment, keeping best alignment
        Update the pheromone
    end for k
    if the alignment is the best, remain it
    if cycle>start and best score this cycle < worst score over last d cycles
        adjust the pheromone
    end if
        Adjust the parameters a, b, c
    end for Cyclenum
    return best alignment and score
end
```

The pseudocode for the FormAlignment part of the Ant Colony Optimization algorithm is below.

```
Input: S -- the original sequences, paths -- the vector of path vectors, and k -- the starting sequence index
Output: alignment -- a vector of strings representing the aligned sequences
Begin
for l in starting sequence length
        calculate new length of alignment based on maximum number of nucleotides that will be added
        for i, match_index in path
                if match_index represents a space
                        append a space to alignment[i]
                else
```

```
                fill in missing nucleotides between this path's selection and last
            path's
                pad with spaces if needed to get to new length - 1
                add the nucleotide given by the match_index in path
            end if
            if last iteration
                append any nucleotides that were after the last match
            end if
        end for i, match_index
end for l
append gaps to alignments to make them all the same size
return alignment
End
```

For the computation complexity, when searching region h is small compared to length of sequence, the ant selects characters on other sequences takes O(Len*N). Ants are equally assigned to every sequence. The Ant Colony function is applied multiple times (Cyclenum) to choose the optimal. The overall complexity is $O(N^2 * Len * Cyclenum)$. Since this is done for roughly log(Len) subsequences, the complexity of the entire Ant Colony Optimization is $O(N^2 * Len * Cyclenum * log(Len))$.

Assembly

The final stage is Assembly. In this step, the aligned subsequences returned from the Ant Colony Optimization step are reassembled to form one long alignment. The subsequences are simply concatenated back together in their original order to form this final alignment.

The pseudocode for this step is below:

```
Input: subsequence_alignments -- the alignments of each subsequence, stored as
a vector of vectors of aligned subsequence strings
Output: final_alignment -- an array of aligned subsequence strings representing
the final alignment
Begin
initialize final_alignment to all empty strings
for each subsequence_alignment in subsequence_alignments
        for each sequence with index i
                final_alignment[i] = final_alignment[i] +
                subsequence_alignment[i]
        end
end
return final_alignment
End
```

For time complexity, there are roughly log(len) subsequence_alignments, where len is the maximum length of a sequence. For each subsequence_alignment, we append to each sequence, of which there are N. Therefore the time complexity is $O(N * log(len))$.

Evaluating Parameters

There were many parameters involved in our algorithm -- the Ant Colony Optimization step alone had over 20 parameters, none of which were given in the paper. Unfortunately, due to time constraints, we had to make educated guesses for many of them, testing just a few different values to get an idea of a reasonable value. A few that seemed significant were the probability of a space (prob_of_space), the number of cycles to consider before globally adjusting pheromone (d), and the number of nucleotides to compare against when finding a match (h), so we tested these in more depth. Using a test set of 4 sequences around length 90, prob_of_space was first varied from 0.00:0.02:0.5, then we focused on the best region, varying prob_of_space by 0.01 and running the algorithm multiple times for each test, taking the optimal score. d was varied from 0:2:10 on the same small set of sequences, and the optimal value was found. h was varied from 5:5:30 on the same set of sequences, and the optimal value was found as well.

Evaluating Performance

Our performance evaluations were concerned with three things: runtime, memory usage, and accuracy. For all experiments we used Rose software to simulate custom DNA sequences with varying number of sequences and lengths of sequences. For the purpose of testing our accuracy, we used the Clustal Omega and Rose reference alignment scores for the same sets of DNA sequences. We used two matching schemes for scoring: Sum-Of-Pairs (+1 for match, 0 otherwise) and Sum-Of-Pairs-With-Penalty (+1 for match, -1 for mismatch, and -2 for a gap).

First we tested accuracy for 8 sets of generated DNA sequences of various lengths and number of sequences. Using the same matching scores, we scored our solutions and the solutions of Rose and Clustal Omega. These scores were tabulated for comparison.

Then we determined how runtime, memory usage, and accuracy varied with respect to number of sequences and sequence length. For the comparisons against length of DNA sequences, we used sequences of length 4 - 200, with the number of DNA sequences set to 4. For comparisons against the number of DNA sequences, we varied the number of sequences from 4-25 with the length of DNA sequences set to 40. Runtime and memory usage were determined using the @timed function of the BenchmarkTools library and plotted against number of sequences and sequence length. To plot accuracy, we plotted both scores (Sum of Pairs and Sum of Pairs with Penalty) of our solutions against the Rose solution scores for the same sequences.

Results and Analysis

The results of varying prob_of_space over a wide range (0 to 50%) are shown below.

[Plot]

We saw that the score was optimal at lower values of prob_of_space, so we focused on the region between 0 and 0.2, which is shown in the plot below. The optimal value in both plots was 0.06.

[Plot]

The results of varying d over a range of 0:2:10 are shown below. The optimal value of d was found to be 4, but there did not seem to be a trend in this plot.

[Plot]

The results of varying h over a range of 1:4:30 are shown below. There is no definite trend, but it seemed to do better for smaller values of h. The optimal value of h after a few of these tests was found to be around 10.

[Plot]

Some examples of aligned sequences are shown below.

[Aligned sequences and visualization]

The results of testing 8 sequences against Rose and Clustal Omega are shown in the table below. In most cases, our results are comparable to the Rose reference alignment, and in some cases (Test2, Test5, Test6, and Test7) we perform better according to the Sum of Pairs with Penalty. Clustal Omega, however, always outperforms us. This is due to the nature of divide and conquer where we look at aligning subsequences of the actual sequences given, so gaps are introduced between subsequences, whereas Clustal Omega performs a global alignment without breaking down the individual sequences (more on this in the Discussion section).

			Sum of Pairs Score			Sum of Pairs with Penalty Score (+1,-1,-2)		
Test Case	Number of Sequences	Length of Sequence	DC-Ant	Rose	Clustal Omega	DC-Ant	Rose	Clustal Omega
Test1	4	90	249	284	249	-437	-278	-256
Test2	5	220	784	1144	916	-2280	-2297	-1741
Test3	6	50	548	588	483	3	237	175
Test4	8	250	1921	2216	2270	-9957	-6370	-5814
Test5	9	150	1384	1473	1724	-7736	-22970	-6127
Test6	9	400	3352	5391	4442	-15730	-31242	-11822
Test7	10	300	3852	6441	5167	-20270	-23619	-9649
Test8	10	100	2495	3225	2709	-6011	-4818	-1873

The results of our benchmarks of time, memory, and accuracy against number of sequences and sequence length are shown below.

[Plots]

Time and memory both increase quadratically with an increasing number of DNA sequences.Time ranges from roughly 13 seconds for 4 sequences of length 40 to 475 seconds (nearly 8 minutes) for 25 sequences. Memory ranges from 8616 MB for 4 sequences to 325,559 MB for 25 sequences. Accuracy, as indicated by the similarity between our scores and Rose's, decreases with

the number of DNA sequences; our scores are very close for a small number of sequences but deviate for a larger number.

Time and memory also appear to increase quadratically as the length of DNA sequences increases. However, we see breaks in these graphs unlike when we vary the number of DNA sequences, because as the length increases the number of subsequences we break the DNA sequences into during the Divide and Conquer stage changes (more on this in the Discussion section). Time ranges from 0.08 seconds for sequences of length 4 to 88 seconds for sequences of length 200, while memory ranges from 40 MB to 78,463 MB for the same range of sequence length. As the length of the DNA sequences increases, the Sum of Pair score of our algorithm stays pretty consistent with the Rose score. The Sum of Pairs with Penalty score stays pretty close to the Rose score as well, but is sometimes significantly lower. There are also a few points where our algorithm performed better.

Discussion

Overall, our algorithm performed decently, but was not as fast or accurate as we expected. It seems to handle a few sequences pretty well, finishing in under a second for short sequences and about a minute and a half for a sequence of length 200. This plot appears to be an nlogn relationship, which is consistent with what we determined our time complexity to be: both Divide and Conquer and Ant Colony Optimization are O(Len*log(Len)) where Len is the length of the longest sequence. However, as the number of sequences increases, even for relatively short sequences of length 40, the algorithm struggles to keep up, taking over 8 minutes for 25 sequences. The quadratic time complexity we see in the plots is consistent with what we determined our time complexity to be: both Divide and Conquer and Ant Colony Optimization are $O(N^2)$ for sequence number. In terms of accuracy, it seems pretty comparable to Rose, sometimes performing better and sometimes worse, but it always performs significantly worse than Clustal Omega. A similar algorithm by Lee et al (2008), published a year after Liu et al (2007), was very closely matched with Clustal Omega, so we did not perform as well as other similar algorithms from that time.

There are many factors at play that affect time/memory and accuracy. In theory, more iterations, of both the Divide and Conquer step and the Ant Colony Optimization step, would produce more accurate results, as both should be working towards optimization with each iteration. However, this will obviously require more time and memory. Another tradeoff comes from the size of the subsequences returned from the Divide and Conquer step. We found that shorter subsequences produce results faster, as the time of the Ant Colony Optimization step increases exponentially with the length of the subsequences it has to align (length^num_of_subsequences) . However, we found that at least for longer sequences, this greatly reduced the Sum of Pairs with Penalty score. The reason for this is that each subsequence alignment has to be the same length, so gaps are added at the end of shorter subsequences when they do not have any more nucleotides left to match. When there are many, shorter subsequences, this results in several gaps being added, which hurts the Sum of Pairs with Penalty score. Therefore, we made this minimum sequence length be proportional to the length of the sequences, which through experimentation seemed to be a good balance.

We also found a tradeoff between our two methods of scoring. As the Sum of Pairs with Penalty score improved, the Sum of Pairs score decreased. We determined that this was due to the gap penalty. With enough gaps, the Sum of Pairs score can be great, because every nucleotide can be “aligned.”

However, we felt that this was misleading, as there should not actually be that many gaps, so we focused on the Sum of Pairs with Penalty score when optimizing our algorithm.

There are many improvements that can be made to our algorithm. A major one is tweaking all of the parameters. There were over 20 parameters in the Ant Colony Optimization step alone, none of which were given a value in the paper, so we made some educated guesses and tested changing many of them, like the pheromone evaporation rates and the velocity of changing the importance of pheromone, matching score, and location deviation. Ideally we would have had the capacity to methodically test how all of these parameters affected the results as we did with the probability of a space, d, and h. Another change would be to the Form_Alignment function. This was another missing piece in the paper; it was briefly described in a couple sentences but no detail or pseudocode was given, so it was up to our interpretation. To deal with the end of alignments, any unused nucleotides in a sequence are simply tacked on the end of the aligned sequence, and differences in length are dealt with by filling in with gaps. This was the best method we could think of but probably not ideal, as this leads to mismatched nucleotides and extra gaps at the end of sub-sequences. One way to avoid the gaps would be to not require that each subsequence is the same length and instead deal with the difference in length at the end of the full alignment, removing many gaps from the middle of the alignment, but this may cause later subsequences to become misaligned. Another method would be to address this in the Divide and Conquer stage, requiring that the subsequences in each set of subsequences are of equal length, but this may result in suboptimal cutoffs.

There were also more experiments we wanted to do but we were limited by time. Our experiments for time, memory, and accuracy vs number of sequences was only done for sequences of length 40, and the comparison of these against length of sequences was only done for 4 sequences. The experiment comparing against the number of sequences should be done for a variety of lengths, and the comparison against length should be done for a variety of the number of sequences. The experiments on the parameters prob_of_space, d, and h, should be performed on more than one test set so we do not only optimize to one set. We also wanted to run experiments on the effects of more parameters. For example, we'd like to see the effects of varying pheromone levels and thresholds, the importance of factors like pheromone, matching score, and location deviation, and the velocities of changing these parameters.

References

Chowdhury B, Garai G. (2017) A review on multiple sequence alignment from the perspective of genetic algorithm, Genomics, 109(5-6), 419-431, DOI: 10.1016/j.ygeno.2017.06.007.

Zne-Jung Lee, Shun-Feng Su, Chen-Chia Chuang, Kuan-Hung Liu (2008) Genetic algorithm with ant colony optimization (GA-ACO) for multiple sequence alignment, Applied Soft Computing, 8:1, 55-78, DOI: 10.1016/j.asoc.2006.10.012.

Wei Liu , Ling Chen & Juan Chen (2007) An efficient algorithm for multiple sequence alignment based on ant colony optimisation and divide-and-conquer method, New Zealand Journal of Agricultural Research, 50:5, 617-626, DOI: 10.1080/00288230709510330