International Journal of Multidisciplinary and Current Educational Research (IJMCER)
ISSN: 2581-7027 ||Volume|| 6 ||Issue|| 4 ||Pages 56-68 ||2024||

Improvement on the Longest Common Subsequence Algorithm Using ANT Colony Optimization

[1,] Maksuda Sultana, DIT, [2,] Christian Nicholas Leal
College of Computer Studies, AMA University

ABSTRACT : Ant Colony Optimization, also known as ACO, is a metaheuristic that uses search agents to traverse the search space in a probabilistic manner. The search agents' behavior is modeled after the motions of ants in real life through the use of a pheromone model that dictates a given search agent's cycle through the search space. This study aims to apply the metaheuristic on the Longest Common Subsequence problem, such that its performance, whether in terms of time or space complexity, may be improved upon.

KEYWORDS: ANT Colony, ANT Colony Optimization, LCSA, Longest Common Subsequence Algorithm

I. INTRODUCTION

The This project explores the applications of Ant Colony Optimization into improving the Time Complexity and Space Complexity of the Longest Common Subsequence (LCS) Problem and to discern the difference in runtime and runs pace of an algorithm where such improvements are applied, to the original O(n^2) dynamic programming-based algorithm. Computer Science as a whole largely concerns itself with the study of both computers and computing, in a theoretical and practical sense. The field itself sees its roots in Mathematics, of which the earliest known computing device, the Abacus, dates back thousands of years before the modern discipline that it is currently. Its pioneers, Charles Babbage and Ada Lovelace, the creator of the first modern computer, and the designer of the first computer algorithm. Another well-known pioneer in the field is Alan Turing, whose concept of the Turing Machine had laid the foundations for modern Computability Theory.

In theoretical Computer Science, one is tasked with studying problems and developing solutions using a given computational model and algorithm, or with analyzing already existing solutions to the problem for their efficiency in terms of time spent and space used by the algorithm in terms of its Computational Complexity. This field is also an application of Mathematical Theories such as Number, Set, and Graph Theory in relation to deducing solutions or approximations to problems. Meanwhile, practical or applied Computer Science makes use of existing breakthroughs in the field to solve real-world problems. This field expresses itself through Software Engineering, Networking and Security, Distributed Systems, and Data Science. Unlike in theoretical Computer Science, one instead devises and realises schematics or maintains already existing systems through the application of skills learned within the field itself. The Longest Common Subsequence Problem is a problem wherein one is tasked to find the longest often non-consecutive sequence of symbols common to a set of strings. That is, the Longest Common Subsequence Problem is a problem where given a set of strings X1, X2..., Xn , one must find the longest subsequence common to all of the given sequences. A closely related problem is the Longest Common String problem, where said sequence must be consecutively ordered. Unlike the Longest Common String problem however, a subsequence of a given sequence is a collection of similarly ordered symbols that may or may not occupy consecutive positions within the sequence itself.

The classic naive solution to the problem involves a recursive algorithm that generates all possible subsequences, then discerns the longest matching common subsequence in all those pairs of sequences. This solution is both exponential, or O(2n) in time and space complexity, as there would be 2n possible combinations. With a method of memorization, duplicate comparisons may be ignored altogether, reducing the time and space complexity needed to compute for the Longest Common Subsequence of an n-length and an m-length string into O(n*m).The problem still stands to be one of the most well-known problems in Computer Science, with exact solutions taking up quadratic time and space in the worst case. However, according to a study by Karl Bringmann, and Marvin Kunnemann, a sub-quadratic exact algorithm for the Longest Subsequence Problem and Levenshtein Distance does not exist, and should one such algorithm exist, would defy the Strong Exponential Time Hypothesis. This lower bound also expands upon other string-related problems, such as the Longest Palindromic Subsequence problem, along with the Longest Tandem Subsequence problem.

As such, the researcher explores algorithms that return approximations of the exact solution in sub-quadratic time, comparing and contrasting its runtime with both the naive algorithm and the dynamic programming approach using tabulation.The aim of the study is to explore approximation methods currently existing for the general case of two n-length strings, specifically the Ant Colony Optimization algorithm. As of this writing, there have been few subquadratic-time approximations for the problem itself, detailed within the Related Literature Section.

The researchers specifically want to address the following goals:

1. To explore the Ant Colony Optimization method as a viable approximation method and compare its accuracy and runtime with the classic Dynamic Programming approach.
2. To devise an algorithm that is based off of the programming logic of Ant Colony Optimization that retains as close an accuracy to the Dynamic Programming-based algorithm, while showing improved runtime and runspace.
3. To devise a dataset for the testing of such an algorithm using a random DNA sequence generator.

II. METHODOLOGY

This problem is a classic in the field of Computer Science and the field of Bioinformatics. Bioinformatics is an interdisciplinary field of Molecular Biology and Computer Science that is concerned with the applications of tools and methods within those fields in order to analyze and convert biological data into useful information, often in a large-scale setting. Meanwhile, Computer Science relates to the problem by virtue of it being a subsection of pattern-matching and string-searching problems.Applications of the study involves software that computes for data comparison, a prominent example being the utility diff in Unix operating systems. This tool allows the user to compare two sets of text, calculating the difference between the given sets of text and presents or outputs the differences between those files. As is, its core algorithm is based off of the Longest Subsequence Problem, as the inverse of the solution to the LCS problem of two sets of strings is the difference between those two sets of strings.

Another application in Computational Linguistics involves a related problem wherein one computes for the edit distance between two given strings. This allows one to evaluate the similarity, or conversely, dissimilarity between strings by calculating how many operations are required in order to transform a given string into another string. Its relevance in the field stems from the discernment of purpose of word forms given a sample set of word forms without further human input. Such is possible through the combination of machine learning techniques and an application of the Longest Common Subsequence algorithm. An application of the Longest Subsequence Problem sees use in the field of Bioinformatics as well, specifically in sequence alignment. Sequence alignment is the method used in Bioinformatics for sequence comparison, of which string-matching algorithms play a vital role in. Some applications of the algorithm in that matter involve the analysis of protein structures, wherein the Longest Common Subsequence problem comes to play. Highly similar subsequences found in proteins imply the existence of a sequence motif, a pattern of proven or assumed biological significance.

The following are the in-depth discussions on the structure that holds or supports the theory of the research study, based on the acquired knowledge of the proponent in the reviewed literature.

Longest Common Subsequence : A subsequence of a given sequence is formally defined as the sequence that is derived from a given parent sequence through the deletion of one or more elements of said parent without the disturbance of the relative positions of the remaining elements. Similar to a subsequence, a substring is a subsequence whose elements are formed from a consecutive run of elements from its parent. That is, a substring is a subsequence, but a subsequence may or may not be necessarily a substring.There are multiple ways for obtaining the LCS of two given strings, with several examples detailed below. For a given subsequence to be considered a common subsequence, it must be a subsequence of all the given parent sequences.

The naive solution to the problem follows a set of simple steps, as detailed below:

- Generate all possible subsequences, or the power set, of one of the parent sequences.
- Compare each generated subsequence to the parent sequences, and check if it is a subsequence of all parent or input sequences.
- Report and return the longest subsequence found.

However, generating all sequences is both time and space inefficient, being O(2^n) time and space, where n is the length of the parent sequence. Another cause for inefficiency is the lack of any form of inference for future data, rendering unnecessary processing and redundancy to happen. Such examples would be the failing of any given

subsequence to turn out to be a subsequence **of** all input sequences, wherein any other found subsequence using this subsequence as a prefix would also not be a subsequence of the input sequences, or that if a longer length subsequence of the given input sequences is found, generating and checking for its' prefixes are unnecessary, as it would also be considered a valid subsequence, albeit with a shorter length value.

In comparison, the recursive based solution performs similarly to the above brute force method, however the recursion starts at the end of the parent sequences instead, recursively iterating up to the initial character in the given sequences.

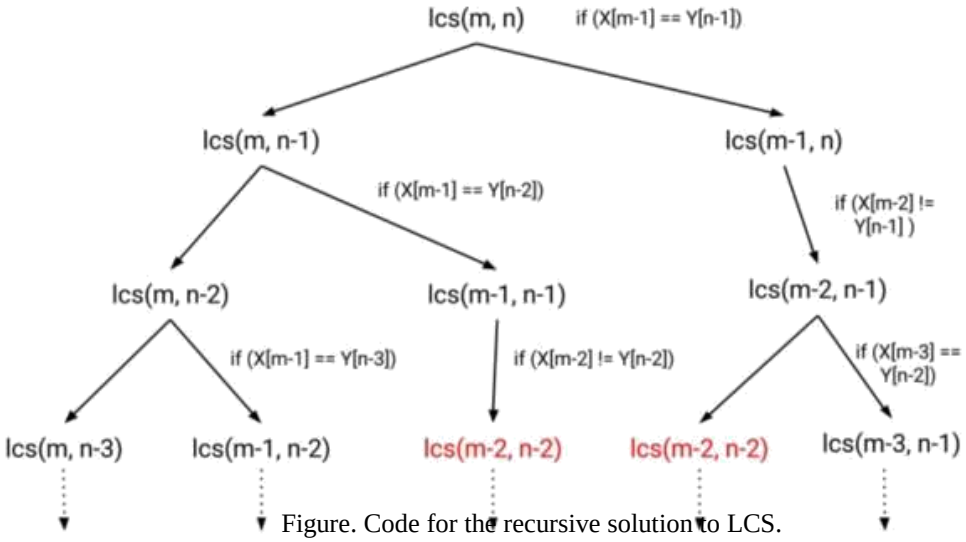

Figure. Code for the recursive solution to LCS.

The algorithm's cases is split into three cases:

- The base case, where if it reaches the first character, then it terminates, showing the maximum value of the length of the LCS returned.
- If a match is found, wherein it increments the stored output by 1, then continues recursion over the subsequences, moving past both matching characters.
- If no such match was found in a specific instance, wherein the recursion branches into two, one including one of the current instance's characters, while excluding the other, and with the other branching instance including the other character of this current instance, while excluding the character that is included on the other branch.

However, because of the algorithm's naivety, performance with regards to runtime will be similar to the above method, however the runspace would only be restricted to the maximum height of the call stack, in comparison to the brute force method. Along with that, the algorithm is susceptible to unneeded recursion, such as the algorithm iterating through characters that will not be a part of the solution set, or overlapping subproblems, wherein the same character instances may be compared multiple times in the program's runtime.

```
int lcs_length(char * A, char * B)
{
    if (*A == '\0' || *B == '\0') return 0;
    else if (*A == *B) return 1 + lcs_length(A+1, B+1);
    else return max(lcs_length(A+1,B), lcs_length(A,B+1));
}
```

Figure. Instance of an overlapping subproblem

This may be solved by performing memorization. This reduces the hefty 2^n recursive calls made by the algorithm through the use of a 2-D array by storing the computed value at position (m, n) in the array if it hasn't been stored

yet, and returning the stored value for any future repeated recursive calls that use the same character indices. This prevents any unnecessary repeated computations from being processed.

Swarm Intelligence: Swarm intelligence is a branch of Artificial Intelligence that is concerned with the concept of decentralized and self-organizing agents based on phenomena both natural and artificial. Such swarm components interact with each other stochastically and locally. Due to such, various emergent properties and behaviors can be observed globally, unbeknownst to the search agents individually. Various mathematical models of swarm intelligence techniques include Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), Artificial Bee Colony (ABC) Optimization, and African Buffalo Optimization (ABO) etc.

Ant Colony Optimization: Ant Colony Optimization is a metaheuristic that is modelled after the behavior of ants searching for food in the wild, eventually discerning a path to the found food source. As these ants return to the colony, pheromone markers are left, tracing their paths such that when other ants come across these path markers, they are also likely to follow it given a certain probability. These ants also drop pheromone markers through their path, strengthening it more and more such that ants are more likely to follow it. Those that don't instead explore for a closer path to the food source, or other food sources entirely, translating into a possibly more optimal solution. These paths must also be regularly updated, with the pheromones decaying over time.

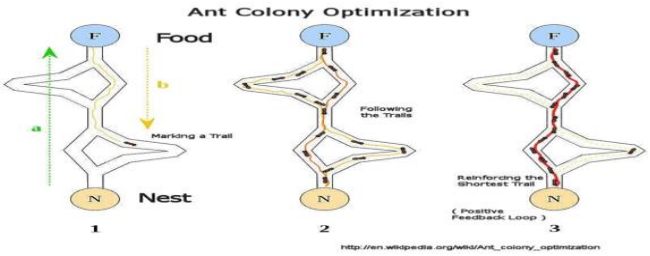

Figure. Ant Colony Optimization algorithm

In its core, the algorithm builds a solution through the reduction of the given computational problem into finding good paths through graphs, with the ants serving as a multi-agent search method whose behavior is inspired by the behavior of real-life ants. A given solution is built through building up promising solutions by each designated ant through the use of a probabilistic decision function. This is used along with a conversion of the search space such that each node has local information stored for the use of the optimization algorithm. Outgoing nodes are read and processed and the decision to move to a given outgoing node is calculated. Common control parameters of this algorithm include the number of ants, rate of evaporation of the pheromone, and the number of iterations for this algorithm.

```
Begin
    Initialize
        While stopping criterion not satisfied do
            Position each ant in a starting node
            Repeat
            For each ant do
              Choose next node by applying
            the state transition rule
              Apply local pheromone update
            End for
            Until every ant has built a solution
                Update best solution
                Apply global pheromone
            update
        End While
End
```

Figure. Pseudocode of the Ant System, the original ACO algorithm

ACO has its advantages such as its robustness, wherein the search agents, or ants would eventually converge into an optimum given enough iterations. Along with that, it has a good capability for discerning the global optimum of a given search space, in which the algorithm returns solutions that are close to what is considered the optimal solution. Additionally, it has an inherent ability to be parallelized, given the usage of multiple agents, or ants,

exploring the search space in concurrence. However, it is prone to its own probabilistic method for determining a given ant's pathing, along with a tendency to detect local optima and consequently result in poor convergence.

Algorithm Overview : A subsequence is a collection of symbols that is obtained from the removal of zero or more symbols from a given sequence. With this, given two input sequences x and y, with lengths of m and n respectively, the pair-wise LCS problem concerns the obtaining of a subsequence t such that t is both a subsequence of x and y, and the length of t is the maximum possible value, given said inputs. In order to create a baseline necessary for the comparison, the proponent shall be running the classic dynamic programming-based algorithm as the baseline in comparison to the proposed algorithm below.

Input Generation : DNA Sequences, or more specifically, DNA Sequencing, refers to the action of determining the order of the four nucleotides found in it; Adenine(A), Cytosine(C), Guanine(G), and Thymine(T). Due to the problem's relation with DNA Sequence Mapping, the proponent had devised an algorithm that generates random pairs of DNA sequences of a given length. Pairs of sequences length 100, 500, 1000, 5000, 10000, 15000, and 20000, are generated at the

corresponding files: "100.txt", "500.txt", "1000.txt", "5000.txt", "10000.txt", "15000.txt", and "20000.txt"

```
1  TAACCGTGATTCTGCATCATTTAGAAAAGATAGCGATGTAAGAAACGACATCTTGCTCTTATTGCGGAATCGTGCGACCATCTTCCCAGCCCTAAAGAGC
2
3  ATTTCCCCCAGTTCCCCGGACACTTTTGCGCCCGAGACCGACCGATGTGCTTGTACGGCTGACAACCTGGAAAATCGAACGCCCAAAAGTACGTTCATGG
4
5
```

Figure. Contents of 100.txt

The file 'main.py' takes in a filename as an argument, of which any of these input files may be chosen in order to compare and contrast the devised algorithm with the dynamic programming-based algorithm.

Base Algorithm: Compared to the naive method discussed before, the baseline algorithm uses a Dynamic Programming approach. This is due to the exponential time nature of the nn method. The naive method solves the problem, albeit inefficiently, by generating all possible subsequences of a given input sequence, then obtaining the longest one through comparing all of the generated subsequences. As this behavior is similar to obtaining the power set of a given set of n elements, this will generate a total of 2^n subsequences. This causes the runtime and runspace to explode even at relatively smaller input sizes.

Meanwhile, the Dynamic Programming approach uses a 2-d matrix, such that index (i, j) contains the length of the LCS for all characters up to the ith character of the first sequence, and the jth character of the second. In addition, this 2-d matrix considers -1 as an index, for the null part of the sequence. This row and column corresponding to it will contain 0, as there is no common character between any character and the value 'null'.

The general case is then as follows:

- If the value xi is equal to yj, then the value in index (i, j), is the value in the index (i – 1, j – 1) + 1. That is, it stores the value in the upper left diagonal index, incremented by 1.
- If the value xi is not equal to yj, then the value in index (i, j) is the max value between (i – 1, j), and (i, j – 1). That is, it stores the larger value between the index above it, and the index left of it.

```
def lcs(X, Y):
    m = len(X)
    n = len(Y)

    dp = [[0] * (n + 1) for item in range(m + 1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif X[i - 1] == Y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else :
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[m][n]
```

Figure. Code for the DP-based solution of the LCS

This behavior ensures that as discussed above, the bottommost right index contains the LCS of these two input sequences.

Proposed Algorithm

In order to apply the ACO algorithm for LCS, consideration for the following must be taken beforehand.

- Conversion of the problem into a structure conducive to the optimization algorithm.
- Development of the State Transition Ruleset that dictates the movements of the search agents.
- Design of the Pheromone Update Rule for the simulation of pheromones decaying in strength over time.
- Discernment of the criteria for the termination of the program.

Conversion of the Problem : Following the Ant System pseudocode as shown in the figure above, for the Ant Colony Optimization algorithm to be applied, the search agents must be initialized and positioned at the starting nodes. This is done by assigning an ant to each of the input sequence. For each of the ants, their assigned input sequence serves as the pivot from which it serves as the base of their traversal. In order to assist with the algorithm's requirements, additional information is stored in each character of each given input sequence. These are the position of each character of each sequence and their respective pheromone values, denoted as p and heuristic values, denoted as h. These nodes, now containing extra local information will hereby be called as states.

In the program's initialization, each state must set an initial pheromone value, and an initial heuristic value. The former defines the "value" of the state, that is, it only increases should the ant consider and store it as part of its generated solution, denoted as s, it will also be updated at the end of each generation. As for the latter, it denotes the local optimality of the given state and will be updated for every movement made by each search agent. This value lowers as the ant considers further states from its current position. This makes for two factors which govern the state transition ruleset as defined below.

With this, solutions to the LCS problem generated by the given ants are a series of feasible states in the given input sequences. The verification of the validity of the given solutions may be done by ensuring that each character of the generated solutions may be found on all the given input sequences, and that they appear in the same order as stored by the ant. The generated solution's length increments as more feasible states are traversed and stored.This was developed by the proponent through the design and implementation of a 'State' class in Python. This class contains the following properties:

- 'Char' - Contains the assigned character at a given position at a given sequence.
- 'Pos' - Contains the index of the assigned character at a given position at a given sequence.
- pherVal' - Contains the pheromone value of this given State, defaulted to 1.
- heuVal' - Contains the heuristic value of this given State, defaulted to 1.

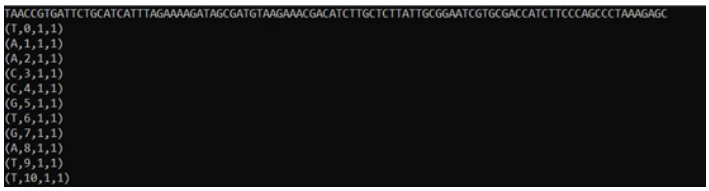

Figure. First sequence in 100.txt, with each character corresponding to the respective states(char, pos, pherVal, heuVal)

These states are collectively found at the variable state List, of which it is a 2-dimensional array such that the first dimension denotes the sequence, while the second dimension denotes the specific State of the respective character. Along with this, the search agent was also implemented as a class in python, named 'Ant', found on the file 'Ant.py'. This contains the necessary properties and methods in order to traverse the converted problem above and generate common subsequences using that as input. Each ant has the following properties:

- Soln, short for solution, is a list of vectors whose initial value is a set of zeroes equal to the number of input sequences. This will contain the indices of matching characters in their respective sequences.

```
[1, 0]
[6, 1]
[9, 2]
[10, 3]
[11, 4]
[14, 5]
[17, 6]
[18, 9]
```

Figure. Sample partial solution generated with '100.txt' as input. [1,0] indicates that the character at index 1 of the first sequence, and the character at index 0 of the second sequence are the same.

- Index, contains an integer relating to the current position of the ant in its assigned sequence.
- Seq, contains an integer relating to the sequence assigned to this ant.
- Running, contains a Boolean such that the ant is considered as 'finished', when it is set to '1', that is, the generated partial solution is found to be unable to be extended.
- step Count, contains an integer relating to the number of steps(movements), this ant has made.
- Extra, contains a list of vectors not dissimilar to soln, however this is used for extending the solution better, by finding matching pairs of characters in between the vectors on soln, discussed below.

Preprocessing of the Input : Initially, as the input is a file containing two sequences of a given length, preprocessing must be made in order to integrate it into the algorithm proper.

```
# Converting the input
for line in lines:
    if line != '\n':
        #print(line.strip())

        # Reset per sequence
        tempList = []
        index = 0

        # Creates a state instance per character, heuVal and pherVal defaulted to 1
        for char in line:
            if(char != '\n'):
                ch = states.state(char, index, 1, 1)
                tempList.append(ch)
            index += 1
        stateList.append(tempList)
```

Figure. Conversion of the input into states.

For each character in each of the input sequences, they are simply converted into an instance of the 'state' class, which contains the index, heuristic, and pheromone values, of which the latter two are initialized to the default of 1. In turn, stateList is filled with a list of these states, organized by their respective sequences, then organized further by their position on those sequences. This allows easy traversal for the ants, as stateList[0], for example, corresponds to all the states of the first sequence in the input, organized by their position.

Development of the State Transition Ruleset : As discussed above, two components must be considered when devising the ruleset for the search agents' movement. These are the state's pheromone value, derived from how often said state is chosen as part of the generated solution, and the heuristic value, a value that governs the likeliness of transitioning onto a given target state, calculated locally. For a given ant to generate a feasible solution to the problem, it considers the next w states, w being the window size defined by the proponent. This bounds each step of the given ant to said w states, or up until the last index, should this exceed the assigned sequence's bounds.

From this, each state within w is considered as a candidate state, wherein the ant may or may not choose to move to the given state. If the ant has chosen a state within w to transition to, it then checks for the feasibility of the newly extended solution. By storing positional vectors of common characters in s, the generation of a feasible solution is created through the mapping of common characters in the other sequences, in relation to the sequence ant Am is assigned to. This is done by first considering the position vector on the last index in s, finding the first instance of the character in its corresponding sequence that is the same as the character in the ant's chosen state, starting from the next index from the stored value in the given vector. This ensures that all indices in each vector correspond to a single character at differing indices on their respective sequences.

As for the transitioning itself, it is governed by a random number q, generated every movement the ant makes, bound by two parameters q0 and q1 such that $0 < q < 1$,and $0 < q0 < q1 < 1$. These two parameters affect whether or not the ant either moves to the calculated local optimum, that is chosen to be the state with the largest combination of pheromone and heuristic value, chooses a random state within w that is affected by the state's own pheromone and heuristic values, or simply does not choose a state at all and moves w steps forward.

As heuristic values are calculated for each step, since each ant's movement is bound by the given window size, only the heuristic values for the next w states are considered. With this, consideration must be made as to which state the ant chooses, as any characters in between its current state and its chosen next state cannot be used as further extensions to the partial solution generated, as the solution set will lose its order. Movements that choose farther away states result in a large number of states rendered invalid through this property, potentially reducing the quality of the solution. As such, the heuristic value of a given state is obtained through the calculation of the total number of states that are rendered invalid if the ant were to move to the given state, this is calculated through finding the first instance of the character across all sequences, starting from the ant's current index on the given sequence. The value 1 is then divided by this total value, resulting in a value that is inversely proportional to the number of invalid characters should that state be chosen.

After the heuristic values of the next w states have been calculated, the probabilities for the ant to transition to each state must then be calculated. This is done through obtaining the product of each state's pheromone value, and the calculated heuristic value, raised by some parameter P, that dictates the weight of the heuristic value relative to the pheromone value in the 'viability' of the given state. The total value is then calculated by the sum of each state's product, of which each probability is calculated by dividing the state's product from the total.The ant then decides its movement through the generation of a random number p, such that its chances of moving to the local optimum(denoted by the state with the highest product over total value), moving to a random state within w, influenced by their individual probabilities, or skipping the next w states are based on the value relative to the parameters p0 and p1 such that $0 < p0 < p1 < 1$. The three possibilities above are then governed by the following conditions.

- Should $p < p0$, the ant immediately moves to the calculated local optimum within w.
- Should $p0 < p < p1$, the ant will transition to a random state governed by the individual probabilities of the next w state.
- Should $p < p1$, then the ant skips all states within w.

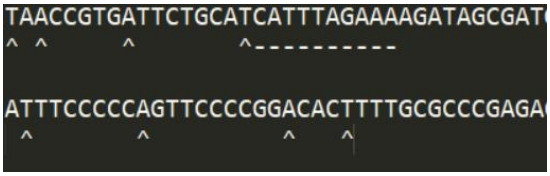

Figure. Sample partial solution with ant assigned onto the first sequence, '-' is used to denote the next w characters, of which it has an option of choosing the most promising state, moving to a state in w, or skipping the entire window.Should the first two cases be chosen by the ant, then it generates a vector of indices to be added to its generated partial solution. This is done by finding matching characters across all sequences, using the values of the last vector in its 'soln' as starting points for searching matching characters. This process is then repeated until it reaches the end of the sequence, to which its 'running' value is then set to 1.

An extra step is then done in order to improve the solution further. This step improves the solution generated by attempting to find matching characters that were skipped, but are valid additions into the generated solution. Through the use of each ant's soln property, it then iterates over the list of characters that were not chosen to be part of the solution in between soln[i], and soln[i+1], in order to preserve order within the solution, note that this improvement does not consider character indices that are already considered as part of soln, and that this is done in a greedy manner, that is, the first matching set of characters are added, as long as it satisfies the requirements for a valid common subsequence.

Implementation of the Pheromone Update Ruleset : Once all ants have successfully generated their solutions, the longest solution generated is designated as the 'generation-best' solution. This generation-best solution is then used in order to alter the pheromone values of all states in accordance to whether or not said state is an element of it.

Initially, all states' pheromone values would be reduced by a factor of (1 - e), such that e is the Evaporation Coefficient. After this step is done, all states found in soln and

extra are then incremented by a value of e. This value is then used as the next generation's pheromone value.

Criteria for Termination : The algorithm terminates once it has reached a total of L loops without generating a longer subsequence than what has already been stored.

Timestamping and Memory Tracking: In order to compare the DP-based solution with the devised algorithm, timestamps were made within the program itself. Three timestamps were made in total.

The first was placed after the input has been processed, but before the ACO-LCS algorithm has been initialized and ran. This serves as the start of the timer for the algorithm itself.

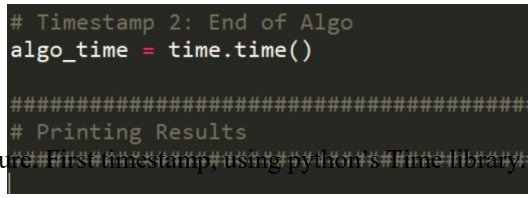

Figure. First timestamp, using python's time library.

The second was then placed after the algorithm had run its course. Subtracting this by the first timestamp would return the total time the algorithm has ran. Alongside this, this second timestamp also serves as the starting point of the DP-LCS algorithm.

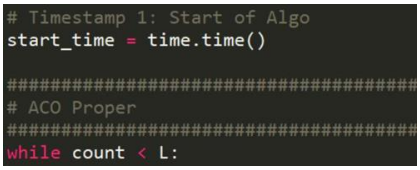

Figure. Second timestamp, located before output is printed.

The third and last timestamp is then placed after the DP-LCS algorithm has terminated, with the actual runtime calculated by subtracting this by the second timestamp.

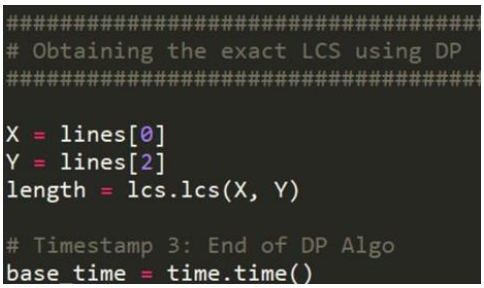

Figure. Third timestamp, located after the DP-LCS algorithm has terminated.

In order to obtain the memory usage of the algorithm, the library tracemalloc was also imported, with the trace itself initialised on where the first timestamp is also located, at the start of the ACO algorithm proper. Two stamps are then implemented on the algorithm proper, both placed where the second and third timestamp above is located. The function tracemalloc.get_traced_memory() returns two values, the first contains the current memory used up by the algorithm, and the second contains the peak amount of memory used by the program, this latter value is then used to compare the memory usage of both algorithms. Alongside that, tracemalloc.resetpeak() is ran in order to reset the second value before the second tracemalloc.get_traced_memory() is ran, in order to obtain the peak value of the DP-based solution, should it return a memory usage value that is lower than the ACO-based solution.

III. RESULTS AND DISCUSSION

Test Platform : The test platform for the experiments was a personal laptop running Windows 10, using an Intel(R) Core(TM) i7-4510U CPU, with 8.00 GB of RAM. The algorithm was devised and implemented in Python, ran on the Python 3.9.5 interpreter.

Parameter	L	w	p0	p1	P	e
Value	20	6	0.92	0.98	1.25	0.05

Table 4.1

These parameters were chosen after testing as it has shown a relatively consistent and efficient performance, compared to the performance of the algorithm using different parameter values. Along with these values, the initial pheromone and heuristic values are also set to 1, with the latter updating per each step made by the ants, and the former updating at the end of each generation.As discussed above, L is the number of loops made by the algorithm that resets each time a better than the currently saved solution is generated, this is set in order to allow possibly bettersolutions to be generated and stored as the pheromone values are updated at the end of each generation. Meanwhile, w is the window size, of which for each movement done, it considers the next w states in calculation for the next possible movement.

The next two values, p0 and p1, are used to determine the preference of the algorithm in choosing the calculated local optimum, moving to a random state within w, or ignoring the next w states entirely. Similar to the previous two variables, P is used as the bias between the pheromone value and the heuristic value in the computation of the local optimum, and the viability of the other states.And lastly, e serves as the modifier from which the pheromone value is either decremented or incremented by at the end of each generation, depending on the values on the generation-best generated solution.

Sequence Length	# Attempts	Average Memory Used(ACO)in seconds	Average Memory Used(DP) in seconds	Efficiency of ACO to DP	Efficiency in Percent
100	1000	0.0914	0.0461	.504	50.4%
500	1000	0.552		.232	23.2%
1000	1000	0.975	0.128	.553	55.3%
5000	500	4.38		3.112	311.2%
10000	100	8.773	0.539	6.727	672.7%
15000	50	12.178	116.104	9.534	953.4%
20000	50	18.368	239.781	13.054	1305.4%

Table 4.2

As shown on table 4.2, for small inputs, the DP-based algorithm performs significantly faster, with the ACO algorithm performing linearly, in comparison to the exponential growth of the runtime of the DP-based algorithm.

This linear growth in runtime would only cause the algorithm to perform significantly faster than the DP-based algorithm on pairs of sequences length > 20000, based on the performance of both algorithms shown above. This allows the generation of approximate solutions within a reasonable timeframe, in comparison to the exact solution being generated in exponentially larger timeframes, as the input length increases.

Sequence Length	# Attempts	Average Memory Used(ACO)in MB	Average Memory Used(DP) in MB	Efficiency of ACO to DP	Efficiency in Percent

100	1000	0.039	0.127	3.25	325%
500	1000	0.190	2.339	12.310	1231%
1000	1000	0.403	11.833	29.362	2936.2%
5000	500	2.109	359.376	170.401	17040.1%
10000	100	4.214	1468.477	348.475	34847.5%
15000	50	6.334	3327.78	525.383	52538.3%
20000	50	8.441	5937.02	703.356	70335.6%

Table : 4.3

As shown on table 4.3, even for small inputs, the ACO algorithm performs significantly better in terms of runspace, performing 325% better at sequence length 100, and the gap between it and the DP-based solution's performance in terms of runspace grows exponentially. The growth of the ACO algorithm also is linear, as opposed to the exponential growth the DP-based solution performs. Based on the results above, this trend will continue for input sequences length > 20000. Due to this exponential growth, the memory requirement for generating exact solutions will increase exponentially and thus, may not be feasible for large enough input sequences.

Sequence Length	# Attempts	Average Lenth of Solutions generated (ACO)	Lenth of Exact Solution	Efficiency of ACO to DP	Efficiency in Percent
100	1000	55.612	60	0.911	91.9%
500	1000	281.499	322	0.874	87.4%
1000	1000	561.299	649	0.865	86.5%
5000	500	2788.02	3280	0.85	85%
10000	100	5527.86	6527	0.847	84.7%
15000	50	8296.18	9781	0.848	84.8%
20000	50	11099.24	13094	0.848	84.8%

Table : 4.4

In terms of solution quality, as input length increases, solution efficiency seems to stay constant at 84.8%, and the constant solution quality trend is assumed to be followed for input length > 20000 on randomly generated DNA strings.

III. SUMMERY AND RECOMMENDATION

The study's main concern is to explore methods in order to improve the Longest Common Subsequence algorithm. The method introduced is the Ant Colony Optimization, or ACO, algorithm, which uses a set of search agents in order to generate solutions on a given search space. For the proponent, the critical problem in this research is into devising the problem that is conducive to how the algorithm behaves. That is, converting the search space into one that allows for the storage of several other properties related to a point in the search space. This is done through the conversion of each character in the given sequence into a state, which contains those properties. Alongside that, the devising of the State Transition Ruleset is also important in ensuring that the search agents move "smartly". That is, it has the ability to discern both the validity of the state and its optimality, at least in a local concern for the latter.

The implementation of Ant Colony Optimization was done through the conversion of the input sequences into states, which contain other variables used by the algorithm itself, aside from the character within the given input sequence. These states are then used by the search agents to traverse throughout the given input, generating subsequences of their own as per the State Transition Ruleset. As the search agent chooses a target state to transition to, it also verifies the validity of the then chosen subsequence before adding the target state in its stored solution. Once these search agents have terminated, that is, a further extension to the stored solution is not found to be possible, an extra greedy step is then implemented in order to improve upon the generated solution. This greedy step finds matching characters in between the states chosen to be part of the solution set, and the validity of the solution is retained as these characters may be inserted in the correct indices in between the congregated solution

set.This allows for the generation of solutions which are both valid subsequences and with solution lengths which are of a consistent average length relative to the exact length of the globally optimal solution. This is despite the biased yet random nature of the State Transition Ruleset, the greedy nature of the ruleset developed along with the extra "improvement" step, and the shortcomings of Ant Colony Optimization as a whole with respect to its tendency for premature convergence.The proponent defines the project scope and limitations of the study in this section as follows in order to specify the boundaries of said study. The existence of the Exponential Time Hypothesis implies the impossibility for a subquadratic-time exact algorithm for the Longest Subsequence Problem.

- As such, the devised algorithm must show an improvement on the pre-existing Dynamic Programming based solution to the problem in terms of its runtime in contrast to its accuracy. Failing that, the devised algorithm must run substantially faster in comparison, while still retaining a good enough solution quality
- The algorithm shall be run on several sets of pre-generated strings, with said strings containing the following alphabetical characters 'A', 'C', 'G', and 'T', to simulate DNA sequences.
- Analysis of the algorithm's performance shall also be done in order to display and discern the difference between how the processing of the given strings are done in the Dynamic Programming approach in comparison to the developed algorithm.

The following list are the recommendations by the researcher for future studies.

1. Improve upon the Pheromone Update Ruleset by separating the method that reduces the pheromone values of each state and increases the pheromone values of the respective generation-best solution.
2. Expand testing to consider other types of inputs, provided it can be categorized.
3. Implement other known approximation algorithms and compare runtime and runspace with respect to the Ant Colony Optimization implementation, and the Dynamic Programming approach.
4. Consider different approaches to calculating the heuristic values of each state in the given window, and consequently, how the local optima are discerned.
5. Separate the variable used for the recalculation of the pheromone values after each generation, e.

IV. CONCLUSION

The proponent was successful in the design and implementation of Ant Colony Optimization within the Longest Common Subsequence problem. As discussed in the results above, the algorithm improves greatly in terms of its runtime and runspace, mainly due to the linear growth it has exhibited throughout testing. This linear growth allows for the generation of solutions that are increasingly faster and use up less space relative to the Dynamic Programming approach due to the latter's quadratic growth in terms of runtime and runspace. However, in terms of solution exactness, the generated solutions are at an average of 84.8% of the exact solution's length, and this behavior is assumed to hold as the input length increases. This can be attributed to the tendency of the algorithm to converge on local optima, which results in solutions that when viewed locally, that is, without any respect for the whole, each piece of the generated solution is "good enough", but when viewed as a whole, results in a solution that isn't as optimal as the exact, or the globally optimal, solution. Note that this measurement holds for randomly generated DNA strings as is stated in the project scope and limitations. This is also limited to pairs of input strings as in its current state, the Dynamic Programming approach implementation is only designed for pairwise LCS.

REFERENCES

[1] Abboud, A., Backurs, A. and Williams, V., 2021. Quadratic-Time Hardness of LCS and other Sequence Similarity Measures. [online] arXiv.org. Available at: <https://arxiv.org/abs/1501.07053> [Accessed 22 June 2021].

[2] Abboud, A., Vassilevska Williams, V. and Weimann, O., 2021. Consequences of Faster Alignment of Sequences. [online] Cs.au.dk. Available at: <https://people.csail.mit.edu/virgi/hardstrings.pdf> [Accessed 22 June 2021].

[3] Nlp.stanford.edu. 2021. Edit distance. [online] Available at: <https://nlp.stanford.edu/IR-book/html/htmledition/edit-distance-1.html> [Accessed 22 June 2021]. PAGASA, "Floods," PAGASA, https://www.pagasa.dost.gov.ph/learning-tools/floods (accessed Mar. 29, 2024).

[4] W. J. Masek and M. S. Paterson, "A Faster Algorithm Computing String Edit Distances* ," thesis, 1978.

[5] Arlazarov, V.; Dinic, E.; Kronrod, M.; Faradžev, I. (1970), "On economical construction of the transitive closure of a directed graph", Dokl. Akad. Nauk SSSR, 194 (11). Original title: "Об экономном построении

транзитивного замыкания ориентированного графа", published in Proceedings of the USSR Academy of Sciences 134

[6] Rubenstein, A. and Song, Z., 2021. Reducing approximate Longest Common Subsequence to approximate Edit Distance. [online] Available at: <https://arxiv.org/abs/1904.05451> [Accessed 22 June 2021].

[7] G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. SIAM Journal on Computing, 27(2):557–582, 1998

[8] Z. Bar-Yossef, T. Jayram, R. Krauthgamer, and R. Kumar. Approximating edit distance efficiently. In Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science, pages 550–559. IEEE Computer Society, 2004.