**DNA Analysis and Comparison using Advanced DSA Techniques**

**Course: Design Analysis and Algorithms**

**Branch: Computer Science and Engineering (CSE)**

**Institution: Indian Institute of Information Technology, Nagpur**

**Team Members:**

| **Name** | **Roll Number** |
| --- | --- |
| Nishchal Verma | BT23CSE160 |
| Divyansh Pal | BT23CSE148 |
| Rishabh Choraria | BT23CSE158 |
| Mohit Bhalotia | BT23CSE181 |
| Aakarsh Maurya | BT23CSE178 |

**1. Introduction**

DNA sequence analysis plays a crucial role in the fields of bioinformatics, genetics, and molecular biology. It involves the study of the nucleotide arrangement (A, T, C, G) in DNA molecules, helping scientists understand genetic variations, identify diseases, and develop treatments.

**Motivation**

This project aims to bridge the gap between theoretical algorithmic approaches and practical bioinformatics applications. By implementing and comparing different algorithmic strategies, we explore their effectiveness in understanding DNA patterns, similarities, and variations across different classes of human DNA.

**2. Project Objectives**

**2.1 Classify DNA Sequences into 7 Groups**

The project categorizes DNA sequences into the following biological classes:

* G Protein Coupled Receptors
* Tyrosine Kinase
* Tyrosine Phosphatase
* Synthetase
* Synthase
* Ion Channel
* Transcription Factor

**2.2 Perform Efficient Operations**

* **Comparison:** Measure sequence similarity (intra-class and inter-class) using LCS and mutation analysis.
* **Assembly Simulation:** Mimic biological DNA fragment assembly.
* **Pattern Detection:** Discover motifs, prefixes, and suffixes using Tries and Backtracking.

**2.3 Apply Multiple DSA Approaches**

The project utilizes five algorithmic strategies:

* Dynamic Programming (DP)
* Greedy Algorithms
* Backtracking
* Divide and Conquer
* Trie Data Structures

**3. Algorithms Used**

**3.1 Dynamic Programming (DP)**

**Applications:**

* Longest Common Subsequence (LCS) Computation
* Mutation Detection

**Example:**

Given sequences:  
A = AGGTABA  
B = GXTXAYBB  
LCS = "GTAB" (Length = 4)

| **Parameter** | **Value** |
| --- | --- |
| Time Complexity | O(m × n) |
| Space Complexity | O(m × n) |

**Advantages:**

* Guarantees optimal results.
* Effective for DNA, RNA, protein sequences.

**Disadvantages:**

* High memory consumption for large sequences.

**3.2 Backtracking**

**Applications:**

* Discover all LCS solutions.
* Motif finding.

| **Parameter** | **Value** |
| --- | --- |
| Time Complexity | O(2(m + n)) |
| Space Complexity | O(m + n) |

**Advantages:**

* Finds all possible motifs.
* Complete exploration of solution space.

**Disadvantages:**

* Exponentially slow for large sequences.

**3.3 Divide and Conquer**

**Applications:**

* Space-efficient DNA sequence comparison.
* Memory Optimization during Sequence Analysis.

| **Parameter** | **Value** |
| --- | --- |
| Time Complexity | O(m × n) |
| Space Complexity | O(m + n) |

**Advantages:**

* Suitable for very large sequences.
* Reduces memory usage drastically.

**Disadvantages:**

* Complex implementation structure.

**3.4 Greedy Algorithm**

**Applications:**

* Approximate DNA fragment assembly.
* DNA Assembly Simulation

| **Parameter** | **Value** |
| --- | --- |
| Time Complexity | O(k × m × n) |
| Space Complexity | O(k × m) |
|  |  |

**Advantages:**

* Fast and efficient for short reads.

**Disadvantages:**

* May miss the globally optimal assembly.

**3.5 Trie (Prefix Tree)**

**Applications:**

* Pattern detection
* Common prefix/suffix identification

| **Parameter** | **Value** |
| --- | --- |
| Time Complexity | O(L) |
| Space Complexity | O(total characters) |

**Advantages:**

* Rapid pattern search.
* Compact storage of repeated patterns.

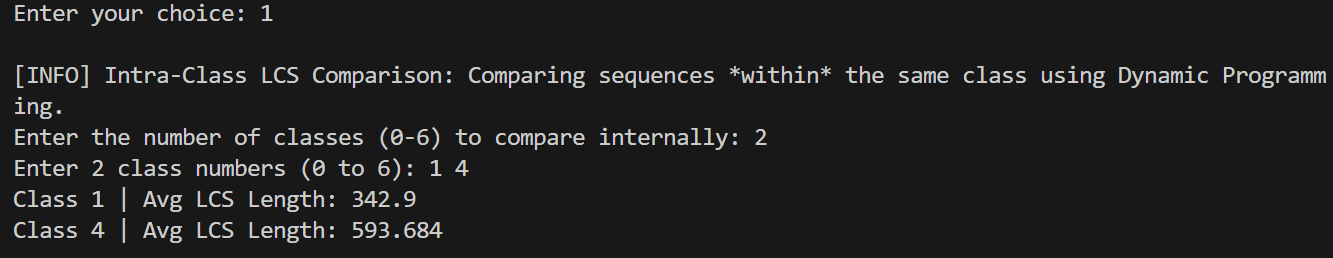
**Disadvantages:**

* High space usage with highly diverse data.

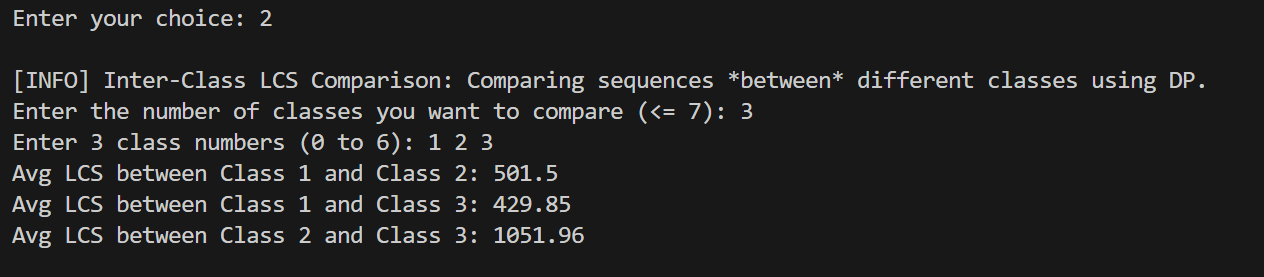
**4. Application Analysis**

**4.1 Dynamic Programming (DP)**

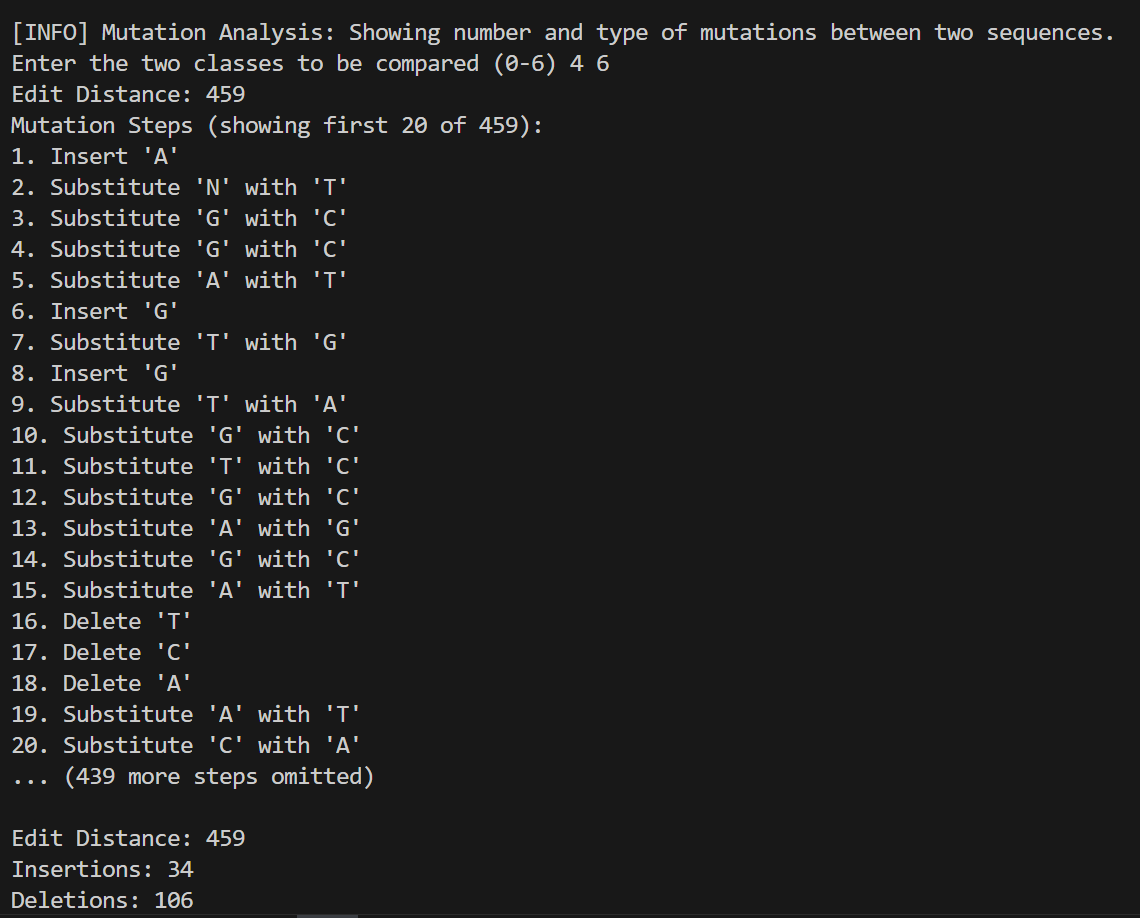
* **LCS Computation Code**
* pair<int, string> lcsq\_dp(const string &a, const string &b)
* {
* int n = a.length(), m = b.length();
* vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
* // DP table construction
* for (int i = 1; i <= n; ++i)
* {
* for (int j = 1; j <= m; ++j)
* {
* if (a[i - 1] == b[j - 1])
* dp[i][j] = 1 + dp[i - 1][j - 1];
* else
* dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
* }
* }
* // Reconstructing the LCS string
* int i = n, j = m;
* string lcs = "";
* while (i > 0 && j > 0)
* {
* if (a[i - 1] == b[j - 1])
* {
* lcs = a[i - 1] + lcs;
* --i;
* --j;
* }
* else if (dp[i - 1][j] > dp[i][j - 1])
* {
* --i;
* }
* else
* {
* --j;
* }
* }
* return {dp[n][m], lcs};
* }
* **Intra-Class Similarity Comparison**
* void Intra\_Class\_Comparison(const vector<vector<string>> &Sequence\_byClass)
* {
* set<int> selectedClasses;
* int n;
* cout << "Enter the number of classes (0-6) to compare internally: ";
* cin >> n;
* cout << "Enter " << n << " class numbers (0 to 6): ";
* for (int i = 0; i < n; ++i)
* {
* int classNum;
* cin >> classNum;
* if (classNum >= 0 && classNum <= 6)
* selectedClasses.insert(classNum);
* else
* cout << "Invalid class number: " << classNum << " (Skipping)\n";
* }
* for (int c : selectedClasses)
* {
* const vector<string> &group = Sequence\_byClass[c];
* if (group.size() < 2)
* {
* cout << "Class " << c << " has insufficient data for comparison.\n";
* continue;
* }
* int totalLCS = 0, count = 0;
* for (size\_t i = 0; i < min((size\_t)20, group.size()); ++i)
* {
* for (size\_t j = i + 1; j < min((size\_t)20, group.size()); ++j)
* {
* pair<int, string> value = lcsq\_dp(group[i], group[j]);
* totalLCS += value.first;
* count++;
* }
* }
* cout << "Class " << c << " | Avg LCS Length: "
* << ((count > 0) ? (double)totalLCS / count : 0) << endl;
* }
* }

**OUTPUT**

* **Intra-Class Similarity Comparison**
* set<int> selectedClasses;
* int n;
* cout << "Enter the number of classes you want to compare (<= 7): ";
* cin >> n;
* if(n > 7) {
* cout << "number of classes must be less than 7, Retry\n";
* Inter\_Class\_Comparison(Sequence\_byClass);
* return;
* }
* cout << "Enter " << n << " class numbers (0 to 6): ";
* for (int i = 0; i < n; ++i)
* {
* int classNum;
* cin >> classNum;
* if (classNum >= 0 && classNum <= 6)
* selectedClasses.insert(classNum);
* else
* cout << "Invalid class number: " << classNum << " (Skipping)\n";
* }
* vector<int> classes(selectedClasses.begin(), selectedClasses.end());
* for (int i = 0; i < classes.size(); ++i)
* {
* for (int j = i + 1; j < classes.size(); ++j)
* {
* int c1 = classes[i], c2 = classes[j];
* const vector<string> &g1 = Sequence\_byClass[c1];
* const vector<string> &g2 = Sequence\_byClass[c2];
* int totalLCS = 0, count = 0;
* for (int x = 0; x < min(10, (int)g1.size()); ++x)
* {
* for (int y = 0; y < min(10, (int)g2.size()); ++y)
* {
* pair<int, string> value = lcsq\_dp(g1[x], g2[y]);
* totalLCS += value.first;
* count++;
* }
* }
* if (count > 0)
* {
* cout << "Avg LCS between Class " << c1 << " and Class " << c2 << ": "
* << (double)totalLCS / count << endl;
* }
* }
* }

**OUTPUT** ****

* **Mutation Summary**
* int m = A.size(), n = B.size();
* vector<vector<int>> dp(m + 1, vector<int>(n + 1));
* for (int i = 0; i <= m; ++i)
* dp[i][0] = i;
* for (int j = 0; j <= n; ++j)
* dp[0][j] = j;
* for (int i = 1; i <= m; ++i)
* for (int j = 1; j <= n; ++j)
* dp[i][j] = (A[i - 1] == B[j - 1]) ? dp[i - 1][j - 1] : 1 + min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]});
* MutationSummary summary;
* int i = m, j = n;
* while (i > 0 || j > 0)
* {
* if (i > 0 && j > 0 && A[i - 1] == B[j - 1])
* {
* --i;
* --j;
* }
* else if (i > 0 && j > 0 && dp[i][j] == dp[i - 1][j - 1] + 1)
* {
* ++summary.substitutions;
* --i;
* --j;
* }
* else if (i > 0 && dp[i][j] == dp[i - 1][j] + 1)
* {
* ++summary.deletions;
* --i;
* }
* else if (j > 0 && dp[i][j] == dp[i][j - 1] + 1)
* {
* ++summary.insertions;
* --j;
* }
* }
* return {dp[m][n], summary};

**OUTPUT** ****

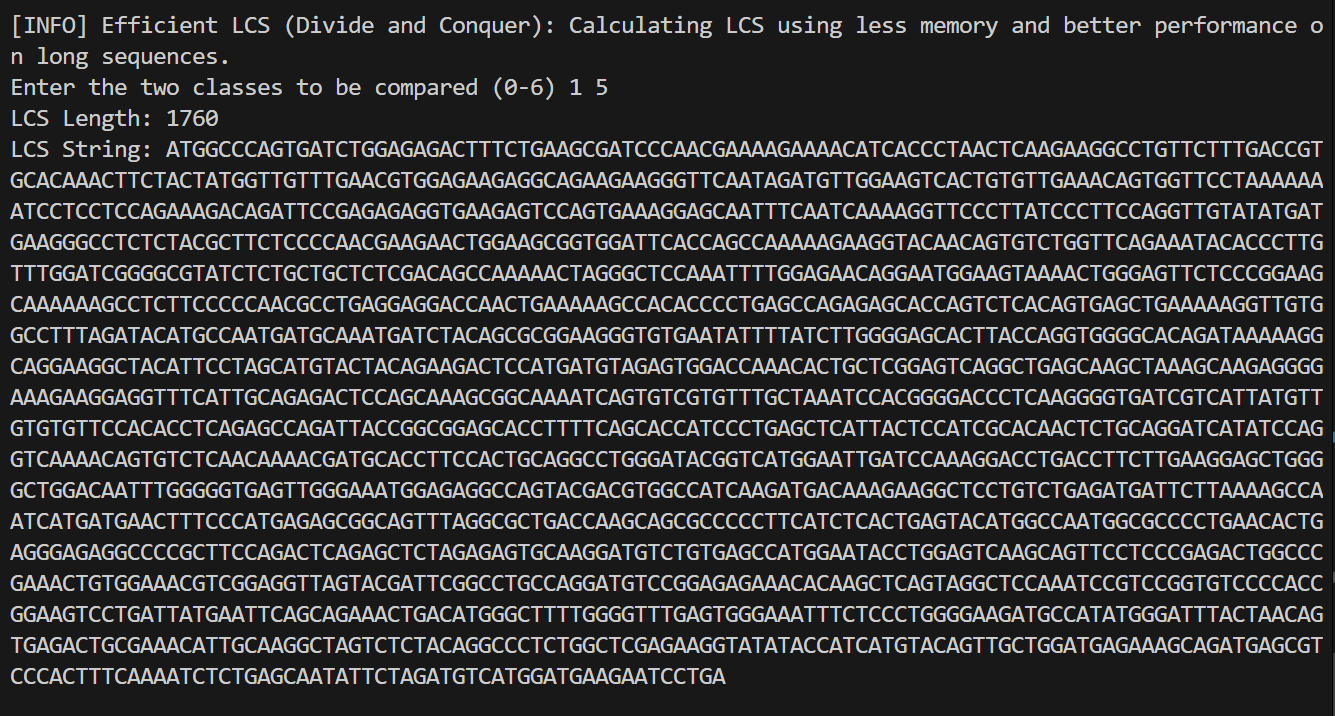
**4.2 Backtracking**

* **Motif finding.**
* void backtrackAllLCS(const vector<vector<int>> &dp, const string &X, const string &Y, int i, int j, string lcs, set<string> &all\_lcs) {
* if (i == 0 || j == 0) {
* reverse(lcs.begin(), lcs.end());
* all\_lcs.insert(lcs);
* return;
* }
* if (X[i - 1] == Y[j - 1]) {
* backtrackAllLCS(dp, X, Y, i - 1, j - 1, lcs + X[i - 1], all\_lcs);
* } else {
* if (dp[i - 1][j] >= dp[i][j - 1])
* backtrackAllLCS(dp, X, Y, i - 1, j, lcs, all\_lcs);
* if (dp[i][j - 1] >= dp[i - 1][j])
* backtrackAllLCS(dp, X, Y, i, j - 1, lcs, all\_lcs);
* }
* }
* set<string> getAllLCS(const string &X, const string &Y) {
* int m = X.length(), n = Y.length();
* vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
* // Fill DP table
* for (int i = 1; i <= m; ++i)
* for (int j = 1; j <= n; ++j)
* if (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]);
* set<string> all\_lcs;
* backtrackAllLCS(dp, X, Y, m, n, "", all\_lcs);
* return all\_lcs;
* }

**OUTPUT**

**4.3 Divide and Conquer**

* **Space-efficient DNA sequence comparison**
* int m = a.length(), n = b.length();
* if (m == 0 || n == 0) return "";
* if (m == 1) {
* for (char ch : b) {
* if (a[0] == ch)
* return string(1, a[0]);
* }
* return "";
* }
* int mid = m / 2;
* string a\_left = a.substr(0, mid);
* string a\_right = a.substr(mid);
* vector<int> l1 = compute\_lcs\_row(a\_left, b);
* string a\_right\_rev = a\_right;
* reverse(a\_right\_rev.begin(), a\_right\_rev.end());
* string b\_rev = b;
* reverse(b\_rev.begin(), b\_rev.end());
* vector<int> l2 = compute\_lcs\_row(a\_right\_rev, b\_rev);
* int maxLen = -1, split = 0;
* for (int i = 0; i <= n; ++i) {
* if (l1[i] + l2[n - i] > maxLen) {
* maxLen = l1[i] + l2[n - i];
* split = i;
* }
* }
* string leftLCS = lcs\_divide\_conquer(a\_left, b.substr(0, split));
* string rightLCS = lcs\_divide\_conquer(a\_right, b.substr(split));
* return leftLCS + rightLCS;

**OUTPUT **

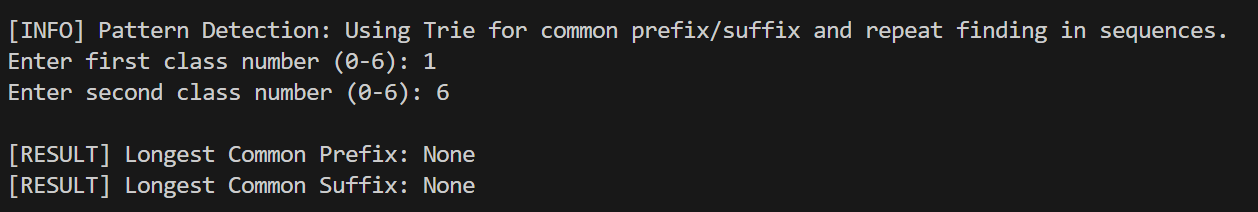
**4.4 Greedy Algorithm**

* **Space-efficient DNA sequence comparison**
* string greedy\_dna\_assembly(vector<string> &fragments) {
* while (fragments.size() > 1) {
* int maxOverlap = -1;
* int l = 0, r = 0;
* string bestMerge = "";
* for (int i = 0; i < fragments.size(); ++i) {
* for (int j = 0; j < fragments.size(); ++j) {
* if (i != j) {
* string merged;
* int overlap = calc\_overlap(fragments[i], fragments[j], merged);
* if (overlap > maxOverlap) {
* maxOverlap = overlap;
* bestMerge = merged;
* l = i;
* r = j;
* }
* }
* }
* }
* if (maxOverlap == -1) break;
* fragments[l] = bestMerge;
* fragments.erase(fragments.begin() + r);
* }
* return fragments.empty() ? "" : fragments[0];
* }

**OUTPUT** ****

**4.5 Trie (Prefix Tree)**

* **Common prefix/suffix identification**
* void analyzeCommonPrefixSuffix(const vector<string>& sequences) {
* if (sequences.empty()) {
* cout << "No sequences to analyze.\n";
* return;
* }
* int total = sequences.size();
* TrieNode\* prefixRoot = new TrieNode();
* for (string s : sequences) {
* s.erase(remove\_if(s.begin(), s.end(), ::isspace), s.end());  // Remove whitespace
* insert(prefixRoot, s);
* }
* string prefix = longestCommon(prefixRoot, total, 0.8);
* TrieNode\* suffixRoot = new TrieNode();
* for (string s : sequences) {
* s.erase(remove\_if(s.begin(), s.end(), ::isspace), s.end());  // Remove whitespace
* reverse(s.begin(), s.end());
* insert(suffixRoot, s);
* }
* string suffix = longestCommon(suffixRoot, total, 0.8);
* reverse(suffix.begin(), suffix.end());
* cout << "\n[RESULT] Longest Common Prefix: " << (prefix.empty() ? "None" : prefix) << endl;
* cout << "[RESULT] Longest Common Suffix: " << (suffix.empty() ? "None" : suffix) << endl;
* }

**OUTPUT **

**5. Summary Chart**

| **Algorithm** | **Time Complexity** | **Space Complexity** | **Best For** | **Weakness** |
| --- | --- | --- | --- | --- |
| Dynamic Programming (DP) | O(m × n) | O(m × n) | Precise DNA similarity and mutation detection | High memory usage |
| Backtracking | O(2^(m+n)) | O(m+n) | Exhaustive LCS and motif discovery | Extremely slow for large sequences |
| Divide and Conquer | O(m × n) | O(m+n) | Large sequence comparison with low memory | Complex recursion |
| Greedy | O(k × m × n) | O(k × m) | Fast DNA assembly simulation | May miss optimal assembly |
| Trie | O(L) | O(total characters) | Fast pattern search, common prefix detection | High space usage |

**6. Conclusion**

This project successfully implemented major DSA techniques for the analysis of human DNA sequences. By categorizing DNA into seven biological classes and utilizing Dynamic Programming, Backtracking, Divide and Conquer, Greedy algorithms, and Trie structures, we performed key operations such as:

* Longest Common Subsequence computation
* Mutation analysis
* DNA assembly simulation
* Motif and pattern detection

The project demonstrates the crucial role that algorithmic strategies play in solving practical bioinformatics problems. Through efficient design and thoughtful implementation, we bridged computational techniques with biological insights, showcasing the potential for algorithms to drive innovation in life sciences.

**End of Report**