Universidad Católica San Pablo

Assembly of DNA Fragments

Computer Science — Computational Molecular Biology

Harold Alejandro Villanueva Borda

---

1. Introducción

En la bioinformática, el ensamblaje de secuencias es un problema fundamental que implica la reconstrucción de una secuencia de ADN a partir de fragmentos cortos. Este proceso es crítico para secuenciaciones genómicas. En este laboratorio, el objetivo es implementar y probar un conjunto de algoritmos destinados a ensamblar segmentos de ADN, siguiendo los principios de algoritmos como el *Shortest Common Superstring (SCS)* y métodos basados en grafos de solapamientos (*overlap graphs*). Además, se busca encontrar una secuencia de consenso aproximada de longitud $l = 55$ a partir de un conjunto de fragmentos, algunos de los cuales pueden estar en su cadena complementaria-reversa.

Los objetivos específicos de este informe son:

- Implementar un algoritmo para ensamblar secuencias de consenso.
- Verificar el correcto ensamblaje, considerando secuencias reverso-complementarias.
- Visualizar y analizar el grafo de solapamientos y determinar el camino Hamiltoniano.
- Discutir la complejidad computacional de los algoritmos utilizados.

2. Desarrollo Experimental

2.1. Algoritmo de Secuencia de Consenso

Se implementó un algoritmo que toma un conjunto de secuencias y busca construir una secuencia de consenso aproximada a una longitud dada l. Este algoritmo se basa en los siguientes pasos:

1. Generar el complemento reverso de cada secuencia, de manera que todas las combinaciones posibles (secuencias originales y sus complementos) se consideren en el proceso de ensamblaje.

2. Utilizar un enfoque *greedy* para maximizar los solapamientos entre las secuencias, seleccionando la secuencia que más se solape con la actual.

3. Repetir el proceso hasta ensamblar una secuencia de consenso de la longitud deseada o aproximada, ajustando su longitud si es necesario.

2.2. Construcción del Grafo de Solapamientos

El grafo de solapamientos se construye comparando cada secuencia con el resto para encontrar el máximo solapamiento entre pares. Se define un *linkage* t que indica el tamaño mínimo de solapamiento para que dos secuencias estén conectadas en el grafo. El siguiente código fue utilizado para generar y visualizar el grafo de solapamientos:

2.3. Camino Hamiltoniano

Para encontrar el camino Hamiltoniano, se emplea un enfoque *greedy* que selecciona la secuencia con mayor solapamiento en cada paso, comenzando desde cualquier nodo del grafo de solapamientos. Este algoritmo busca maximizar el peso total de las aristas seleccionadas para construir un superstring corto que pase por todas las secuencias.

3. Análisis de complejidad

3.1. `calculateOverlap`

Esta función recibe dos secuencias de longitud m y n, respectivamente, y calcula el máximo solapamiento entre ellas.

3.1.1. Código relevante:

```
int calculateOverlap(const string& a, const string& b) {
    for (int i = min(a.length(), b.length()); i > 0; --i) {
        if (a.substr(a.length() - i) == b.substr(0, i)) {
            return i;
        }
    }
    return 0;
}
```

3.1.2. Análisis paso a paso:

- La longitud de las cadenas a y b son m y n, respectivamente.
- El bucle itera $\text{mín}(m, n)$ veces.
- En cada iteración, la operación de obtener un substring y compararlo tiene una complejidad de $O(i)$, donde i es el tamaño del substring.

La complejidad en el peor caso es:

$$O(1) + O(2) + O(3) + \cdots + O(\text{mín}(m, n)) = O\left(\frac{\text{mín}(m, n)^2}{2}\right)$$

Por lo tanto, la complejidad final es:

$$O(\text{mín}(m, n)^2)$$

3.2. `findConsensusSequence`

Esta función encuentra la secuencia de consenso utilizando todas las secuencias directas y sus complementos reversos, eligiendo la que tiene mayor solapamiento.

3.2.1. Código relevante:

```
string findConsensusSequence(const vector<string>& sequences, int
    targetLength) {
     vector<string> allSeqs = sequences;
     for (const auto& seq : sequences) {
         allSeqs.push_back(reverseComplement(seq));
     }

     auto it = max_element(allSeqs.begin(), allSeqs.end(),
                           [](const string& a, const string& b) { return
                              a.length() < b.length(); });
     consensus = *it;
     used[it - allSeqs.begin()] = true;

     while (consensus.length() < targetLength) {
         for (int i = 0; i < allSeqs.size(); ++i) {
             if (!used[i]) {
                 int overlapStart = calculateOverlap(consensus,
                    allSeqs[i]);
```

```
16                  int overlapEnd = calculateOverlap(allSeqs[i],
                        consensus);
17              }
18          }
19      }
20
21      return consensus;
22  }
```

3.2.2. Análisis paso a paso:

- Se duplican las secuencias en el vector `allSeqs`, añadiendo los complementos reversos. Esto toma $O(N \cdot m)$, donde N es el número de secuencias y m es la longitud promedio.
- La búsqueda de la secuencia más larga con `max_element` toma $O(2N)$.
- El bucle principal se ejecuta hasta que la longitud de `consensus` alcance `targetLength`. En el peor caso, esto toma $O(2N)$ iteraciones.
- En cada iteración del bucle, evaluamos todas las secuencias no utilizadas, lo que implica realizar $O(2N \cdot m^2)$ operaciones.

La complejidad total es:

$$O(N \cdot m) + O(2N \cdot (2N \cdot m^2)) = O(N^2 \cdot m^2)$$

3.3. `findHamiltonianPath`

Esta función utiliza un enfoque voraz (greedy) para encontrar un camino hamiltoniano en un grafo de secuencias, donde el peso de las aristas está dado por el solapamiento entre secuencias.

3.3.1. Código relevante:

```
1  string findHamiltonianPath(const vector<string>& sequences, const
       vector<vector<int>>& graph) {
2      int n = sequences.size();
3      vector<bool> visited(n, false);
4      string result;
5
6      int current = 0;
7      visited[current] = true;
```

```
    result = sequences[current];

    for (int step = 1; step < n; ++step) {
        int bestNext = -1;
        int bestWeight = 0;

        for (int i = 0; i < n; ++i) {
            if (!visited[i] && graph[current][i] > bestWeight) {
                bestWeight = graph[current][i];
                bestNext = i;
            }
        }

        if (bestNext == -1) break;

        result += sequences[bestNext].substr(bestWeight);
        visited[bestNext] = true;
        current = bestNext;
    }

    return result;
}
```

3.3.2. Análisis paso a paso:

- Inicialización de `visited` y `result` toma $O(n)$.
- El bucle principal se ejecuta n veces.
- En cada iteración, se busca la mejor arista, lo que toma $O(n)$.
- La actualización de la secuencia `result` toma $O(m)$ en cada iteración, donde m es la longitud promedio de las secuencias.

Por lo tanto, la complejidad total es:

$$O(n^2 \cdot m)$$

3.4. Resumen de complejidades

- `calculateOverlap`: $O(\text{mín}(m, n)^2)$

- `findConsensusSequence`: $O(N^2 \cdot m^2)$
- `findHamiltonianPath`: $O(n^2 \cdot m)$

Donde:

- N es el número de secuencias.
- m es la longitud promedio de las secuencias.

4. Resultados y Análisis

4.1. Secuencia de Consenso

El algoritmo fue ejecutado con las siguientes secuencias de entrada:

- f1: ATCCGTTGAAGCCGCGGGC
- f2: TTAACTCGAGG
- f3: TTAAGTACTGCCCG
- f4: ATCTGTGTCGGG
- f5: CGACTCCCGACACA
- f6: CACAGATCCGTTGAAGCCGCGGG
- f7: CTCGAGTTAAGTA
- f8: CGCGGGCAGTACTT

La secuencia de consenso obtenida tiene una longitud de 55 y es la siguiente:

```
AGTACTGCCCGACTCCCGACACAGATCCGTTGAAGCCGCGGGCAGTACTTAACTC
```

4.2. Visualización del Grafo de Solapamientos

El grafo de solapamientos entre las secuencias fue construido usando un valor de $t = 3$, y las siguientes conexiones fueron encontradas:

Donde:

- A: ATCCGTTGAAGCCGCGGGC
- B: CGCGGGCAGTACTT

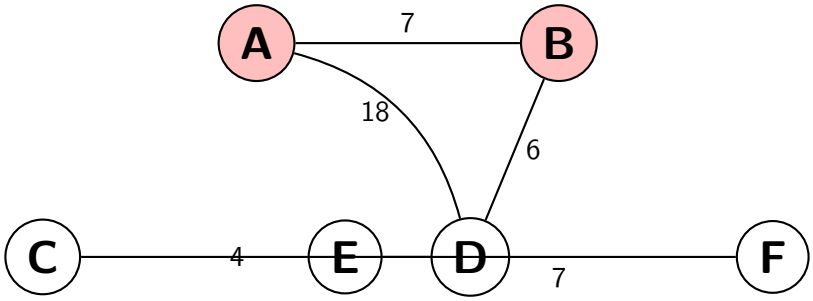

Figura 1: Grafo de overlaps de secuencias

- C: CGACTCCCGACACA
- D: CACAGATCCGTTGAAGCCGCGGG
- E: CTCGAGTTAAGTA
- F: TTAAGTACTGCCCG

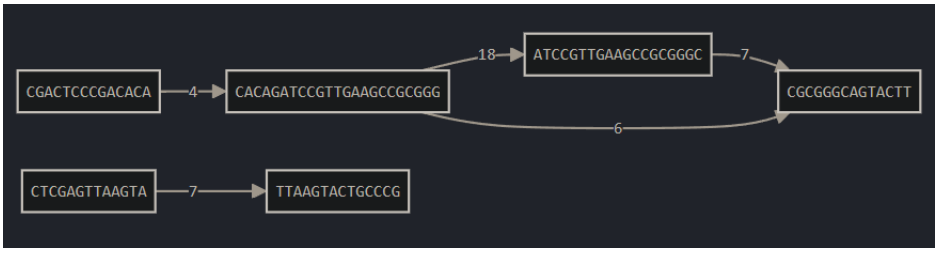

Figura 2: Grafo de solapamientos entre secuencias.

4.3. Camino Hamiltoniano

El camino Hamiltoniano encontrado, que conecta todas las secuencias en un superstring corto, es el siguiente:

```
ATCCGTTGAAGCCGCGGGCAGTACTT
```

Este camino fue generado al seleccionar iterativamente la arista de mayor solapamiento en cada paso.

4.4. Complejidad Computacional

El algoritmo para encontrar la secuencia de consenso tiene una complejidad de $O(n^2 \cdot m)$, donde n es el número de secuencias y m es la longitud promedio de las secuencias. Esto se debe a que el cálculo de los solapamientos entre cada par de secuencias toma $O(m)$ y se compara cada secuencia con todas las demás.

La construcción del grafo de solapamientos también tiene una complejidad de $O(n^2 \cdot m)$, ya que requiere comparar todas las secuencias entre sí.

El algoritmo *greedy* para encontrar el camino Hamiltoniano tiene una complejidad de $O(n^2)$, ya que en cada paso se selecciona la mejor arista entre las secuencias no visitadas.

5. Implementación en Github

El código fuente del análisis se encuentra disponible en GitHub: GitHub.

6. Conclusiones

El ensamblaje de secuencias de ADN es un problema computacionalmente desafiante, pero puede abordarse de manera efectiva utilizando algoritmos *greedy* y basados en grafos de solapamientos. En este laboratorio, se implementaron métodos eficientes para encontrar la secuencia de consenso y visualizar las conexiones entre fragmentos mediante un grafo de solapamientos. El enfoque propuesto permitió ensamblar una secuencia de consenso aproximada de longitud 55, cumpliendo con los objetivos planteados.

Sin embargo, este enfoque *greedy* no garantiza la solución óptima, ya que existen múltiples secuencias posibles que pueden generar un ensamblaje correcto.