

## Laboratorio 2 - Parte 1: Algoritmos de detección

Algoritmos de corrección de errores:

- Pruebas
  - Sin errores:

Entrada Original	Trama Codificada	Salida codificada por Viterbi
1011	11100001	1011
110	110101	110
01101	0011010100	01101

The screenshot shows a VS Code editor with a file named `receptor.py` open. The script defines a `viterbi_decode` function that takes an encoded string and returns the decoded string. The function uses a Viterbi algorithm to find the most likely path through a state transition matrix. The terminal output shows the script being executed with the input `11100001`, resulting in the decoded output `1011`.

```

24 def viterbi_decode(encoded_bits: str) -> str:
25     for bit_in in ['0', '1']:
26         next_state, expected_out = transitions[state][bit_in]
27         distance = hamming_distance(segment, expected_out)
28         total_cost = cost + distance
29         if next_state not in new_paths or total_cost < new_paths[next_state][0]:
30             new_paths[next_state] = (total_cost, path + bit_in)
31     paths = new_paths
32     # Buscar el camino con menor costo
33     best_state = min(paths, key=lambda s: paths[s][0])
  
```

Terminal Output:

```

PS C:\Users\WSI\Desktop\info\Escritorio\UG\8vo Semestre\Redes\Lab2\Lab2_Redes> .\Viterbi\emisorViterbi
Ingrese la trama binaria a codificar: 1011
Trama codificada: 11100001
PS C:\Users\WSI\Desktop\info\Escritorio\UG\8vo Semestre\Redes\Lab2\Lab2_Redes>
  
```

```
def viterbi_decode(encoded_bits: str) -> str:
    for bit_in in ['0', '1']:
        next_state, expected_out = transitions[state][bit_in]
        distance = hamming_distance(segment, expected_out)
        total_cost = cost + distance

        if next_state not in new_paths or total_cost < new_paths[next_state][0]:
            new_paths[next_state] = (total_cost, path + bit_in)

    paths = new_paths

    # Buscar el camino con menor costo
    best_state = min(paths, key=lambda s: paths[s][0])
```

Terminal output:

```
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> ./Viterbi/emisorViterbi
Ingrese la trama binaria a codificar: 1011
Trama codificada: 11100001
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> ./Viterbi/emisorViterbi
Ingrese la trama binaria a codificar: 110
Trama codificada: 110101
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes>
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 11100001
Mejor estado final: 11
Costo total (errores corregidos): 0
Bits originales decodificados: 1011
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 110101
Mejor estado final: 01
Costo total (errores corregidos): 0
Bits originales decodificados: 110
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi>
```

```
def viterbi_decode(encoded_bits: str) -> str:
    for bit_in in ['0', '1']:
        next_state, expected_out = transitions[state][bit_in]
        distance = hamming_distance(segment, expected_out)
        total_cost = cost + distance

        if next_state not in new_paths or total_cost < new_paths[next_state][0]:
            new_paths[next_state] = (total_cost, path + bit_in)

    paths = new_paths

    # Buscar el camino con menor costo
    best_state = min(paths, key=lambda s: paths[s][0])
```

Terminal output:

```
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> ./Viterbi/emisorViterbi
Ingrese la trama binaria a codificar: 1011
Trama codificada: 11100001
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> ./Viterbi/emisorViterbi
Ingrese la trama binaria a codificar: 110
Trama codificada: 110101
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> ./Viterbi/emisorViterbi
Ingrese la trama binaria a codificar: 01101
Trama codificada: 0011010100
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes>
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 11100001
Mejor estado final: 11
Costo total (errores corregidos): 0
Bits originales decodificados: 1011
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 110101
Mejor estado final: 01
Costo total (errores corregidos): 0
Bits originales decodificados: 110
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 0011010100
Mejor estado final: 10
Costo total (errores corregidos): 0
Bits originales decodificados: 01101
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi>
```

- 1 Error

Entrada original	Codificado	Bit alterado	Trama con error	Salida decodificada	Costo
1011	11100001	bit 3 (1→0)	<b>11000001</b>	1011	1
110	110101	bit 3 (0→1)	<b>111101</b>	110	1
01101	0011010100	bit 3 (1→0)	0001010100	01101	1

- Mismos 3 entradas en el emisor, pero alteradas para input al receptor

```

def viterbi_decode(encoded_bits: str) -> str:
    for bit_in in ['0', '1']:
        next_state, expected_out = transitions[state][bit_in]
        distance = hamming_distance(segment, expected_out)
        total_cost = cost + distance

        if next_state not in new_paths or total_cost < new_paths[next_state][0]:
            new_paths[next_state] = (total_cost, path + bit_in)

    paths = new_paths

    # Buscar el camino con menor costo
    best_state = min(paths, key=lambda s: paths[s][0])

```

```

PS C:\Users\MSI\Desktop\info\Escritorio\UMG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> .\Viterbi\emisorViterbi
Ingrese la trama binaria a codificar: 1011
Trama codificada: 11100001
PS C:\Users\MSI\Desktop\info\Escritorio\UMG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> .\Viterbi\emisorViterbi
Ingrese la trama binaria a codificar: 110
Trama codificada: 110101
PS C:\Users\MSI\Desktop\info\Escritorio\UMG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> .\Viterbi\emisorViterbi
Ingrese la trama binaria a codificar: 01101
Trama codificada: 0011010100
PS C:\Users\MSI\Desktop\info\Escritorio\UMG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes>

```

```

Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 11000001
Mejor estado final: 11
Costo total (errores corregidos): 1
Bits originales decodificados: 1011
PS C:\Users\MSI\Desktop\info\Escritorio\UMG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 111101
Mejor estado final: 01
Costo total (errores corregidos): 1
Bits originales decodificados: 110
PS C:\Users\MSI\Desktop\info\Escritorio\UMG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi> python receptor.py
Ingrese la trama codificada de 2 bits por símbolo: 0001010100
Mejor estado final: 10
Costo total (errores corregidos): 1
Bits originales decodificados: 01101
PS C:\Users\MSI\Desktop\info\Escritorio\UMG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\Viterbi>

```

- Preguntas:
  - ¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no? En caso afirmativo, con su implementación

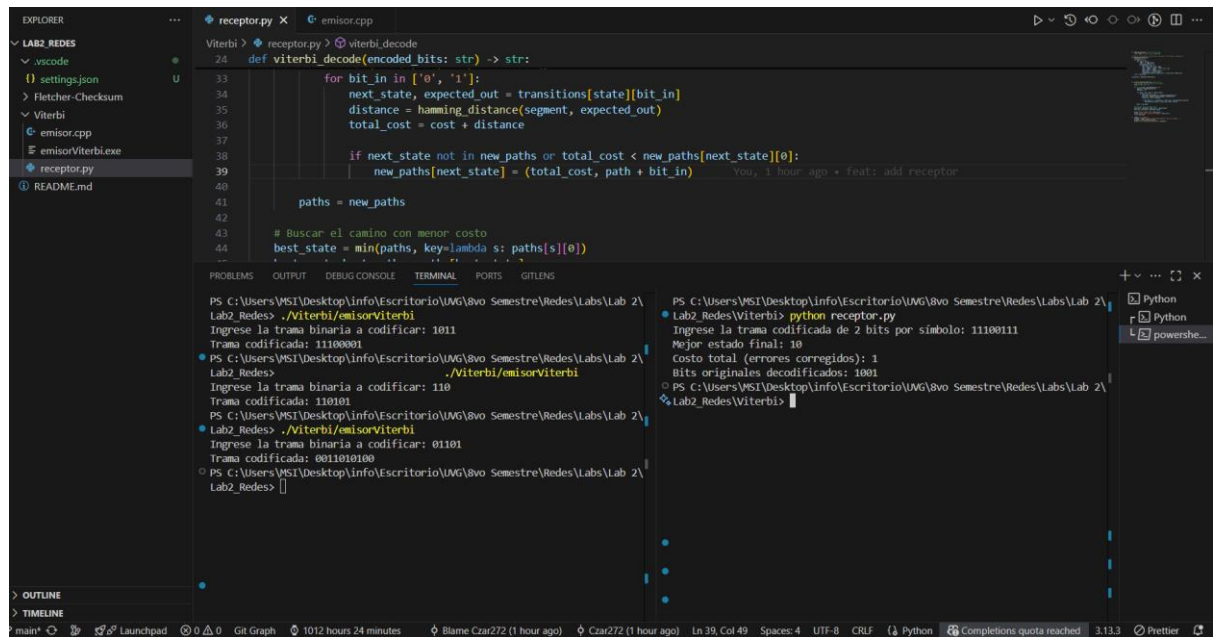
Sí, es posible cambiar algunos bits de forma que el algoritmo de Viterbi no note el error. Esto pasa porque este algoritmo no está hecho para comprobar si el mensaje está completamente bien, sino para tratar de adivinar cuál era el mensaje original más probable basándose en lo que recibió. Si los bits se cambian de cierta manera y el resultado aún "parece correcto" dentro de lo que el algoritmo espera, puede decodificar un mensaje equivocado sin darse cuenta. Esto suele ocurrir cuando el error no se nota fácilmente porque se parece mucho al mensaje original.

Por ejemplo:

Entrada original = 1011;

Entrada codificada = 11100001

Trama codificada errónea: 11100111



The screenshot shows a VS Code editor with a file explorer on the left containing files like settings.json, Fletcher-Checksum, Viterbi, emisor.cpp, emisorViterbi.exe, receptor.py, and README.md. The main editor displays the Python code for the Viterbi decoder in receptor.py. The code defines a viterbi\_decode function that takes encoded bits as a string and returns the decoded string. It uses a transition table and a path table to find the most likely original message. The terminal at the bottom shows the execution of the program, where the user enters the encoded message '11100111' and the program outputs the decoded message '1001', indicating a correction of one error.

```
def viterbi_decode(encoded_bits: str) -> str:
    for bit in ['0', '1']:
        next_state, expected_out = transitions[state][bit_in]
        distance = hamming_distance(segment, expected_out)
        total_cost = cost + distance

        if next_state not in new_paths or total_cost < new_paths[next_state][0]:
            new_paths[next_state] = (total_cost, path + bit_in)

    paths = new_paths

    # Buscar el camino con menor costo
    best_state = min(paths, key=lambda s: paths[s][0])
```

```
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> .\Viterbi\emisorViterbi
Ingrese la trama binaria a codificar: 1011
Trama codificada: 11100001
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> .\Viterbi\emisorViterbi
Ingrese la trama binaria a codificar: 110
Trama codificada: 110101
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes> .\Viterbi\emisorViterbi
Ingrese la trama binaria a codificar: 01101
Trama codificada: 0011010100
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes>
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\> python receptor.py
Ingrese la trama codificada de 2 bits por simbolo: 11100111
Mejor estado final: 10
Costo total (errores corregidos): 1
Bits originales decodificados: 1001
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\Lab2_Redes\>
```

El receptor se equivoca y devuelve:

Costo total (errores corregidos): 1

Bits originales decodificados: 1001

- En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos?

Durante las pruebas que realicé, el algoritmo de Viterbi demostró ser muy poderoso cuando se trata de corregir errores, incluso cuando se altera más de un bit. A diferencia de Fletcher o CRC, Viterbi no solo detecta que hubo un error, sino que intenta recuperar el mensaje original, lo cual lo hace especialmente útil en entornos donde los errores de transmisión son frecuentes o inevitables. Una de sus grandes ventajas es justamente esa: la capacidad de corrección. En los casos donde se introdujo un solo error, o incluso en algunos con dos errores, el receptor logró reconstruir correctamente el mensaje original. Además, no requiere reenvío de información, lo cual mejora la eficiencia cuando las condiciones del canal no son confiables.

Sin embargo, también observé varias desventajas. La más evidente es que la implementación es más compleja que la de Fletcher o CRC. Hay que codificar un autómata, gestionar estados, calcular caminos y mantener un registro del “costo” de cada trayectoria, lo cual requiere más procesamiento y más memoria

Algoritmos de detección de errores:

- Pruebas
  - Sin errores:

```

39 }
40
41 int main() {
42     string received;
43     cout << "Ingrese la trama recibida (datos + checksum): ";
44     cin >> received;
45
46     if (received.length() < 16) {
47         cout << "Error: Trama demasiado corta para contener un checksum válido." << endl;
48         return 1;
49     }
50
51     string databits = received.substr(0, received.length() - 16);
52     pair<int, int> expected = extractChecksum(received);
  
```

```

PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> python emisor.py
Ingrese la trama en binario: 10110011
Trama emitida con checksum: 101100111011001110
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> python receptor.py
Ingrese la trama en binario: 11110000
Trama emitida con checksum: 111100001111000011
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> python emisor.py
Ingrese la trama en binario: 1100101010010110
Trama emitida con checksum: 11001010100101100110001001100
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum>
  
```

```

PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> g++ receptor.cpp -o receptorFletcher
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> ./receptorFletcher
Ingrese la trama recibida (datos + checksum): 101100111011001110110011
Trama valida. No se detectaron errores.
Trama original: 10110011
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> ./receptorFletcher
Ingrese la trama recibida (datos + checksum): 111100001111000011110000
Trama valida. No se detectaron errores.
Trama original: 11110000
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> ./receptorFletcher
Ingrese la trama recibida (datos + checksum): 110010101001011001100001001011
Trama valida. No se detectaron errores.
Trama original: 1100101010010110
PS C:\Users\MSI\Desktop\info\Escritorio\UAG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum>
  
```

Trama Original	Trama Codificada	Salida Receptor
10110011	101100111011001110 110011	Aceptada sin errores
11110000	111100001111000011 110000	Aceptada sin errores
1100101010010110	110010101001011001 10000100101100	Aceptada sin errores

- 1 Error:

Trama Original	Trama Codificada	Bit alterado	Trama con error	Salida Receptor
10110011	1011001110 1100111011 0011	Bit 10 (0 -> 1)	1011001111 1100111011 0011	Checksum esperado: s1 = 243, s2 = 179

				Checksum calculado: s1 = 179, s2 = 179
11110000	1111000011 1100001111 0000	Bit 7 (0 -> 1)	1111001011 1100001111 0000	Checksum esperado: s1 = 240, s2 = 240 Checksum calculado: s1 = 242, s2 = 242
1100101010 010110	1100101010 0101100110 0001001011 00	Bit 12 (0 -> 1)	1100101010 0001101100 1000001011 00	Checksum esperado: s1 = 200, s2 = 44 Checksum calculado: s1 = 81, s2 = 28

- Preguntas:

- ¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no? En caso afirmativo, con su implementación

Sí, es posible modificar ciertos bits de forma que el algoritmo de Fletcher-Checksum no detecte el error. Esto se debe a que Fletcher utiliza sumas acumulativas (s1 y s2) para calcular un valor de verificación, pero no tiene la capacidad de identificar la posición exacta de los errores ni detectar todos los patrones posibles.

Por ejemplo, si se cambian dos bits en posiciones distintas, y esos cambios están cuidadosamente balanceados (uno aumenta y el otro disminuye el valor total), el nuevo checksum puede coincidir con el original. En ese caso, el receptor asumiría que la trama está bien, aunque realmente se hayan corrompido datos. Esto sucede porque el algoritmo solo valida la suma total, no el contenido real ni su estructura interna.

Por ejemplo:

Trama original: 111100001111000011110000

Trama Modificada: 111100001111000011110011 -> últimos dos bits se cambiaron

Por ende, el algoritmo se equivoca y no muestra el checksum correcto esperado

```
Fletcher-Checksum > G receptor.cpp > main()
33 pair<int, int> extractChecksum(string bits) {
39 }
40
41 int main() {
42     string received;
43     cout << "Ingrese la trama recibida (datos + checksum): ";
44     cin >> received;
45
46     if (received.length() < 16) {
47         cout << "Error: Trama demasiado corta para contener un checksum válido." << endl;
48         return 1;
49     }
50
51     string dataBits = received.substr(0, received.length() - 16);
52     pair<int, int> expected = extractChecksum(received);

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GIT LENS
Ingrese la trama en binario: 11110000
Trama emitida con checksum: 111100001111000011110000
PS C:\Users\MSI\Desktop\info\Escritorio\UVG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> python emisor.py
Ingrese la trama en binario: 1100101010010110
Trama emitida con checksum: 11001010100101100110000100101100
PS C:\Users\MSI\Desktop\info\Escritorio\UVG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> python emisor.py
Ingrese la trama en binario: 1100101010010110
Trama emitida con checksum: 11001010100101100110000100101100
PS C:\Users\MSI\Desktop\info\Escritorio\UVG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum>

PS C:\Users\MSI\Desktop\info\Escritorio\UVG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum> ./receptorFletcher
Ingrese la trama recibida (datos + checksum): 111100001111000011110011
Error detectado. La trama se descarta.
Checksum esperado: s1 = 240, s2 = 243
Checksum calculado: s1 = 240, s2 = 240
PS C:\Users\MSI\Desktop\info\Escritorio\UVG\8vo Semestre\Redes\Labs\Lab 2\La
b2_Redes\Fletcher-Checksum>
```

- En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos?

Una de las principales ventajas de Fletcher es que es muy sencillo de implementar. Solo requiere sumar bytes y calcular dos sumas acumulativas (s1 y s2), lo cual lo hace rápido y eficiente, incluso en dispositivos con pocos recursos. Además, su redundancia es baja (solo añade 16 bits al final del mensaje), lo que significa que el overhead que genera es mínimo comparado con otros métodos más complejos como Viterbi.

Sin embargo, también noté varias desventajas. La más importante es que no puede corregir errores, solo detectarlos. Y ni siquiera detecta todos: en mis pruebas, si se cambian dos bits de forma "balanceada", el checksum no cambia y el receptor no detecta el error. En ese sentido, es menos robusto que algoritmos como CRC o Viterbi, que tienen mejor cobertura frente a errores múltiples o más estructurados.