Resolución de Tarea 5 - Grafos (Fecha: 3 de Noviembre de 2025)

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5651 - Diseño de Algoritmos I
Septiembre - Diciembre 2025
Estudiante: Junior Miguel Lara Torres (17-10303)

Tarea 5 (9 puntos)

Indice

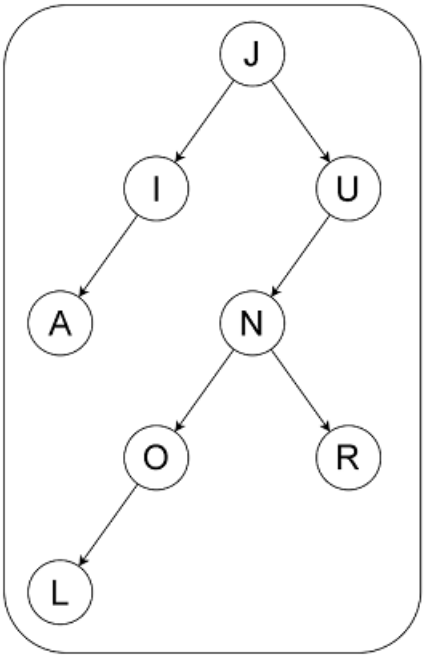
Pregunta 1

Mi nombre y apellido es: **Junior Lara**

- Cadena completa en minúsculas: “juniorlara”.
- Eliminación de repeticiones (conservando el orden de aparición): **j, u, n, i, o, r, l, a**
- Cadena de caracteres resultante (S): “juniorla” (n=8 caracteres).

Parte (a)

El árbol binario de búsqueda se muestra a continuación

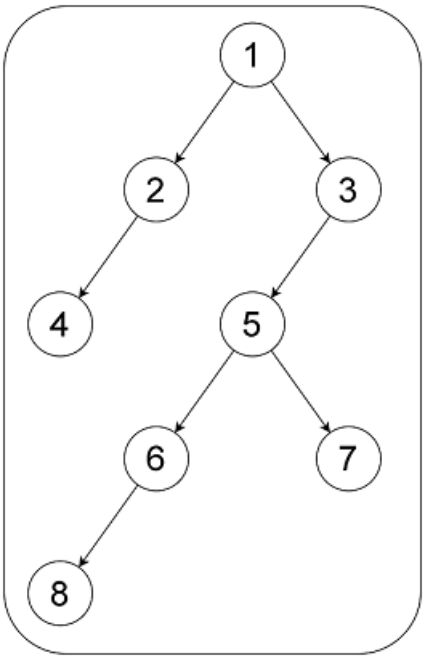


Parte (b)

El recorrido preorder es el siguiente: **j, i, a, u, n, o, l, r**

Parte (c)

Realizando la enumeración por niveles



- El recorrido de Euler es el siguiente: **1, 2, 3, 2, 1, 2, 3, 4, 5, 5, 3, 4, 3, 2, 1**

- A nivel de caracteres tenemos: **j, i, a, i, j, u, n, o, l, o, n, r, n, u, j**

Parte (d)

La cadena es "juniorla". Los nodos a considerar son 'l' y 'a'.

El Ancestro Común Más Bajo (LCA) de dos nodos u y v en un árbol es el nodo con la profundidad mínima (menor nivel) en el sub-arreglo del Recorrido de Euler (L) comprendido entre la primera aparición de u y la primera aparición de v.

Esto reduce el problema de LCA a un problema de Consulta de Rango Mínimo. (Nota: la justificación es en base 1-indexación)

Tenemos los arreglos

- $EulerLevel = [1, 2, 3, 2, 1, 2, 3, 4, 5, 5, 3, 4, 3, 2, 1]$
- $EulerNodos = [j, i, a, i, j, u, n, o, l, o, n, r, n, u, j]$

1. Identificar las primeras ocurrencias:
   - Posición primera ocurrencia de l: 9.
   - Posición primera ocurrencia de a: 3.
2. Determinar los niveles en el rango [3..9]:
   - $EulerLevel[3..9] = [3, 2, 1, 2, 3, 4, 5]$.
   - $EulerNodos[3..9] = [a, i, j, u, n, o, l]$.
3. Encontrar el Mínimo Nivel:
   - El valor mínimo en la secuencia de niveles [3..9] es 1, el cual ocurre en el índice 5.
4. Identificar el LCA:
   - El nodo en la secuencia E en el índice 5 es j. Por lo tanto, el Ancestro Común Más Bajo entre 'l' y 'a' es j.

Pregunta 2

El problema pide determinar si es posible orientar las aristas de un grafo no dirigido de encuentros $G = (A, E)$ (donde los agentes A son los nodos y los encuentros E son las aristas) de tal forma que toda la información pueda alcanzar a todos los agentes. Esto equivale a encontrar una **orientación fuertemente conexa** (G'), es decir, un grafo dirigido donde existe un camino de u a v y de v a u para todo par de nodos u, v.

El **Teorema de Robbins** establece que un grafo no dirigido G puede tener una orientación fuertemente conexa si, y solo si, G es **conexo** y **no contiene puentes**.

Si el grafo G es conexo y la cantidad de puentes es nula, el Teorema de Robbins se cumple. En este caso, la orientación fuertemente conexa resultante garantiza un **plan de propagación de información** tal que, al asignar roles de "hablante" y "oyente" (orientación de las aristas), la información generada por cualquier agente puede alcanzar a todos los demás.

Algoritmo

Para resolver el problema en el tiempo lineal requerido, $O(|A| + |E|)$, utilizaremos una única pasada de Búsqueda en Profundidad (DFS) modificada para cumplir dos objetivos:

1. **Verificación de Conexidad:** Un DFS iniciado desde un nodo arbitrario explorará el grafo completo si y solo si es conexo. Si no todos los nodos son visitados, el grafo es disconexo.

2. **Detección de Puentes:** Utilizaremos la extensión del DFS (similar al algoritmo de Tarjan) que calcula los tiempos de descubrimiento (`dfs_num`) y el valor más bajo alcanzable (`dfs_low`) para cada nodo. Una arista de árbol (u, v) es un **puente** si, al salir del subárbol de v, el punto más alto que se puede alcanzar es el propio v o un nodo que está más abajo, es decir, si **dfs_low**$(v) >$ **dfs_num**(u).

Si el algoritmo verifica que el grafo es conexo y encuentra que el número total de puentes es cero, entonces existe un plan de propagación de información.

Implementación en C++

El archivo funcional y documentado, con algunos ejemplos de prueba se encuentra en agents.cpp.

```cpp
// Constantes y estructuras auxiliares para la detección de puentes (Bridges)
typedef vector<vector<int>> AdjList;

// Variables globales para el DFS
int timer_counter;
vector<int> dfs_num;  // Tiempo de descubrimiento (prenum)
vector<int> dfs_low;  // Menor prenum alcanzable desde el subárbol
vector<bool> visited; // Para el chequeo de conectividad
int total_vis;        // Total visitados en DFS.
int num_bridges;

void find_bridges(int u, int p, const AdjList& G) {
    visited[u] = true;
    total_vis++;
    dfs_num[u] = dfs_low[u] = timer_counter++;

    for (int v : G[u]) {
        if (v == p) continue; // Ignorar la arista de vuelta al padre

        if (dfs_num[v] != -1) {
            // Caso 1: Vértice 'v' ya visitado (back edge)
            // Actualizamos dfs_low[u] con el tiempo de descubrimiento (dfs_num) de 'v'
            dfs_low[u] = min(dfs_low[u], dfs_num[v]);
        } else {
            // Caso 2: Vértice 'v' no visitado (tree edge)
            find_bridges(v, u, G);

            // Después de la llamada recursiva, actualizamos dfs_low[u]
            // con el valor más bajo alcanzado por 'v'
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);

            // Detección de Puente: si el subárbol de 'v' no puede alcanzar
            // a un ancestro estricto de 'u' (o 'u' mismo)
            if (dfs_low[v] > dfs_num[u]) {
                // Si dfs_low[v] > dfs_num[u], la arista (u, v) es un puente
                num_bridges++;
            }
        }
    }
}

bool isStronglyConnectable(int N, const vector<pair<int, int>>& E) {
    if (N <= 1) return true; // Si hay 0 o 1 agente, trivialmente es SC.
```

```
    // 1. Construcción del Grafo (Lista de Adyacencia)
    AdjList G(N);
    for (const auto& edge : E) {
        G[edge.first].push_back(edge.second);
        G[edge.second].push_back(edge.first);
    }

    // 2. Inicialización de estructuras auxiliares
    timer_counter = 0;
    num_bridges = 0;
    total_vis = 0;
    // -1 indica UNVISITED, usado para inicializar dfs_num y asegurar el chequeo de conectividad
    dfs_num.assign(N, -1);
    dfs_low.assign(N, 0);
    visited.assign(N, false);

    // 3. Ejecutar DFS modificado (empezamos desde el nodo 0)
    find_bridges(0, -1, G);

    // 4. Verificación de Conexidad y Cero Puentes (Biconexidad de Aristas)
    // Por el Teorema de Robbins, si es conexo y no tiene puentes, admite orientación SC.
    if (num_bridges == 0 && total_vis == N) {
        return true;
    } else {
        return false;
    }
}
```

Pregunta 3

Para el problema planteado se tiene que Jugador 1 (MAX, usa '-') y Jugador 2 (MIN, usa '|'). Para resolverlo, aplicamos la técnica Minimax.

Una **configuración (estado)** W debe contener toda la información necesaria para determinar los movimientos legales y el resultado del juego. Dado que el tablero es de 3×3 (9 casillas) y existe una restricción sobre la jugada anterior, definimos el estado de la siguiente manera:

1. **Tablero:** Una matriz 3×3 donde cada celda $c_{i,j}$ contiene el símbolo del Jugador 1 (-), el símbolo del Jugador 2 (|), ambos (+) o ninguno (vacío). Para modelarlo internamente, cada celda se representa como un par booleano: (Hay $-$, Hay $|$).

Hay '$-$'	Hay '$\|$'	Celda
$False$	$False$	
$False$	$True$	$\|$
$True$	$False$	$-$
$True$	$True$	$+$

2. La coordenada de la casilla donde se jugó en el turno inmediatamente anterior se define como **Última Posición Jugada** ($último\_mov$). Esto es esencial para aplicar la regla de la restricción.

3. Una transición es un movimiento legal que lleva de una configuración W a una configuración sucesora X. El conjunto de sucesores de W, $Sucesores(W, \text{Jugador})$, se define como todas las jugadas posibles que cumplen:

- El jugador de turno elige una casilla $c_{i,j}$ que sea diferente a $\acute{u}ltimo\_mov(W)$, que esté vacía o contenga sólo el símbolo del contrincante.
- Si el jugador es MAX ('-'), se añade '-' a $c_{i,j}$. Si es MIN ('|'), se añade '|' a $c_{i,j}$. Si la celda ya contenía el símbolo del oponente, ahora contendrá un '+'.
- La posición $c_{i,j}$ seleccionada pasa a ser la nueva $\acute{u}ltimo\_mov$ en X.

La función `eval(w)` asigna un valor a una configuración w. Esta función se utiliza para **configuraciones terminales** (aquellas donde se ha alcanzado la profundidad máxima o el juego ha terminado).

El valor de la configuración w se calcula como:

- **Ganador MAX:** Si MAX forma tres '+' en línea (fila, columna o diagonal): $eval(w) = 1000$.
- **Ganador MIN:** Si MIN forma tres '+' en línea: $eval(w) = -1000$.
- **Empate:** Si el tablero está lleno y no hay ganador: $eval(w) = 0$.

El valor debe ser **más positiva si el jugador 1 va ganando** y **más negativa si el jugador 2 va ganando**.

Se tienen las siguientes funciones auxiliares:

```
función check_terminal(w: configuración) -> booleano
    // Retorna 'cierto' si hay 3 '+' en línea o si no quedan movimientos legales.
    si hay_3_mas_en_linea(w) retornar cierto
    si sucesores(w, jugador_actual) es vacio retornar cierto
    retornar falso

función eval(w: configuración) -> entero
    // Evalúa una configuración terminal.
    si hay_3_mas_en_linea(w, '-') retornar 1000 // MAX gana
    si hay_3_mas_en_linea(w, '|') retornar -1000 // MIN gana
    retornar 0 // Empate
```

Las funciones `primero` y `segundo` son recursivas. La Poda Alfa-Beta utiliza dos valores: α (el valor mínimo que tiene asegurado el jugador que maximiza) y β (el valor máximo que tiene asegurado el jugador que minimiza). Si $\beta \leq \alpha$, se poda la rama.

```
función primero(w: configuración, alfa: entero, beta: entero) -> entero
    // Jugador MAX (Maximiza)
    si check_terminal(w)
        retornar eval(w)

    mejor_valor <- -infinito // Representando el valor más bajo posible

    para cada x en sucesores(w, '-')
        valor <- segundo(x, alfa, beta)
        mejor_valor <- max(mejor_valor, valor)
        alfa <- max(alfa, mejor_valor)

        si beta <= alfa
            salir del ciclo // Poda Beta

    retornar mejor_valor

función segundo(w: configuración, alfa: entero, beta: entero) -> entero
    // Jugador MIN (Minimiza)
    si check_terminal(w)
        retornar eval(w)

    mejor_valor <- infinito // Representando el valor más alto posible
```

```
    para cada x en sucesores(w, '|')
        valor <- primero(x, alfa, beta)
        mejor_valor <- min(mejor_valor, valor)
        beta <- min(beta, mejor_valor)

        si beta <= alfa
            salir del ciclo // Poda Alfa

    retornar mejor_valor

// Función principal para iniciar la búsqueda
función resolver_minimax(config_inicial: configuración) -> entero
    alfa_inicial <- -infinito
    beta_inicial <- infinito
    retornar primero(config_inicial, alfa_inicial, beta_inicial)
```

Finalmente, para determinar si hay una estrategia ganadora, el algoritmo `resolver_minimax` se invoca con la configuración inicial y:

- Si el resultado es **positivo**, el Jugador 1 (MAX) tiene una estrategia ganadora.
- Si el resultado es **negativo**, el Jugador 2 (MIN) tiene una estrategia ganadora.
- Si el resultado es **cero**, el resultado óptimo es un empate.

Pregunta 4

Definimos un **grafo de conflicto** $G = (C, E)$, donde:

- E contiene una arista (x, y) si $x, y \in C$ y su suma $x + y$ es un número primo.

Un grafo es **bipartito** si sus vértices pueden ser divididos en dos conjuntos disjuntos A y B tales que cada arista conecta un vértice de A con uno de B.

Se define entonces que:

1. La suma de dos números, $x + y$, es un número primo. Dado que 2 es el único número primo par, y que los números en C son distintos (lo que implica que $x + y \neq 2$ si $x \neq y \wedge x, y > 0$), cualquier suma prima $x + y = P$ debe ser un número primo **impar** ($P > 2$).
2. Para que la suma de dos enteros sea impar, uno debe ser par y el otro debe ser impar.
3. Si definimos A como el conjunto de números impares en C, y B como el conjunto de números pares en C, toda arista $(x, y) \in E$ necesariamente conecta un nodo en A con un nodo en B.

Concluimos que el grafo de conflicto G es **bipartito**.

El objetivo es encontrar el subconjunto máximo de vértices $C'' \subseteq C$ tal que no haya aristas entre ningún par de vértices en C''. Este subconjunto es conocido como el **Conjunto Independiente Máximo (MIS)**. El número mínimo de elementos a eliminar es, por definición, $|C| - |MIS|$.

El **Teorema de König** establece una equivalencia fundamental en grafos bipartitos: **la cardinalidad del emparejamiento máximo (MCBM) es igual a la cardinalidad del cubrimiento de vértices mínimo (MVC)**.

$$|MCBM| = |MVC|$$

Además, en cualquier grafo, el Conjunto Independiente Máximo ($|MIS|$) y el Cubrimiento de Vértices Mínimo ($|MVC|$) se relacionan por:

$$|MIS| = |V| - |MVC|$$

Sustituyendo el Teorema de König en esta relación, para nuestro grafo bipartito G:

$$|MIS| = |C| - |MCBM|$$

Dado que buscamos el **mínimo número de números a eliminar** ($|R|$), y $|R| = |C| - |MIS|$, entonces:

$$|R| = |C| - (|C| - |MCBM|) = |MCBM|$$

Por lo tanto, el problema se reduce a calcular la cardinalidad del **Emparejamiento Bipartito Máximo (MCBM)** en el grafo de conflicto G.

Sin embargo, hay que considerar los siguientes puntos

- Si el número 1 está presente en el conjunto inicial C y la restricción exige que no existan x, y tales que $x + y$ sea primo (permitiendo $x = y$), la suma $1 + 1 = 2$ constituye un conflicto que debe ser eliminado.
- Este conflicto ocurre dentro del conjunto de números impares A, rompiendo la naturaleza bipartita simple del grafo para el conjunto original C.
- Solución: Para preservar la estructura bipartita G, el número 1 debe ser EXCLUIDO obligatoriamente si está presente. Se incrementa el conteo de eliminaciones mínimas en 1, y se resuelve el problema sobre el conjunto restante $C' = C/\{1\}$. Mediante esta corrección, el algoritmo total calcula el número mínimo de eliminaciones como:

  $$\text{Eliminaciones Minimas} = (1 \text{ si } 1 \in C \text{ si no } 0) + \textbf{MBCM(G')}$$

  donde G' es el grafo bipartito generado por los elementos en $C/\{1\}$.

Algoritmo

El algoritmo consiste en dos fases principales:

1. Construcción del grafo
2. Cálculo del MCBM.

Fase 1: Construcción del Grafo

1. **Precomputación de Primos:** Se debe determinar si las sumas $x+y$ son primas. El problema indica que se puede suponer que consultar si un número es primo $O(1)$, sin embargo como sabemos que el máximo numero a consultar es $C_{max} + C_{max} = 2 * C_{max} = M$, utilizamos Criba de Eratóstenes para encontrar todos los números primos hasta un límite dado, en nuestro caso M, en tiempo $O(M \cdot log(log(M)))$. De esta forma ahora tenemos consulta de un número primo en O(1) .
2. **Construcción de Aristas:** Iteramos sobre todos los pares (x, y) donde $x \in A$ (impares) y $y \in B$ (pares). El número total de pares es n^2.
   - Si $x + y$ es primo, agregamos la arista (x, y) a E.
   - El tiempo de construcción es $O(n^2) \times O(1) = O(n^2)$.

Fase 2: Cálculo del Emparejamiento Bipartito Máximo (MCBM)

La cardinalidad del MCBM la calculamos con el algoritmo de **Hopcroft-Karp** para una complejidad de tiempo de $O(Edges \cdot \sqrt{Nodos})$.

Sustituyendo $Edges = n^2$ y $Nodos = |C| = n$ tenemos $O(n^2 \cdot \sqrt{n})$

Complejidad

Tenemos:

1. Criba de Eratóstenes en $O(M \cdot log(log(M)))$
2. Construcción de grafo en $O(n^2)$

3. Hopcroft-Karp en $O(n^2 \cdot \sqrt{n})$

Nos queda entonces $O(n^2 \cdot \sqrt{n} + M \cdot log(log(M)))$ cuyo tiempo es el más fiel a la implementación real. Por otro lado, tomando la indicación del problema en cuanto a saber si un número es primo o no es $O(1)$, entonces tenemos $O(n^2 \cdot \sqrt{n})$ finalmente.

Implementación en C++

El archivo funcional y documentado, con algunos ejemplos de prueba se encuentra en min_removals.cpp.

```
vector<int> sieve_array; // Array 0/1 para la criba (1 = Primo)

void sieve(int max_val) {
    if (max_val < 2)
        max_val = 2;

    sieve_array.assign(max_val + 1, 1);
    sieve_array[0] = sieve_array[1] = 0; // 0 y 1 no son primos

    for (int p = 2; p * p <= max_val; p++) {
        if (sieve_array[p] == 1) {
            for (int i = p * p; i <= max_val; i += p)
                sieve_array[i] = 0; // Marca los múltiplos como no primos
        }
    }
}

bool is_prime(int v) {
    if (v < 0 || v >= (int)sieve_array.size())
        return false;
    return sieve_array[v] == 1;
}

namespace HopcroftKarp {
    int V_L, V_R;             // Tamaños de los conjuntos L y R
    vector<vector<int>> Adj; // Lista de adyacencia del grafo bipartito
    vector<int> match_R;     // match_R[v] = el nodo en L emparejado con v en R
    vector<int> match_L;     // match_L[u] = el nodo en R emparejado con u en L
    vector<int> dist;        // Distancias usadas en BFS para caminos de aumento más cortos

    // Constantes para distancias
    const int NIL = 0;
    const int INF = 1000000000;

    // BFS: Fase 1 de Hopcroft-Karp. Encuentra caminos de aumento más cortos.
    bool bfs() {
        vector<int> Q;
        for (int u = 1; u <= V_L; u++) {
            if (match_L[u] == NIL) {
                dist[u] = 0;
                Q.push_back(u);
            } else
                dist[u] = INF;
        }
        dist[NIL] = INF;
```

```
    int head = 0;
    while (head < Q.size()) {
        int u = Q[head++];
        if (dist[u] < dist[NIL]) {
            for (int v : Adj[u]) {
                if (dist[match_R[v]] == INF) {
                    dist[match_R[v]] = dist[u] + 1;
                    Q.push_back(match_R[v]);
                }
            }
        }
    }
    // Retorna true si se encontró un camino de aumento (dist[NIL] != INF)
    return dist[NIL] != INF;
}

// DFS: Fase 2 de Hopcroft-Karp. Maximiza el número de caminos de aumento disjuntos.
bool dfs(int u) {
    if (u != NIL) {
        for (int v : Adj[u]) {
            // Buscamos un camino de aumento más corto
            if (dist[match_R[v]] == dist[u] + 1) {
                if (dfs(match_R[v])) {
                    // Invertimos el camino (toggle)
                    match_R[v] = u;
                    match_L[u] = v;
                    return true;
                }
            }
        }
        dist[u] = INF;
        return false;
    }
    return true;
}

// Función principal que calcula el MCBM
int max_matching(int n_L, int n_R, const vector<vector<int>> &graph_adj) {
    V_L = n_L;
    V_R = n_R;
    Adj = graph_adj;

    // Inicialización de estructuras
    match_L.assign(V_L + 1, NIL);
    match_R.assign(V_R + 1, NIL);
    dist.assign(V_L + 1, INF);

    int result = 0;
    // Repetir mientras existan caminos de aumento
    while (bfs()) {
        for (int u = 1; u <= V_L; u++) {
            // Si u es libre e inicializa un camino de aumento en la capa más corta
            if (match_L[u] == NIL && dfs(u))
```

```
                    result++;
                }
            }
            return result;
        }
    }

    int min_removals(const vector<int> &C) {
        int minimum_removals = 0;

        if (C.empty())
            return minimum_removals;

        // Paso 1a: Encontrar el máximo valor
        int c_max = *max_element(C.begin(), C.end());

        // Paso 1b: Criba de Eratóstenes hasta 2 * Cmax
        // No usamos un límite fijo: creamos la criba hasta 2 * c_max (como mínimo 2)
        int limit = max(2, 2 * c_max);
        sieve(limit);

        // Paso 2a: Separar C en conjuntos L (impares) y R (pares)
        vector<int> L_vals, R_vals;
        vector<int> map_to_index(2 * c_max + 1, 0); // Mapea valor a índice en L/R
        for (int x : C) {
            if (x % 2 != 0) {
                if (x == 1)             // Excluimos al 1 si está presente
                    minimum_removals++; // Asegurarse de contar el 1 si está presente
                else
                    L_vals.push_back(x);
            }
            else
                R_vals.push_back(x);
        }

        // Si la suma es prima (mayor que 2), debe ser impar, por lo tanto (Impar + Par).
        // El único primo par es 2. Si x+y=2, x=1, y=1, pero los elementos en C son distintos.
        // Si C contiene 1, 1 es impar. Si C contiene 2, 2 es par. 1+2=3 (Primo).
        // La bipartición en par/impar es fundamental.

        int n_L = L_vals.size();
        int n_R = R_vals.size();

        // Mapeamos los valores de L y R a índices 1-basados para Hopcroft-Karp
        // L: 1..n_L, R: 1..n_R. NIL (0) es para nodos no emparejados.

        vector<vector<int>> adj_hk(n_L + 1); // Lista de adyacencia solo para el conjunto L

        // Paso 2b: Construir aristas
        for (int i = 0; i < n_L; ++i) {
            for (int j = 0; j < n_R; ++j) {
                int sum = L_vals[i] + R_vals[j];
                // Si la suma es prima, añadimos una arista entre el nodo i+1 (en L)
                // y el nodo j+1 (en R)
```

```
            if (sum <= limit && is_prime(sum))
                adj_hk[i + 1].push_back(j + 1);
        }
    }

    // Paso 3: Calcular MCBM usando Hopcroft-Karp
    int mcbm_size = HopcroftKarp::max_matching(n_L, n_R, adj_hk);

    // Resultado: Eliminaciones totales = 1 (si se eliminó el 1) + MCBM(C')
    minimum_removals += mcbm_size;

    return minimum_removals;
}
```