
# PROGRAMACIÓN III

## RED SOCIAL UNIVERSITARIA

Informe Técnico

Docente: **María Angela León**

Alumno: **Lucas Nicolás Miño**
Legajo: **1022935**
Entrega: **15/02/2026**

# Resumen ejecutivo

Este informe presenta la implementación y análisis de cuatro problemas algorítmicos fundamentales aplicados a una red social universitaria. El proyecto cubre los cuatro paradigmas principales de diseño de algoritmos vistos en clase: Divide y Conquista, Programación Dinámica, Greedy y Backtracking.

**Problemas implementados:**

1. Gestión de Publicaciones - Merge Sort (**Divide y Conquista**)
2. Asignación de Publicidad – Problema de la mochila (**Programación Dinámica**)
3. Recomendación de Amigos - Dijkstra (**Greedy**)
4. Simulación de Bloqueos - DFS + **Backtracking con pod**a

# Índice

# Introducción

El presente trabajo integrador de la materia se aborda el diseño e implementación de una **Red Social Universitaria (RSU**) como contexto para aplicar cuatro paradigmas algorítmicos fundamentales vistos durante en la cursada.

La red social se modela como un **grafo no dirigido ponderado** implementado mediante **listas de adyacencia**, donde los vértices representan usuarios y las aristas representan relaciones de amistad con un peso asociado a la cantidad de interacciones entre ellos. Esta estructura permite representar de forma eficiente las conexiones en una red dispersa.

El proyecto esta desarrollado íntegramente **en Java puro** (JDK 21), sin dependencias externas ni frameworks, utilizando únicamente la biblioteca estándar del lenguaje.

La arquitectura separa claramente el modelo de datos, los algoritmos y las pruebas de la siguiente manera:

- **modelo**: Clases de dominio (Usuario, Publicación, Anuncio, Grafo, Arista)
- **algoritmos**: Implementaciones organizadas por paradigma (4 subpaquetes)
- **test**: Clases de prueba para cada problema

Cada problema aborda un aspecto funcional distinto de la red social y emplea el paradigma algorítmico más adecuado para resolverlo, justificando la elección frente a alternativas viables.

**Tabla resumen de los problemas**

| Problema | Paradigma | Algoritmo | Complejidad Temporal | Tipo de análisis | Observaciones |
|---|---|---|---|---|---|
| **Gestión de Publicaciones** | Divide y Conquista | Merge Sort | O(n log n) | Peor caso | |
| **Asignación de Publicidad** | Programación Dinámica | Mochila | O(n · P) | Peor caso | n=cant. anuncios, P=presupuesto. |
| **Recomendación de Amigos** | Greedy | Dijkstra | O((n + a) log n) | Peor caso | n=usuarios, a=aristas. |
| **Simulación de Bloqueos** | Backtracking | DFS + poda | $O(2^n)$ | Peor caso | |

El proyecto se encuentra publicado en **https://github.com/lucas-mino/UADE-RSU.git**

# Problema 1: Gestión de Publicaciones

## P1.1. Descripción del problema

El sistema debe permitir visualizar publicaciones en dos modos diferentes:

- **Vista Cronológica**: publicaciones ordenadas por fecha, de la más reciente a la más antigua.
- **Vista por Relevancia**: publicaciones ordenadas según un puntaje calculado que combina el nivel de interacción (cantidad de "me gusta" y comentarios) con un factor de antigüedad.

La fórmula de relevancia es:

score = (likes * 2 + comentarios) * factorTiempo
factorTiempo = 1.0 / (1.0 + minutosDesdePublicacion / 60.0)

De esta manera, las publicaciones recientes con mayor cantidad de interacciones tienden a ubicarse en las primeras posiciones, mientras que las publicaciones más antiguas disminuyen progresivamente su relevancia debido al factor de decaimiento temporal.

## P1.2. Algoritmo seleccionado

Para este problema utilicé **Merge Sort**, del paradigma **Divide y Conquista**, por las siguientes razones:

- Complejidad garantizada: O(n log n) en TODOS los casos.
- Estable: Mantiene el orden relativo de elementos con igual clave.
- Predecible: Sin casos degenerados (a diferencia de Quick Sort).
- Escalable: para grandes volúmenes de datos.
- Cumple perfectamente con el paradigma Divide y Conquista.

**Comparación con otros algoritmos**

| Algoritmo | Mejor Caso | Promedio | Peor Caso |
|---|---|---|---|
| Merge Sort | O(n log n) | O(n log n) | O(n log n) |
| Quick Sort | O(n log n) | O(n log n) | $O(n^2)$ |
| Selección | O(n log n) | O(n log n) | O(n log n) |
| Inserción | O(n) | $O(n^2)$ | $O(n^2)$ |

Si bien la alternativa de Selección tiene la misma complejidad temporal que Merge Sort, en este caso el enunciado daba indicios de aplicar este algoritmo, como lo detallo más adelante.

## P1.3. Justificación de la estrategia

Este problema se modela como un problema de ordenamiento sobre grandes volúmenes de datos. La consigna sugiere explícitamente:

*"…dividir el conjunto de datos en partes más pequeñas y procesarlas de forma independiente antes de unificarlas…"*

Esto corresponde directamente al paradigma Divide y Conquista, cuya estrategia consiste en:

1. Dividir el conjunto en subproblemas de menor tamaño.
2. Resolver cada subproblema de forma independiente.
3. Combinar las soluciones parciales en una solución global.

El algoritmo **Merge Sort** implementa exactamente esta estructura:

- Divide el arreglo en mitades.
- Ordena recursivamente cada mitad.
- Combina ambas en orden lineal.

## P1.4. Análisis de complejidad

### Complejidad Temporal

Ecuación de Recurrencia

- $T(n) = 2 \times T(n/2) + O(n)$

Donde:

- $2 \times T(n/2)$: Tiempo para ordenar ambas mitades
- $O(n)$: Tiempo para combinar las mitades ordenadas

Resolución por Teorema Maestro

- Forma: $T(n) = a \times T(n/b) + f(n)$ → **División**

Para Merge Sort:

- $a = 2$ (dos llamadas recursivas)
- $b = 2$ (dividir entre 2)
- $k = 1$ → $f(n) = O(n)$ (merge lineal)

$a = b^k$ → $2 = 2^1$ → Caso 2 del Teorema Maestro

- $O(n^k \log(n))$ → **$O(n \log n)$**

Resultado: **$O(n \log n)$**

### Complejidad Espacial

- Arrays temporales: $O(n)$ → Se necesita un array temporal del tamaño del array original para el merge
- Pila de recursión: $O(\log n)$ → Profundidad máxima de la recursión es $\log_2(n)$
- Espacio total: $O(n) + O(\log n) = O(n)$

El término $O(n)$ domina, por lo que la **complejidad espacial es O(n**).

## P1.5. Implementación en Java

### Código principal

```
public class GestorPublicaciones {

 private long comparaciones = 0; // Para análisis de rendimiento

 public List<Publicacion> vistaCronologica(List<Publicacion> publicaciones) {
  List<Publicacion> copia = new ArrayList<>(publicaciones);
  comparaciones = 0;

  // Comparador: fecha más reciente primero
  Comparator<Publicacion> comparadorFecha = (p1, p2) -> {
   comparaciones++;
   return p2.getFecha().compareTo(p1.getFecha()); // Descendente
  };

  mergeSort(copia, 0, copia.size() - 1, comparadorFecha);
  return copia;
 }

 public List<Publicacion> vistaPorRelevancia(List<Publicacion> publicaciones) {
  List<Publicacion> copia = new ArrayList<>(publicaciones);
  comparaciones = 0;

  // Comparador: score más alto primero
  Comparator<Publicacion> comparadorRelevancia = (p1, p2) -> {
   comparaciones++;
   double diff = p2.calcularScore() - p1.calcularScore(); // Descendente
   if (diff > 0) return 1;
   if (diff < 0) return -1;
   return 0;
  };

  mergeSort(copia, 0, copia.size() - 1, comparadorRelevancia);
  return copia;
 }
```

```java
private void mergeSort(
  List<Publicacion> lista,
  int inicio,
  int fin,
  Comparator<Publicacion> comparador
) {
  // Caso base: subarreglo de 1 elemento o vacío
  if (inicio >= fin) {
    return;
  }

  // DIVIDE: calcular punto medio
  int medio = inicio + (fin - inicio) / 2;

  // CONQUISTA: ordenar recursivamente ambas mitades
  mergeSort(lista, inicio, medio, comparador); // Mitad izquierda
  mergeSort(lista, medio + 1, fin, comparador); // Mitad derecha

  // COMBINA: fusionar las dos mitades ordenadas
  merge(lista, inicio, medio, fin, comparador);
}

private void merge(
  List<Publicacion> lista,
  int inicio,
  int medio,
  int fin,
  Comparator<Publicacion> comparador
) {
  // Crear copias temporales de los subarreglos
  List<Publicacion> izquierda = new ArrayList<>();
  List<Publicacion> derecha = new ArrayList<>();

  for (int i = inicio; i <= medio; i++) {
    izquierda.add(lista.get(i));
  }
```

```
    for (int i = medio + 1; i <= fin; i++) {
      derecha.add(lista.get(i));
    }

    // Índices para recorrer los subarreglos y el arreglo principal
    int i = 0; // Índice para izquierda
    int j = 0; // Índice para derecha
    int k = inicio; // Índice para lista principal

    // Combinar elementos en orden
    while (i < izquierda.size() && j < derecha.size()) {
      if (comparador.compare(izquierda.get(i), derecha.get(j)) <= 0) {
        lista.set(k, izquierda.get(i));
        i++;
      } else {
        lista.set(k, derecha.get(j));
        j++;
      }
      k++;
    }

    // Copiar elementos restantes de izquierda (si hay)
    while (i < izquierda.size()) {
      lista.set(k, izquierda.get(i));
      i++;
      k++;
    }

    // Copiar elementos restantes de derecha (si hay)
    while (j < derecha.size()) {
      lista.set(k, derecha.get(j));
      j++;
      k++;
    }
  }
```

El diseño utiliza un Comparator<Publicacion> genérico, lo que permite reutilizar el mismo algoritmo de Merge Sort para ambos criterios de ordenamiento (cronológico y por relevancia) sin duplicar código.

# Problema 2: Asignación de Publicidad

## P2.1. Descripción del problema

El sistema debe asignar anuncios publicitarios a usuarios para **maximizar el alcance total** sin exceder el presupuesto disponible.

Cada anuncio posee:

- Un costo (peso)
- Un valor de alcance (beneficio)

Existe una restricción presupuestaria.

## P2.2. Algoritmo seleccionado

Para este problema utilicé **el problema de la Mochila**, del paradigma **Programación Dinámica**, por las siguientes razones:

El modelado se realizó mapeando los conceptos del problema de la mochila de la siguiente manera:

| Problema de la Mochila | Equivalente en el problema |
|---|---|
| Capacidad de la mochila (W) | Presupuesto |
| Objetos | Anuncios |
| Peso del objeto | Costo del anuncio |
| Valor del objeto | Alcance potencial |
| Maximizar valor total | Maximizar alcance |

## P2.3. Justificación de la estrategia

La primera pista está en el enunciado, donde se indica explícitamente:
*"...construir una solución óptima a partir de subproblemas más pequeños, recordando la solución para no calcularla dos veces…"*

Esto sugiere exactamente el paradigma de **Programación Dinámica**, que se basa en:

- **Principio de optimalidad**: La solución óptima se construye a partir de soluciones óptimas de subproblemas más pequeños. Si el anuncio n está en la solución óptima S*, entonces S* {n} es solución óptima para (n-1 anuncios, presupuesto W - costo[n]).
- **Descomposición en subproblemas dependientes**: Al resolver recursivamente, los mismos subproblemas se calculan múltiples veces.
- **Almacenamiento de resultados intermedios**: se almacenan en la tabla DP.

Además, el objetivo es "**maximizar el alcance total sin exceder el presupuesto"** y el problema se adecua a las características del problema de la mochila, donde:

- Cada anuncio puede ser incluido o no.
- No se permiten fracciones (no puedo usar "medio anuncio").

- Se busca maximizar el valor total bajo una restricción de capacidad.

### Alternativas evaluadas

**Fuerza Bruta:** probar todas las combinaciones

- Complejidad: $O(2^n)$
- Para n=20 anuncios → 1.048.576 combinaciones
- Para n=30 anuncios → 1.073.741.824 combinaciones
- Conclusión: Impracticable para n > 25

**Algoritmo Greedy:** ordenar por eficiencia

- Complejidad: O(n log n)
- No garantiza solución óptima

**Backtracking con Poda**

- Complejidad: $O(2^n)$ peor caso, mejora con poda
- Sigue siendo exponencial

**Programación Dinámica**

- Complejidad: O(n × W)
- Para n=50, W=1000: 50.000 operaciones
- Conclusión: Solución óptima garantizada y complejidad pseudo-polinomial (eficiente si W no es muy grande)

**Resumen de alternativas evaluadas**

| Algoritmo | Complejidad | Optimalidad |
|---|---|---|
| Fuerza Bruta | $O(2^n)$ | Sí |
| Greedy | O(n log n) | No |
| Backtracking | $O(2^n)$ | Sí |
| DP (Mochila) | O(n×W) | Sí |

## P2.4. Análisis de complejidad

### Complejidad temporal

La ecuación de recurrencia que define los subproblemas es:

```
dp[i][w] = max(
   dp[i-1][w],                    // NO incluir anuncio i
   dp[i-1][w - costo[i]] + alcance[i]      // SI incluir anuncio i (si costo[i] <= w)
)
```

Con casos base: dp[0][w] = 0 para todo w, dp[i][0] = 0 para todo i.

La construcción de la tabla requiere dos ciclos anidados:

```
for i = 1 to n:         // n iteraciones
   for w = 0 to W:      // W+1 iteraciones
      dp[i][w] = ...   // O(1)
```

La reconstrucción de la solución recorre la tabla hacia atrás en O(n).

**Complejidad total: O(n x W)**

- Mejor caso: O(n * W)
- Caso promedio: O(n * W)
- Peor caso: O(n * W)

### Complejidad Espacial

- Versión estándar (matriz 2D): **O(n * W)**
- Versión optimizada (arreglo 1D): **O(W)** - implementada en el método "asignarAnunciosOptimizado()", que recorre el arreglo de derecha a izquierda para evitar sobrescribir valores necesarios. La limitación de esta versión es que no permite reconstruir que anuncios fueron seleccionados.

## P2.5. Implementación en Java

### Código principal

```java
public class AsignadorPublicidad {

    // Tabla DP para memorización
    private int[][] dp;

    // Para análisis de rendimiento
    private long operaciones = 0;
    private long tiempoEjecucion = 0;

    public ResultadoAsignacion asignarAnuncios(Usuario usuario, List<Anuncio> anuncios, int presupuesto) {
        operaciones = 0;
        long inicio = System.nanoTime();

        // Filtrar anuncios aplicables al perfil del usuario
        List<Anuncio> anunciosAplicables = filtrarAnunciosAplicables(usuario, anuncios);

        int n = anunciosAplicables.size();
```

```java
    // Caso base: sin anuncios o presupuesto cero
    if (n == 0 || presupuesto <= 0) {
        tiempoEjecucion = System.nanoTime() - inicio;
        return new ResultadoAsignacion(new ArrayList<>(), 0, 0);
    }

    // Inicializar tabla DP
    // dp[i][w] = máximo alcance usando los primeros i anuncios con presupuesto w
    dp = new int[n + 1][presupuesto + 1];

    // Construir tabla DP (bottom-up)
    construirTablaDP(anunciosAplicables, presupuesto);

    // Reconstruir solución (qué anuncios fueron seleccionados)
    List<Anuncio> anunciosSeleccionados = reconstruirSolucion(anunciosAplicables, presupuesto);

    // Calcular alcance total y costo total
    int alcanceTotal = dp[n][presupuesto];
    int costoTotal = anunciosSeleccionados.stream()
            .mapToInt(Anuncio::getCosto)
            .sum();

    tiempoEjecucion = System.nanoTime() - inicio;

    return new ResultadoAsignacion(anunciosSeleccionados, alcanceTotal, costoTotal);
}

private void construirTablaDP(List<Anuncio> anuncios, int presupuesto) {
    int n = anuncios.size();

    // Inicializar primera fila (sin anuncios) = 0
    for (int w = 0; w <= presupuesto; w++) {
        dp[0][w] = 0;
        operaciones++;
    }

    // Llenar tabla DP
```

```
    for (int i = 1; i <= n; i++) {
        Anuncio anuncio = anuncios.get(i - 1);
        int costo = anuncio.getCosto();
        int alcance = anuncio.getAlcancePotencial();

        for (int w = 0; w <= presupuesto; w++) {
            operaciones++;

            // Opción 1: No incluir el anuncio actual
            dp[i][w] = dp[i - 1][w];

            // Opción 2: Incluir el anuncio actual (si cabe en presupuesto)
            if (costo <= w) {
                int alcanceIncluyendo = dp[i - 1][w - costo] + alcance;

                // Elegir la mejor opción (DECISIÓN ÓPTIMA)
                if (alcanceIncluyendo > dp[i][w]) {
                    dp[i][w] = alcanceIncluyendo;
                }
            }
        }
    }
}
```

El método “filtrarAnunciosAplicables()” verifica que cada anuncio sea relevante para el perfil del usuario antes de ejecutar el algoritmo de mochila, reduciendo el tamaño efectivo de la entrada.

# Problema 3: Recomendación de Amigos

## P3.1. Descripción del problema

El sistema debe recomendar amistades potenciales a un usuario basándose en la "distancia" en la red social. Se considera amigo potencial a un usuario que:

1. No sea amigo directo del usuario.
2. Este alcanzable en la red (conectado a través de otros usuarios).
3. Tenga la menor distancia posible en la red.

La red social se modela como un grafo no dirigido ponderado donde el peso de cada arista representa la cercanía entre dos usuarios, considerando que a mayor número de interacciones entre dos amigos, menor es el peso (más cercanos están).

El algoritmo debe encontrar los caminos más cortos desde un usuario origen a todos los demás, y luego recomendar los N usuarios no-amigos más cercanos.

## P3.2. Algoritmo seleccionado

Para este problema utilicé el **algoritmo de Dijkstra**, del paradigma **Greedy**, por las siguientes razones:

El primer motivo de elección, es la pista que se da en el enunciado, donde dice:
*"…encontrar la ruta de menor costo desde un usuario hacia todos los demás…"*

Además, este algoritmo posee:

- Complejidad aceptable si se implementa bien → $O((n + a) \log n)$
- Garantiza caminos mínimos en grafos con pesos positivos.
- Escalable a miles de usuarios.

Por lo tanto, Dijkstra resulta la estrategia adecuada para generar recomendaciones basadas en cercanía estructural en la red.

## P3.3. Justificación de la estrategia

El algoritmo de **Dijkstra** pertenece al paradigma **Greedy** y funciona bajo la siguiente lógica:

1. En cada paso selecciona el nodo no visitado con menor distancia acumulada.
2. Marca esa distancia como definitiva.
3. Descarta las aristas adyacentes.

La propiedad clave es que cuando se selecciona el nodo con menor distancia provisional, se garantiza que esa distancia es óptima.

Esto cumple con el principio de elección local óptima propio de los algoritmos Greedy.

Requisito fundamental: todos los pesos deben ser no negativos. En mi modelo, peso = max(1, 100 - interacciones) >= 1, por lo que esta condición se cumple siempre.

## P3.4. Análisis de complejidad

### Complejidad Temporal

Las estructuras de datos utilizadas son:
PriorityQueue<NodoDijkstra>: insercion y extraccion en O(log V). - HashMap<Usuario, Integer>: acceso y escritura en O(1) promedio. - HashSet<Usuario>: verificacion y insercion en O(1) promedio.

Análisis del algoritmo:

1. **Inicialización**: se asigna distancia infinita a los V vértices → O(V)
   Antes de arrancar, se le asigna distancia infinita a cada usuario. Si hay V usuarios, se hacen V asignaciones. Eso es **O(V)**.
2. 

3. **Bucle principal**: cada vértice se extrae de la cola una vez → O(V log V)
   El algoritmo usa una cola que siempre te da el usuario con menor distancia. Cada usuario se saca de la cola una sola vez → son V extracciones. Cada cuesta log V.

   Entonces:
   V extracciones × log V cada una = **O(V log V)**

4. **Mirar vecinos**: cada arista se analiza una vez → O(E log V)
   Cada vez que se saca un usuario de la cola, se mira a todos sus amigos (vecinos). En total, sumando todos los usuarios, se mira cada arista una sola vez, eso es E aristas en total. Por cada arista, si se encuentra un camino mejor, se agrega al vecino en la cola, lo cual cuesta log V.

   Entonces:
   E aristas × log V cada inserción = O(E log V)

Resultado: T(V, E) = O(V) + O(V log V) + O(E log V) = **O((V + E) log V)**

Para grafos dispersos donde E ~ k*V con k = grado promedio constante:

T(V, E) = O((V + kV) log V) = O(V log V)

- Mejor caso: O((V + E) log V)
- Caso promedio: O((V + E) log V)
- Peor caso: O((V + E) log V)

### Complejidad Espacial

S(V) = O(V) + O(V) + O(V) = O(V)

- O(V) para el mapa de distancias.
- O(V) para el conjunto de visitados.
- O(V) para peor caso.

El grafo en si ocupa O(V + E) pero es parte del input, no del algoritmo.

## P3.5. Implementación en Java

### Código principal

```
public class RecomendadorAmigos {

    // Para análisis de rendimiento
    private long operaciones = 0;
    private long tiempoEjecucion = 0;

    private static class NodoDijkstra implements Comparable<NodoDijkstra> {
        Usuario usuario;
        int distancia;

        public NodoDijkstra(Usuario usuario, int distancia) {
            this.usuario = usuario;
            this.distancia = distancia;
        }

        @Override
        public int compareTo(NodoDijkstra otro) {
            return Integer.compare(this.distancia, otro.distancia);
        }
    }

    public Map<Usuario, Integer> calcularDistancias(Grafo grafo, Usuario origen) {
        operaciones = 0;
        long inicio = System.nanoTime();

        // Inicializar estructuras de datos
        Map<Usuario, Integer> distancias = new HashMap<>();
        Set<Usuario> visitados = new HashSet<>();
        PriorityQueue<NodoDijkstra> cola = new PriorityQueue<>();
```

```
// Inicializar todas las distancias como infinito
for (Usuario u : grafo.getUsuarios()) {
    distancias.put(u, Integer.MAX_VALUE);
    operaciones++;
}

// La distancia al origen es 0
distancias.put(origen, 0);
cola.offer(new NodoDijkstra(origen, 0));

// ALGORITMO DE DIJKSTRA
while (!cola.isEmpty()) {
    operaciones++;

    // GREEDY: Seleccionar el nodo no visitado con menor distancia
    NodoDijkstra actual = cola.poll();
    Usuario usuarioActual = actual.usuario;

    // Si ya fue visitado, saltar
    if (visitados.contains(usuarioActual)) {
        continue;
    }

    // Marcar como visitado
    visitados.add(usuarioActual);

    // Relajar todas las aristas adyacentes
    for (Arista arista : grafo.getVecinos(usuarioActual)) {
        operaciones++;

        Usuario vecino = arista.getDestino();

        // Si ya fue visitado, saltar
        if (visitados.contains(vecino)) {
            continue;
        }
```

```
            // Calcular nueva distancia potencial
            int distanciaActual = distancias.get(usuarioActual);
            int nuevaDistancia = distanciaActual + arista.getPeso();

            // Si encontramos un camino más corto, actualizar
            if (nuevaDistancia < distancias.get(vecino)) {
                distancias.put(vecino, nuevaDistancia);
                cola.offer(new NodoDijkstra(vecino, nuevaDistancia));
            }
        }
    }

    tiempoEjecucion = System.nanoTime() - inicio;
    return distancias;
}

public List<RecomendacionAmigo> recomendar(Grafo grafo, Usuario usuario, int n) {
    // Calcular distancias con Dijkstra
    Map<Usuario, Integer> distancias = calcularDistancias(grafo, usuario);

    // Obtener amigos directos para excluirlos de las recomendaciones
    Set<Usuario> amigosDirectos = new HashSet<>();
    for (Arista arista : grafo.getVecinos(usuario)) {
        amigosDirectos.add(arista.getDestino());
    }

    // Crear lista de candidatos (no son amigos directos, ni el mismo usuario)
    List<RecomendacionAmigo> candidatos = new ArrayList<>();
    for (Map.Entry<Usuario, Integer> entry : distancias.entrySet()) {
        Usuario candidato = entry.getKey();
        int distancia = entry.getValue();

        // Excluir: el mismo usuario, amigos directos, y usuarios inalcanzables
        if (!candidato.equals(usuario) &&
            !amigosDirectos.contains(candidato) &&
            distancia != Integer.MAX_VALUE) {
```

```
            // Reconstruir ruta
            List<Usuario> ruta = reconstruirRuta(grafo, usuario, candidato, distancias);

            RecomendacionAmigo rec = new RecomendacionAmigo(candidato, distancia, ruta);
            candidatos.add(rec);
        }
    }

    // Ordenar por distancia (menor primero)
    candidatos.sort(Comparator.comparingInt(RecomendacionAmigo::getDistancia));

    // Retornar top N
    return candidatos.subList(0, Math.min(n, candidatos.size()));
}
```

El método “recomendar()” ejecuta Dijkstra una sola vez y obtiene las distancias a todos los usuarios, luego filtra los amigos directos y los usuarios inalcanzables, y retorna los N candidatos más cercanos. Además, reconstruye la ruta más corta hasta cada candidato para mostrar la cadena de conexiones.

# Problema 4: Simulación de Bloqueos

## P4.1. Descripción del problema

Cuando un usuario bloquea a otro en la red social, se elimina la arista que los conecta en el grafo.

El sistema debe:

1. **Verificar conectividad**: determinar si el grafo sigue siendo conexo después del bloqueo.
2. **Restaurar conectividad**: si el grafo se desconecta, encontrar el conjunto mínimo de nuevas conexiones necesarias para restablecer la conectividad total.

## P4.2. Algoritmo seleccionado

Para este problema utilice el algoritmo **Búsqueda en profundidad(DFS)**, en **Backtracking con poda**. En este caso, podría haber utilizado otros algoritmos, pero quise implementar alguno distinto, y dado que la complejidad se mantuvo aceptable lo implemente de esta manera.

Además, en la consigna sugiere:
*"…modelar como una búsqueda exhaustiva de combinaciones de conexiones…"*

## P4.3. Justificación de la estrategia

El problema de restaurar la conectividad del grafo requiere encontrar el mínimo conjunto de aristas nuevas que reconecten todas las componentes. Esto implica evaluar distintas combinaciones de conexiones posibles, lo cual es por naturaleza un problema de búsqueda combinatoria. Se necesita un enfoque que explore sistemáticamente las opciones y garantice que la solución encontrada sea optima.

El flujo general del algoritmo es:

1. Copiar el grafo para no mutar el original.
2. Eliminar la arista del bloqueo.
3. Verificar conectividad con DFS.
4. Si se desconectó, identificar componentes conexas, generar conexiones candidatas entre componentes, y ejecutar backtracking con poda para encontrar el conjunto mínimo de reconexión.

**Alternativas evaluadas**

**Fuerza bruta** (probar todas las combinaciones de conexiones) tiene complejidad $O(2^C)$ sin ninguna optimización.

**Greedy** (conectar componentes de forma voraz). Es rápido , O(V + E), y encuentra una solución valida de tamaño n-1, pero no considera todas las alternativas posibles.

**Programación Dinámica** no es aplicable directamente porque el problema no presenta subproblemas superpuestos de forma natural: cada combinación de conexiones produce un grafo distinto cuya conectividad debe verificarse independientemente.

**Backtracking** combina la exhaustividad de la fuerza bruta con la inteligencia de tres estrategias de poda que descartan ramas del árbol de búsqueda sin explorarlas:

1. **Poda por cota superior**: si la solución parcial actual ya tiene tantas o más conexiones que la mejor solución conocida, no puede mejorarla, por lo que se descarta toda la rama.

2. **Poda por cota inferior** (mínimo teórico): matemáticamente, para conectar n componentes se necesitan al menos n-1 aristas (propiedad fundamental de los árboles). Si ya se encontró una solución de tamaño n-1, es imposible encontrar algo mejor, y el algoritmo termina.

3. **Poda por validación temprana**: cuando al agregar conexiones el grafo ya resulta conexo, no tiene sentido seguir agregando más conexiones a esa rama.

Estas tres podas combinadas reducen el espacio de búsqueda entre un 90% y un 99% en casos típicos, haciendo que el algoritmo sea practicable para redes de tamaño moderado.

Además, se implementó el método auxiliar "encontrarConexionesRapido()", con estrategia Greedy, que permite comparar ambos enfoques y ofrecer una solución rápida cuando no se requiere exploración exhaustiva.

## P4.4. Análisis de complejidad

### Complejidad Temporal

- **Verificación de conectividad (DFS)**: O(V + E) → cada vértice y cada arista se visita una vez.
- **Identificación de componentes:** O(V + E) → DFS múltiples desde cada nodo no visitado; en total se visita cada nodo y arista una sola vez.
- **Generación de candidatas:** $O(n^2)$ donde n = número de componentes. Se genera un par representante por cada par de componentes: C = n*(n-1)/2.
- **Backtracking sin poda:** $O(2^C)$ → árbol binario de profundidad C.
- **Backtracking con poda:** $O(2^k)$ donde k << C. La poda es extremadamente efectiva.

### Complejidad Espacial

S(V, C) = O(V) + O(C) = O(V + C)

- O(V) para el conjunto de visitados del DFS y la pila de recursión.
- O(C) para la solución actual y la mejor solución en el backtracking.

## P4.5. Implementación en Java

### Código principal

```
public class SimuladorBloqueos {
    // Para análisis de rendimiento
    private long operaciones = 0;
    private long tiempoEjecucion = 0;
    private int nodosExplorados = 0;
    private int nodosRecortados = 0;

    public ResultadoBloqueo simularBloqueo(Grafo grafo, Usuario bloqueador, Usuario bloqueado) {
        operaciones = 0;
        nodosExplorados = 0;
        nodosRecortados = 0;
        long inicio = System.nanoTime();

        // Crear copia del grafo para no modificar el original
        Grafo grafoCopia = grafo.copiar();

        // Verificar que la conexión existe
        if (!grafoCopia.sonAmigos(bloqueador, bloqueado)) {
            tiempoEjecucion = System.nanoTime() - inicio;
            return new ResultadoBloqueo(
                bloqueador, bloqueado, true, 0,
                new ArrayList<>(), "No existe conexión para bloquear"
            );
        }

        // Eliminar la conexión (bloqueo)
        grafoCopia.eliminarAmistad(bloqueador, bloqueado);
        operaciones++;

        // Verificar si el grafo sigue siendo conexo
        boolean esConexo = verificarConectividad(grafoCopia);

        if (esConexo) {
```

```
            // El grafo sigue conexo, no se necesitan nuevas conexiones
            tiempoEjecucion = System.nanoTime() - inicio;
            return new ResultadoBloqueo(
                bloqueador, bloqueado, true, 0,
                new ArrayList<>(), "El grafo sigue conexo después del bloqueo"
            );
        }

        // El grafo no es conexo, encontrar conexiones mínimas
        List<ParUsuarios> conexionesMinimas = encontrarConexionesMinimas(grafoCopia);

        tiempoEjecucion = System.nanoTime() - inicio;

        return new ResultadoBloqueo(
            bloqueador, bloqueado, false, conexionesMinimas.size(),
            conexionesMinimas, "Se requieren " + conexionesMinimas.size() + " conexiones nuevas"
        );
    }

    public boolean verificarConectividad(Grafo grafo) {
        if (grafo.estaVacio()) {
            return true;
        }

        Set<Usuario> usuarios = grafo.getUsuarios();
        if (usuarios.isEmpty()) {
            return true;
        }

        // Iniciar DFS desde el primer usuario
        Usuario inicio = usuarios.iterator().next();
        Set<Usuario> visitados = new HashSet<>();

        dfs(grafo, inicio, visitados);
        operaciones += visitados.size();

        // Si visitamos todos los usuarios, el grafo es conexo
```

```
    return visitados.size() == usuarios.size();
}

private void dfs(Grafo grafo, Usuario actual, Set<Usuario> visitados) {
    visitados.add(actual);

    for (Arista arista : grafo.getVecinos(actual)) {
        Usuario vecino = arista.getDestino();
        if (!visitados.contains(vecino)) {
            dfs(grafo, vecino, visitados);
        }
    }
}

public List<Set<Usuario>> identificarComponentes(Grafo grafo) {
    List<Set<Usuario>> componentes = new ArrayList<>();
    Set<Usuario> visitados = new HashSet<>();

    for (Usuario usuario : grafo.getUsuarios()) {
        if (!visitados.contains(usuario)) {
            Set<Usuario> componente = new HashSet<>();
            dfs(grafo, usuario, componente);
            componentes.add(componente);
            visitados.addAll(componente);
        }
    }

    return componentes;
}

public List<ParUsuarios> encontrarConexionesMinimas(Grafo grafo) {
    // Identificar componentes desconectadas
    List<Set<Usuario>> componentes = identificarComponentes(grafo);

    if (componentes.size() <= 1) {
        // Ya es conexo
        return new ArrayList<>();
```

```
    }

    // Generar todas las conexiones candidatas (entre componentes diferentes)
    List<ParUsuarios> candidatas = generarConexionesCandidatas(componentes);

    // Búsqueda con backtracking
    List<ParUsuarios> mejorSolucion = new ArrayList<>();
    List<ParUsuarios> solucionActual = new ArrayList<>();

    // El mínimo teórico es (número de componentes - 1)
    int minimoTeorico = componentes.size() - 1;

    backtrack(grafo, candidatas, 0, solucionActual, mejorSolucion, minimoTeorico);

    return mejorSolucion;
}

private void backtrack(Grafo grafo, List<ParUsuarios> candidatas, int index,
                       List<ParUsuarios> actual, List<ParUsuarios> mejor, int minimoTeorico) {

    nodosExplorados++;

    // CASO BASE: Verificar si la solución actual hace el grafo conexo
    if (!actual.isEmpty()) {
        Grafo grafoPrueba = grafo.copiar();
        for (ParUsuarios par : actual) {
            grafoPrueba.agregarAmistad(par.getUsuario1(), par.getUsuario2(), 1);
        }

        if (verificarConectividad(grafoPrueba)) {
            // Encontramos una solución válida
            if (mejor.isEmpty() || actual.size() < mejor.size()) {
                mejor.clear();
                mejor.addAll(actual);

                // PODA ÓPTIMA: Si alcanzamos el mínimo teórico, no hay mejor solución
                if (mejor.size() == minimoTeorico) {
```

```
                    return;
                }
            }
            return; // No seguir explorando esta rama
        }
    }

    // CASO BASE: Exploradas todas las candidatas
    if (index >= candidatas.size()) {
        return;
    }

    // PODA: Si la solución actual ya es >= mejor conocida, no seguir
    if (!mejor.isEmpty() && actual.size() >= mejor.size()) {
        nodosRecortados++;
        return;
    }

    // DECISIÓN 1: INCLUIR la conexión actual
    actual.add(candidatas.get(index));
    backtrack(grafo, candidatas, index + 1, actual, mejor, minimoTeorico);
    actual.remove(actual.size() - 1); // BACKTRACK

    // DECISIÓN 2: NO INCLUIR la conexión actual
    backtrack(grafo, candidatas, index + 1, actual, mejor, minimoTeorico);
  }
}
```

# Conclusiones

El presente trabajo permitió aplicar de manera integrada los principales paradigmas de diseño y análisis de algoritmos en un contexto realista. Más allá de la implementación funcional en Java, el foco estuvo puesto en la correcta elección y justificación de cada estrategia algorítmica frente a alternativas posibles.

Cada problema abordado requirió analizar su estructura matemática subyacente antes de decidir la técnica de resolución:

- En **Gestión de Publicaciones**, el problema se redujo a un ordenamiento masivo de datos, donde el paradigma Divide y Conquista mediante Merge Sort ofreció una complejidad garantizada y comportamiento estable ante grandes volúmenes de información.
- En **Asignación de Publicidad**, el carácter combinatorio y la necesidad de optimalidad estricta condujeron naturalmente al modelo de Programación Dinámica (Mochila 0/1), descartando enfoques Greedy por no garantizar solución óptima.
- En **Recomendación de Amigos**, modelado como grafo ponderado permitió aplicar Dijkstra bajo el paradigma Greedy, garantizando caminos mínimos en tiempo eficiente para grafos dispersos.
- En **Simulación de Bloqueos**, el desafío combinatorio de restaurar conectividad motivó la aplicación de Backtracking con poda, demostrando cómo una estrategia exhaustiva puede volverse viable cuando se incorporan cotas teóricas y validaciones tempranas.

Uno de los aprendizajes centrales del proyecto fue comprender que la elección del algoritmo no depende únicamente de la eficiencia teórica, sino también de:

- La naturaleza del problema (optimización, búsqueda, ordenamiento, conectividad).
- Las propiedades estructurales (subproblemas superpuestos, optimalidad local vs global).
- Las restricciones del dominio (presupuesto, conectividad mínima, pesos no negativos).
- El tamaño esperado de la entrada.

Asimismo, el análisis de complejidad permitió comparar formalmente las soluciones implementadas, evidenciando la diferencia entre algoritmos polinomiales eficientes O(n log n, O((V+E) log V)), pseudo-polinomiales O(n x W) y exponenciales $O(2^n)$, y cómo la incorporación de técnicas de poda puede reducir significativamente el espacio práctico de búsqueda.

Desde el punto de vista de ingeniería, el desarrollo modular en Java, la separación por paquetes y la instrumentación para medición de operaciones y tiempos de ejecución permitieron validar empíricamente los análisis teóricos, fortaleciendo la coherencia entre teoría y práctica.

En conclusión, el trabajo no solo resolvió los problemas planteados, sino que evidenció la importancia de seleccionar el paradigma adecuado para cada situación, justificando formalmente dicha elección y comprendiendo sus implicancias en términos de rendimiento, escalabilidad y optimalidad.