Informe Moderador Del Conflicto Interno

PROFESOR:

Jesus Alexander Bueno Aranda

PRESENTADO POR:

Angie Carolina Joya Duarte - 2322609

Emily Nuñez Ordoñez - 2240156

Sheila Marcela Valencia Chito - 2243011

Victoria Andrea Volveras Parra - 2241874

ANÁLISIS Y DISEÑO DE ALGORITMO II

FACULTAD DE INGENIERÍA

UNIVERSIDAD DEL VALLE

CALI, VALLE

2025

INTRODUCCIÓN

Las redes sociales son una fuente de información ampliamente utilizada entre las personas. Esto les permite tener diferentes opiniones sobre un mismo tema, lo que genera conflicto interno. Cuando el valor del conflicto interno es alto significa que hay alta maleabilidad de las opiniones. Por consiguiente, el presente informe presenta la implementación de tres algoritmos para tratar de reducir el conflicto de opiniones en una red social.

ALGORITMO DE FUERZA BRUTA

Descripción

El algoritmo de fuerza bruta calcula todas las posibles estrategias de cambio de opinión E y el conflicto interno de cada una de ellas. Luego, selecciona la tupla E que tenga menor conflicto interno y esta es la solución que retorna.

Complejidad

Complejidad temporal

Para desarrollar el algoritmo de fuerza bruta se tiene un función interna recursiva *generarCombinacion*, la cual genera todas las combinaciones posibles de agentes a modificar.

Para cada posición i, puede tomar valores desde 0 hasta m_i, donde m_i es la cantidad máxima de agentes por cada grupo i. Por lo tanto:

$$Total\ combinaciones = \prod_{i=0}^{n-1} (m_i + 1)$$

En el peor caso, si cada grupo tiene el mismo número de agentes máximo m, entonces:

$$Total\ combinaciones = (m + 1)^n$$

Por lo tanto, su complejidad será:

$$O((m + 1)^n)$$

Donde n es el número de grupos de agentes en la red.

Por cada una de las combinaciones se realizan las siguientes operaciones:

- *calcularEsfuerzoRed(rs, combinacion):* $O(n)$ (Recorre todos los grupos de agentes)
- *obtenerNuevaRed(red, combinacion)*: $O(n)$ (Recorre todos los grupos de agentes)
- *calcularCI(...):* $O(n)$ (Recorre todos los grupos de agentes)

Teniendo esto en cuenta, la complejidad temporal se puede ver como:

$$O(n * (m + 1)^n)$$

Como se puede observar se tiene una complejidad temporal exponencial.

Complejidad espacial

- *maximos:* $O(n)$ (Arreglo de tamaño n).
- *combinacion*: $O(n)$ (Arreglo de tamaño n).
- *e*: $O(n)$ (Arreglo de tamaño n).
- *solucion*: $O(n)$ (Copia de la red social).
- *nuevaRed*: $O(n)$ (Ocupa el mismo espacio que la red social).
- *Recursion*: $O(n)$ (La recursión de *generarCombinacion* tiene una profundidad máxima de n).

Por lo tanto, la complejidad espacial se puede ver como:

$$O(n)$$

Se puede evidenciar que la complejidad espacial es de forma lineal.

Corrección

El algoritmo de fuerza bruta siempre encontrará la solución correcta, ya que genera todas las posibles combinaciones, considerando la cantidad de agentes en cada grupo. Donde para n grupos con m_i agentes examina $\prod_{i=0}^{n-1}(m_i + 1)$ combinaciones.

Adicionalmente, filtra las combinaciones teniendo en cuenta que el esfuerzo de moderar esa cantidad de agentes sea menor al esfuerzo máximo disponible. Es decir:

$$\sum_{i=0}^{n-1}(\lceil |o_{i,1} - o_{i,2}| * r_i * e_i \rceil) \leq r\_max$$

Garantizando que solo se evalúen estrategias dentro del presupuesto establecido.

También calcula el conflicto interno para cada una de las combinaciones que cumplieron con la restricción anterior, este es comparado con la mejor solución encontrada hasta el momento, donde se conserva la que tenga el menor conflicto interno. Por lo tanto, esto asegura que siempre retornará la solución óptima.

ALGORITMO VORAZ

Descripción

Primero, se ordenan los grupos de agentes (sag) según su beneficio, utilizando un montículo de máximos. El beneficio de un agente en un grupo de agentes i se calcula como el cociente entre la diferencia de sus opiniones $(|o1 - o2|)$ y su rigidez (r) es decir:

$$beneficio = \left(\frac{|o1 - o2|}{r}\right).$$

Una vez construido el montículo , se extrae iterativamente los grupos de agentes con mayor beneficio. Para dicho grupo de agentes , se calcula el esfuerzo requerido al moderar n agentes del grupo. Si este esfuerzo supera el límite máximo disponible para moderar opiniones $\left(r_{max}\right)$, se ajusta el valor de 'n' al máximo posible que no exceda dicho límite. Este ajuste se realiza calculando el piso del cociente entre r_{max} y el esfuerzo necesario para ese agente .

El valor ajustado se almacena en una tupla e, que representa el número de agentes a los cuales se les cambió la opinión en un grupo de agentes i . Este procedimiento se repite hasta agotar los recurso r_{max} o hasta que no queden más grupos de agentes por procesar.

Finalmente, una vez recorrido todos lo agentes del grupo (sag), se retorna la tupla e, el esfuerzo total y el nuevo valor de conflicto interno de la red (ci), que constituyen la solución obtenida mediante el enfoque voraz.

Casos de Prueba

Las soluciones obtenidas mediante el algoritmo voraz se verifican comparándolas con los resultados del algoritmo de fuerza bruta.

Casos de pruebas propios

Las siguientes cuatro pruebas, tienen la misma cantidad de grupos de agentes, de agentes en cada grupo. Se diferencian en el valor de las opiniones y la rigidez .

- **Prueba 1**

  Todos los grupos tienen una opinión completamente favorable sobre la afirmación 1, mientras que todos tienen una opinión neutral sobre la afirmación 2. La rigidez de los agentes está por encima de 0.5.

  $RS = < SAG , 1000 >$

  $SAG = << 5, -100, 0, 0.6 > , < 5, -100, 0, 0.7 > ,$

  $< 5, -100, 0, 0.8 > , < 5, -100, 0, 0.9 >, < 5, -100, 0, 0.5 >>$

- **Prueba 2**

  Todos los grupos tienen una opinión completamente favorable sobre la afirmación 1, mientras que todos tienen diferentes opiniones sobre la segunda afirmación. Todos tienen una rigidez de 1.

$$RS = < SAG , 1000 >$$

$$SAG = << 5, -100, -30, 1 > , < 5, -100, -10, 1 > ,$$

$$< 5, -100, 50, 1 > , < 5, -100, 1, 1 >, < 5, -100, -50, 1 >>$$

- **Prueba 3**

  Todos los grupos tienen una opinión completamente desfavorable sobre la afirmación 1 y completamente favorable sobre la afirmación 2. Todos los agentes tienen niveles de rigidez bajos . El valor de moderación es de 100.

$$RS = < SAG , 100 >$$

$$SAG = << 5, -100, 100, 0.02 > , < 5, -100, 100, 0.1 > ,$$

$$< 5, -100, 100, 0.05 >, < 5, -100, 100, 0.03 >, < 5, -100, 100, 0.2 >>$$

- **Prueba 4**

  Todos los grupos tienen opiniones favorables sobre las afirmaciones 1 y 2. Su nivel de rigidez es alto y un valor de moderación de 1000.

$$RS = < SAG , 1000 >$$

$$SAG = << 5, 90, 80, 0.8 > , < 5, 100, 87, 0.976 > ,$$

$$< 5, 85, 98, 0.853 > , < 5, 92, 88, 1 >, < 5, 91, 88, 0.921 >>$$

Prueba	Solución	¿Solución óptima?	Costo
2.3.1	$esfuerzo = 15033,33$ $ci = 77$ $e = [1, 1, 0]$	Si	77
2.3.2	$esfuerzo = 13333,33$ $ci = 380$ $e = [2, 2, 4]$	Si	380
Prueba Propia 1	$esfuerzo = 18000$ $ci = 980$ $e = [5, 5, 1, 0, 5]$	Si	980
Prueba Propia 2	$esfuerzo = 25201$ $ci = 800$	Si	800

	$e = [0, 0, 5, 0, 1]$		
Prueba Propia 3	$esfuerzo = 80000$ $ci = 100$ $e = [5, 0, 5, 5, 0]$	Si	100
Prueba Propia 4	$esfuerzo = 0$ $ci = 194$ $e = [5, 5, 5, 5, 5]$	Si	194

Complejidad

Complejidad temporal

Primero, se debe calcular la complejidad de la función $crearHeap(sag)$.

- $crearHeap(sag)$

  Esta función tiene una complejidad de $O(n \lg n)$, ya que incluye un ciclo for que recorre un arreglo de n posiciones (sag) [1] . Durante cada iteración, se insertan elementos en un montículo de acuerdo a la prioridad de $beneficio$, operación que tiene un costo de $\lg n$. Por lo tanto:

$$\Sigma_{i=1}^{n} \lg n = \lg n * (n - 1 + 1) = n * \lg n$$

Luego, se analiza la complejidad del ciclo while:

- Peor caso : El bucle realiza n repeticiones, ya que se procesan todos los elementos del montículo.
- Caso promedio: El bucle realiz k repeticiones, con $k < n$, si el recurso r_{max} se agota rápidamente.
- Mejor caso: El primer elemento del montículo consume todos los recursos de r_{max}.

Dentro del bucle $while$, la mayoría de las operaciones tienen complejidad constante, a excepción del acceso y extracción de elementos del montículo, que tiene una complejidad logarítmica.

Por ende la complejidad del bucle sería:

- Peor caso

$$\Sigma_{i=1}^{n} \lg n = \lg n * (n - 1 + 1) = O(n * \lg n)$$

[1] 'n' representa la cantidad de grupo de agentes que tiene una red social.

- Caso promedio

$$\Sigma_{i=1}^{k} lg\, n \;=\; lg\, n \;*\; (k - 1 + 1) \;=\; O(\; k \;*\; lg\, n)$$

- Mejor caso

$$\Sigma_{i=1}^{1} lg\, n \;=\; O(lg\, n)$$

Finalmente, la complejidad sería:

- Peor caso

$$O(n\, lg\, n) \;+\; O(n\, lg\, n) \;=\; O(n\, lg\, n)$$

- Caso promedio

$$O(n\, lg\, n) \;+\; O(k\, lg\, n) \;=\; O(n\, lg\, n)$$

- Mejor caso

$$O(n\, lg\, n) \;+\; O(\, lg\, n) \;=\; O(n\, lg\, n)$$

En conclusión, el algoritmo $modciV(redSocial)$ tiene una complejidad $O(n\, lg\, n\,)$ para cualquier caso.

Complejidad espacial

Las principales estructuras de datos empleadas en el algoritmo con enfoque voraz son:

- Array e (solución)
- Montículo $heap$

Ambas estructuras tienen una complejidad espacial lineal, dado que su tamaño siempre depende de la cantidad de grupos de agentes de la red social. Es decir, de la longitud del arreglo sag. Por otra parte, las variables y funciones auxiliares utilizadas en el algoritmo tienen una complejidad constante.

En conclusión, se afirma que la complejidad espacial del algoritmo con enfoque voraz es lineal $O(n)$.

Corrección

El algoritmo voraz prioriza los grupos con mayor impacto, la métrica utilizada identifica los grupos donde la reducción de conflicto interno por unidad de esfuerzo es mayor, estos son

organizados de manera descendente, posteriormente se modera la mayor cantidad de agentes por cada grupos, teniendo en cuenta que no se supere el esfuerzo disponible.

El enfoque utilizado siempre retorna la solución óptima cuando el esfuerzo disponible es suficiente para moderar todos los agentes, adicionalmente, se puede evidenciar que en los casos donde falla, el valor de la solución es cercano al óptimo, por esta razón se obtuvo un error de solamente el 0,13%.

La estrategia puede fallar para los casos donde sea mejor moderar parcialmente ciertos grupos, en lugar de todos los agentes posibles por cada grupo, esto se puede evidenciar en la prueba 8, donde:

Solución óptima: [0, 1, 6, 1, 0, 0, 8, 10, 0, 7]

Solución del algoritmo voraz: [0, 10, 4, 1, 0, 0, 10, 10, 0, 7]

Grupos: $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9]$

Al calcular el beneficio de cada uno, se ordena descendentemente:

$$[a_9, a_7, a_6, a_3, a_2, a_4, a_0, a_8, a_1, a_5]$$

En el caso de a_6 era mejor moderar parcialmente este grupo, donde solo se cambia la opinión de 8 agentes en lugar de 10, al moderar 10 agentes, el esfuerzo disponible para los grupos restantes se ve afectado.

ALGORITMO DINÁMICO

Caracterización de la Solución Óptima

Sea:

- RS: Red social inicial
- RS': Red social resultante al aplicar la solución
- E: Solución final
- $Es_{a,b}$: Esfuerzo de cambiar a personas del grupo b
- R_max: Máximo esfuerzo que se puede usar
- n: Cantidad de grupos

Se define:

$CI(RS')$: Valor mínimo de CI que puede tomar la red social original después de aplicarle la solución óptima.

E: Solución óptima

Si R_max = 0 entonces:

$$RS' = RS$$

$$CI(RS') = CI(RS)$$

$$E = (0, \ldots, 0)$$

Si $Es_{a,b}$ > R_max entonces:

Posibles soluciones:

$$E_0 = (x_0, \ldots, 0, \ldots, x_{n-1})$$

$$E_1 = (x_0, \ldots, 1, \ldots, x_{n-1})$$

. . .

$$E_{a-1} = (x_0, \ldots, a-1, \ldots, x_{n-1})$$

Redes sociales generadas:

$$RS_0 = < \ldots, < n_b^{RS} - 0, O_{b,1}^{RS}, O_{b,1}^{RS}, r_b^{RS} >, \ldots >$$

$$RS_1 = < \ldots, < n_b^{RS} - 1, O_{b,1}^{RS}, O_{b,1}^{RS}, r_b^{RS} >, \ldots >$$

. . .

$$RS_{a-1} = < \ldots, < n_b^{RS} - (a-1), O_{b,1}^{RS}, O_{b,1}^{RS}, r_b^{RS} >, \ldots >$$

$$CI(RS') = min(CI(RS_0), CI(RS_1), \ldots, CI(RS_{a-1}))$$

RS': $Red\ social\ con\ el\ mínimo\ CI$

E: $Solución\ que\ produjo\ la\ red\ social\ con\ el\ menor\ CI$

Si $Es_{a,b} \leq$ R_max entonces:

Posibles soluciones:

$$E_0 = (x_0, \ldots, 0, \ldots, x_{n-1})$$

$E_1 = (x_0, \ldots, 1, \ldots, x_{n-1})$

...

$E_a = (x_0, \ldots, a, \ldots, x_{n-1})$

Redes sociales generadas:

$RS_0 = < \ldots, < n_b^{RS} - 0, O_{b,1}^{RS}, O_{b,1}^{RS}, r_b^{RS} >, \ldots >$

$RS_1 = < \ldots, < n_b^{RS} - 1, O_{b,1}^{RS}, O_{b,1}^{RS}, r_b^{RS} >, \ldots >$

...

$RS_a = < \ldots, < n_b^{RS} - a, O_{b,1}^{RS}, O_{b,1}^{RS}, r_b^{RS} >, \ldots >$

$CI(RS') = min(CI(RS_0), CI(RS_1), \ldots, CI(RS_a))$

RS': $Red\ social\ con\ el\ mínimo\ CI$

E: $Solución\ que\ produjo\ la\ red\ social\ con\ el\ menor\ CI$

Definición recursiva del valor de la solución óptima

$CI(RS')$: Valor del CI de la red social después de aplicar la solución óptima.

$$CI(RS') = \begin{cases} CI(RS) & Si\ R\_max = 0 \\ \min(CI(RS_0), CI(RS_1), .., CI(RS_{a-1})) & Si\ Es_{a,b} > R\_max \\ \min(CI(RS_0), CI(RS_1), .., CI(RS_a)) & Si\ Es_{a,b} \leq R\_max \end{cases}$$

Algoritmo de costo de la solución óptima

INICIO

Entrada: redSocial
Salida: Valor del conflicto interno después de aplicar la solución óptima a la red social de la entrada.

sag <- Secuencia de grupos de agentes
cIInicial <- CI de la red social inicial
cantidadGrupos <- Cantidad de grupos de la red social

esfuerzos <- Matriz que en cada celda contiene el esfuerzo de cambiar la opinion de cierta cantidad de personas en cierto grupo.
esfuerfoMaximo <- Valor máximo con el que se cuenta para moderar
matrizCI <- Matriz con cantidadGrupos cantidad de filas y esfuerzoMaximo+1 cantidad de columnas

```
PARA i <- 0 HASTA cantidadGrupos HACER
    cantCambiar <- 0

    PARA j <- 0 HASTA esfuerzoMaximo+1 HACER
        esfuerzoSiguiente = esfuerzos[i][cantCambiar+1]

    MIENTRAS QUE j = esfuerzoSiguiente HACER
        cantCambiar + 1: Aumentar la cantidad de agentes a los que se le puede cambiar la opinion
        esfuerzoSiguiente = esfuerzos[i][cantCambiar+1]: Conocer el esfuerzo de cambiar la opinion de un agente más

    FIN_MIENTRAS_QUE

    SI i = 0 y cantCambiar > 0 ENTONCES
        e <- Vector con la solución parcial, inicializada con cantidadGrupos 0's
        e[0] <- cantCambiar
        redModificada <- ModCI(redSocial, e)
        conflictoVarible <- calcularCI(redModificada)
        matrizCI[j][i] <- conflictoVariable

    FIN_SI

    SINO SI i > 0 ENTONCES

        izquierda = matrizCI[j][i-1]

        SI cantCambiar = 0 ENTONCES

            matrizCI[j][i] = izquierda

        FIN_SI

        SINO ENTONCES
            valoresComparar <- [[izquierda, solucionParcial]]

            PARA k <- 0 HASTA cantCambiar+1 HACER
```

```
                        posRef <- j-esfuerzos[i][k]
                        e <- Solución obtenida hasta el momento
                        e[i] <- k


                        redModificada <- ModCI(redSocial, e)
                        valorComparar = calcularCI(redModificada.sag)
                        valoresComparar.agregar([valorComparar, e])
                FIN_PARA

                ciMinimo <- valor minimo comparando la posición 0 de valoresComparar
                matrizCI[j][i] = ciMinimo[0]

            FIN_SINO
        FIN_SINO

        FIN_PARA
FIN_PARA

Devolver salida

FIN
```

Algoritmo de la solución óptima

```
INICIO

Entrada: redSocial
Salida: Solución optima.

sag <- Secuencia de grupos de agentes
cIInicial <- CI de la red social inicial
cantidadGrupos <- Cantidad de grupos de la red social
esfuerzos <- Matriz que en cada celda contiene el esfuerzo de cambiar la opinion de cierta cantidad de personas en cierto grupo.
esfuerfoMaximo <- Valor máximo con el que se cuenta para moderar
matrizSolucion <- Matriz con cantidadGrupos cantidad de filas y esfuerzoMaximo+1 cantidad de columnas

PARA i <- 0 HASTA cantidadGrupos HACER
        cantCambiar <- 0

        PARA j <- 0 HASTA esfuerzoMaximo+1 HACER
```

```
            esfuerzoSiguiente = esfuerzos[i][cantCambiar+1]

        MIENTRAS QUE j = esfuerzoSiguiente HACER
            cantCambiar + 1: Aumentar la cantidad de agentes a los que se le puede cambiar la opinion
            esfuerzoSiguiente = esfuerzos[i][cantCambiar+1]: Conocer el esfuerzo de cambiar la opinion de un agente más

        FIN_MIENTRAS_QUE

        SI i = 0 y cantCambiar > 0 ENTONCES
            e <- Vector con la solución parcial, inicializada con cantidadGrupos 0's
            e[0] <- cantCambiar
            redModificada <- ModCI(redSocial, e)
            conflictoVarible <- calcularCI(redModificada)
            matrizSolucion[j][i] = e

        FIN_SI

        SINO SI i > 0 ENTONCES

            izquierda <- CI de la red social aplicando la solución de la izquierda

            SI cantCambiar = 0 ENTONCES

                matrizSolucion[j][i] = matrizSolucion[j][i-1]

            FIN_SI

            SINO ENTONCES
                valoresComparar <- [[izquierda, matrizSolucion[j][i-1]]]

                PARA k <- 0 HASTA cantCambiar+1 HACER

                    posRef <- j-esfuerzos[i][k]
                    e <- matrizSolución[posRef][i-1] (Copia)
                    e[i] <- k

                    redModificada <- ModCI(redSocial, e)
                    valorComparar = calcularCI(redModificada.sag)
                    valoresComparar.agregar([valorComparar, e])
                FIN_PARA
```

ciMinimo <- valor minimo comparando la posición 0 de valoresComparar
matrizSolucion[j][i] = ciMinimo[1]

FIN_SINO
FIN_SINO

FIN_PARA
FIN_PARA

Devolver salida

FIN

Complejidad

Complejidad Temporal

Para definir la complejidad temporal debemos definir las variables que la van a caracterizar:

- ***n*** = cantidad de grupos de agentes de la red
- ***R_max*** = Esfuerzo máximo que se puede aplicar sobre la red
- ***n_i_max*** = Cantidad máxima de agentes en un grupo de la red. Puesto que cantidad de agentes varía por grupo, la complejidad se calcula en el caso pesimista donde todos los grupos tienen la misma cantidad de agentes
- ***k*** = Representa una constante, un valor que no depende de las variables definidas anteriormente. $O(n) = O(1)$, $O(kh) = O(h)$ y $O(h+k) = O(h)$

Algunas de las instrucciones del algoritmo son llamados a otras funciones que no tienen complejidad $O(1)$, estas son las siguientes:

- ***calcularCI(red)***: Función auxiliar que recibe el SAG de una red y calcula el conflicto interno de esta. Contiene un ciclo for que recorre la lista de grupos de agentes, la cual tiene un tamaño n, así que es $O(n)$.
- ***obtenerNuevaRed(redSocial, e)****:* Función auxiliar que recibe una red social y una lista e con las cantidades de agentes a cambiar en cada grupo. A partir de estos genera una nueva red social RS'. Contiene un ciclo for que recorre la lista de grupos de agentes, la cual tiene un tamaño n, así que es $O(n)$.
- ***matrizEsfuerzo(redSocial):*** Función auxiliar que recibe el SAG de una red y calcula el esfuerzo necesario para cambiar cada cantidad posible de agentes para cada grupo. Esto se realiza con el fin de disminuir los cálculos realizados, pues después de construida se pueden referenciar las celdas por posición lo que tiene un costo $O(1)$. Contiene dos ciclos for anidados, uno que va de 0 a n y otro que va de 0 a n_i;

como el valor de ni es diferente para cada grupo, se maneja el caso pesimista reemplazando n_i por n_i_max. Así que la complejidad es $O(n*n_i\_max)$

- ***min(lista):*** Función que retorna el menor valor de un conjunto de datos que recibe como parámetro. En este algoritmo, se usa en una lista cuyo tamaño máximo es n_i_max+1, así que la complejidad es $O(n_i\_max+1)$ o $O(n_i\_max+k)$ que es igual a $O(n_i\_max)$

Con estas definiciones, podemos representar la complejidad temporal del algoritmo de la siguiente manera:

$$O(n) + O(n * n_i\_max) + k + (n * R\_max + 1) + (n * R\_max + 1 * n) +$$
$$\sum_{0}^{n}[\ \sum_{0}^{R\_max+1}[\ \sum_{0}^{n_i\_max}(k) + max\{(k + O(n) + O(n)), (k + max\{(k), (k + \sum_{1}^{n_i\_max+1}($$
$$k + O(n) + O(n)\ ) + O(n_i\_max))\})\}]\ ]$$

Los $max\{(i_1), (i_2)\}$ representan los condicionales if, donde si se cumple una condición de ejecuta un conjunto de instrucciones y si no se cumple se ejecuta otro conjunto. Como O representa la complejidad del caso pesimista, debemos analizar ambos conjuntos de instrucciones y realizar el cálculo con el conjunto de mayor complejidad.

Resolviendo, obtenemos:

$$O(n * (1 + n_i\_max)) + O(n * R\_max) + O(n^2 * R\_max) + \sum_{0}^{n}[\ \sum_{0}^{R\_max+1}[\ \sum_{0}^{n_i\_max}(k) +$$
$$max\{(2 * O(n)), (max\{(k), (\sum_{1}^{n_i\_max+1}(2 * O(n)) + O(n_i\_max))\})\}]]$$

Simplificando y solucionando los max obtenemos:

$$O(n * n_i\_max) + O(n^2 * R\_max) + \sum_{0}^{n}[\ \sum_{0}^{R\_max+1}[\ \sum_{0}^{n_i\_max}(k) + \sum_{1}^{n_i\_max+1}(O(n)) + O(n_i\_max)]]$$

Y luego al resolver: $O(n * n_i\_max) + O(n^2 * R\_max) + \sum_{0}^{n}[\ \sum_{0}^{R\_max+1}[(k * n_i\_max + 1)$
$+ (O(n) * n_i\_max) + O(n_i\_max)]]$

Simplificamos lo que está en la sumatoria interna:

$$O(n * n_i\_max) + O(n^2 * R\_max) + \sum_{0}^{n}[\ \sum_{0}^{R\_max+1}[O(n_i\_max) + (O(n) * n_i\_max)]]$$

Resolvemos las sumatorias:

$O(n * n_i\_max) + O(n^2 * R\_max) + n * R\_max + 1 * (O(n_i\_max) + (O(n) * n_i\_max))$

Y nos da: $O(n * n_i\_max) + O(n^2 * R\_max) + (O(n) * O(R\_max) * O(n * n_i\_max))$

Luego obtenemos: $O(n * n_i\_max) + O(n^2 * R\_max) + O(n^2 * R\_max * n_i\_max)$

Que es igual a $O(n^2 * R\_max * n_i\_max)$

Concluimos que la complejidad temporal es $O(n^2*R\_max*n_i\_max)$

Complejidad Espacial

Para determinar la complejidad espacial, debemos determinar las estructuras de datos utilizadas y sus respectivos tamaños. Las estructuras que se crean al inicio del algoritmo y se utilizan durante toda la ejecución son:

- ***esfuerzos:*** Es una lista que almacena el esfuerzo necesario para cambiar cada cantidad posible de agentes para cada grupo de la red. Tiene tamaño $n*n_i$, así que su complejidad es $O(n*n_i\_max)$
- ***matrizCI:*** Lista donde se van a almacenar los conflictos internos de las soluciones parciales. Tiene tamaño $n*R\_max+1$, así que su complejidad es $O(n*R\_max)$
- ***matrizSolucion:*** Lista donde se van a almacenar las soluciones parciales. Tiene tamaño $n*R\_max+1*n$, así que su complejidad es $O(n^2*R\_max)$

Las estructuras temporales que solo se usan en una porción de algoritmo son:

- ***e:*** Es una lista que almacena la cantidad de agentes a cambiar de cada grupo. Tiene tamaño n, así que su complejidad es $O(n)$
- **arregloValores:** Lista que almacena las tuplas con las posibles cantidades de agentes a cambiar y la solución parcial que se lleva hasta esa cantidad. Tiene longitud n y cada celda tiene una lista de longitud 2, donde la primera posición es un entero y la segunda posición es una lista de longitud n.Su complejidad es $O(n_i\_max)$ *(O(1)+O(n)),* lo que es igual a $O(n_i\_max*n)$

Entonces podemos decir que la complejidad espacial del algoritmo dinámico es:

$O(n * n_i\_max) + O(n * R\_max) + O(n^2 * R\_max) + O(n) + O(n * n_i\_max)$

Que es equivalente a :

$2 * O(n * n_i\_max) + O(n * R\_max) + O(n^2 * R\_max) + O(n)$

Simplificando obtenemos:

$O(n * (R\_max + n_i\_max)) + O(n^2 * R\_max) + O(n)$

Y finalmente obtenemos como resultado:

$O(n * (R\_max + n_i\_max)) + O(n^2 * R\_max)$

¿Es útil en la práctica?

- **Tiempo:** La complejidad temporal del algoritmo es $O(n^2 * R\_max * n_i\_max)$. Si $n = 2^k$, entonces la complejidad se puede escribir como $O(2^{2k} * R\_max * n_i\_max)$. Si tenemos un computador que procesa $3 * 10^8$ operaciones por minuto, y vamos a considerar como aceptable un tiempo máximo de un año. Un año se puede representar como 518.400 minutos

  Entonces debemos hallar el valor de k tal que $2^{2k} * R\_max * ni\_max <= 518400$, que es equivalente a $4^k * R\_max * ni\_max <= 518400$. Resolvemos la inecuación, obtenemos que $4^k <= \frac{518400}{R\_max * ni\_max}$, aplicamos logaritmo para despejar la exponencial: $k <= log_4(\frac{518400}{R\_max * ni\_max})$. Si R_max y ni_max fueran iguales a 1, tendríamos que $k <= 9,49$, pero de acuerdo a los casos analizados, el valor de R_max puede ser mucho mayor. Si tomamos R_max = 121.144 y ni_max = 24 (Valores máximos con los que se hicieron pruebas) obtenemos que $k <= -1,243$, que es un valor que en la realidad no tiene sentido, pues no pueden haber menos de un grupo de agentes. Ya que ni_max toma valores considerablemente menores a R_max, vamos a analizar que valor debe tomar R_max para que $log_4(\frac{518400}{R\_max}) = 0$, de tal manera que obtengamos un n de 1. Resolviendo obtenemos que R_max=518.400; entonces podemos concluir que $k <= 9,49$ y el máximo valor de R_max para que el problema sea factible es 518.400.

- **Espacio:** La complejidad espacial del algoritmo es $O(n * (R\_max + n_i\_max)) + O(n^2 * R\_max)$ Si $n = 2^k$, entonces la complejidad se puede escribir como $O(2^k * (R\_max + n_i\_max)) + O(2^{k^2} * R\_max)$, y por propiedades de los exponentes es igual a $O(2^k * (R\_max + n_i\_max)) + O(4^k * R\_max)$. Vamos a considerar como aceptable

un uso de 12 GB (tomando como referencia un computador de 16 GB de RAM donde al menos 4 GB son ocupadas por el sistema operativo) o 12.884.901.888 Bytes, que vamos a aproximar como $12 * 10^9$ Bytes. Las celdas de las estructuras de datos almacenan enteros, y un entero tiene 4 Bytes de tamaño (o 0,05 MB), entonces debemos hallar el valor de k tal que $(2^k * (R\_max + n_i\_max)) + (4^k * R\_max) <= 12 * 10^9$. R_max y ni_max fueran iguales a 1, tendríamos que $2^{k+1} + 4^k <= 12 * 10^9$, al despejar k obtenemos $k <= log_2(\sqrt{(12 * 10^9 + 1)} - 1$ que es equivalente a $k <= 16,74$ .Si tomamos R_max = 121.144 y ni_max = 24 (Valores máximos con los que se hicieron pruebas) obtenemos que $(2^k * 121168) + (4^k * 121144) <= 12 * 10^9$, y al despejar k obtenemos $k <= 8,3$

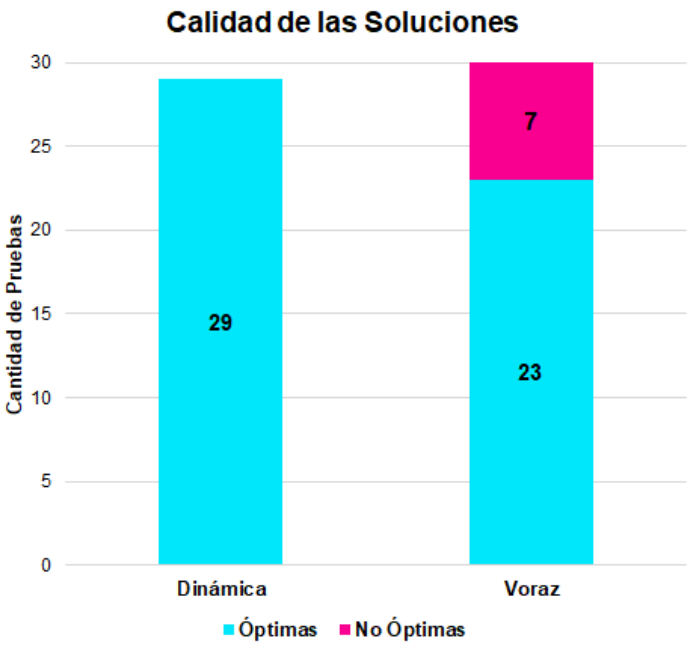
COMPARACIÓN DE RESULTADOS

Para realizar la comparación de las soluciones y los tiempos de ejecución de los algoritmos se creó una hoja de cálculo para recopilar la información, a partir de la cual se crearon gráficos que ayudaran a ilustrar nuestras conclusiones.

Calidad

La calidad de la solución se determinó para los algoritmos Voraz y Dinámico al comparar las soluciones obtenidas con las óptimas. Las soluciones óptimas de referencia fueron las obtenidas por el algoritmo de Fuerza Bruta, para los casos es que este fue posible ejecutarse, y el valor de la solución proporcionado como parte de la batería de pruebas para los casos restantes.

Obtenemos que el algoritmo dinámico generó la solución óptima para los 29 casos que se pudo probar, que es el 100% y el algoritmo voraz generó la solución óptima para 23 de los 30 casos, que es el 76,7%. Sin embargo, esto puede sugerir que el error de este algoritmo es del 23,3%, y esto no es correcto.



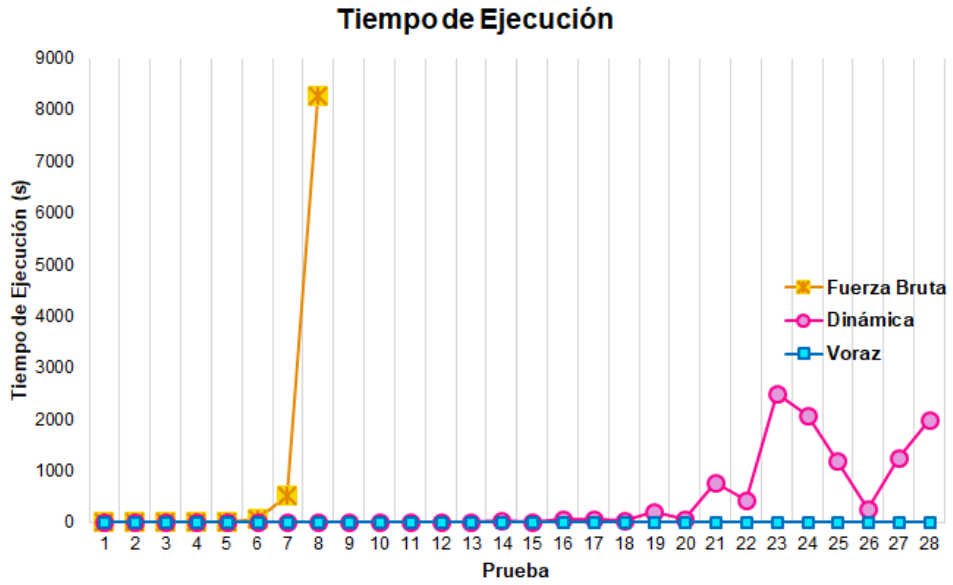
Para calcular el error del algoritmo debemos comparar los valores de la solución generados con los óptimos. Entonces se calcula, para cada prueba, el resultado de restar uno menos la división entre el valor de la solución óptima y el valor de la solución voraz. Para las 23 soluciones óptimas el error es cero, para las restantes los errores son los siguientes.

Prueba	Error
8	0,49%
11	1,15%
14	0,17%
18	0,02%
20	1,14%
27	0,03%
28	0,78%

Al promediar los errores de las 30 pruebas obtenemos un error promedio del 0,13%.

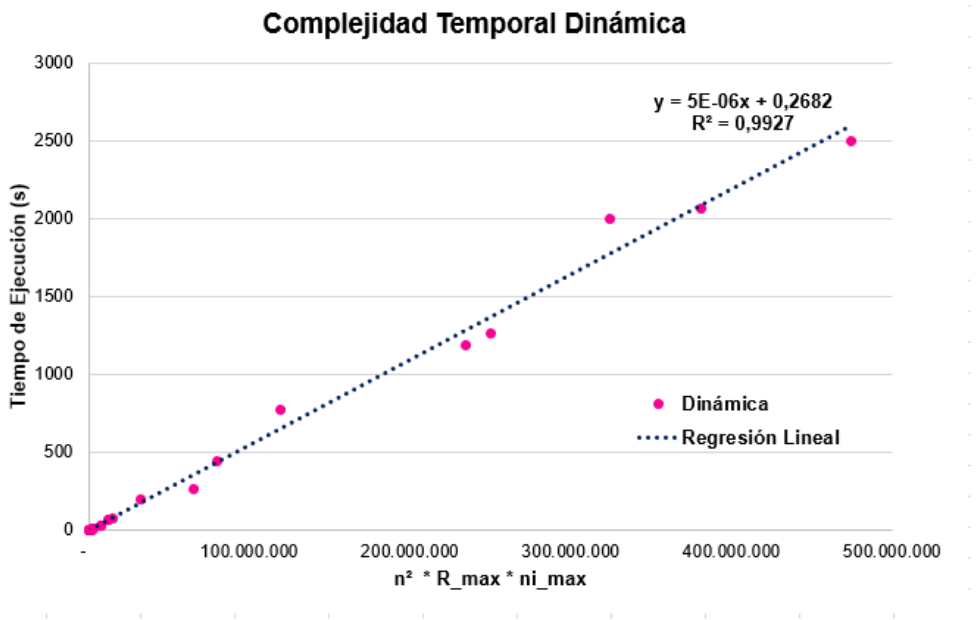
Tiempo

Para analizar y comparar los tiempos realizamos una gráfica de los tiempos de ejecución de cada algoritmo para cada prueba.

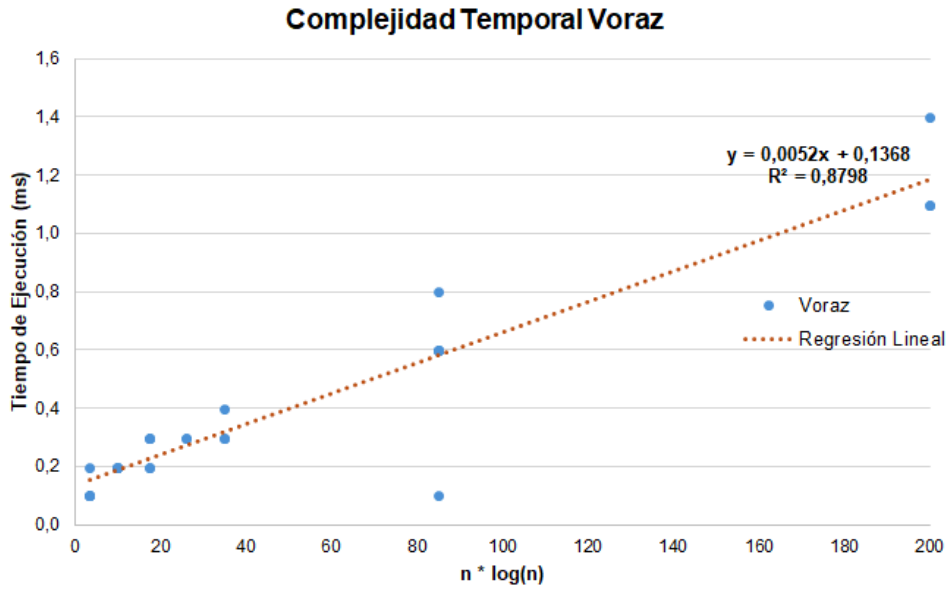


Podemos observar que cuando las variables de decisión toman valores pequeños (primeras pruebas), los 3 algoritmos tienen un tiempo de ejecución similar; pero cuando los valores de las variables aumentan, los algoritmos de fuerza bruta y dinámico empiezan a crecer, el algoritmo de fuerza bruta con un crecimiento más rápido que el dinámico. Sin embargo el algoritmo voraz no se ve afectado, tomando un milisegundo en ejecutar la misma prueba que al dinámico le toma 6,6 horas. Esto es debido a que el algoritmo de fuerza bruta tiene una complejidad exponencial, mientras que el dinámico tiene complejidad n^2*R_max*n_i_max.

Además de comparar las soluciones entre sí, también podemos ver si la complejidad temporal teórica se refleja en los datos reales, ya que depende de variables cuyo valor conocemos para cada prueba. En el eje x se escribe el resultado de la complejidad (Que es n^2*R_max*n_i_max para el algoritmo dinámico) y en el eje y los tiempos de ejecución. El gráfico resultante debe tener un comportamiento aproximadamente lineal, lo cuál se evidencia en este caso. El algoritmo se puede aproximar a una función lineal con una calidad de 99,27%

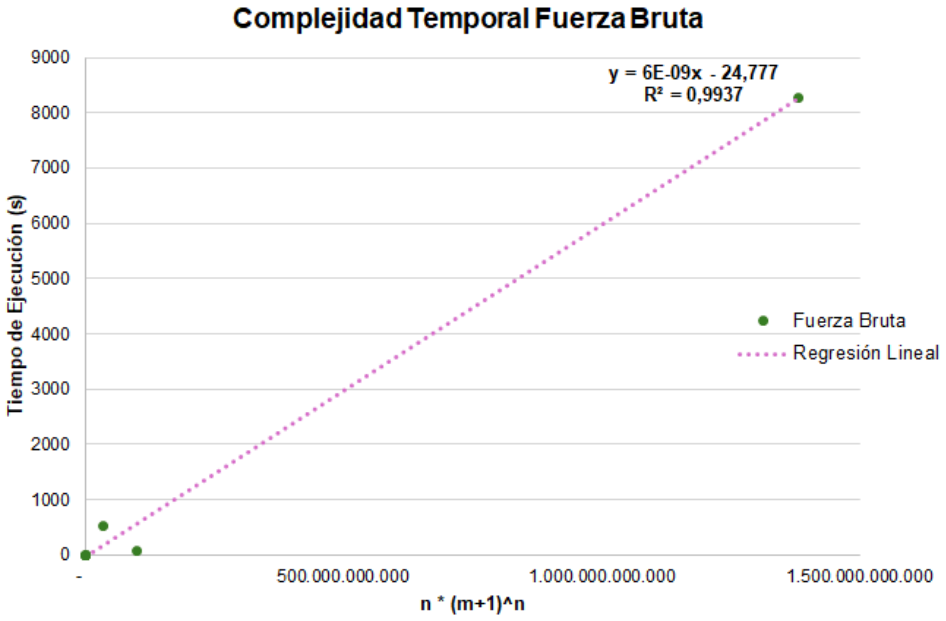


Lo mismo se hace para el algoritmo voraz, poniendo en el eje x su complejidad de nlogn. También se realiza una regresión lineal con una calidad de 87,98%, menor que la solución dinámica. Esto se debe a que n va de 5 a 100 en estas pruebas, por lo que nlogn va de 3 a 200, y estos valores son muy pequeños para analizar el comportamiento. Podemos ver que el máximo tiempo de ejecución es 1,6 milisegundos, y con estos valores tan pequeños hay mucha susceptibilidad al scheduling que implementa el sistema operativo y otras razones en general que no dependen directamente del programa en ejecución.

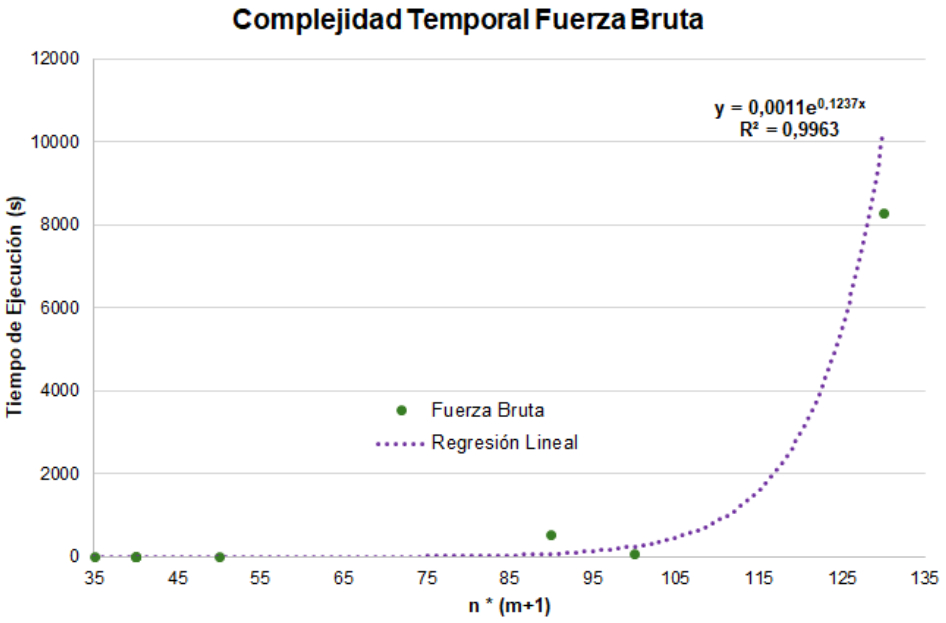


Con el algoritmo de fuerza bruta tenemos el problema opuesto, la complejidad es n*(m+1)^n lo que la hace exponencial, entonces los tiempo de ejecución crecen tan rápidamente que solo pudimos ejecutar las primeras 11 pruebas, y al graficar y hallar la

regresión lineal obtenemos una pendiente tan pequeña (Del orden de 10^{-9}) que no provee muchas información.



Entonces utilizamos una estrategia diferente. En el eje x escribimos los valores de n*(m+1) de cada prueba, y realizamos una aproximación a una función exponencial. La función obtenida no es igual a la complejidad real puesto que es de la forma e^{an} en vez de $(m+1)^n$, pero nos muestra efectivamente el tipo crecimiento de la función y por eso la calidad de la aproximación es de 99,63%



Pruebas propias

Se ejecutaron las cuatro pruebas propias previamente descritas en la estrategia voraz, con el objetivo de comparar los resultados obtenidos con el enfoque dinámico y de fuerza bruta. Los resultados se presentan en la siguientes tablas:

Fuerza bruta

Prueba	Solución	¿Solución óptima?	Tiempo
Prueba Propia 1	$esfuerzo = 18000$ $ci = 980$ $e = [4, 5, 2, 0, 5]$	Si	0.287
Prueba Propia 2	$esfuerzo = 25201$ $ci = 800$ $e = [0, 0, 5, 0, 1]$	Si	0.296
Prueba Propia 3	$esfuerzo = 80000$ $ci = 100$ $e = [5, 0, 5, 5, 0]$	Si	0.06
Prueba Propia 4	$esfuerzo = 0$ $ci = 194$ $e = [5, 5, 5, 5, 5]$	Si	0.234

Dinámica

Prueba	Solución	¿Solución óptima?	Tiempo
Prueba Propia 1	$esfuerzo = 18000$ $ci = 980$ $e = [0, 0, 4, 5, 0]$	Si	0.273
Prueba Propia 2	$esfuerzo = 25201$ $ci = 800$ $e = [0, 0, 5, 0, 1]$	Si	0.289
Prueba Propia 3	$esfuerzo = 80000$ $ci = 100$ $e = [5, 0, 5, 5, 0]$	Si	0.0078
Prueba Propia 4	$esfuerzo = 0$ $ci = 194$ $e = [5, 5, 5, 5, 5]$	Si	0.310

Voraz

Prueba	Solución	¿Solución óptima?	Tiempo
Prueba Propia 1	$esfuerzo = 18000$ $ci = 980$ $e = [5, 5, 1, 0, 5]$	Si	0.210
Prueba Propia 2	$esfuerzo = 25201$ $ci = 800$ $e = [0, 0, 5, 0, 1]$	Si	0.12
Prueba Propia 3	$esfuerzo = 80000$ $ci = 100$ $e = [5, 0, 5, 5, 0]$	Si	0.0052
Prueba Propia 4	$esfuerzo = 0$ $ci = 194$ $e = [5, 5, 5, 5, 5]$	Si	0.02

En los tres casos se obtiene la solución óptima, y los tiempos de ejecución resultan ser bastante similares entre sí. Se puede afirmar que los tres algoritmos se comportan similar para estos casos de pruebas.

CONCLUSIONES

Se logró diseñar satisfactoriamente las tres estrategias de programación requeridas para la moderación del conflicto interno en una red social. Los enfoques de fuerza bruta y programación dinámica demostraron ser exactos y correctos, ya que garantizan la solución óptima en cualquier caso. Por otro lado, el algoritmo voraz mostró un buen desempeño, aunque con menor confiabilidad, presentando un error promedio del 0,13 % en las pruebas realizadas. Esto indica que el enfoque voraz no siempre asegura una solución óptima.

Por otra parte, la complejidad temporal de la estrategia voraz es mejor en comparación con las estrategias dinámica y de fuerza bruta. Su desempeño se mantiene estable independientemente de la cantidad de grupos de agentes en la red social, mientras que en el caso del algoritmo de fuerza bruta, el tiempo de ejecución crece de manera exponencial. Por su parte, el algoritmo dinámico, al tener una complejidad polinomial, presenta inicialmente un tiempo de ejecución estable, pero cuando el número de grupos de agentes supera los 50, comienza a crecer de forma considerable.

En cuanto a la complejidad espacial, tanto la estrategia voraz como la de fuerza bruta presentan una complejidad lineal, mientras que la estrategia dinámica posee una complejidad polinomial.

Por ende, la estrategia voraz resulta más beneficiosa que las demás, ya que tiene una baja complejidad y un nivel de exactitud bastante alto. La estrategia dinámica es también una buena opción, ya que garantiza soluciones óptimas en todos los casos; sin embargo, sus tiempos de ejecución se vuelven muy altos para redes sociales de gran tamaño. Finalmente, la estrategia de fuerza bruta resulta muy ineficiente en términos de tiempo de ejecución, por lo que se emplea únicamente como referencia debido a que siempre encuentra la solución correcta.