Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo práctico 3
Comunidades NP-Completas

5 de mayo de 2025

Barletta Brenda
112184

Docampo Torrico Daniel Rodolfo
112395

Rivas Sofia Belen
112216

1. Consideraciones

Este informe presenta el análisis y desarrollo correspondiente al Trabajo Práctico N.º 3, titulado *"Comunidades NP-Completas"*, cuyo eje central es abordar un problema de particionado de grafos desde una perspectiva algorítmica y teórica. El objetivo es evaluar la factibilidad de separar un grafo no dirigido y no pesado en comunidades (o *clusters*) de manera que se minimice la distancia máxima entre vértices dentro de cada uno.

1.1. Contexto del problema

En el marco de una narrativa ficticia, nos encontramos colaborando con una organización que busca identificar potenciales divisiones internas dentro de su red de contactos. A partir de información obtenida sobre relaciones de cercanía entre miembros, se construye un grafo cuyas aristas representan vínculos entre personas. Nuestro objetivo es agrupar a estos individuos en comunidades cohesivas, siguiendo criterios estructurales que garanticen una baja dispersión interna.

En un principio, se consideró maximizar la distancia mínima entre comunidades, lo cual podía resolverse eficientemente. Sin embargo, esta métrica fue descartada por su escasa utilidad práctica. En su lugar, se decidió trabajar con una métrica mucho más exigente: minimizar la distancia máxima dentro de cada comunidad. Esta reformulación transforma el problema en uno considerablemente más complejo, y se sospecha que pertenece a la clase de problemas NP-Completos.

1.2. Consigna

El trabajo requiere abordar múltiples enfoques para estudiar el problema desde distintas perspectivas:

- Demostrar que el problema de **Clustering por Bajo Diámetro** pertenece a NP.
- Probar que dicho problema es **NP-Completo**, mediante una reducción adecuada desde un problema conocido.
- Implementar una solución exacta por **backtracking** para la versión de optimización, minimizando la peor distancia máxima intra-cluster.
- Generar instancias de prueba para validar la correctitud del algoritmo y realizar mediciones de tiempo.
- Proponer e implementar un **modelo de Programación Lineal** para resolver el problema, utilizando distancias precomputadas.
- Comparar empíricamente el desempeño de ambos enfoques exactos (backtracking y PL).
- Analizar una **estrategia alternativa** basada en la **Modularidad** de la red, implementando el algoritmo *greedy* de Louvain.
- Estudiar la calidad de las soluciones obtenidas por Louvain como aproximación al clustering por bajo diámetro, especialmente sobre instancias de gran tamaño.
- (Opcional) Explorar otras heurísticas o algoritmos aproximados, comparando su rendimiento.

1.3. Objetivos del informe

Este trabajo tiene como propósito principal el estudio integral de un problema de particionado en grafos desde una perspectiva algorítmica, con énfasis en su dificultad computacional (NP) y en la comparación entre distintos enfoques resolutivos. En particular, se busca:

- Justificar formalmente la pertenencia del problema a la clase NP y demostrar su NP-Completitud.
- Diseñar e implementar algoritmos exactos y aproximados, argumentando sus decisiones técnicas.
- Evaluar empíricamente los tiempos de ejecución y la calidad de las soluciones para distintas configuraciones del problema.
- Reflexionar sobre la aplicabilidad real de cada enfoque y las limitaciones prácticas del clustering por bajo diámetro.
- Documentar el proceso de desarrollo, validación y análisis con el fin de garantizar la reproducibilidad de los resultados.

2. Análisis del problema

El problema abordado consiste en particionar un grafo no dirigido y no pesado en k comunidades disjuntas, de forma tal que se minimice la mayor distancia entre pares de vértices dentro de cada cluster. Esta formulación —denominada **Clustering por Bajo Diámetro**— impone una métrica restrictiva de cohesión interna para los grupos, y presenta un desafío computacional significativo.

2.1. Enfoque inicial y entendimiento del problema

Desde una primera lectura, se identificó que el objetivo consiste en asignar cada vértice a exactamente uno de los k clusters, respetando dos condiciones principales:

- Todo vértice debe pertenecer a un único cluster.
- La distancia máxima entre cualquier par de vértices dentro de un mismo cluster debe ser mínima entre todas las posibles particiones.

Se evaluó inicialmente una formulación basada en la versión de decisión del problema, que consiste en verificar si existe una partición válida en a lo sumo k clusters tal que el diámetro de cada uno sea $\leq C$, para un valor de C dado. Esta versión resulta clave para analizar su pertenencia a la clase NP.

El enfoque adoptado implica observar el grafo como una estructura donde los vértices deben agruparse bajo restricciones estrictas de conectividad interna. Esto difiere sustancialmente de otras métricas de clustering más permisivas (como la modularidad), y orienta el análisis hacia técnicas de resolución exactas y análisis de complejidad.

2.2. Pertinencia al conjunto NP y NP-Completitud

A partir del análisis del problema en su versión de decisión, se comprobó que este pertenece a la clase **NP**, ya que, dada una solución candidata (una partición de vértices), es posible verificar en tiempo polinomial que:

- Cada vértice pertenece a exactamente un cluster.
- Se utilizan a lo sumo k clusters.
- La distancia máxima dentro de cada cluster es menor o igual a C.

Dicho proceso fue implementado en un validador que, utilizando búsquedas BFS, computa las distancias internas de cada grupo. Como consecuencia, se concluye que la validación de cualquier solución propuesta puede realizarse en tiempo polinomial respecto al tamaño del grafo.

Por otra parte, se demostrará que el problema es **NP-Completo** mediante una reducción polinomial desde el problema de **Coloreo de Grafos**, el cual es conocido por ser NP-Completo. En particular, se mostrará que, para el caso especial en el que se exige un diámetro máximo igual a 1, el problema de clustering se comporta de manera análoga a una asignación de colores, donde los nodos conectados no pueden compartir grupo. Esta equivalencia justifica formalmente su NP-Completitud.

2.3. Algoritmo exacto por backtracking con podas

Dado que el objetivo es minimizar la mayor distancia interna entre vértices de cualquier cluster, se desarrolló un algoritmo exacto utilizando la técnica de **backtracking con poda**. Esta estrategia recorre recursivamente todas las posibles asignaciones de vértices a clusters, incorporando múltiples condiciones de poda para evitar ramificaciones innecesarias:

- **Poda por simetría**: evita explorar configuraciones equivalentes mediante ordenamiento de clusters.
- **Poda por desbalance**: evita clusters que crezcan desproporcionadamente respecto al resto.
- **Poda por imposibilidad**: si los vértices restantes no alcanzan para completar los clusters vacíos, se descarta la rama.
- **Poda por optimalidad parcial**: se evita continuar una asignación si el diámetro parcial ya excede al óptimo conocido.

El algoritmo garantiza encontrar la solución óptima para instancias de tamaño reducido o moderado. La evaluación del diámetro de cada cluster se realiza mediante un recorrido BFS entre todos los pares de vértices de un mismo grupo, lo que permite determinar su distancia interna máxima.

2.4. Modelo de Programación Lineal

Además del algoritmo exacto por backtracking, se propuso una estrategia alternativa basada en la formulación del problema como un modelo de **Programación Lineal Entera**. Este modelo permite abordar el problema desde una perspectiva de optimización exacta, utilizando un solver para encontrar una partición factible con la menor cantidad posible de clusters bajo una cota de distancia C.

Para construir el modelo, se definieron dos tipos de variables binarias:

- $x_{v,i} \in \{0,1\}$: vale 1 si el vértice v es asignado al cluster i, 0 en caso contrario.
- $y_i \in \{0,1\}$: vale 1 si el cluster i es utilizado, 0 en caso contrario.

La función objetivo busca minimizar la cantidad total de clusters activos:

$$\text{mín} \sum_{i=1}^{k} y_i$$

Las restricciones impuestas sobre el modelo son las siguientes:

1. **Cobertura total**: cada vértice debe ser asignado a exactamente un único cluster.

$$\sum_{i=1}^{k} x_{v,i} = 1 \quad \forall v \in V$$

2. **Activación de cluster**: un cluster se considera utilizado si al menos un vértice le fue asignado.

$$x_{v,i} \leq y_i \quad \forall v \in V, \forall i \in \{1, \ldots, k\}$$

3. **Límite de clusters**: no se permite usar más de k clusters.

$$\sum_{i=1}^{k} y_i \leq k$$

4. **Restricción de diámetro**: dos vértices cuya distancia sea mayor a C no pueden estar en el mismo cluster.

$$x_{u,i} + x_{v,i} \leq 1 \quad \forall i, \forall u, v \in V \text{ tal que } d(u,v) > C$$

Se calculó previamente la matriz de distancias entre todos los pares de vértices mediante BFS, lo que permitió agregar únicamente aquellas restricciones relevantes al modelo (aquellos pares cuya distancia supera C).

El modelo tiene una complejidad combinatoria importante: en el peor caso, el número de variables es $|V| \cdot k + k$, y las restricciones pueden llegar a ser del orden de $O(k \cdot |V|^2)$, aunque en la práctica esto se reduce gracias a la poda previa de pares inválidos.

2.5. Estrategia alternativa: Optimización de la modularidad

Dado que la métrica de diámetro resulta difícil de manejar para grandes volúmenes de datos, se propone estudiar un enfoque alternativo basado en la **modularidad**, una métrica que premia la densidad de conexiones internas dentro de cada comunidad.

Se considerará la implementación del **Algoritmo de Louvain**, una heurística greedy ampliamente utilizada en el campo de detección de comunidades. Aunque este algoritmo no está diseñado para minimizar diámetros, se analizará empíricamente su comportamiento como estrategia aproximada para el problema original.

A diferencia de los enfoques exactos propuestos (backtracking y programación lineal), que buscan explícitamente minimizar la cantidad de clusters manteniendo acotada la distancia máxima entre vértices de cada comunidad, el algoritmo de Louvain optimiza una función objetivo distinta: la modularidad. Esto representa un cambio significativo en la formulación del problema, ya que deja de lado la minimización directa del diámetro y el número de clusters, para favorecer la **cohesión estructural interna** de las comunidades.

Al incorporar esta heurística, no se buscó una solución óptima bajo las métricas originales, sino explorar una perspectiva alternativa que nos permite comparar distintos criterios de calidad de partición y valorar la eficiencia y utilidad de enfoques aproximados en instancias donde los métodos exactos resultan computacionalmente costosos, o no escalan bien.

2.6. Relación con problemas clásicos

El problema analizado presenta conexiones conceptuales con varios problemas combinatorios ampliamente estudiados:

- **Coloreo de grafos**: la restricción de diámetro uno equivale a una prohibición de asignación conjunta a nodos adyacentes, estableciendo un paralelismo directo con el coloreo.
- **K-partitioning con restricciones**: el problema requiere segmentar un conjunto bajo criterios estructurales precisos.
- **Problemas de cobertura mínima**: se relaciona con estructuras donde se busca cubrir un conjunto respetando propiedades internas (conectividad, cohesión, etc.).

Este contexto reafirma la complejidad y el interés algorítmico del problema planteado, justificando la exploración de soluciones tanto exactas como heurísticas.

2.7. Condiciones para una solución válida

Una solución será considerada válida si cumple con todas las siguientes condiciones:

- La partición contiene a lo sumo k clusters disjuntos.
- Cada vértice del grafo pertenece a un único cluster.
- No se repiten vértices ni se omiten en la asignación.
- La distancia máxima entre cualquier par de vértices dentro de cada cluster es menor o igual que un valor C, o es la mínima posible (según la versión del problema).

2.8. Algoritmo Greedy para maximizar la modularidad

Como alternativa, se implementó un algoritmo greedy que busca maximizar la **modularidad**, una métrica que evalúa la densidad de conexiones internas dentro de las comunidades.

El procedimiento parte de una partición trivial, donde cada vértice forma su propia comunidad. En cada iteración, se consideran todas las posibles fusiones de pares de comunidades, se calcula el incremento en modularidad que implicaría cada fusión, y se selecciona aquella que produzca la mayor mejora. Esta operación se repite hasta que no se logre incrementar más la modularidad total del grafo.

Si bien esta estrategia no controla explícitamente el diámetro de los clusters ni el número final de comunidades, se utilizó como punto de comparación empírica frente a los modelos exactos. En particular, se evaluó el diámetro interno de las comunidades generadas una vez obtenida la partición, observando si respetaban —o no— el umbral C establecido en el planteo original.

3. Demostración de pertenencia a NP

Para demostrar que el problema de **Clustering por Bajo Diámetro** pertenece a la clase NP, es suficiente con exhibir un *validador polinomial*, es decir, un algoritmo que permita verificar, en tiempo polinomial, si una solución propuesta es válida para una instancia dada del problema.

3.1. Versión de decisión del problema

La versión de decisión del problema puede enunciarse de la siguiente manera:

> Dado un grafo no dirigido y no ponderado $G = (V, E)$, un número entero k, y un valor C, ¿existe una partición de V en a lo sumo k clusters disjuntos tal que:
>
> - Cada vértice pertenezca a un único cluster,
> - Todos los vértices estén asignados (la partición es total),
> - Y que la distancia máxima entre cualquier par de vértices dentro de cada cluster sea menor o igual a C?

3.2. validador polinomial

Para verificar una solución candidata, se implementó un algoritmo que realiza las siguientes comprobaciones:

1. Que se utilicen a lo sumo k clusters.
2. Que todos los vértices estén presentes, y que ninguno esté repetido.
3. Que el diámetro (la máxima distancia entre pares de nodos) dentro de cada cluster no supere el valor C.

La verificación de la condición 3 se realiza calculando la distancia entre todos los pares de vértices de cada cluster mediante un algoritmo BFS. En caso de que no exista camino entre dos vértices del mismo cluster, se asume una distancia infinita y se rechaza la solución.

3.3. Descripción del algoritmo validador

El algoritmo `validador_clustering` toma como entrada un grafo $G = (V, E)$, un número máximo de clusters k, un umbral de diámetro permitido C, y una solución candidata representada como un diccionario que asigna vértices a clusters. Su propósito es verificar si la solución cumple con las condiciones de validez del problema de Clustering por Bajo Diámetro.

El procedimiento consta de los siguientes pasos:

1. **Cantidad de clusters:** se verifica que la solución no utilice más de k clusters. Si se supera ese número, la solución es rechazada.
2. **Cobertura sin repeticiones:** se construye un conjunto `usados` que contiene todos los vértices asignados. Durante la iteración:
   - Si un vértice aparece más de una vez (es decir, si ya está en el conjunto), se rechaza la solución.
   - Si, al finalizar, el conjunto de vértices utilizados no coincide exactamente con el conjunto de vértices del grafo, la solución también es rechazada, ya que implica que hay vértices sin asignar o en exceso.

3. **Verificación de diámetros internos:** para cada cluster, se calcula la distancia máxima entre pares de vértices utilizando un recorrido BFS. Esta operación se realiza mediante una función auxiliar llamada `calcular_distancia_max_cluster`. Si la distancia máxima encontrada en un cluster excede el valor C, la solución se rechaza.

4. **Aprobación:** si se cumplen todas las condiciones anteriores, la solución se considera válida y el algoritmo retorna `True`.

3.4. Código del validador

A continuación se muestra la implementación utilizada para verificar soluciones candidatas al problema, incluida en el archivo `validador.py`:

```python
def validador_clustering(grafo, k, c, solucion_propuesta):

    if len(solucion_propuesta) > k:
        return False

    usados = set()
    total_vertices = set(grafo.obtener_vertices())

    for cluster, vertices in solucion_propuesta.items():
        for vertice in vertices:
            if vertice not in usados:
                usados.add(vertice)
            else:
                return False

    if usados != total_vertices:
        return False

    for cluster, vertices in solucion_propuesta.items():
        distancia_maxima = calcular_distancia_max_cluster(grafo, vertices)
        if distancia_maxima > c:
            return False

    return True
```

3.5. Complejidad del validador

El tiempo de ejecución del validador es polinomial en el tamaño del grafo:

- Verificar que no haya vértices repetidos ni omisiones se puede hacer en $O(V)$.
- Para cada cluster, se calcula la distancia entre pares de vértices con BFS, cuya complejidad es $O(V + E)$ por par.
- Como hay a lo sumo k clusters, cada uno con a lo sumo V vértices, el número de pares por cluster es $O(V^2)$.

En conjunto, la complejidad total del validador es:

$$O(k \cdot V^2 \cdot (V + E))$$

3.6. Conclusión

Dado que la validación de una solución propuesta puede realizarse en tiempo polinomial con respecto al tamaño de la entrada, se concluye que el problema de Clustering por Bajo Diámetro pertenece a la clase **NP**.

4. Demostración de NP-Completitud

En esta sección se demostrará que el problema de **clustering por bajo diámetro** es **NP-Completo**, mediante una reducción en tiempo polinomial desde el problema de **k-coloreo**, el cual es conocido por ser NP-Completo.

4.1. Versión de decisión del problema

Entrada: Un grafo no dirigido y no pesado $G = (V, E)$, un número entero k (cantidad máxima de clusters), y un número entero C (diámetro máximo permitido dentro de cada cluster).

Pregunta: ¿Existe una partición de V en a lo sumo k clusters disjuntos $C_1, C_2, \ldots, C_k$, tal que la distancia máxima entre cualquier par de vértices dentro de cada cluster C_i sea a lo sumo C?

4.2. Reducción desde k-coloreo

Dado que el problema de **k-coloreo** es NP-Completo, reduciremos este problema a una instancia del problema de clustering por bajo diámetro. La instancia del problema de k-coloreo es:

Entrada: Un grafo no dirigido $G = (V, E)$, y un entero p (cantidad máxima de colores).

Pregunta: ¿Existe una asignación de colores $f : V \to \{1, \ldots, p\}$ tal que ningún par de vértices adyacentes tenga el mismo color?

4.3. Construcción de la instancia reducida

A partir de una instancia de k-coloreo (G, p), se construye una instancia del problema de clustering por bajo diámetro (G', k, C) de la siguiente forma:

- $G' = G$ (el mismo grafo original).
- $k = p$ (la cantidad de clusters es igual a la cantidad de colores).
- $C = 0$ (el diámetro máximo permitido en cada cluster es cero).

4.4. Demostración en una dirección (ida)

Si k-coloreo tiene solución, entonces clustering por bajo diámetro también.

Supongamos que existe una coloración válida $f : V \to \{1, \ldots, p\}$ del grafo G, es decir, tal que $f(u) \neq f(v)$ para todo $(u, v) \in E$.

Definimos los clusters de la instancia del problema de clustering por bajo diámetro como:

$$C_i = \{v \in V \mid f(v) = i\}, \quad \text{para } i = 1, \ldots, p$$

Dado que ningún par de vértices adyacentes comparte el mismo color, dentro de cada conjunto C_i no hay aristas: cada cluster es un conjunto independiente.

Bajo la definición adoptada, si un cluster no contiene aristas internas, se considera que su diámetro máximo es 0, ya sea porque tiene un solo vértice o porque la distancia entre cualquier par de vértices es infinita (y se toma como válida para $C = 0$).

Por lo tanto, esta partición en clusters cumple con todas las condiciones del problema de clustering por bajo diámetro con $k = p$ y $C = 0$.

4.5. Demostración en la otra dirección (vuelta)

Si clustering por bajo diámetro tiene solución, entonces k-coloreo también.

Supongamos que existe una partición de V en a lo sumo p clusters $C_1, C_2, \ldots, C_k$, tal que la distancia máxima dentro de cada cluster es ≤ 0.

Esto implica:

- Si un cluster contiene uno o cero vértices, su diámetro es 0 por definición.
- Si un cluster contiene dos o más vértices, la única forma de que su diámetro sea 0 es que no haya aristas entre ninguno de sus vértices. Es decir, cada cluster es un conjunto independiente.

A partir de esta partición, definimos una coloración del grafo original G asignando un color distinto a cada cluster. Como ningún cluster contiene aristas internas, ningún par de vértices adyacentes comparte el mismo color. Por lo tanto, esta asignación constituye una coloración válida con a lo sumo p colores.

4.6. Conclusión

Se ha demostrado que una instancia del problema de k-coloreo puede transformarse en una instancia equivalente del problema de clustering por bajo diámetro con $C = 0$, y viceversa, en tiempo polinomial.

Como k-coloreo es NP-Completo, y clustering por bajo diámetro pertenece a la clase NP (ver sección anterior), concluimos que el problema de **clustering por bajo diámetro es NP-Completo**.

5. Algoritmo por Backtracking

Con el objetivo de encontrar una solución óptima al problema de Clustering por Bajo Diámetro, se implementó un algoritmo exacto basado en **backtracking con múltiples podas**. El criterio de optimalidad es minimizar el mayor de los diámetros entre todos los clusters generados.

5.1. Descripción general

Dado un grafo no dirigido y no pesado $G = (V, E)$ y un número entero k, el problema consiste en particionar el conjunto V en k clusters disjuntos, de modo tal que cada vértice pertenezca a un único cluster y se minimice el peor caso de cohesión interna:

$$\min_{\text{partición}} \left(\max_{1 \leq i \leq k} \text{diam}(C_i) \right)$$

Donde $\text{diam}(C_i)$ es la distancia máxima entre pares de vértices del cluster C_i.

5.2. Descripción detallada de la función `clustering_bt`

La función `clustering_bt(grafo, vertices, actual, sol_optima, sol_temporal, k)` implementa la lógica central del algoritmo. Utiliza una estrategia de exploración en profundidad para recorrer el espacio de asignaciones posibles de vértices a clusters. A su vez, incorpora podas agresivas que permiten reducir la cantidad de configuraciones exploradas.

Caso base

Cuando se asignaron todos los vértices (`actual >= len(vertices)`), se evalúa la partición construida:

- Se calcula el diámetro máximo actual de los clusters en `sol_temporal`.
- Si es mejor que el de la mejor solución conocida hasta el momento (`sol_optima`), se actualiza esta última.

Caso recursivo

Se selecciona el vértice en la posición actual y se prueba su asignación a cada uno de los clusters disponibles. Para cada intento de asignación, se aplican sucesivamente las siguientes podas:

1. **Poda por simetría:** si se intenta asignar un vértice a un cluster vacío y ya se intentó con otro cluster vacío en una posición anterior, la rama se descarta. Esto evita explorar permutaciones equivalentes (por ejemplo, intercambiar nombres de clusters sin cambiar la estructura).
2. **Poda por insuficiencia de vértices:** si la cantidad de vértices restantes no alcanza para llenar todos los clusters vacíos, no vale la pena continuar esa rama. Se calcula cuántos clusters vacíos quedan y se compara con la cantidad de vértices aún no asignados.
3. **Poda por desbalance:** se evita que un cluster contenga más vértices de los necesarios. Para eso se calcula un máximo permitido por cluster: $\left\lceil \frac{|V|}{k} \right\rceil$.
4. **Poda por subóptimo:** si el diámetro parcial de la solución temporal ya supera el diámetro de la mejor solución conocida hasta el momento, no se continúa explorando esa rama.

Si ninguna de las podas se activa, se hace una llamada recursiva a `clustering_bt` con el siguiente vértice. Al retornar, se actualiza la mejor solución si la nueva partición es mejor.

5.3. Ordenamiento previo

Antes de comenzar la búsqueda, los vértices se ordenan por grado (cantidad de vecinos) en orden decreciente. Los vértices con mayor conectividad imponen mayores restricciones y conviene asignarlos primero, ya que podrían aumentar tempranamente el diámetro de un cluster.

5.4. Evaluación de soluciones

Para cada asignación completa, se utiliza la función `calcular_mayor_diametro_cluster`, que invoca BFS para determinar el diámetro máximo entre los clusters. El valor óptimo es el menor de estos máximos.

5.5. Complejidad teórica

La complejidad en el peor caso es exponencial, dado que se exploran todas las posibles asignaciones de vértices a clusters. Sin embargo, gracias a las podas implementadas, el algoritmo logra resolver eficientemente instancias de tamaño pequeño y mediano.

El orden de búsqueda se corresponde con una permutación de asignaciones de n vértices a k clusters, pero el espacio de búsqueda real es significativamente más pequeño debido a las restricciones internas del problema.

5.6. Código del problema

A continuación se presenta el fragmento principal de la función implementada:

```
def clustering_bt(grafo, vertices, actual, sol_optima, sol_temporal, k):
    if actual >= len(vertices):
        diametro_actual = calcular_mayor_diametro_cluster(grafo, sol_temporal)
        if clusters_vacios(sol_optima) or diametro_actual <
    calcular_mayor_diametro_cluster(grafo, sol_optima):
            return copy.deepcopy(sol_temporal)
        else:
            return sol_optima

    vertice = vertices[actual]
    mejor_solucion = sol_optima
    indice_cluster = 0

    for cluster, v in sol_temporal.items():
        if len(v) == 0 and es_cluster_vacio_anterior(sol_temporal, indice_cluster):
            indice_cluster += 1
            continue

        sol_temporal[cluster].append(vertice)
        vertices_restantes = len(vertices) - (actual + 1)

        if not alcanzan_vertices(vertices_restantes, sol_temporal):
            sol_temporal[cluster].pop()
            continue

        if cluster_desbalanceado(sol_temporal, len(vertices), k):
            sol_temporal[cluster].pop()
            continue

        if not clusters_vacios(sol_optima):
            diametro_parcial = calcular_mayor_diametro_cluster(grafo, sol_temporal)
            diametro_optimo = calcular_mayor_diametro_cluster(grafo, sol_optima)
            if diametro_parcial >= diametro_optimo:
                sol_temporal[cluster].pop()
                continue

        incluyendo = clustering_bt(grafo, vertices, actual + 1, mejor_solucion,
    sol_temporal, k)
```

```
37          diametro_incluyendo = calcular_mayor_diametro_cluster(grafo, incluyendo)
38          diametro_mejor_solucion = calcular_mayor_diametro_cluster(grafo,
    mejor_solucion)
39
40          if clusters_vacios(mejor_solucion) or diametro_incluyendo <
    diametro_mejor_solucion:
41              mejor_solucion = copy.deepcopy(incluyendo)
42
43          sol_temporal[cluster].pop()
44          indice_cluster += 1
45
46      return mejor_solucion
```

5.7. Algoritmo por Programación Lineal

5.7.1. Entrada del algoritmo.

La función recibe los siguientes parámetros:

- Un grafo $G = (V, E)$, representado como una estructura que expone `obtener_vertices()`.
- Un entero $k \in \mathbb{N}$, que representa la cantidad máxima de clusters permitidos.
- Un valor umbral $C \in \mathbb{R}$, que determina la distancia máxima permitida entre vértices del mismo cluster.
- Una matriz de distancias $d : V \times V \to \mathbb{R}$, precalculada mediante BFS.

5.7.2. Variables de decisión.

Se definen las siguientes variables binarias:

- $x_{v,i} \in \{0, 1\} \quad \forall v \in V, \forall i \in \{1, \ldots, k\}$: indica si el vértice v pertenece al cluster i.
- $y_i \in \{0, 1\} \quad \forall i \in \{1, \ldots, k\}$: indica si el cluster i fue utilizado.

La cantidad total de variables del modelo es:

$$\underbrace{|V| \cdot k}_{\text{Variables } x_{v,i}} + \underbrace{k}_{\text{Variables } y_i}$$

5.7.3. Función objetivo.

El objetivo es minimizar la cantidad de clusters utilizados:

$$\text{mín} \sum_{i=1}^{k} y_i$$

5.7.4. Restricciones.

Se incorporan al modelo las siguientes restricciones:

1. **Asignación única:**
$$\sum_{i=1}^{k} x_{v,i} = 1 \quad \forall v \in V$$
Cada vértice debe ser asignado exactamente a un único cluster.

2. **Activación del cluster:**
$$x_{v,i} \leq y_i \quad \forall v \in V, \forall i \in \{1, \ldots, k\}$$
Un cluster se considera utilizado si al menos un vértice es asignado a él.

3. **Cantidad máxima de clusters:**
$$\sum_{i=1}^{k} y_i \leq k$$
Se garantiza que no se utilicen más de k clusters.

4. **Restricciones de diámetro:**

   Para cada par de vértices (u, v) tal que $d(u, v) > C$, se impide que ambos estén en el mismo cluster:

   $$x_{u,i} + x_{v,i} \leq 1 \quad \forall i \in \{1, \ldots, k\}, \forall (u, v) \in V^2 \text{ tal que } d(u, v) > C$$

   Estas restricciones aseguran que ningún cluster contenga pares de vértices cuya distancia geodésica supere el umbral permitido.

5.7.5. Cantidad total de restricciones.

El número de restricciones impuestas al modelo es:

$$\underbrace{|V|}_{\text{Asignación única}} + \underbrace{|V| \cdot k}_{\text{Activación}} + \underbrace{1}_{\text{Límite de clusters}} + \underbrace{k \cdot |\{(u, v) \in V^2 : d(u, v) > C\}|}_{\text{Distancia prohibida}}$$

5.7.6. Mejoras computacionales.

Para optimizar la construcción del modelo se aplicaron las siguientes técnicas:

- Se considera solo un subconjunto de pares (u, v) tal que $u < v$, reduciendo la duplicación de restricciones simétricas.
- La matriz $d(u, v)$ se calcula por única vez mediante BFS para cada nodo $v \in V$, con complejidad $O(|V| \cdot (|V| + |E|))$.
- Solo se generan restricciones de distancia para los pares donde $d(u, v) > C$, lo cual reduce significativamente el espacio de búsqueda cuando C es grande.

5.7.7. Extracción de la solución.

Una vez resuelto el modelo, se reconstruyen los clusters observando las variables con valor 1:

$$(v \in V) \wedge (x_{v,i} = 1) \Rightarrow v \in \text{cluster } i$$

Se genera así una partición disjunta $\mathcal{C} = \{C_1, \ldots, C_k\}$ con los clusters no vacíos.

5.7.8. Observaciones finales.

El enfoque de programación lineal entera permite obtener soluciones óptimas y validables para el problema de clustering bajo restricciones de cohesión estructural estrictas. No obstante, su aplicabilidad práctica está limitada por la explosión combinatoria en instancias grandes, dado que el número de restricciones crece en el peor caso con $O(k \cdot |V|^2)$.

6. Algoritmo de Louvain

Para abordar instancias de gran escala en el problema de *Clustering por Bajo Diámetro*, se incorporó el algoritmo de Louvain como una **estrategia heurística**, aprovechando su eficiencia en al detección de comunidades en grafos grandes.

6.1. Descripción General

El algoritmo busca particionar un grafo en comunidades (clusters) disjuntas tales que la **modularidad** de la partición sea máxima. La modularidad es una métrica que mide la densidad de enlaces dentro de las comunidades, comparada con la esperada si las conexiones fueran aleatorias.

El objetivo principal del algoritmo no está alineado directamente con la métrica de la distancia máxima dentro de cada cluster, sino con la cohesión estructural general del grafo. No obstante, se utilizó su salida como una **aproximación práctica** al problema original, evaluando empíricamente la calidad de las particiones encontradas (en términos de cantidad de clusters generados y diámetro máximo observado).

6.2. Fundamento teórico

El algoritmo de Louvain intenta maximizar la modularidad Q de una partición de un grafo G. Esta métrica evalúa qué tan bien dividida está la estructura del grafo en comunidades densamente conectadas internamente, y con pocas conexiones hacia el exterior.

La modularidad se define como:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

Donde:

- m es el número total de aristas en el grafo (o la suma de pesos si es ponderado).
- A_{ij} es el valor de la arista entre i y j (1 si existe, 0 si no).
- k_i es el grado del nodo i.
- c_i es la comunidad a la que pertenece el nodo i.
- $\delta(c_i, c_j)$ es 1 si $c_i = c_j$, y 0 en otro caso.

6.3. Descripción del enfoque y diferencias clave

A diferncia de las estrategias exactas, el algoritmo de Louvain:

- No busca minimizar la cantidad de clusters ni el diámetro máximo explícitamente.
- No utiliza como entrada un valor fijo de k, el número de comunidades surge de la **maximización de la modularidad**.
- Permite obtener resultados en tiempos considerablemente menores, incluso en grafos de gran tamaño.

Pese a estas diferencias, se evaluó empíricamente qué tan buenas fueron las particiones generadas por el algoritmo de Louvain en términos del objetivo original: minimizar el diámetro máximo restringiendo a un valor k la cantidad de clusters.

6.4. Descripción del algoritmo implementado

El algoritmo implementado se estructura en dos fases iterativas:

1. **Fase de mejora local:** cada vértice se mueve al cluster vecino que más aumente la modularidad. Si no hay mejora, se conserva su posición. Esta fase termina cuando no se logra ninguna mejora adicional.
2. **Fase de agregación:** se construye un nuevo grafo donde cada comunidad detectada se convierte en un nuevo nodo. Se repite entonces la fase 1 sobre este nuevo grafo, y así sucesivamente hasta alcanzar un punto fijo.

Estas fases se repiten hasta que no haya más mejoras en la modularidad global. El resultado final es una partición del conjunto de vértices en comunidades que, en teoría, reflejan estructuras cohesivas del grafo.

6.5. Primera fase: optimización local

Inicialmente, cada nodo se considera como su propia comunidad. Luego, se exploran los vecinos de cada nodo y se calcula el cambio en la modularidad (ΔQ) al trasladarlo a la comunidad de cada uno. El noto se mueve a la comunidad que produzca el mayor aumento de modularidad, o se queda en su comunidad si el cambio no implica una mejora.

El valor (ΔQ) puede calcularse eficientemente, mediante la siguiente fórmula:

$$\Delta Q = \left[\frac{\text{sum\_in} + 2k_i^{in}}{2m} - \left(\frac{\text{sum\_tot} + k_i}{2m}\right)^2\right] - \left[\frac{\text{sum\_in}}{2m} - \left(\frac{\text{sum\_tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

Donde:

- k_i es el grado del nodo i.
- k_i^{in} es la suma de pesos entre el nodo y la comunidad candidata.
- sum_in es la suma de pesos internos de la comunidad.
- sum_tot es la suma de grados de los nodos de la comunidad.

6.6. Segunda fase: agregación

Una vez que no hay más mejoras posibles, se construye un nuevo grafo en el que cada comunidad detectada se convierte en un nuevo nodo.Las aristas entre estas comunidades se ponderan de acuerdo con la suma de las aristas oiginales que conectaban nodos de distintas comunidades.

Luego, se repite la primera fase sobre este nuevo grafo. El algoritmo termina cuando ninguna iteración produce un aumento en la modularidad total.

6.7. Código de la implementación

A continuación, se presenta el fragmento principal del algoritmo implementado:

```
1 def louvain(grafo):
2     nodo_a_comunidad = {v: v for v in grafo.obtener_vertices()}
3     comunidades = {v: set([v]) for v in grafo.obtener_vertices()}
4
5     while True:
```

```
        nodo_a_comunidad, comunidades = primera_etapa(grafo, comunidades,
    nodo_a_comunidad)
        nuevo_grafo, mapeo = segunda_etapa(grafo, comunidades)

        if len(nuevo_grafo.obtener_vertices()) == len(grafo.obtener_vertices()):
            break  # no hubo fusiones nuevas

        grafo = nuevo_grafo
        nodo_a_comunidad = {nodo: comunidad for nodo, comunidad in mapeo.items()}
        comunidades = defaultdict(set)
        for nodo, comunidad in nodo_a_comunidad.items():
            comunidades[comunidad].add(nodo)

    resultado = defaultdict(set)
    for nodo, comunidad in nodo_a_comunidad.items():
        resultado[comunidad].add(nodo)

    return list(resultado.values())
```

6.8. Evaluación de soluciones

A pesar de no estar orientado directamente al problema de clustering por bajo diámetro, el resultado de Louvain se evaluó bajo los mismos criterios que los enfoques exactos:

- Se calcularon los diámetros de cada cluster resultante y se tomó el máximo entre ellos como cota empírica.
- Se registró la cantidad de comunidades generadas.
- Se compararon estos valores con los obtenidos por las estrategias exactas.

Los resultados muestran que Louvain es considerablemente más eficiente en términos de tiempo de ejecución. Mientras que los enfoques exactos requerían segundos, minutos, o incluso no terminaban de ejecutarse en grafos medianos, Louvain fue capaz de procesar todos los casos probados en tiempos casi instantáneos.

Sin embargo, esta mejora en eficiencia computacional viene acompañada de una diferencia significativa en la naturaleza de las soluciones:

- Louvain no recibe como entrada el parámetro k, por lo que el número final de clusters generados es emergente del algoritmo y puede diferir del deseado.
- En varias instancias, Louvain generó **más de k clusters**, como consecuencia de su objetivo de maximizar modularidad, en lugar de controlar explícitamente la cantidad de particiones.
- En otros casos, la cantidad de clusters fue menor a k, lo que puede traducirse en comunidades más grandes, y por ende, con diámetros internos mayores.

6.9. Implicancias sobre el diámetro

La diferencia más relevante observada con respecto a los enfoques exactos es que Louvain, si bien tiende a formar comunidades densamente conectadas (alta cohesión estructural), no necesariamente minimiza el **diámetro máximo** entre pares de vértices en los clusters.

En algunas ejecuciones, particularmente en grafos donde existen nodos de alta centralidad o estructuras tipo estrella, Louvain agrupó varios nodos en una única comunidad, elevando considerablemente su diámetro interno. En cambio, los algoritmos exactos priorizaban siempre minimizar dicho valor, aunque a costa de mayor tiempo de ejecución.

En contraste, si el problema demanda una solución con un número específico de clusters y un control estricto sobre su diámetro interno, los métodos exactos son más apropiados, siempre que el tamaño de las instancias lo permita.

Louvain complementa, por tanto, los otros enfoques: no como reemplazo, sino como alternativa práctica en escenarios de gran escala o como herramienta para obtener soluciones iniciales o aproximadas.

6.10. Ejemplos comparativos

Para ilustrar las diferencias prácticas entre el enfoque exacto basado en backtracking y el algoritmo de Louvain, se presentan a continuación dos ejemplos concretos.

Ejemplo 1: Grafo 22_3.txt

En este grafo pequeño, se evaluó el problema con $k = 10$.

Backtracking

- **Cantidad de clusters:** 10
- **Máxima distancia intra-cluster:** 1
- **Tiempo de ejecución:** 40.25 segundos

Louvain

- **Cantidad de clusters:** 7
- **Máxima distancia intra-cluster:** 2
- **Tiempo de ejecución:** 0.00 segundos

Observaciones

Louvain logró una partición en tiempo instantáneo, sacrificando sin embargo el control sobre la cantidad de clusters y el diámetro interno. En este caso particular, agrupó los vértices en 7 clusters en lugar de 10, lo que llevó a un incremento en la distancia máxima dentro de algunos clusters. Si bien la calidad estructural (en términos de modularidad) puede ser alta, la solución no cumple con las restricciones originales del problema.

Ejemplo 2: Grafo 45_3.txt

En este grafo más grande se utilizó $k = 7$ en el enfoque exacto.

Backtracking

- **Cantidad de clusters:** 7 (como requerido)
- **Máxima distancia intra-cluster:** 3
- **Tiempo de ejecución:** *varios minutos* (incompleto debido a tiempos prolongados)

Louvain

- **Cantidad de clusters:** 13
- **Máxima distancia intra-cluster:** 2
- **Tiempo de ejecución:** 0.01 segundos

Observaciones

En este caso, Louvain generó una cantidad significativamente mayor de clusters (13), excediendo el límite de 7 clusters que restringía a los algoritmos exactos. Esta sobresegmentación condujo a una reducción del diámetro dentro de los clusters, aunque a costa de romper con la restricción del problema original. Por su parte, backtracking tardó varios minutos en encontrar una solución válida respetando exactamente $k = 7$, aunque con un diámetro levemente superior (3).

Conclusiones generales

- Louvain actúa como una heurística rápida, generando comunidades con buena cohesión estructural, pero sin garantizar el cumplimiento de restricciones sobre k ni sobre el diámetro máximo.
- Backtracking garantiza una solución óptima para el problema planteado, respetando exactamente el número de clusters y minimizando el diámetro máximo, pero su costo computacional escala rápido.

Cabe destacar, con el algoritmo de Louvain pudimos obtener resultados extremadamente rápido (pocos segundos) probando con grafos de un volumen muchísimo mayor (por ejemplo, con un grafo conexo compuesto de 100 cliques de 100 vértices cada uno). Claro que, con grafos de este calibre, no podemos hacer una comparación de resultados con el algoritmo de backtracking porque sería prácticamente imposible medirlo (tardaría demasiado, considerando también todos los valores de k con los que podemos probar el algoritmo para el mismo grafo), pero la diferencia en cuanto a eficiencia temporal del algoritmo de Louvain con respecto al algoritmo de backtracking no es menor.

7. Algoritmo Greedy para maximizar la modularidad

Como estrategia alternativa a los enfoques exactos, se implementó un algoritmo **greedy** que busca **maximizar la modularidad**, una métrica comúnmente utilizada en detección de comunidades dentro de grafos. Este enfoque no impone restricciones explícitas sobre el diámetro de las comunidades, pero favorece la agrupación de nodos densamente conectados.

7.1. Definición de modularidad.

La *modularidad* Q es una medida que evalúa la calidad de una partición de un grafo $G = (V, E)$ en comunidades $\{C_1, C_2, \ldots, C_r\}$. Se define como:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

donde:

- A_{ij}: peso de la arista entre los nodos i y j,
- k_i: grado del nodo i,
- m: suma total de pesos en el grafo,
- $\delta(c_i, c_j) = 1$ si i y j pertenecen a la misma comunidad, 0 en caso contrario.

7.2. Regla greedy.

El algoritmo comienza con una partición trivial en la que cada vértice forma su propia comunidad. En cada iteración se consideran todas las posibles fusiones de pares de comunidades existentes. Para cada par (C_i, C_j), se simula la fusión y se calcula el nuevo valor de modularidad Q'. Se selecciona la fusión que produce el mayor *incremento positivo de modularidad*:

$$\Delta Q = Q(\mathcal{C}_{i \cup j}) - Q(\mathcal{C})$$

y se aplica solo si $\Delta Q > 0$. El proceso se repite hasta que ninguna fusión posible mejora la modularidad actual.

7.3. Descripción del algoritmo.

1. Inicialización: cada nodo $v \in V$ forma su propia comunidad $C_v = \{v\}$.
2. Mientras haya una fusión con $\Delta Q > 0$:
   - Calcular el valor de modularidad actual Q.
   - Para cada par (C_i, C_j), simular la fusión y calcular Q'.
   - Seleccionar el par cuya fusión maximiza $\Delta Q = Q' - Q$.
   - Aplicar la fusión si $\Delta Q > 0$.
3. Retornar la partición final.

7.4. Complejidad y comportamiento.

El algoritmo presenta una complejidad combinatoria considerable, ya que en cada iteración se evalúan $O(n^2)$ posibles fusiones y se calcula la modularidad desde cero. Esto resulta en una complejidad general aproximada de:

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

donde n es la cantidad de comunidades actuales (inicialmente $|V|$) y T_m es el tiempo necesario para calcular la modularidad.

7.5. Código del algoritmo planteado

```
def maximizar_modularidad(grafo):
    comunidades = []
    for nodo in grafo.obtener_vertices():
        comunidades.append(set([nodo]))

    total_peso_aristas = sum(grafo.peso_arista(v, w) for v, w in grafo.aristas()) /
     2

    hubo_mejora = True
    while hubo_mejora:
        hubo_mejora = False
        mejor_incremento_modularidad = 0
        mejor_par_a_fusionar = None

        for i in range(len(comunidades)):
            for j in range(i + 1, len(comunidades)):
                comunidad_1 = comunidades[i]
                comunidad_2 = comunidades[j]

                comunidades_fusionadas = []
                for k in range(len(comunidades)):
                    if k != i and k != j:
                        comunidades_fusionadas.append(comunidades[k])
                nueva_comunidad = comunidad_1.union(comunidad_2)
                comunidades_fusionadas.append(nueva_comunidad)

                modularidad_actual = calcular_modularidad(grafo, comunidades,
    total_peso_aristas)
                modularidad_fusionada = calcular_modularidad(grafo,
    comunidades_fusionadas, total_peso_aristas)
                incremento = modularidad_fusionada - modularidad_actual

                if incremento > mejor_incremento_modularidad:
                    mejor_incremento_modularidad = incremento
                    mejor_par_a_fusionar = (i, j)
                    hubo_mejora = True

        if hubo_mejora:
            i, j = mejor_par_a_fusionar
            nueva_comunidad = comunidades[i].union(comunidades[j])
            nuevas_comunidades = []
            for k in range(len(comunidades)):
                if k != i and k != j:
                    nuevas_comunidades.append(comunidades[k])
            nuevas_comunidades.append(nueva_comunidad)
            comunidades = nuevas_comunidades

    return comunidades
```

7.6. Ventajas y limitaciones.

- **Ventajas**: no requiere parámetros externos (como k), y aprovecha la estructura del grafo para agrupar vértices de forma eficiente.
- **Limitaciones**: no garantiza soluciones óptimas ni respeta necesariamente restricciones de diámetro; el algoritmo puede estancarse en óptimos locales.

7.7. Aplicación al problema.

Dado que el problema original busca particiones con cohesión estructural (bajo diámetro), este enfoque permite evaluar si la maximización de modularidad conduce naturalmente a comunidades que cumplan (aunque no estén garantizadas) con la restricción de distancia máxima entre pares de vértices. En particular, se utilizó la función `calcular_distancia_max_cluster` para verificar a posteriori si las comunidades generadas respetaban el umbral C.

7.8. Comparación con el Algoritmo de Louvain

El algoritmo Greedy implementado en este trabajo comparte con Louvain el objetivo de maximizar la **modularidad** en un grafo no dirigido, lo que implica encontrar una partición de los vértices en comunidades densamente conectadas internamente y débilmente conectadas entre sí. No obstante, sus estrategias, su complejidad y su rendimiento varían de manera considerable.

El **algoritmo Greedy** parte de una configuración inicial en la que cada nodo se encuentra en su propia comunidad. En cada iteración, evalúa exhaustivamente todas las posibles fusiones entre pares de comunidades y selecciona aquella que maximiza el incremento de modularidad global. Este proceso continúa hasta que no existe ninguna combinación que mejore la modularidad. Aunque este enfoque garantiza una mejora progresiva, su costo computacional es elevado, ya que implica recalcular la modularidad de múltiples fusiones potenciales en cada ciclo. Su simplicidad conceptual lo vuelve didáctico, pero ineficiente en grafos de gran tamaño.

En contraste, el **algoritmo Louvain** sigue una heurística mucho más eficiente y escalable. Está estructurado en dos fases principales que se repiten iterativamente:

- **Primera etapa (optimización local)**: se considera el traslado de cada nodo a las comunidades de sus vecinos inmediatos, evaluando si dicha reubicación mejora la modularidad. Este procedimiento continúa mientras existan mejoras.
- **Segunda etapa (agrupación)**: se construye un nuevo grafo donde cada comunidad descubierta en la primera fase es un supernodo, y se repite el proceso sobre este grafo reducido.

Esta estrategia jerárquica no solo mejora la eficiencia, sino que también permite detectar comunidades a diferentes niveles de granularidad, capturando estructuras anidadas. Además, Louvain evita recalcular la modularidad completa al trabajar con *delta modularidad*, lo que le otorga una ventaja computacional significativa.

En términos empíricos, Louvain tiende a producir resultados de alta calidad (modularidades elevadas) en tiempos significativamente menores que Greedy, especialmente en grafos grandes o densos. La tabla siguiente resume las principales diferencias entre ambos algoritmos:

Criterio	Greedy	Louvain
Estrategia	Fusión iterativa de comunidades que maximiza modularidad global.	Reasignación local de nodos y agregación jerárquica de comunidades.
Inicialización	Cada nodo en su propia comunidad.	Cada nodo en su propia comunidad.
Evaluación	Considera todas las fusiones posibles entre pares de comunidades.	Solo considera movimientos a comunidades vecinas del nodo.
Optimización	Global, con recomputación completa de modularidad.	Local, usando incrementos (ΔQ) y estructuras auxiliares.
Eficiencia	Costosa en tiempo para grafos medianos o grandes.	Altamente eficiente y escalable.
Escalabilidad	Limitada por crecimiento cuadrático en número de comunidades.	Óptima: puede manejar grafos con millones de nodos.
Detección jerárquica	No.	Sí, mediante grafos agregados por nivel.
Facilidad de implementación	Alta (algoritmo directo).	Media (requiere estructuras adicionales y cálculos incrementales).

Cuadro 1: Comparación entre el algoritmo Greedy y Louvain para detección de comunidades

7.9. Sobre los óptimos alcanzados

En las pruebas realizadas, el algoritmo Louvain demostró alcanzar valores de modularidad consistentemente más altos que el algoritmo Greedy, especialmente en grafos con estructuras comunitarias no triviales. Esto se debe a su capacidad de reorganizar dinámicamente las comunidades mediante reubicaciones locales de nodos, lo que le permite escapar de óptimos locales que limitan al enfoque Greedy. En cambio, Greedy, al basarse únicamente en fusiones globales irreversibles, tiende a consolidar tempranamente comunidades que no necesariamente reflejan la estructura óptima del grafo.

Ambos algoritmos, sin embargo, deben entenderse como *aproximaciones*, ya que la maximización de la modularidad es un problema NP-hard y no se conoce un algoritmo polinomial que garantice el óptimo global. En este contexto, Louvain actúa como una *aproximación más refinada*, al explorar más profundamente el espacio de soluciones posibles mediante optimización local iterativa y agregación jerárquica. Esto le permite acercarse más al óptimo en la práctica, mientras que Greedy suele estabilizarse en soluciones subóptimas por su menor flexibilidad.

En resumen, aunque ninguno de los dos garantiza alcanzar el óptimo global, Louvain representa una estrategia de aproximación más robusta y eficaz, tanto en calidad de la solución como en escalabilidad computacional.

8. Mediciones

En esta sección se realizarán mediciones empíricas con el fin de contrastar la complejidad teórica de los algoritmos implementados con su comportamiento real. Según lo analizado, la complejidad de los algoritmos depende principalmente de dos factores:

- V: Cantidad de vértices del grafo.
- k: Cantidad límite de clusters.

El término dominante en la complejidad de ambos algoritmos (backtracking y programación lineal) es $\mathcal{O}(k^V)$. Por lo tanto, se probará que, al mantener k constante, el crecimiento sea exponencial respecto a V.

Entonces para cada algoritmo se realizaran dos pruebas iniciales:

- Medición de tiempo de ejecución con un k = 3, y V variable.
- Medición de tiempo de ejecución con un k = 6, y V variable.

Los datos se generaron de manera aleatoria creando grafos con n cantidad de vértices y aristas 3n - 1. Las mediciones se realizaron promediando 7 ejecuciones por tamaño para reducir la variabilidad. Es importante destacar que se redujo ese valor (inicialmente era 10) para minimizar tiempos ya que se llegaron a obtener tiempos realmente inviables.

8.1. Análisis para algoritmo Backtracking

8.1.1. Análisis con V variable ($k = 3$ fijo)

En esta primera instancia se midió el tiempo de ejecución para grafos con cantidades de vértices crecientes, manteniendo constante el k máximo de clusters en $k = 3$. Los tamaños utilizados para los grafos fueron: **10, 12, 15, 17, 20**, se tomaron esos valores ya que al crecer exponencialmente se demoraba demasiado con valores más grandes.

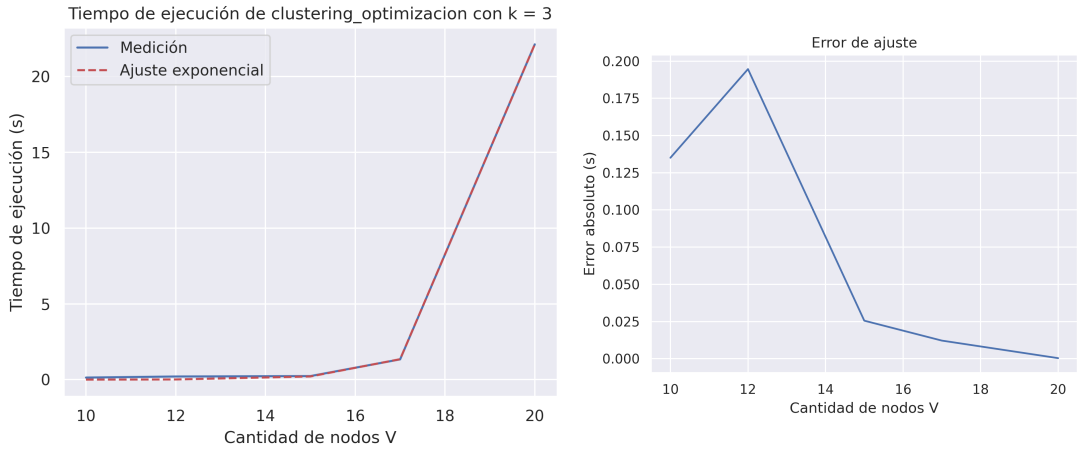

Figura 1: Ajuste exponencial para V variable. Error cuadrático total = **0.05693380316546821**

Como puede observarse, el gráfico presenta una relación exponencial clara respecto del tamaño del grafo y el tiempo de ejecución. Es interesante agregar que realizar la medición del algoritmo tomó 169.42s

8.1.2. Análisis con V variable ($k = 6$ fijo)

En esta segunda prueba se utilizó el mismo set de datos pero con un k mayor, por lo que todos los tiempos resultaron mayores.

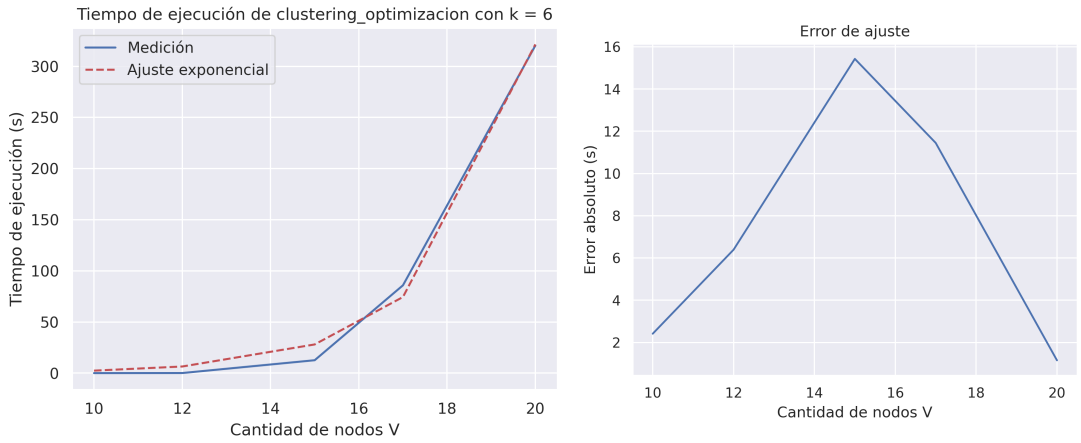

Figura 2: Ajuste exponencial para V variable. Error cuadrático total = **417.07352172676906**

En este caso, el tiempo de ejecución crece de abrupta demorando en total 2933.43s. Se aprecia que el ajuste se adapta correctamente a los datos medidos, validando la hipótesis de que la complejidad es exponencial. Si bien el error cuadrático total es bastante grande, esto se debe a que los tiempos también crecieron, por lo que si bien el gráfico se ajusta a simple vista a la complejidad propuesta, las pequeñas diferencias que se aprecian son también grandes valores.

8.2. Análisis para algoritmo Programación Lineal

8.2.1. Análisis con V variable ($k = 3$ fijo)

En esta primera instancia se midió el tiempo de ejecución para grafos con cantidades de vértices crecientes, manteniendo constante el k máximo de clusters en $k = 3$. Como el enunciado lo indica el set de datos utilizado fue el mismo por lo que los grafos fueron de tamaño **10, 12, 15, 17, 20**.

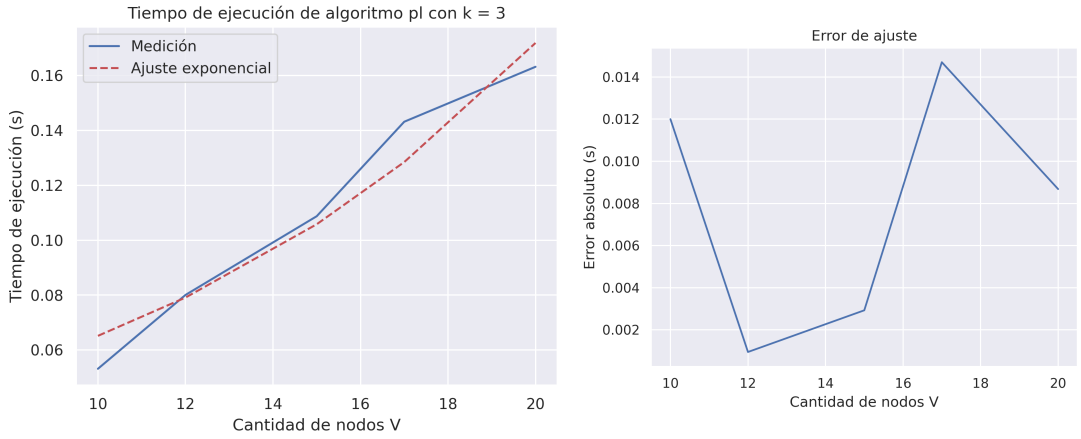

Figura 3: Ajuste exponencial para V variable. Error cuadrático total = **0.0004449706485052979**

Observamos que el gráfico no termina de encajar con el ajuste, aunque el error total tan pequeño

nos hace sospechar. Es por eso que decidimos hacer una prueba con un ajuste logarítmico. El tiempo de ejecución fue de 4.90s.

8.2.2. Análisis con V variable ($k = 3$ fijo) ajuste logarítmico

Mantuvimos el set de datos y k = 3 para ajustar a una función diferente.

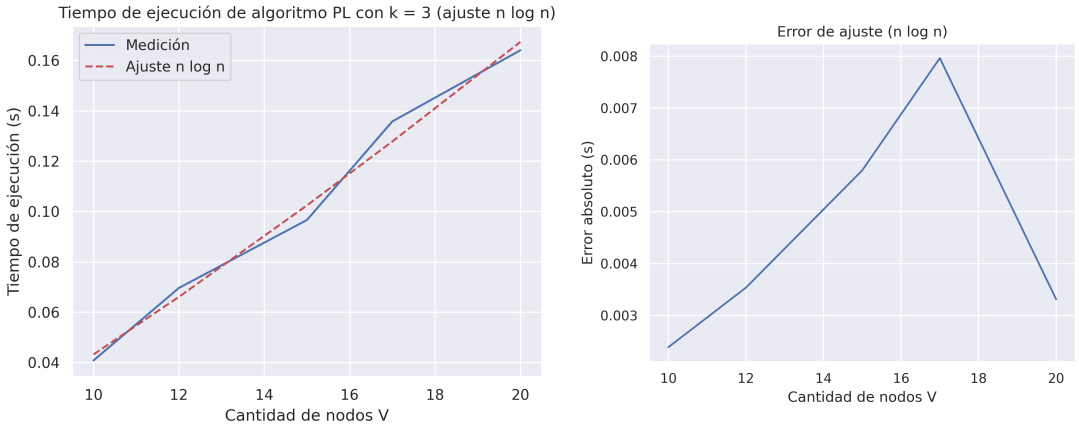

Figura 4: Ajuste logarítmico para V variable. Error cuadrático total = **0.00012624563485940898**

Nuevamente obtenemos un tiempo muy pequeño (4.59s) y un error también pequeño. Pero todavía no podíamos estar seguros de que la complejidad fuera una u otra, por lo que continuamos con las pruebas.

8.2.3. Análisis con V variable ($k = 6$ fijo)

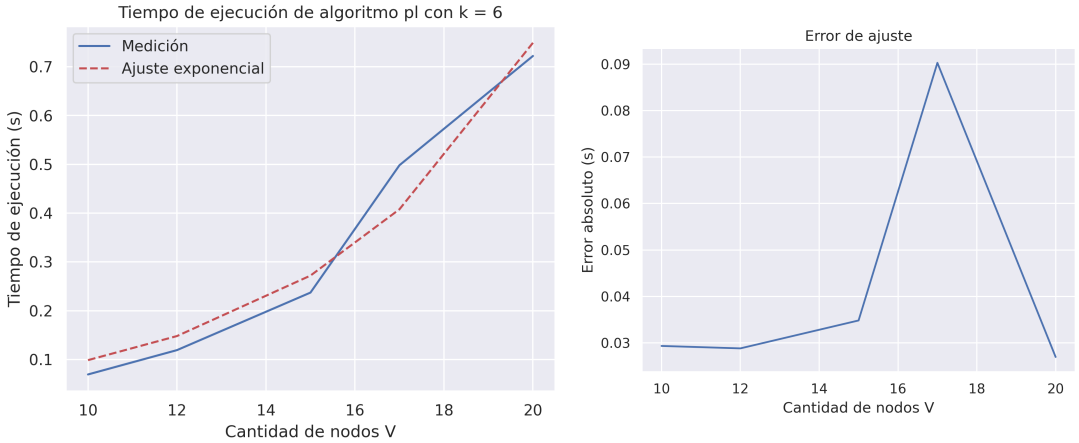

Figura 5: Ajuste exponencial para V variable. Error cuadrático total = **0.011780692777802911**

El tiempo aumenta a 12.55s, lo cual tiene sentido ya que aumentamos el k, sin embargo el gráfico sigue sin darnos una seguridad total sobre si el ajuste es el indicado.

8.2.4. Análisis con V variable ($k = 6$ fijo) ajuste logarítmico

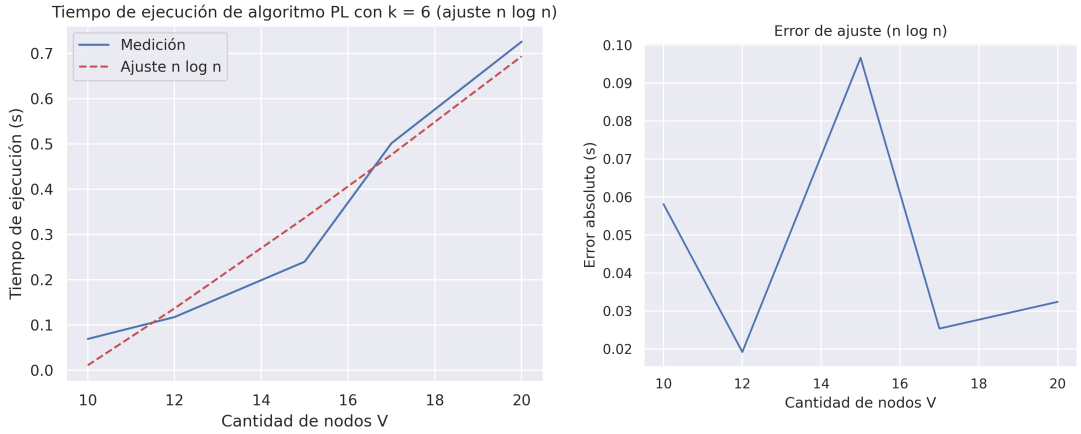

Figura 6: Ajuste logarítmico para V variable. Error cuadrático total = **0.01478876101023637**

Podemos ver que el tiempo es similar al caso anterior, obteniendo 12.80s, lo cual nuevamente es lógico ya que en este caso mantuvimos el k = 6, el error también es bastante similar al caso exponencial. Para tratar de elegir finalmente un ajuste decidimos aumentar nuevamente k.

8.2.5. Análisis con V variable ($k = 9$ fijo)

En este caso al ser un algoritmo más rápido pudimos extender las pruebas a un k mayor para además cerciorarnos de cuál era la función a la que debíamos ajustar la complejidad.

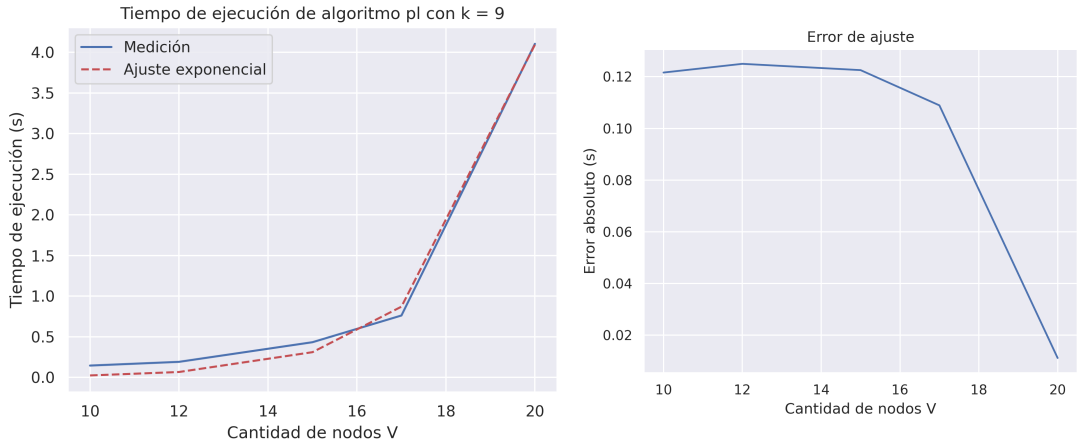

Figura 7: Ajuste exponencial para V variable. Error cuadrático total = **0.057407087759339614**

Obtuvimos un tiempo considerablemente mayor a los casos anteriores (40.55s) y un error que estaba dentro de lo esperado. Además ahora podemos confirmar viendo el gráfico que el ajuste correcto es exponencial de la misma forma que backtracking. Podemos ver como las pendientes van cambiando levemente en un principio para luego dispararse enormemente con un V mayor.

8.3. Análisis para algoritmo de Louvain

8.3.1. Análisis con *cliques* variables (*tamaño* = 100 fijo)

Para este algoritmo realizamos mediciones diferentes ya que su naturaleza lo permite, en este caso lo que variamos fue la cantidad de cliques del grafo y tuvimos un tamaño de 100 para cada clique

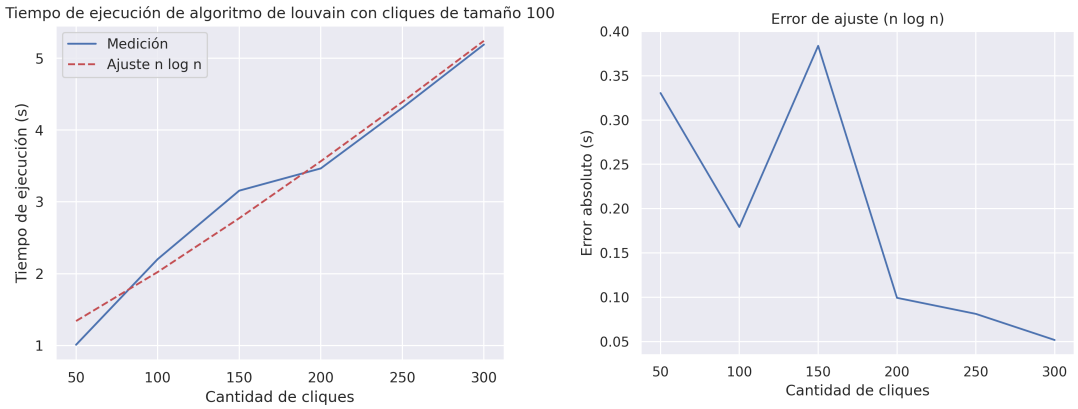

Figura 8: Ajuste exponencial para V variable. Error cuadrático total = **0.30778280201355995**

Observamos que el gráfico encaja bastante bien con el ajuste logarítmico propuesto. El error obtenido es muy pequeño y además algo a destacar es que para un grafo con 25000 vértices demoró 620.94s.

Conclusión

Los resultados obtenidos son coherentes con las complejidades teóricas. Se observó que: El algoritmo de backtracking si bien es exacto no es bueno para grafos de gran o mediano tamaño, mientras que el algoritmo de programación lineal si bien es exponencial es un poco mejor que el anterior ya que demora menos y se adapta mejor a grafos más grandes o mayor cantidad de clusters. Por otro lado, el algoritmo de Louvain es por lejos mucho mejor para grafos de gran tamaño, tanto en términos de escalabilidad como de tiempo de ejecución. Si bien no garantiza encontrar la solución óptima, su enfoque permite obtener resultados de buena calidad en tiempos considerablemente menores. Esto lo convierte en una opción sumamente recomendable para aplicaciones prácticas donde el tamaño del grafo impide utilizar algoritmos exactos o de complejidad alta. En definitiva, la elección del algoritmo adecuado dependerá del tamaño del problema, la necesidad de precisión en la solución y las restricciones de tiempo computacional disponibles.

9. Conclusión

A lo largo del presente informe se abordó el problema de *Clustering por Bajo Diámetro*, una formulación compleja que combina elementos de teoría de grafos, optimización y análisis algorítmico. El objetivo principal fue diseñar estrategias que permitieran particionar un grafo en comunidades disjuntas, minimizando la distancia máxima entre pares de vértices dentro de cada cluster, una métrica particularmente exigente en términos de cohesión estructural interna.

El trabajo incluyó tanto el estudio teórico del problema como el desarrollo e implementación de distintas soluciones. Se justificó formalmente que el problema pertenece a la clase **NP**, y se demostró su **NP-Completitud** mediante una reducción desde el problema clásico de k-coloreo. Esta fundamentación permitió enmarcar el problema dentro del conjunto de desafíos algorítmicos más complejos y motivó la exploración de múltiples enfoques resolutivos.

Entre los principales aportes del trabajo, se destacan:

- La implementación de un algoritmo exacto basado en **backtracking con múltiples podas**, diseñado para explorar el espacio de soluciones de manera controlada y eficiente. Este enfoque demostró ser eficaz para instancias de tamaño reducido o moderado, gracias al uso de estrategias de poda como la simetría, el desbalance, la imposibilidad estructural y la optimalidad parcial.
- El desarrollo de un **modelo de Programación Lineal Entera**, que formaliza el problema como una minimización sobre variables binarias, con restricciones de cobertura, activación de clusters y control explícito de la distancia máxima. Esta formulación permitió utilizar solvers para obtener soluciones óptimas en plazos razonables, y sirvió como referencia de calidad frente a otros enfoques.
- La incorporación de un **validador polinomial** capaz de verificar si una partición propuesta cumple con todas las condiciones del problema. Esta herramienta permitió establecer un marco riguroso de validación, tanto para soluciones exactas como aproximadas.
- La evaluación de **métodos aproximados guiados por la estructura del grafo**, tales como el algoritmo de Louvain y un enfoque greedy de maximización de modularidad. Estos métodos no están diseñados para optimizar directamente el criterio de diámetro, pero buscan generar particiones con alta cohesión interna. Se utilizaron como aproximaciones exploratorias sobre instancias de gran tamaño, evaluando empíricamente el diámetro resultante en las comunidades detectadas.
- El análisis comparativo entre enfoques, tanto en términos de *calidad estructural* (diámetro y cohesión) como de *desempeño empírico* (tiempo de ejecución, escalabilidad y simplicidad de implementación).

Además, se exploraron las condiciones bajo las cuales los métodos basados en modularidad podían —aunque no garantizado— generar soluciones aceptables en términos del criterio de diámetro. Esta comparación permitió discutir los alcances y limitaciones de cada enfoque, y reflexionar sobre su aplicabilidad práctica según el tamaño y la estructura del grafo.

En síntesis, el trabajo combinó fundamentos teóricos sólidos con desarrollos computacionales concretos. Las decisiones tomadas en cada etapa —desde la formulación de restricciones hasta las optimizaciones en la construcción de modelos— resultaron en una solución robusta, validable y flexible.

10. Anexo

10.1. Demostración de pertenencia a NP

Para demostrar que el problema de **Clustering por Bajo Diámetro** pertenece a la clase NP, es suficiente con exhibir un validador polinomial.

10.1.1. Versión de decisión del problema

La versión de decisión del problema puede enunciarse de la siguiente manera:

> Dado un grafo no dirigido y no pesado $G = (V, E)$, un número entero k, y un valor C, ¿existe una partición de V en a lo sumo k clusters disjuntos tal que:
>
> - Cada vértice pertenezca a un único cluster,
> - Todos los vértices estén asignados (la partición es total),
> - Y que la distancia máxima entre cualquier par de vértices dentro de cada cluster sea menor o igual a C?

Si un cluster queda vacío o con un único vértice, se considera que su diámetro es 0. Las distancias se calculan sobre el grafo original, considerando cualquier camino existente (no solo aristas internas al cluster).

10.1.2. Validador polinomial

Para verificar una solución candidata, se implementó un algoritmo que realiza las siguientes comprobaciones:

1. Que se utilicen a lo sumo k clusters.
2. Que todos los vértices estén presentes, y que ninguno esté repetido.
3. Que el diámetro (la máxima distancia entre pares de nodos) dentro de cada cluster no supere el valor C.

La verificación de la condición 3 se realiza calculando la distancia entre todos los pares de vértices de cada cluster mediante un algoritmo BFS. En caso de que no exista camino entre dos vértices del mismo cluster, se asume una distancia infinita y se rechaza la solución.

10.1.3. Descripción del algoritmo validador

El algoritmo `validador_clustering` toma como entrada un grafo $G = (V, E)$, un número máximo de clusters k, un umbral de diámetro permitido C, y una solución candidata representada como un diccionario que asigna vértices a clusters. Su propósito es verificar si la solución cumple con las condiciones de validez del problema de Clustering por Bajo Diámetro.

El procedimiento consta de los siguientes pasos:

1. **Cantidad de clusters:** se verifica que la solución no utilice más de k clusters. Si se supera ese número, la solución es rechazada.
2. **Cobertura sin repeticiones:** se construye un conjunto `usados` que contiene todos los vértices asignados. Durante la iteración:
   - Si un vértice aparece más de una vez (es decir, si ya está en el conjunto), se rechaza la solución.

- Si, al finalizar, el conjunto de vértices utilizados no coincide exactamente con el conjunto de vértices del grafo, la solución también es rechazada.

3. **Verificación de diámetros internos:** para cada cluster, se calcula la distancia máxima entre pares de vértices utilizando un recorrido BFS. Esta operación se realiza mediante una función auxiliar llamada `calcular_distancia_max_cluster`. Si la distancia máxima encontrada en un cluster excede el valor C, la solución se rechaza.

4. **Aprobación:** si se cumplen todas las condiciones anteriores, la solución se considera válida y el algoritmo retorna `True`.

10.2. Código del validador

```python
def validador_clustering(grafo, k, c, solucion_propuesta):

    if len(solucion_propuesta) > k:
        return False

    usados = set()
    total_vertices = set(grafo.obtener_vertices())

    for cluster, vertices in solucion_propuesta.items():
        for vertice in vertices:
            if vertice in usados:
                return False  # nodos repetidos
            usados.add(vertice)

    if usados != total_vertices:
        return False

    for cluster, vertices in solucion_propuesta.items():
        distancia_maxima = calcular_distancia_max_cluster(grafo, vertices)
        if distancia_maxima > c:
            return False

    return True
```

10.2.1. Complejidad del validador

El tiempo de ejecución del validador es polinomial en el tamaño del grafo:

- Verificar que no haya vértices repetidos ni omisiones se puede hacer en $O(V)$.
- Para cada cluster, se calcula la distancia entre pares de vértices con BFS, cuya complejidad es $O(V + E)$ por par.
- Como hay a lo sumo k clusters, cada uno con a lo sumo V vértices, el número de pares por cluster es $O(V^2)$.

En conjunto, la complejidad total del validador es:

$$O(k \cdot V^2 \cdot (V + E))$$

10.2.2. Conclusión

Dado que la validación de una solución propuesta puede realizarse en tiempo polinomial con respecto al tamaño de la entrada, se concluye que el problema de **Clustering por Bajo Diámetro** pertenece a la clase **NP**.

10.3. Demostración de NP-Completitud

En la sección anterior se ha demostrado que el problema de **Clustering por Bajo Diámetro** pertenece a la clase NP, mediante la construcción de un validador polinomial que verifica si una partición dada cumple con las condiciones del problema. A continuación, completaremos la demostración de NP-completitud mediante una reducción desde un problema conocido como NP-completo.

10.3.1. Enunciado formal del problema de Clustering por Bajo Diámetro:

Dado un grafo no dirigido y no pesado $G = (V, E)$, un número entero k, y un valor C, ¿es posible separar los vértices en a lo sumo k grupos o clusters disjuntos $C_1, C_2, \ldots, C_k$, de tal forma que:

- Cada vértice pertenezca exactamente a un único cluster,
- Todos los vértices estén asignados (la partición es total),
- La distancia máxima entre cualquier par de vértices dentro de cada cluster (medida en el grafo original) sea a lo sumo C.

Si un cluster queda vacío o con un único vértice, se considera que su diámetro es 0.

Al calcular las distancias se tienen en cuenta tanto las aristas entre vértices dentro del cluster como cualquier otro camino dentro del grafo: es decir, se considera la distancia geodésica en el grafo original.

10.3.2. Estrategia:

Demostraremos que el problema es NP-completo mediante una reducción polinomial desde el problema de **Separación en R Cliques (SRC)**, el cual es conocido por ser NP-completo.

10.3.3. Enunciado formal de SRC:

Dado un grafo no dirigido $G = (V, E)$ y un número entero R, ¿existen subconjuntos disjuntos $S_1, S_2, \ldots, S_k \subseteq V$, con $\bigcup_{i=1}^{k} S_i = V$ y $k \leq R$, tales que cada subgrafo inducido por S_i sea un *clique* (es decir, un subgrafo completamente conexo)?

10.3.4. Transformación:

Dada una instancia de SRC, (G, R), construimos una instancia del problema de Clustering por bajo diámetro (G', k, C) como sigue:

- $G' = G$ (mismo grafo original),
- $k = R$ (misma cantidad máxima de grupos),
- $C = 1$ (valor de diámetro máximo permitido).

10.3.5. Demostración de equivalencia:

(Ida) Supongamos que existe una partición de V en a lo sumo R cliques $S_1, \ldots, S_k$. Por definición de clique, todos los pares de vértices dentro de cada S_i están directamente conectados por una arista en el grafo. Entonces, la distancia entre cualquier par de vértices es 1. Por lo tanto, esta partición cumple con las condiciones del problema de Clustering por bajo diámetro con $C = 1$.

(Vuelta) Supongamos que existe una partición de V en a lo sumo $k \leq R$ clusters $C_1, \ldots, C_k$, donde la distancia máxima entre cualquier par de vértices (medida en el grafo original) es $= 1$. Esto implica que cualquier par de vértices dentro de un mismo cluster está directamente conectado (ya que una distancia de 1 requiere una arista directa). Por lo tanto, cada cluster induce un clique. La partición obtenida es válida para el problema SRC.

10.3.6. Conclusión:

La transformación descrita se puede realizar en tiempo polinomial, y la equivalencia entre las instancias es correcta en ambas direcciones. Como:

- El problema SRC es NP-completo,
- Y el problema de Clustering por bajo diámetro pertenece a NP (ver sección anterior),

Concluimos que el problema de **Clustering por bajo diámetro es NP-completo**.

10.4. Algoritmo por Backtracking

Con el objetivo de minimizar el mayor de los diámetros entre todos los clusters generados a partir de un grafo no dirigido y no pesado, se diseñó un algoritmo exacto basado en **backtracking con múltiples podas** complementado con una **solución inicial heurística**, **preprocesamiento de distancias** y un seguimiento incremental del **diámetro parcial** de cada cluster durante la búsqueda.

10.4.1. Instancia del problema

Dado un grafo no dirigido y no pesado $G = (V, E)$ y un entero positivo k, se busca particionar el conjunto de vértices V en k clusters disjuntos $C_1, C_2, \ldots, C_k$, minimizando el peor caso de cohesión interna, entendida como el mayor diámetro entre los clusters:

$$\min_{\text{partición}} \left(\max_{1 \leq i \leq k} \text{diam}(C_i) \right)$$

donde el diámetro de un cluster C_i se define como la mayor distancia entre pares de vértices dentro del mismo cluster.

10.4.2. Solución inicial heurística

Para establecer una cota superior inicial que permita guiar las podas del algoritmo exacto, se utiliza una solución heurística basada en la maximización de la modularidad. El algoritmo greedy `maximizar_modularidad` genera una partición en comunidades, que luego se asignan rotativamente a los k clusters disponibles. Los nodos no asignados por la heurística se agregan también de forma rotativa.

10.4.3. Preprocesamiento de distancias

Para evitar cálculos repetidos de distancias durante la ejecución del algoritmo, se realiza un **BFS desde cada vértice** del grafo. El resultado es una estructura que almacena las distancias mínimas entre todos los pares de vértices, accesibles en tiempo constante mediante un diccionario de claves ordenadas.

10.4.4. Ordenamiento inicial

Los vértices del grafo se ordenan en forma descendente por grado (cantidad de vecinos). Esta estrategia prioriza la asignación de vértices más conectados al comienzo de la búsqueda, ya que estos suelen ser los que más influyen en el aumento del diámetro dentro de los clusters.

10.4.5. Estrategia de búsqueda

La función `clustering_bt` implementa una estrategia de backtracking con evaluación incremental de los diámetros. Se recorren los vértices ordenados y se intenta asignarlos a cada uno de los k clusters, actualizando el diámetro del cluster de forma incremental en cada paso.

10.4.6. Podas implementadas

1. **Poda por simetría:** evita asignar un vértice a un cluster vacío si ya se intentó otro cluster vacío en una posición anterior, reduciendo permutaciones equivalentes.
2. **Poda por insuficiencia de vértices:** si la cantidad de vértices restantes no alcanza para completar los clusters vacíos, la rama se descarta.
3. **Poda por desbalance:** evita que algún cluster supere el tamaño máximo estimado como $\left\lceil \frac{|V|}{k} \right\rceil$.
4. **Poda por subóptimo:** si el diámetro parcial actual ya iguala o supera el de la mejor solución encontrada hasta el momento, no se continúa con esa asignación.

10.4.7. Evaluación y actualización

Cuando todos los vértices han sido asignados, se evalúa el diámetro máximo de la partición generada. Si esta solución mejora la mejor conocida hasta el momento, se guarda como nueva solución óptima. La copia se realiza de forma profunda para evitar efectos colaterales.

10.4.8. Código del Backtracking

```
def clustering_bt(grafo, vertices, actual, sol_optima, sol_temporal, k, distancias,
     diametros_actuales, mejor_diametro_actual):
    if actual >= len(vertices):
        diametro_actual = max(diametros_actuales.values())
        if mejor_diametro_actual is None or diametro_actual < mejor_diametro_actual
    :
            return copy.deepcopy(sol_temporal), diametro_actual
        return sol_optima, mejor_diametro_actual

    vertice = vertices[actual]

    for cluster in sol_temporal:
        if not sol_temporal[cluster] and es_cluster_vacio_anterior(sol_temporal,
    list(sol_temporal.keys()).index(cluster)):
            continue

        nuevo_diametro = diametros_actuales[cluster]
        for otro in sol_temporal[cluster]:
            clave = tuple(sorted((vertice, otro)))
            nuevo_diametro = max(nuevo_diametro, distancias.get(clave, float('inf')
    ))

        diametro_anterior = diametros_actuales[cluster]
        sol_temporal[cluster].append(vertice)
        diametros_actuales[cluster] = nuevo_diametro

```

```
        if not alcanzan_vertices(len(vertices) - (actual + 1), sol_temporal):
            sol_temporal[cluster].pop()
            diametros_actuales[cluster] = diametro_anterior
            continue

        if cluster_desbalanceado(sol_temporal, vertices_restantes, k):
            sol_temporal[cluster].pop()
            diametros_actuales[cluster] = diametro_anterior
            continue

        if mejor_diametro_actual is not None and max(diametros_actuales.values())
    >= mejor_diametro_actual:
            sol_temporal[cluster].pop()
            diametros_actuales[cluster] = diametro_anterior
            continue

        incluyendo, nuevo_diametro_sol = clustering_bt(grafo, vertices, actual + 1,
     sol_optima, sol_temporal, k, distancias, diametros_actuales,
    mejor_diametro_actual)

        if mejor_diametro_actual is None or nuevo_diametro_sol <
    mejor_diametro_actual:
            sol_optima = copy.deepcopy(incluyendo)
            mejor_diametro_actual = nuevo_diametro_sol

        sol_temporal[cluster].pop()
        diametros_actuales[cluster] = diametro_anterior

    return sol_optima, mejor_diametro_actual
```

10.4.9. Observaciones finales

Este enfoque permite explorar el espacio de soluciones de manera eficiente y exacta, combinando técnicas de optimización combinatoria con un greedy inicial y estructuras de datos apropiadas. El uso de podas efectivas y evaluación incremental del diámetro permite escalar a instancias de tamaño medio sin comprometer la optimalidad de la solución.

10.5. Cota empírica de aproximación

Dado que el algoritmo de Louvain no garantiza cumplir con las restricciones del problema (cantidad de clusters k ni diámetro máximo), se realizó una evaluación comparativa contra el algoritmo exacto por backtracking, que sí ofrece soluciones óptimas (cuando el tiempo de cómputo lo permite).

La métrica utilizada para cuantificar la calidad relativa de Louvain respecto a la solución óptima es la siguiente:

$$\text{Cota empírica} = \frac{\text{Diam(Louvain)} - \text{Diam(Óptimo)}}{\text{Diam(Óptimo)}} \times 100$$

Esta cota representa el porcentaje de incremento (o reducción) del peor diámetro de cluster obtenido por Louvain respecto al óptimo.

Instancia	k	**Diam(Óptimo)**	**Diam(Louvain)**	**Clusters(Louvain)**	**Cota empírica**
22_3.txt	10	1	2	7	+100 %
45_3.txt	7	3	2	13	-33 %

Análisis:

- En la instancia `22_3.txt`, Louvain generó menos clusters de los requeridos, lo que aumentó el diámetro dentro de algunas comunidades. Esto explica la cota empírica positiva del 100 %.
- En `45_3.txt`, el algoritmo de Louvain generó una sobresegmentación del grafo (13 clusters), lo que redujo el diámetro interno pero a costa de incumplir la restricción de $k = 7$.

Conclusión:

Esta comparación empírica revela que Louvain puede ofrecer buenos resultados en términos de cohesión (diámetro bajo), aunque no necesariamente cumpliendo con las restricciones de cardinalidad. Por tanto, su uso es apropiado como heurística rápida, especialmente útil en grafos grandes o como solución inicial para algoritmos exactos.

Nota: Las observaciones anteriores se basan en los resultados presentados previamente en la subsección de ejemplos comparativos dentro de la sección 6, donde se detallan tanto el número de clusters como los diámetros obtenidos por cada algoritmo.

10.6. Mediciones

En esta sección se volverán a realizar las mediciones de los algoritmos de backtracking y programación lineal (dado que se realizaron modificaciones tanto sobre el backtracking como sobre los sets de datos utilizados para medir estos dos). Entonces nuevamente, la complejidad de los algoritmos depende principalmente de dos factores:

- V: Cantidad de vértices del grafo.
- k: Cantidad límite de clusters.

El término dominante en la complejidad de ambos algoritmos (backtracking y programación lineal) es $\mathcal{O}(k^V)$. Por lo tanto, se probará que, al mantener k constante, el crecimiento sea exponencial respecto a V.

Entonces para cada algoritmo se realizarán dos pruebas iniciales:

- Medición de tiempo de ejecución con un k = 3, y V variable.
- Medición de tiempo de ejecución con un k = 6, y V variable.

Los datos se generaron de manera aleatoria creando grafos con n cantidad de vértices y aristas 4/3n - 1 (esto se cambió respecto de las mediciones originales ya que notamos que se generaban grafos muy densos que aumentaban demasaido los tiempos de medición). Las mediciones se realizaron promediando 7 ejecuciones por tamaño para reducir la variabilidad. Es importante destacar que se redujo ese valor (inicialmente era 10) para minimizar tiempos ya que se llegaron a obtener tiempos realmente inviables. También se cambió respecto de la vez anterior los grafos que se creaban pasaron de ser **10, 12, 15, 17, 20** a ser **20, 27, 35, 42, 50**, esto para acercarnos más a los ejemplos que proveía la cátedra, obteniendo tiempos mucho más grandes que la medición anterior pero que a la vez demuestran una mejoría importante en el algoritmo de backtracking ya que si bien el set de datos creció mucho, los tiempos lo hicieron pero de la misma manera.

10.6.1. Análisis para algoritmo Backtracking

- Análisis con V variable ($k = 3$ fijo)

En esta primera instancia se midió el tiempo con el set ya mencionado y manteniendo constante el k máximo de clusters en $k = 3$.

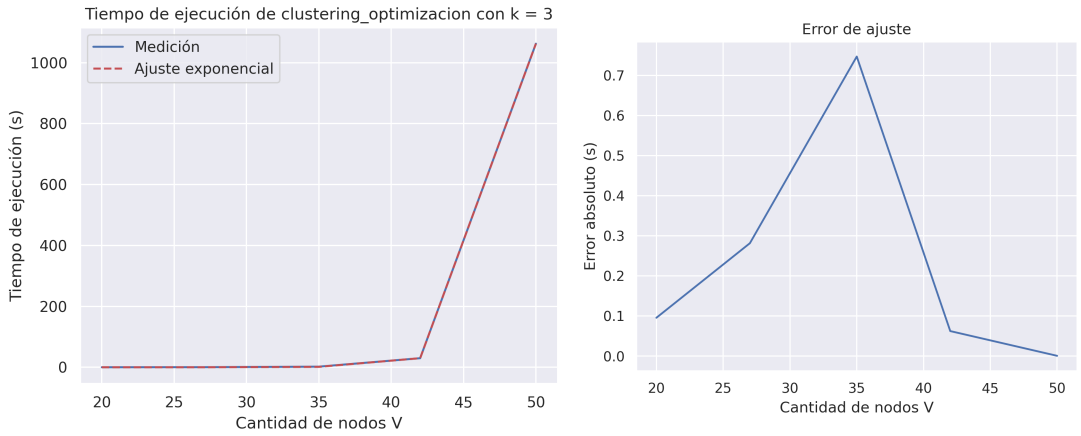

Figura 9: Ajuste exponencial para V variable. Error cuadrático total = **0.6508555824227805**

Nuevamente, el gráfico presenta una relación exponencial respecto del tamaño del grafo y el tiempo de ejecución. Es interesante agregar que realizar la medición del algoritmo tomó 7663.60s

- Análisis con V variable ($k = 6$ fijo)

En esta segunda prueba se aumentó el valor de k, midiendo con $k = 6$

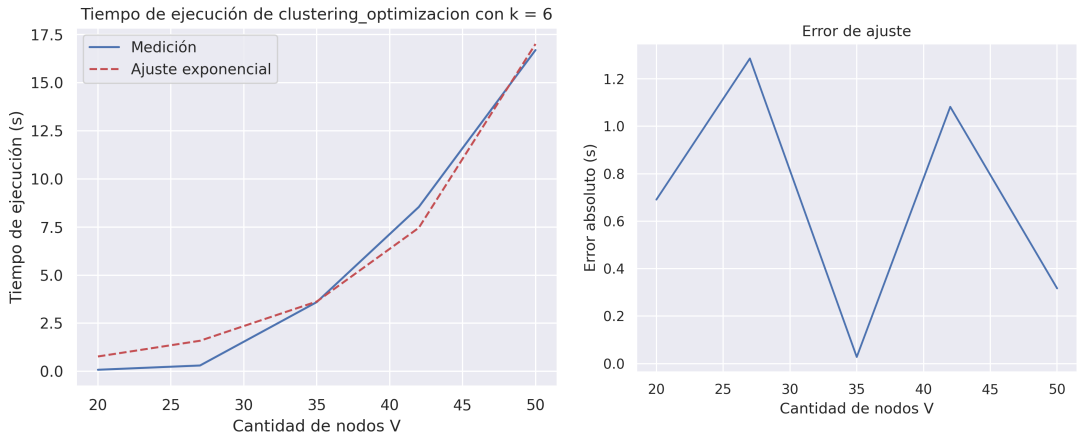

Figura 10: Ajuste exponencial para V variable. Error cuadrático total = **3.4027106668773985**

En este caso, el tiempo de ejecución es 205.33s, si bien es alto, es mucho más bajo que con $k = 3$, esto se debe a que al haber más clusters donde asignar vértices se recorren menos posibilidades hasta llegar a la correcta. Se aprecia que el ajuste se adapta correctamente a los datos medidos, validando la hipótesis de que la complejidad es exponencial.

10.6.2. Análisis para algoritmo Programación Lineal

- Análisis con V variable ($k = 3$ fijo)

Se utilizó el mismo set de datos con ($k = 3$ para medir nuevamente el algoritmo de programación lineal

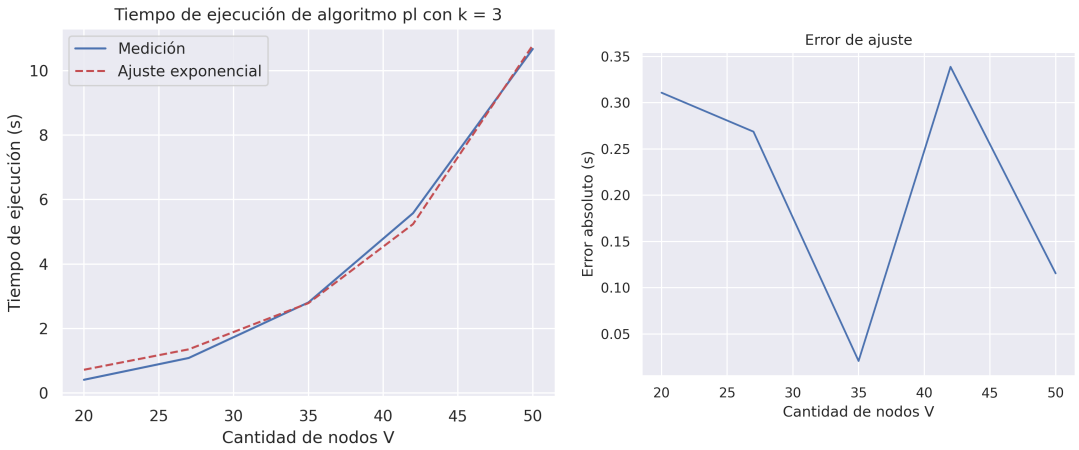

Figura 11: Ajuste exponencial para V variable. Error cuadrático total = **0.2974281745599414**

El tiempo que demoró fueron 144.64s.

10.6.3. Análisis con V variable ($k = 3$ fijo) ajuste logarítmico

Mantuvimos el set de datos y k = 3 para ajustar a una función diferente.

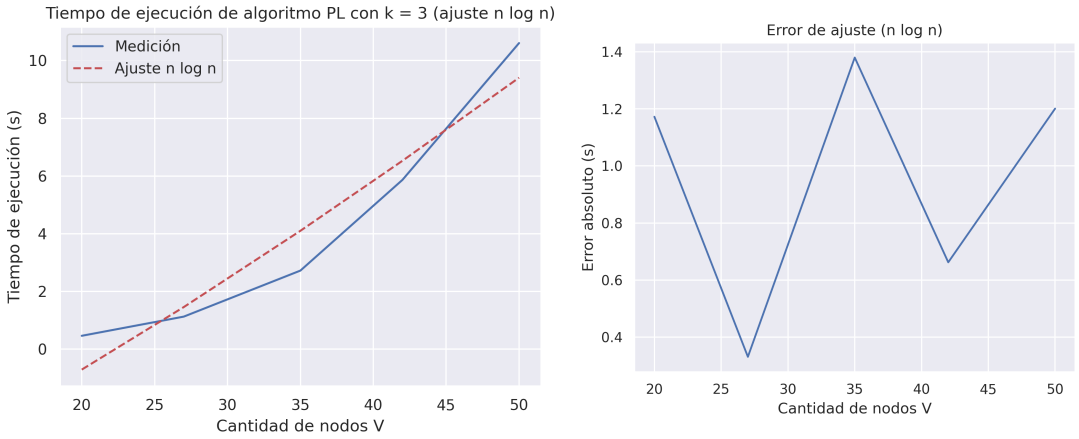

Figura 12: Ajuste logarítmico para V variable. Error cuadrático total = **5.267768310007417**

El tiempo que demoró fueron 146.19s

Al aumentar los nodos de nuestros grafos, podemos observar ahora que intentar con un ajuste logaritmico ya no tendría mucho sentido, ya que se llega a percibir viendo el gráfico y los errores obtenidos que se trata de una relación exponencial, tal como habíamos pensado. Sin embargo, realizamos el resto de pruebas tal como la vez anterior, para salir de dudas.

10.6.4. Análisis con V variable ($k = 6$ fijo)

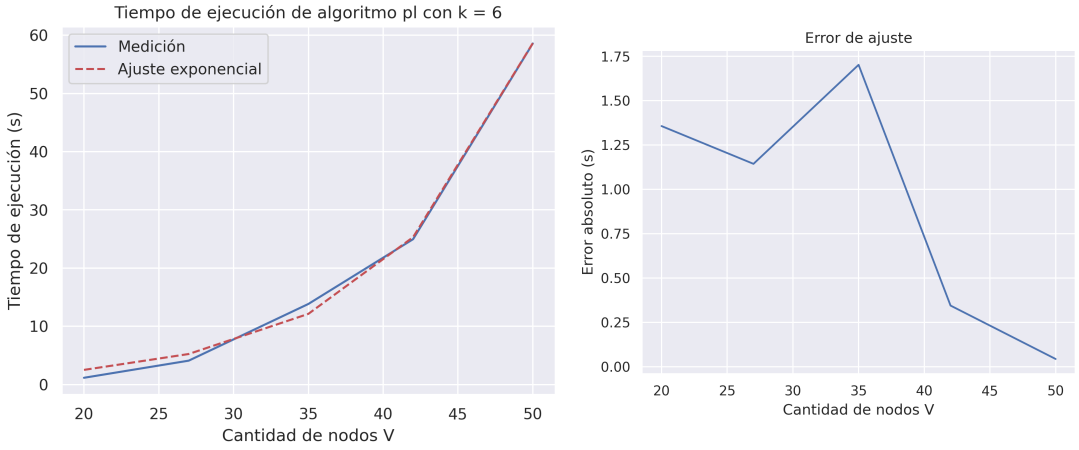

Figura 13: Ajuste exponencial para V variable. Error cuadrático total = **6.159233102842426**

El tiempo aumenta a 719.33s, lo cual tiene sentido ya que aumentamos el k.

10.6.5. Análisis con V variable ($k = 6$ fijo) ajuste logarítmico

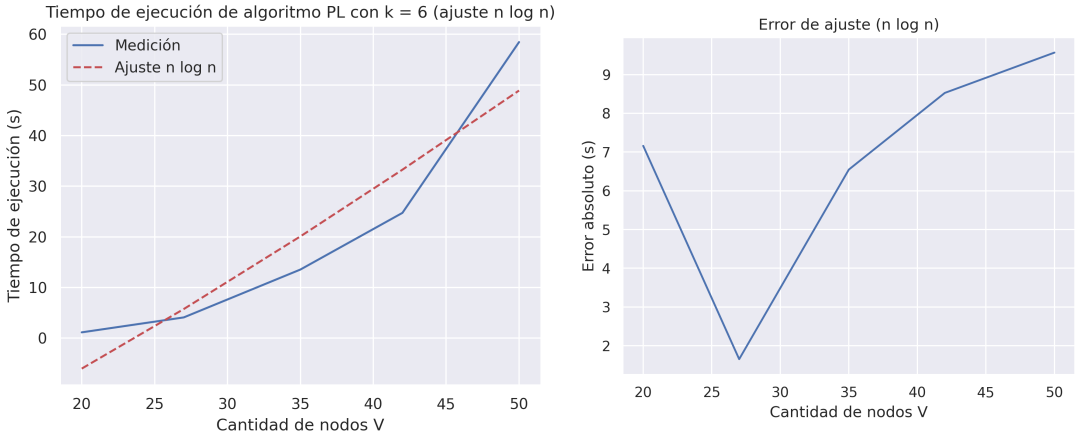

Figura 14: Ajuste logarítmico para V variable. Error cuadrático total = **261.1167876171511**

Aquí el tiempo también aumento, en este caso a 714.86s. Pero esta vez si podemos asegurar que no se asemeja para nada a una relación logarítmica, cosa que no podíamos asegurar a estas alturas en las mediciones con menos vértices realizadas anteriormente.

10.6.6. Análisis con V variable ($k = 9$ fijo)

Al ser un algoritmo más rápido pudimos extender a un ($k = 9$

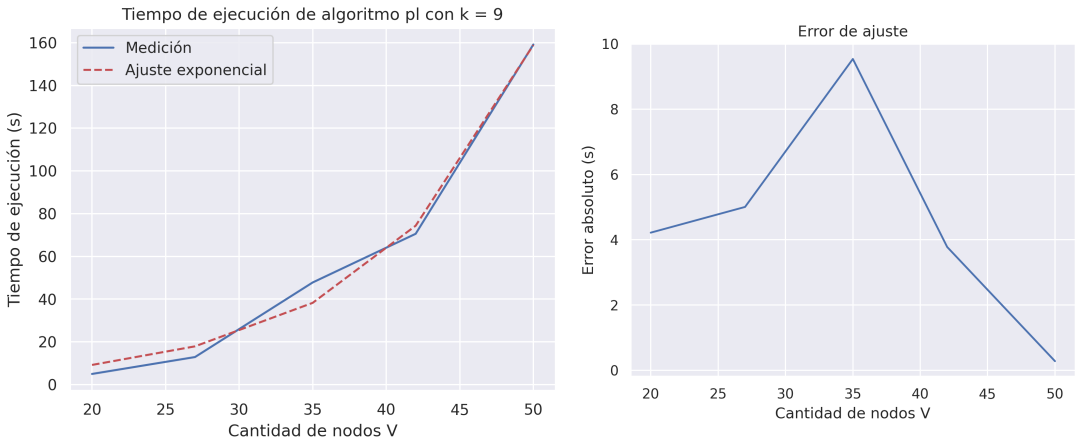

Figura 15: Ajuste exponencial para V variable. Error cuadrático total = **148.21555988420957**

Obtuvimos un tiempo considerablemente mayor a los casos anteriores (2068.74s), algo esperable dado que el k es mucho más grande, y un error que si bien es alto refleja los inmensos valores que se están manejando, ya que si miramos el gráfico podemos observar que la relación es exponencial. Podemos ver como las pendientes van cambiando levemente en un principio para luego dispararse enormemente con un V mayor.

Conclusión

Los resultados obtenidos son coherentes con las complejidades teóricas. Se observó respecto de la vez anterior que: Al aumentar los valores de k, pudimos ver más claramente la distancia que había entre el ajuste logaritmico y el exponencial en programación lineal. Que el nuevo algoritmo propuesto para Backtracking mejoró el rendimiento enormemente pero que sigue sin ser bueno para grafos de gran o mediano tamaño, mientras que el algoritmo de programación lineal si bien es exponencial es un poco mejor que el anterior ya que demora menos y se adapta mejor a grafos más grandes o mayor cantidad de clusters. Y que el algoritmo de Louvain si bien no fue medido nuevamente, sigue siendo ampliamente mejor para grafos de gran tamaño que los últimos dos mencionados. En definitiva, la elección del algoritmo adecuado dependerá del tamaño del problema, la necesidad de precisión en la solución y las restricciones de tiempo computacional disponibles.