.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Comunidades NP-Completas

16 de Junio de 2025

Franco Guardia
109374

Mateo Requejo
110109

Santino
Regidor
112147

1. Introducción

Luego de los grandes resultados obtenidos en los trabajos anteriores, fuimos nuevamente ascendidos. Ahora somos la mano derecha del Gringo.

En función de poder evitar que situaciones así vuelvan a suceder, obtuvo la información de qué integrante de la organización es cercano a otro. Con esta información construyó un grafo no dirigido y no pesado (vértices: personas, aristas: si las personas son cercanas). Nos pidió que separemos el grafo en comunidades. Es decir, separar los vértices en K clusters (grupos).

Todo venía bien, nuestra primera idea era la de maximizar la distancia mínima entre comunidades, lo cual puede resolverse con un algoritmo de árboles de tendido mínimo (y siendo este grafo no pesado, aún más fácil), pero Amarilla Pérez nos indicó que esa métrica da muy malos resultados. Que en vez de esto, mejor minimizar la distancia máxima entre dos vértices dentro de la misma comunidad.

Pasamos de un un problema que hasta se podía resolver de forma lineal, a un problema que, creemos, es NP-Completo. Debemos convencer a Amarilla Pérez de que esta no es una buena idea, que demorará mucho. Para esto, igualmente, vamos a tener que venir con demostraciones, mediciones, y al menos una alternativa

Luego de plantear el problema de clustering por bajo diámetro como un problema de decisión se nos pide primero la demostración de que es un problema NP completo. Luego al replantear el problema como si fuera de optimización se nos pide implementar un algoritmo de backtracking y programación lineal que den la solución óptima al problema

Finalmente con un algoritmo de solución ya planteado poder saber qué tan buena es la solución aproximada que nos da para resolver el problema, realizando mediciones que corroboren esa aproximación

2. Demostraciones

2.1. Demostración NP

Para poder demostrar que el problema de decisión de clustering por bajo diámetro está en NP, primero debemos saber si dada una posible solución, podemos verificar su validez en tiempo polinomial

Supongamos que tenemos una solución propuesta, es decir un conjunto de a lo sumo K clusters disjuntos cada uno con un subconjunto de los vértices y que cumplen que la distancia máxima entre vértices dentro de cada cluster sea a lo sumo C. Para verificar esta solución, utilizaremos el siguiente algoritmo en el cual se reciben:

- Un grafo no dirigido y no pesado a partir del cual se pueden formar los clusters
- Un número entero K que indica a lo sumo cuantos clusters se quieren formar
- Un número entero C que indica la máxima distancia permitida entre cada par de vértices en cada cluster
- La solución propuesta representada como una lista de listas donde cada sublista es un cluster y dentro están los vértices que lo conforman

```
from collections import deque

def calcular_distancias_bfs(grafo, origen):
    cola = deque()
    visitados = set()
    dist = {}
    cola.append(origen)
    visitados.add(origen)
```

```
    for v in grafo:
        dist[v] = 0

    while not len(deque) == 0:
        v = cola.popleft()
        for w in grafo.adyacentes(v):
            if w not in visitados:
                cola.appendleft(w)
                dist[w] = dist[v] + 1
                visitados.add(w)

    return dist



def validar_solucion_clustering(grafo, K, C, solucion):
    if len(solucion) > K:
        return False

    vertices = grafo.obtener_vertices()
    cubiertos = set()

    for cluster in solucion:
        for v in cluster:
            if v not in vertices:
                return False

            if v in cubiertos:
                return False #hay un vertice en dos clusters
            else:
                cubiertos.add(v)

            distancia = calcular_distancias_bfs(grafo, v)
            for w in cluster:
                if v == w:
                    continue

                if distancia[(v, w)] > C:
                    return False

    if len(cubiertos) != len(vertices):
        return False

    return True
```

1) Verificar que la cantidad de clusters sea de a lo sumo K: $\mathcal{O}(1)$

2) Para cada vértice verificar tanto que se encuentre en el grafo como que la distancia a cualquier otro vértice del cluster sea de a lo sumo C, para esto calcular la distancia en cantidad de aristas hacia todos los otros vértices del grafo usando BFS (porque el grafo es no pesado). Esto por cada cluster y el cálculo de distancias a su vez por cada vértice: $\mathcal{O}((V + E) * K * V^2)$

3) Finalmente verificar que todos los vértices del grafo se encuentren en algún cluster: $\mathcal{O}(1)$

Dado que cada uno de los pasos anteriores puede realizarse en tiempo polinomial, podemos verificar una solución propuesta en tiempo polinomial. Por lo tanto, el problema de clustering por bajo diámetro está en NP

2.2. Demostración de NP-Complejidad

Para demostrar que el problema de Clustering por bajo diámetro es NP-Completo, realizamos una reducción desde el problema de coloreo de grafos (Coloreo), el cual es NP-Completo.

Problema de coloreo Dado un grafo $G = (V, E)$ y un entero K, decidir si es posible asignar uno de K colores a cada vértice de manera que no exista ninguna arista cuyos extremos compartan el mismo color.

Reducción Dado un grafo $G = (V, E)$ y un entero K, construimos el grafo complemento $G^c = (V, E^c)$, donde

$$(u, v) \in E^c \iff (u, v) \notin E.$$

Fijamos el parámetro $C = 1$ para el problema de clustering por bajo diámetro.

Entonces, planteamos la siguiente pregunta: ¿es posible separar el conjunto de vértices V en a lo sumo K clusters, de modo que la distancia máxima dentro de cada cluster en G^c sea a lo sumo $C = 1$?

Demostración de la equivalencia

- **Si G es K-coloreable, entonces existe una partición de V en K clusters con diámetro ≤ 1 en G^c:**

  Sea una asignación válida de colores a los vértices de G. Cada color define un conjunto de vértices que no tienen aristas internas en G, es decir, un conjunto independiente. Por definición del grafo complemento G^c, dentro de cada conjunto independiente los vértices están conectados por aristas en G^c. Por lo tanto, la distancia entre cualquier par de vértices dentro de ese cluster en G^c es 1, y el diámetro del cluster es a lo sumo 1. W

- **Si existe una partición de V en K clusters con diámetro ≤ 1 en G^c, entonces G es K-coloreable:**

  Si cada cluster tiene diámetro máximo ≤ 1 en G^c, entonces cualquier par de vértices dentro de un cluster está conectado directamente por una arista en G^c. Esto implica que no existen aristas internas en G entre vértices del mismo cluster. Por lo tanto, asignando un color diferente a cada cluster se obtiene una coloración válida de G con K colores.

Conclusión Dado que el problema de colore es NP-Completo y pudimos reducirlo en tiempo polinomial al problema de clustering por bajo diámetro con $C = 1$, se concluye que el problema de clustering por bajo diámetro es también NP-Completo, es decir,

$$\text{Coloreo} \leq_p \text{Clustering por bajo diámetro.}$$

3. Algoritmo Backtracking

3.1. Código

A continuación, se propone un algoritmo de Backtracking que permita resolver el problema planteado en su versión de optimización: "*Dado un grafo no dirigido y no pesado, y un valor K, determinar los K clusters para que la distancia máxima de cada cluster sea mínima*".

Lo que se busca es agrupar los nodos del grafo en K conjuntos conectados internamente donde el clúster más largo, en términos de distancia, sea lo más chico posible.

El algoritmo está dividido en varias secciones, primero tenemos una función auxiliar llamada `obtener_todas_las_distancias_minimas(grafo, nodos)`. El código de la función es el siguiente:

```
1 def obtener_todas_las_distancias_minimas(grafo, nodos=None):
2     """Devuelve una lista de nodos, un diccionario que mapea el indice de un nodo
      en la lista, y una matriz dist[i][j] con la distancia minima entre el nodo i y
      j"""
3     if nodos is None:
4         nodos = list(grafo.nodes())
5
6     n = len(nodos)
7     indices = {}
```

```
    for i, nodo in enumerate(nodos):
        indices[nodo] = i

    dist = [[math.inf]*n for _ in range(n)]

    for i in range(n):
        dist[i][i] = 0

    for u, d in nx.all_pairs_shortest_path_length(grafo):
        if u not in indices:
            continue
        ui = indices[u]
        for v, dv in d.items():
            if v in indices:
                dist[ui][indices[v]] = dv

    return nodos, indices, dist
```

Como explica el comentario de la función, calcula la distancia mínima entre todos los pares de nodos utilizando la función `all_pairs_shortest_path_length` de la librería *networkx*. Se decidió usar una matriz de distancias por una cuestión de optimización, luego de observar que el algoritmo de backtracking podía ser poco performante usando otras estructuras de datos.

Por otro lado tenemos la función base: `k_clusters_backtracking(grafo, k)` . El código de la misma es el siguiente:

```
def k_clusters_backtracking(grafo: nx.Graph, k: int):
    """
    Dado un grafo no dirigido y no pesado y un valor K, determina los
    K clusters para que la distancia maxima de cada cluster sea minima.
    Devuelve una lista con los clusters y la distancia maxima.
    """

    orden = sorted(grafo.nodes(), key=lambda u: grafo.degree(u), reverse=True)
    # Este orden sirve para poner los nodos de mayor grado primero (optimizacion)

    nodos, indices, dist = obtener_todas_las_distancias_minimas(grafo, orden)

    clusters = [[] for _ in range(k)]
    mejor_solucion = {
        "clusters": None,
        "distancia_maxima": float('inf')
    }

    _k_clusters_backtracking(grafo, nodos, dist, k, clusters, 0, 0, mejor_solucion)

    resultado = [
        [nodos[i] for i in lista_indices_resultado]
        for lista_indices_resultado in mejor_solucion["clusters"]
    ]

    return resultado, mejor_solucion["distancia_maxima"]
```

Este es el punto de entrada de nuestro backtracking. En primer lugar, se ordenan los nodos por grado. Esto también es una decisión de optimización para mejorar el rendimiento del algoritmo, sacándonos de encima los nodos más "centrales" primero. Esta función almacenará la mejor solución que nos devuelva la función recursiva y devolverá una lista de los clusters y la distancia máxima alcanzada.

Y finalmente, tenemos la función recursiva donde se realizará el algoritmo de backtracking. A continuación presentamos el código:

```
def _k_clusters_backtracking(grafo, nodos, dist, k, clusters, indice, dist_actual,
    mejor_solucion):
    if dist_actual >= mejor_solucion["distancia_maxima"]:
        return

    if indice == len(nodos):
        if any(len(c) == 0 for c in clusters):
```

```
            return

        for c in clusters:
            sub = grafo.subgraph(nodos[i] for i in c)
            if not nx.is_connected(sub):
                return

        diametro = 0
        for c in clusters:
            for i in range(len(c)):
                for j in range(i+1, len(c)):
                    diametro = max(diametro, dist[c[i]][c[j]])

        if diametro < mejor_solucion["distancia_maxima"]:
            mejor_solucion["distancia_maxima"] = diametro
            mejor_solucion["clusters"] = [list(c) for c in clusters]
        return


    indice_u = indice

    for i in range(k):
        if not clusters[i] and any(not clusters[j] for j in range(i)):
            continue

        clusters[i].append(indice_u)

        if len(clusters[i]) == 1:
            diam_i = 0
        else:
            diam_i = max(dist[indice_u][indice_v] for indice_v in clusters[i])

        nuevo_diam = max(dist_actual, diam_i)

        if nuevo_diam < mejor_solucion["distancia_maxima"]:
            _k_clusters_backtracking(
                grafo, nodos, dist, k,
                clusters, indice+1, nuevo_diam,
                mejor_solucion
            )

        clusters[i].pop()
```

Esta función recursiva asigna los nodos uno a uno a alguno de los K clusters. Descarta ramas si:

- Se superó la distancia mínima encontrada hasta el momento
- Hay clusters vacíos
- Algún cluster no está conectado

Cuando se han asignado todos los nodos a los clusters, se verifica que todos los nodos estén conectados. Calcula la distancia de cáda cluster usando las distancias precalculadas. Si mejoramos la solución, se guarda.

En cuanto a la exploración, para cada nodo intenta agregarlo a alguno de los clusters. Para cada nodo:

- Calcula la nueva distancia parcial
- Si no supera la mejor encontrada, sigue recursivamente.
- Luego, deshace la asignación.

Resumidamente, el algoritmo busca encontrar una partición de K clusters conectados tal que la peor distancia sea lo más pequeña posible. El backtracking evitará ramas innecesarias que ya no mejoren la solución, usa distancias precalculadas para la eficiencia, y verifica la conectividad y vacíos para evitar clusteres inválidos.

3.2. Complejidad

Cuando hablamos de complejidad, sabemos que Backtracking no suele ser muy performante. En el peor de los casos, intenta asignar cada nodo a uno de los K clusters, donde obtendremos una complejidad de $O(K^n)$, donde n es la cantidad de nodos del grafo. Para cada asignación se puede verificar conectividad y calcular la distancia del clúster lo que también tiene su costo de $O(n+m)$ para la conectividad, y $O(n^2)$ para la distancia dentro de un cluster, porque se recorren pares dentro de cada clúster.

En la práctica, gracias a las optimizaciones de ordenamiento por grado, poda por distancia y clusters vacíos, y memoización de las distancias precalculadas, el algoritmo es efectivo para grafos relativamente pequeños, de 15 o 20 nodos aproximadamente, pero escala muy negativamente para los grafos grandes. Esto se podrá comprobar en las mediciones de tiempo realizadas.

3.3. Pruebas

Para verificar la funcionalidad del algoritmo, se realizaron pruebas usando los sets de datos y resultados esperados otorgados por la cátedra. A continuación, detallamos los resultados:

10-3.txt, K = 2

- Máxima distancia dentro del clúster: 2
- Cluster 1: [1, 3, 5, 7, 9, 4, 6, 2]
- Cluster 2: [8, 0]
- Tiempo de ejecución: 0.001s

10-3.txt, K = 5

- Máxima distancia dentro del clúster: 1
- Cluster 1: [1, 3, 5]
- Cluster 2: [7, 6]
- Cluster 3: [9, 4]
- Cluster 4: [8, 0]
- Cluster 5: [2]
- Tiempo de ejecución: 0.001s

22-3.txt, K = 3

- Máxima distancia dentro del clúster: 2
- Cluster 1: [19, 5, 4, 3, 16, 1, 21, 2, 0, 12, 11, 8, 6]
- Cluster 2: [9, 17, 18, 20]
- Cluster 3: [7, 10, 13, 15, 14]
- Tiempo de ejecución: 0.002s

22-3.txt, K = 4

- Máxima distancia dentro del clúster: 2
- Cluster 1: [19, 5, 4, 3, 16, 1, 21, 9, 2, 12, 11, 8, 20]

- Cluster 2: [7, 10, 0, 13, 14]
- Cluster 3: [17, 15]
- Cluster 4: [18, 6]
- Tiempo de ejecución: 0.001s

22-3.txt, K = 10

- Máxima distancia dentro del clúster: 1
- Cluster 1: [19, 4, 12]
- Cluster 2: [5, 16, 13]
- Cluster 3: [3, 21]
- Cluster 4: [1, 8]
- Cluster 5: [9, 20]
- Cluster 6: [2, 11]
- Cluster 7: [7, 17, 15]
- Cluster 8: [10, 14]
- Cluster 9: [0]
- Cluster 10: [18, 6]
- Tiempo de ejecución: 0.183s

22-5.txt, K = 2

- Máxima distancia dentro del clúster: 2
- Cluster 1: [10, 11, 14, 19, 4, 21, 1, 2, 16, 18, 17, 12, 13, 8, 7, 20, 9, 0, 15]
- Cluster 2: [6, 5, 3]
- Tiempo de ejecución: 0.001s

22-5.txt, K = 7

- Máxima distancia dentro del clúster: 1
- Cluster 1: [10, 19, 21]
- Cluster 2: [11, 14, 1, 20]
- Cluster 3: [4, 18, 12, 13]
- Cluster 4: [2, 16, 15]
- Cluster 5: [17, 5, 3]
- Cluster 6: [8, 0]
- Cluster 7: [7, 9, 6]
- Tiempo de ejecución: 0.003s

30-3.txt, K = 2

- Máxima distancia dentro del clúster: 3
- Cluster 1: [4, 13, 3, 9, 5, 12, 26, 0, 2, 14, 23, 22, 20, 16, 6, 19, 15, 21, 17, 7, 10, 28, 1, 25]
- Cluster 2: [24, 18, 8, 11, 29, 27]
- Tiempo de ejecución: 0.058s

30-3.txt, K = 6

- Máxima distancia dentro del clúster: 2
- Cluster 1: [4, 3, 13, 9, 12, 26, 0, 23, 22, 15, 18]
- Cluster 2: [5, 2, 14, 6, 19, 28, 1]
- Cluster 3: [24, 7, 10, 8, 11]
- Cluster 4: [20, 21, 25]
- Cluster 5: [16, 17, 27]
- Cluster 6: [29]
- Tiempo de ejecución: 1.807s

30-5.txt, K = 5

- Máxima distancia dentro del clúster: 2
- Cluster 1: [5, 14, 21, 23, 24, 2, 0, 25, 4, 15, 7, 1, 17, 19, 11, 12, 16, 10, 13, 22, 26, 20, 8, 18]
- Cluster 2: [6, 27]
- Cluster 3: [28, 29]
- Cluster 4: [3]
- Cluster 5: [9]
- Tiempo de ejecución: 0.091s

40-5.txt, K = 3

- Máxima distancia dentro del clúster: 2
- Cluster 1: [0, 32, 29, 24, 36, 33, 16, 34, 25, 13, 6, 7, 3, 10, 1, 2, 30, 37, 19, 38]
- Cluster 2: [27, 22, 14, 28, 9, 17, 12, 8, 18, 31, 35, 15, 39]
- Cluster 3: [26, 23, 5, 11, 21, 20, 4]
- Tiempo de ejecución: 30.832s

Para los grafos más grandes, por ejemplo **45-3.txt con K=7** y **50-3.txt con K = 3** no logramos terminar la ejecución en nuestros dispositivos, el tiempo se extendía a horas en lugar de segundos o minutos.

Como esperábamos, a medida que crece K y la cantidad de nodos, el algoritmo se vuelve muy ineficiente. Sin embargo, los resultados obtenidos son los esperados, este algoritmo es óptimo. En la sección de 'mediciones' de este informe, se puede ver bien detalladas las mediciones de tiempo junto a algunos gráficos.

4. Algoritmo por Programación Lineal Entera

En este caso propondremos un algoritmo de programación lineal entera que resuelva el mismo problema de optimización, y con el mismo objetivo que es encontrar la solución óptima. Para ello primero comenzaremos definiendo las variables del modelo:

Para cada vértice del grafo se define una variable $X_{i,c}$ binaria que vale 1 si el vértice i se encuentra en el cluster c. Luego se define una variable entera D que representa el máximo diámetro de todos los clusters

Considerando estas variables las restricciones a definir ahora para ellas son:

1. $\forall i : \sum_{c=1}^{k} X_{i,c} = 1$, es decir cada vértice debe estar en un y solo un cluster y no en más (pero tampoco puede quedarse sin asignar en alguno)
2. $\forall c : \forall i \neq j : D \geq dist(i,j) - M * (2 - X_{i,c} - X_{j,c})$, esto quiere decir que para dos vértices distintos que estén en un mismo cluster se debería cumplir que su distancia sea a lo sumo de valor D (el lado derecho de la igualdad queda solo el valor de la distancia entre los vértices), pero si no están en el mismo cluster (solo uno está o ninguno) entonces la restricción se relaja haciendo que puedan tener una distancia incluso mayor a D ya que este en ese caso no los restringe (el lado derecho de la igualdad es un valor muy pequeño y siempre se satisface). El valor de M se toma como la máxima distancia entre dos vértices del grafo

Finalmente, la función objetivo que se desea minimizar es:

$$MIN(D)$$

Ya que se busca minimizar el máximo de las distancias máximas (es decir lo que representa la variable D definida antes) para hacer que se cumpla la restricción de las distancias en cada cluster. Utilizando la librería Pulp, podemos implementar este modelo en Python de la siguiente forma:

```python
def k_clustering_pl(grafo, k):
    V = list(grafo.nodes())
    n = len(V)
    dist = {i: calcular_distancias_bfs(grafo, V[i]) for i in range(n)}

    M = max(dist[i][V[j]]
            for i in range(n)
            for j in range(n)
            if V[j] in dist[i])

    prob = pulp.LpProblem("Min_K_Clustering", pulp.LpMinimize)

    x = {(i,c): pulp.LpVariable(f"x_{i}_{c}", cat="Binary")
         for i in range(n) for c in range(k)}
    D = pulp.LpVariable("D", lowBound=0, cat="Integer")


    for i in range(n):
        prob += pulp.lpSum(x[i,c] for c in range(k)) == 1


    for i in range(n):
        for j in range(i+1, n):
            d_ij = dist[i].get(V[j], M)
            for c in range(k):
                prob += D >= d_ij - M*(2 - x[i,c] - x[j,c])

    prob += D

    prob.solve()

    D_opt = int(pulp.value(D))
    clusters = [[] for _ in range(k)]
```

```
    for i in range(n):
        for c in range(k):
            if pulp.value(x[i,c]) > 0.5:
                clusters[c].append(V[i])


    return D_opt, clusters
```

4.1. Pruebas de ejecución

Para comparar este algoritmo con los resultados dados por backtracking, lo ejecutaremos con los casos de prueba provistos:

10-3.txt, K = 2

- Máxima distancia dentro del cluster: 2
- Cluster 1: [0, 4, 6]
- Cluster 2: [1, 8, 3, 5, 9, 2, 7]
- Tiempo de ejecución: 0.03532552719116211 segundos

10-3.txt, K = 5

- Máxima distancia dentro del cluster: 1
- Cluster 1: [0, 1]
- Cluster 2: [5, 2]
- Cluster 3: [3, 6, 7]
- Cluster 4: [8, 9]
- Cluster 5: [4]
- Tiempo de ejecución: 0.33311939239501953 segundos

22-3.txt, K = 3

- Máxima distancia dentro del cluster: 2
- Cluster 1: [3, 1, 18, 21, 8, 9, 6]
- Cluster 2: [0, 12, 16, 5, 19, 11, 7, 10, 14]
- Cluster 3: [13, 4, 2, 17, 15, 20]
- Tiempo de ejecución: 0.3616139888763428 segundos

22-3.txt, K = 4

- Máxima distancia dentro del cluster: 2
- Cluster 1: [0, 12, 10, 14]
- Cluster 2: [13, 4, 5, 7, 17, 15, 20]
- Cluster 3: [3, 16, 1, 8, 6, 11]
- Cluster 4: [18, 21, 9, 2, 19]

- Tiempo de ejecución: 0.4367411136627197 segundos

22-3.txt, K = 10

- Máxima distancia dentro del cluster: 1
- Cluster 1: [13, 14]
- Cluster 2: [18, 11]
- Cluster 3: [5, 10]
- Cluster 4: [7, 15]
- Cluster 5: [0, 4, 12]
- Cluster 6: [1, 8]
- Cluster 7: [3, 16]
- Cluster 8: [19, 20]
- Cluster 9: [2, 6]
- Cluster 10: [21, 9, 17]
- Tiempo de ejecución: 3443.3793137073517 segundos

22-5.txt, K = 2

- Máxima distancia dentro del cluster: 2
- Cluster 1: [0, 4, 7, 20, 11, 18, 5, 10, 2, 15, 12, 13, 6]
- Cluster 2: [16, 8, 21, 1, 14, 17, 19, 9, 3]
- Tiempo de ejecución: 0.17812299728393555 segundos

22-5.txt, K = 7

- Máxima distancia dentro del cluster: 1
- Cluster 1: [16, 10, 15]
- Cluster 2: [11, 2, 9]
- Cluster 3: [4, 12, 13]
- Cluster 4: [7, 14, 19, 6]
- Cluster 5: [1, 20, 18]
- Cluster 6: [5, 17, 3]
- Cluster 7: [0, 8, 21]
- Tiempo de ejecución: 11.340739965438843 segundos

30-3.txt, K = 2

- Máxima distancia dentro del cluster: 3
- Cluster 1: [0, 1, 14, 11, 29]

- Cluster 2: [20, 4, 2, 3, 13, 16, 23, 28, 5, 6, 12, 19, 15, 18, 21, 22, 26, 17, 24, 25, 7, 10, 8, 27, 9]
- Tiempo de ejecución: 1.4887099266052246 segundos

30-3.txt, K = 6

- Máxima distancia dentro del cluster: 2
- Cluster 1: [13, 12, 22, 10, 11, 29]
- Cluster 2: [28, 18, 7, 27, 9]
- Cluster 3: [23, 21, 25]
- Cluster 4: [3, 16, 8]
- Cluster 5: [20, 2, 1, 14, 5, 6, 19]
- Cluster 6: [0, 4, 15, 26, 17, 24]
- Tiempo de ejecución: 179.76169610023499 segundos

30-5.txt, K = 5

- Máxima distancia dentro del cluster: 2
- Cluster 1: [0, 25, 14, 7, 4, 19, 6, 13]
- Cluster 2: [15, 11, 12, 22]
- Cluster 3: [9, 17, 5, 23, 28]
- Cluster 4: [1, 2, 16, 8, 18]
- Cluster 5: [24, 27, 20, 21, 3, 10, 26, 29]
- Tiempo de ejecución: 1.8028182983398438 segundos

40-5.txt, K = 3

- Máxima distancia dentro del cluster: 2
- Cluster 1: [0, 32, 5, 30, 38, 26, 9, 17, 35, 39]
- Cluster 2: [36, 33, 14, 3, 7, 29, 37, 27, 21, 34, 4, 20, 23, 18, 12]
- Cluster 3: [6, 10, 11, 16, 1, 22, 2, 28, 24, 25, 13, 8, 19, 31, 15]
- Tiempo de ejecución: 4.479216814041138 segundos

45-3.txt, K = 7

- Máxima distancia dentro del cluster: 2
- Cluster 1: [0, 42, 18, 22, 26, 28, 19]
- Cluster 2: [1, 2, 16, 33, 3, 20, 44]
- Cluster 3: [31, 5, 21, 37, 12]
- Cluster 4: [6, 9, 41, 13, 23, 36, 15]
- Cluster 5: [4, 7, 39, 35, 34]

- Cluster 6: [32, 29, 38, 10, 8, 43]
- Cluster 7: [24, 27, 14, 30, 17, 25, 11, 40]
- Tiempo de ejecución: 13314.964541196823 segundos

50-3.txt, K = 3

- Máxima distancia dentro del cluster: 3
- Cluster 1: [0, 22, 18, 32, 15, 46, 13, 31, 35, 7, 30, 26, 17, 25, 12, 39, 29]
- Cluster 2: [11, 9, 1, 42, 48, 5, 43, 10, 6, 37, 47, 44, 49, 34, 16]
- Cluster 3: [28, 45, 4, 24, 20, 2, 21, 19, 3, 23, 36, 33, 40, 8, 41, 14, 38, 27]
- Tiempo de ejecución: 2.3777973651885986 segundos

Como se puede observar en este caso el algoritmo obtuvo la solución óptima en todos los casos (al menos expresado en la máxima distancia dentro de cada cluster, ya que cómo se repartan los vértices puede variar siempre y cuando se respete esa distancia máxima) y en cuanto a los tiempos de ejecución se observa que en ciertos casos con un valor un poco más grande de cantidad de clusters K estos tiempos se incrementan considerablemente, esto entendemos se debe a que el algoritmo tiene una mayor cantidad de combinaciones distintas que probar y también se incrementan la cantidad de restricciones (teniendo en cuenta que en estas se tienen en cuenta la cantidad de clusters que haya). Comparado con el algoritmo de BT se nota una mejora para valores de K chicos pero a valores de K más grandes el tiempo que tarda tiende a igualarse

5. Algoritmo de aproximación de Louvain

5.1. Introducción

El método de Louvain es un algoritmo de detección de comunidades, diseñado para particionar grafos en grupos (o clusters) de nodos altamente conectados internamente y poco conectados entre sí. El objetivo principal es maximizar la **modularidad**, una métrica para definir la densidad de aristas dentro de una comunidad. A continuación, presentamos una implementación del algoritmo adaptado para resolver el problema tratado en este informe, el problema de los K clusters. Además, proporcionaremos información respecto a la aproximación que brinda este algoritmo respecto de la solución óptima, pruebas con diferentes sets de datos, etc.

5.2. Algoritmo

Inicialmente, se asigna cada nodo a su propia comunidad única. Calculamos la modularidad inicial Q. La modularidad se modela bajo la siguiente fórmula:

$$Q = \frac{1}{2m} \sum_i \sum_j \left(peso(v_i, v_j) - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

Donde:

- $peso(v_i, v_j)$: el peso de la arista entre i y j (0 si no están unidos, y en nuestro caso 1 sí si lo están).
- k_i: es la suma de las aristas del vértice i (en nuestro caso, su grado).
- $2m$: es la suma de todos los pesos de las aristas.
- c_i: es la comunidad del vértice i.
- δ: función delta de Kronecker, que es básicamente 1 si ambas comunidades son iguales, 0 si son diferentes.

Figura 1: Fórmula de la modularidad

Para poder obtener este dato, implementamos la siguiente función en Python:

```python
def calcular_modularidad(grafo, asignacion_comunidades):
    m = grafo.size(weight='weight')

    grado = {n: grafo.degree(n, weight='weight') for n in grafo.nodes()}
    acumulado = 0.0

    for i in grafo.nodes():
        for j, datos in grafo[i].items():
            peso_ij = datos.get('weight', 1.0)
            if asignacion_comunidades[i] == asignacion_comunidades[j]:
                acumulado += peso_ij - (grado[i] * grado[j]) / (2.0 * m)

    return acumulado / (2.0 * m)
```

El algoritmo inicia en una primera fase de movimientos locales, donde para cada nodo se lo retira temporalmente de su comunidad y se evalúa la ganancia de modularidad ΔQ al instertarlo en la comunidad de cada vecino (así también como el coste de sacarlo de su comunidad original).

Se elige el movimiento que maximiza la mejora de ΔQ. Incluso, podría ser quedarse en la misma comunidad. Este proceso se repite de manera iterativa hasta que ningún nodo cambie de comunidad.

Esta primera fase la podemos ver implementada en el siguiente algoritmo:

```python
def primera_fase(grafo):
    # cada nodo en su propia comunidad
    asignacion = {n: n for n in grafo.nodes()}
    hubo_mejora = True

    while hubo_mejora:
        hubo_mejora = False
        for nodo in grafo.nodes():
            comunidad_original = asignacion[nodo]
            # retirar nodo
            asignacion[nodo] = None

            # comunidades candidatas
            comunidades_vecinas = {
                asignacion[vecino]
                for vecino in grafo[nodo]
                if asignacion[vecino] is not None
            }
            candidatas = comunidades_vecinas.union({comunidad_original})

            mejor_delta = 0.0
```

```
            mejor_comunidad = comunidad_original

            # probar candidatas
            for com_cand in candidatas:
                asignacion[nodo] = com_cand
                delta_meter_nodo = calcular_delta_modularidad(grafo, nodo, com_cand
    , asignacion) # deltaQ(i -> com_cand)
                delta_sacar_nodo_comunidad_actual = -calcular_delta_modularidad(
    grafo, nodo, comunidad_original, asignacion) # deltaQ(D -> i) = -deltaQ(i -> D)
                if delta_meter_nodo + delta_sacar_nodo_comunidad_actual >
    mejor_delta:
                    mejor_delta = delta_meter_nodo +
    delta_sacar_nodo_comunidad_actual
                    mejor_comunidad = com_cand

            # asignar la mejor
            asignacion[nodo] = mejor_comunidad
            if mejor_comunidad != comunidad_original:
                hubo_mejora = True

    return asignacion
```

Este nos devolverá un diccionario de asignaciones, donde se indica la comunidad a la que pertenece cada nodo. Por otro lado, el cálculo de ΔQ sigue la siguiente fórmula:

$$\Delta Q(i \to C) = \left[\frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

Donde...

Σ_{in} es la suma de los pesos de entre los nodos *en* C

Σ_{tot} es la suma de los pesos totales de los nodos de C

$k_{i,in}$ es la suma de los pesos entre i y los nodos de C

k_i es el grado del nodo i

Figura 2: Enter Caption

Y está implementada de la siguiente manera:

```
def calcular_delta_modularidad(grafo, nodo, comunidad_destino,
    asignacion_comunidades):
    m = grafo.size(weight='weight')
    k_i = grafo.degree(nodo, weight='weight')

    # k_i,in: peso de aristas entre nodo y miembros de comunidad_destino
    k_i_in = sum(
        datos.get('weight', 1.0)
        for vecino, datos in grafo[nodo].items()
        if asignacion_comunidades.get(vecino) == comunidad_destino
    )

    # suma de grados de todos los nodos en comunidad_destino
    suma_tot = sum(
        grafo.degree(n, weight='weight')
        for n, com in asignacion_comunidades.items()
        if com == comunidad_destino
    )

```

```
    # peso de aristas internas de comunidad_destino
    miembros = [n for n, com in asignacion_comunidades.items() if com ==
    comunidad_destino]
    subgrafo = grafo.subgraph(miembros)
    suma_in = subgrafo.size(weight='weight')

    # delta Q -> Formula de diapositivas "06 - Comunidades"
    parte1 = (suma_in + k_i_in) / (2.0 * m) - ((suma_tot + k_i) / (2.0 * m))**2
    parte2 = suma_in / (2.0 * m) - (suma_tot / (2.0 * m))**2 - (k_i / (2.0 * m))**2
    return parte1 - parte2
```

El algoritmo de Louvain continúa en una segunda fase de agregación. Cada comunidad resultante será colapsada o unida en un único nodo. Así, se construye un grafo donde el peso de la arista entre dos super-nodos es la suma de los pesos de todas las conexiones entre sus nodos originales (y los bucles representarán las aristas internas).

Podemos ver esta fase implementada en el siguiente código:

```
def segunda_fase(grafo, asignacion_comunidades):
    G2 = nx.Graph()
    comunidades = set(asignacion_comunidades.values())
    G2.add_nodes_from(comunidades)

    for u, v, datos in grafo.edges(data=True):
        peso = datos.get('weight', 1.0)
        cu = asignacion_comunidades[u]
        cv = asignacion_comunidades[v]
        if G2.has_edge(cu, cv):
            G2[cu][cv]['weight'] += peso
        else:
            G2.add_edge(cu, cv, weight=peso)

    return G2
```

Luego de ambas fases, se regresa a la primer fase sobre el grafo reducido. Esto va a generar una "jerarquia"de particiones hasta lograr la convergencia. A continuación, mostramos la función principal del algoritmo:

```
def louvain(grafo_original, K):
    grafo_actual = grafo_original.copy()
    mapeo_global = {n: n for n in grafo_original.nodes()}

    # Modularidad inicial
    Q_prev = calcular_modularidad(grafo_original, mapeo_global)

    while True:
        asignacion = primera_fase(grafo_actual)

        nuevo_mapeo = {
            nodo_orig: asignacion[mapeo_global[nodo_orig]]
            for nodo_orig in mapeo_global
        }

        Q_new = calcular_modularidad(grafo_original, nuevo_mapeo)

        # Solo confirmamos el cambio si hay mejora
        if Q_new > Q_prev:
            mapeo_global = nuevo_mapeo
            Q_prev = Q_new
            grafo_actual = segunda_fase(grafo_actual, asignacion)
            if len(set(mapeo_global.values())) <= K:
                return mapeo_global
        else:
            # No hay ganancia
            break

    # Fusionamos si hay mas de K
    asign = mapeo_global.copy()
    while len(set(asign.values())) > K:
        Q_actual = calcular_modularidad(grafo_original, asign)
```

```
        mejor_ganancia = None
        mejor_asignacion = None

        comunidades = list(set(asign.values()))
        # pruebar todas las parejas de comunidades
        for i in range(len(comunidades)):
            for j in range(i+1, len(comunidades)):
                c1, c2 = comunidades[i], comunidades[j]
                # fusionar c2 en c1 temporalmente
                temp = {}
                for nodo, c in asign.items():
                    temp[nodo] = c1 if c == c2 else c
                Q_temp = calcular_modularidad(grafo_original, temp)
                gain = Q_temp - Q_actual
                if mejor_ganancia is None or gain > mejor_ganancia:
                    mejor_ganancia = gain
                    mejor_asignacion = temp
        asign = mejor_asignacion
    return asign
```

Como podemos ver, hay una adpatación para el problema de los clusters. Una vez que alcanzamos K comunidades o menos, el algoritmo terminará allí mismo. Por otro lado, en caso de no converger hasta menos de K comunidades, el algoritmo realizará un "mergeo"de las comunidades que menos perjudiquen (o más mejores) la modularidad, hasta alcanzar los K clusters.

5.3. Complejidad

- Primera Fase: En cada iteración se visitan todos los nodos y para cada uno se prueban tantas comunidades candidatas como vecinos tenga. En el peor de los casos se aproxima a O(E).
- Segunda Fase: Comprime el grafo en $O(E)$.
- El número de iteraciones hasta convergencia no suele ser muy grande, es prácticamente constante. La complejidad observada es aproximadamente $O(nlogn)$ siendo n la cantidad de nodos.
- La etapa de fusión luego del proceso para garantizar K clusters explora todas las parejas de comunidades. Para cada uno se crea una asignacion temporal y se mide la modularidad (que tiene costo E ya que recorre las aristas). Entonces, el costo de esto es $O(c^2E)$ siendo c el número actual de clusters.

6. Mediciones

Se realizaron pruebas empíricas para evaluar el rendimiento de dos algoritmos exactos para el problema de optimización de *Clustering por bajo diámetro*: uno basado en **programación lineal** y otro en **backtracking**. El objetivo fue analizar cuán viable resulta su uso en la práctica en términos de tiempo de ejecución.

Los resultados mostraron que ambos algoritmos, debido a su complejidad temporal, son altamente sensibles a cambios en los parámetros de entrada, principalmente la cantidad de nodos n y la cantidad de clústers k. En general, el tiempo de ejecución crece de forma exponencial con estos valores, lo que limita severamente su aplicabilidad en escenarios reales. Esto volvió inviable realizar mediciones significativas para valores altos de n o k, debido a los tiempos de cómputo excesivos requeridos.

En el caso del algoritmo de programación lineal, el número de restricciones del modelo, que es del orden de $\mathcal{O}(n^2 \cdot k)$, impacta directamente en el rendimiento del *solver*, haciendo que muchas instancias resulten intratables incluso con un número moderado de nodos, especialmente si el valor de k es elevado.

El algoritmo de backtracking presentó un comportamiento aún más limitado, debido a su espacio de búsqueda de tamaño $\mathcal{O}(k^n)$. Esto lo vuelve impráctico incluso para grafos pequeños o moderadamente densos, ya que el tiempo de ejecución se vuelve rápidamente inmanejable.

Además, se observaron variaciones significativas en los tiempos de ejecución entre instancias con la misma cantidad de nodos. Por ejemplo, para dos grafos con $n = 30$ y $k = 3$, aquellos cuya estructura se acercaba a la de un cliqué requerían tiempos de cómputo mucho mayores. Esto evidencia que la estructura del grafo también influye fuertemente en la dificultad del problema. En particular, grafos más densos o con mayor conectividad entre clústers tienden a incrementar la complejidad del proceso de búsqueda de soluciones óptimas.

En resumen, si bien ambos algoritmos producen soluciones óptimas, su uso en la práctica está fuertemente restringido a instancias pequeñas o muy controladas. Para aplicaciones reales o de mayor escala, resulta necesario recurrir a enfoques heurísticos o aproximados que permitan obtener soluciones razonables en tiempos acotados.

6.1. Comparación con Louvain

Con el objetivo de encontrar alternativas para resolver instancias más grandes, se compararon los resultados del algoritmo de **backtracking** con los obtenidos mediante el algoritmo de **Louvain**, el cual, aunque no busca minimizar el diámetro de k clústers, al maximizar la modularidad de la partición y luego hacer un posprocesado para dejar k clústers, nos da una solución bastante cercana a lo que se necesita, gastando una cantidad mucho menor de tiempo.

Se utilizó una serie de grafos de tamaño creciente (entre 10 y 40 nodos), y se calculó el *diámetro máximo por clúster* resultante de aplicar cada algoritmo. En la Figura se puede observar cómo se comportan ambos algoritmos respecto a esta métrica.

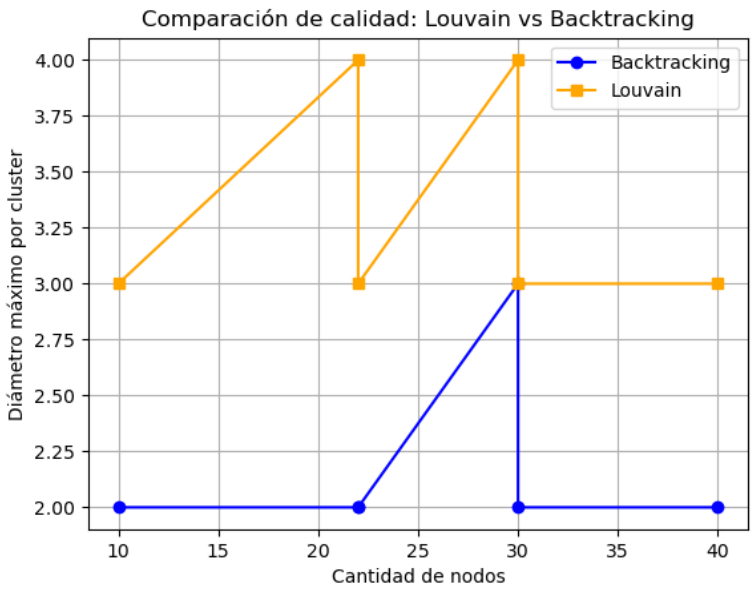

Figura 3: Comparación de los resultados entre Louvain y Backtracking

A continuación dejamos una tabla con los resultados obtenidos en las distintas instancias de ejecución:

n	k	Tiempo BT (s)	Distancia BT	Tiempo LV (s)	Distancia LV	Ratio LV/BT
10	3	0.00	2	0.03	3	1.50
22	3	0.00	2	0.16	4	2.00
22	5	0.03	2	0.18	3	1.50
30	3	0.77	3	0.14	4	1.33
30	5	0.30	2	0.29	3	1.50
40	5	29.50	2	0.46	3	1.50

Cuadro 1: Comparación empírica entre los algoritmos de Backtracking y Louvain.

A partir de los datos obtenidos en las instancias evaluadas, se puede calcular el cociente entre las distancias máximas por clúster de Louvain y las de backtracking:

$$\rho = \frac{D_{\text{LV}}}{D_{\text{BT}}}$$

El valor máximo observado es:

$$\rho_{\max} = 2{,}00$$

- **Cota empírica:** En las instancias consideradas, el algoritmo de Louvain se comportó como, a lo sumo, una **2-aproximación** para el problema de clustering por bajo diámetro.
- **Aclaración:** Esta cota se basa únicamente en las instancias evaluadas, las cuales, debido a la complejidad temporal del algoritmo de backtracking, corresponden a grafos de tamaño reducido. Para obtener una cota más representativa y generalizable, sería necesario considerar una mayor variedad de grafos (en cuanto a tamaño, densidad y distribución), y calcular:

$$\max_{\text{todas las instancias}} \left(\frac{D_{\text{LV}}}{D_{\text{BT}}} \right)$$

En resumen, aunque el algoritmo de *backtracking* encuentra soluciones óptimas, su uso en la práctica está fuertemente restringido a instancias pequeñas o muy controladas. Para aplicaciones reales o de mayor escala, resulta necesario recurrir a enfoques aproximados que permitan obtener soluciones razonables en tiempos acotados. Uno de esos enfoques aproximados podría ser el algoritmo de Louvain, que, según la cota empírica de aproximación previamente mostrada, logra soluciones razonablemente cercanas al óptimo, a pesar de no optimizar explícitamente la métrica de diámetro. Además, presenta una mejora sustancial en tiempo de ejecución.

7. Conclusión

Los métodos exactos, como backtracking y programación lineal, son útiles para instancias pequeñas, ya que garantizan soluciones óptimas, pero presentan limitaciones significativas en escalabilidad. Por otro lado, el algoritmo de Louvain ofrece una solución aproximada eficiente para grandes volúmenes de datos, proporcionando resultados razonablemente cercanos a la óptima en términos de distancia máxima. La modularidad es una métrica valiosa para identificar comunidades reales, aunque no minimiza explícitamente el diámetro. En definitiva, para aplicaciones prácticas con grandes grafos, las aproximaciones constituyen una herramienta fundamental para evitar los largos tiempos de ejecución.

8. Correcciones

A continuación, presentamos las correcciones correspondientes al informe. Se muestran y se detallan diferentes mediciones de tiempo para los algoritmos con sus respectivos gráficos. Para

cada uno, haremos un análisis y comentaremos las dificultades encontradas.

8.1. Backtracking

Para Backtracking hemos realizado dos mediciones de tiempo:

- Una medición dejando un valor K de clústers fijo, y usando diferentes cantidades de nodos.
- Otra medición usando una cantidad de nodos fija, y diferentes valores de K.

En primer lugar, se muestra la medición hecha para un K = 5, variando diferentes tamaños de grafos:

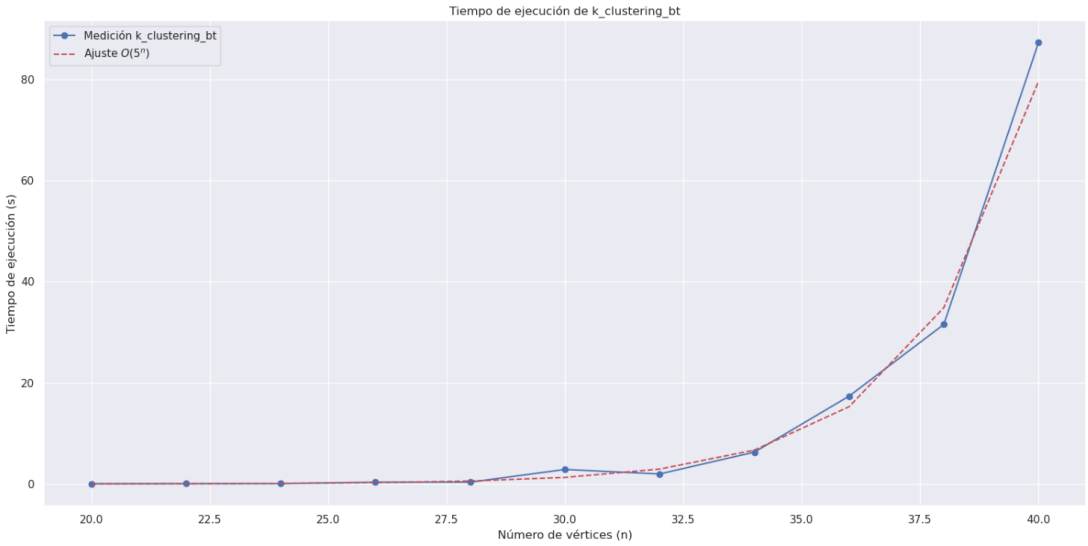

Figura 4: K = 5, N varia

Como vemos en el gráfico, la cantidad de tiempo que demora en ejecutar el algoritmo es cada vez más grande. De hecho, comprobamos la complejidad teórica de $O(K^n)$ haciendo un ajuste con $O(5^n)$. Como habíamos mencionado en la sección de Backtracking, este algoritmo se vuelve muy poco performante para grafos cada vez más grandes y densos, por lo que se nos ha dificultado ejecutar mediciones de tiempo para grafos mucho mayores, ya que las ejecuciones no solían terminar.

Esto se ve aún más reflejado para la segunda medición que realizamos: Manteniendo una cantidad fija de 20 nodos, y variando la cantidad de clusters:

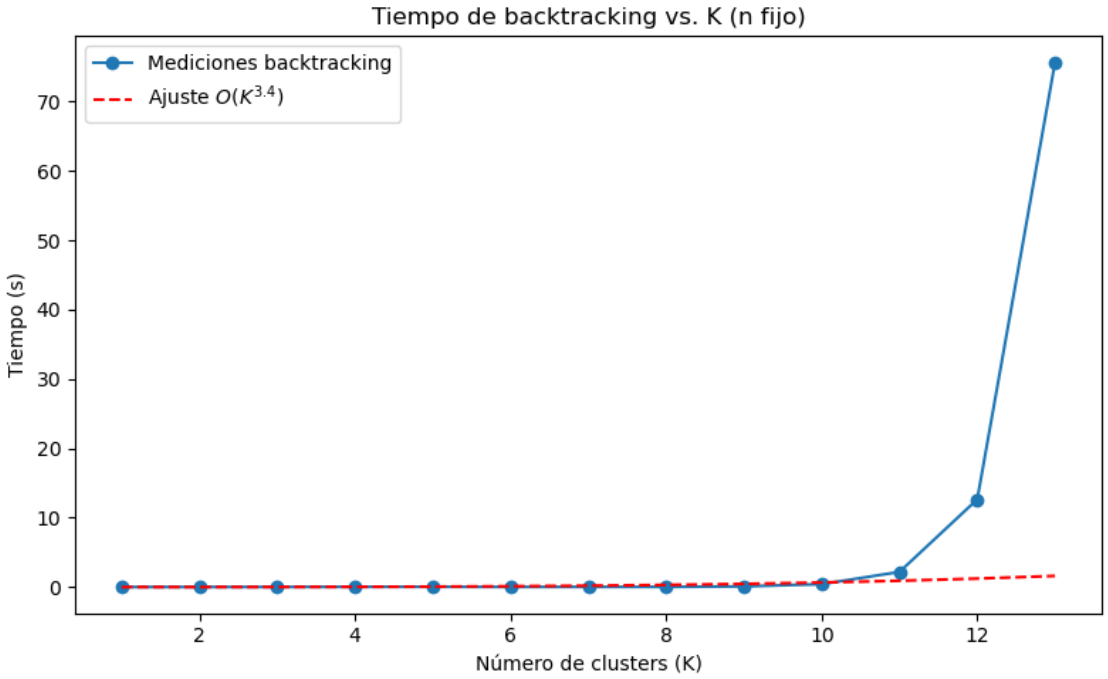

Figura 5: 20 Nodos fijos, K varia

En este grafico podemos ver un crecimiento muy violento en los tiempos de ejecución. De hecho, para valores de K apenas más grandes, como 15, el algoritmo no lograba terminar de ejecutar en un tiempo razonable. Eso hizo que se dificulte el ajuste mediante cuadrados mínimos ya que no había una función que se aproxime lo suficiente para esta cantidad de valores tan pequeña. Para un ajuste más óptimo, deberíamos contar con una muestra de muchos K diferentes, pero se nos hizo imposible debido a la performance.

Tiene sentido que este gráfico crezca muy violentamente, ya que si recordamos la complejidad de este algoritmo: $O(K^n)$ ahora teniendo un valor de nodos fijo se vuelve $O(K^{20})$). Si vamos incrementando el valor de K cada vez más, la complejidad crece cada vez más.

8.2. Programación lineal

Para el algoritmo de Programación Lineal realizamos también dos mediciones de tiempo:

- Una medición manteniendo un valor de $K = 5$ fijo y variando la cantidad de nodos.
- Otra medición manteniendo una cantidad fija de nodos ($N = 20$) y variando el valor de K.

En primer lugar, se presenta la medición realizada con $K = 5$, variando el tamaño de los grafos:

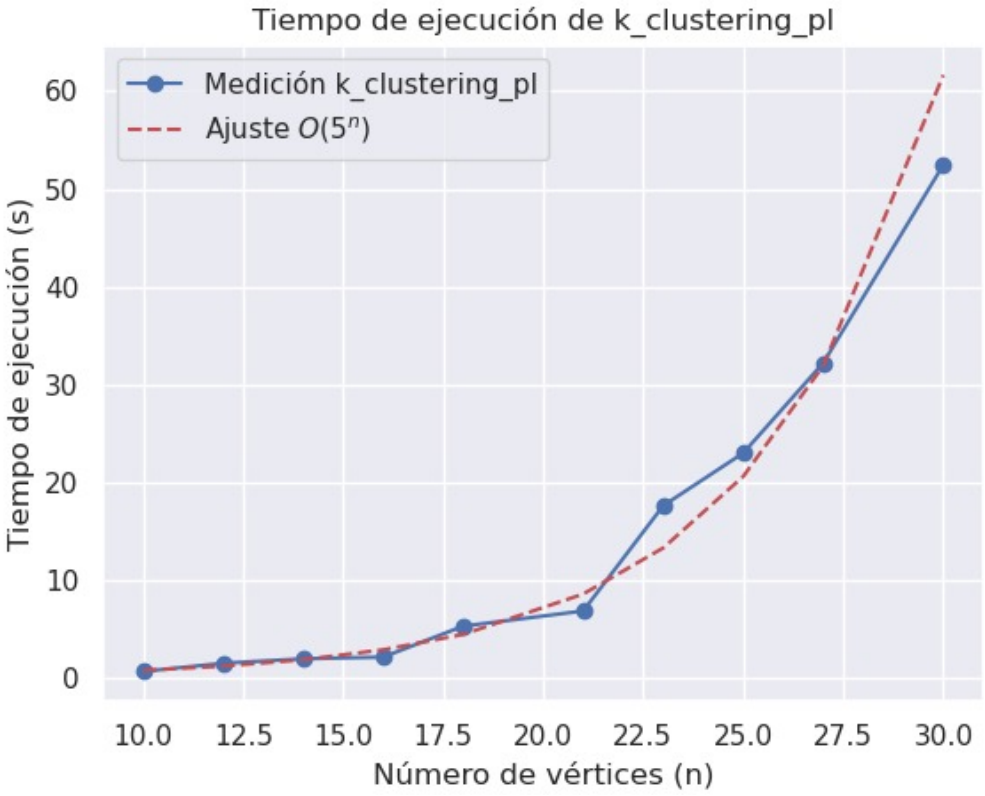

Figura 6: Nodos varian, K=5

Como se puede observar en el gráfico, el tiempo de ejecución del algoritmo aumenta significativamente a medida que aumenta la cantidad de nodos. De hecho, al ajustar los datos con una función de la forma $O(5^n)$, se aprecia que el crecimiento es exponencial, lo cual concuerda con el análisis teórico del problema.

El tiempo de resolución sigue estando fuertemente condicionado por el tamaño del grafo, especialmente en instancias densas o con muchas restricciones.

En segundo lugar, se muestra la medición realizada con una cantidad fija de 20 nodos, variando el número de clusters K:

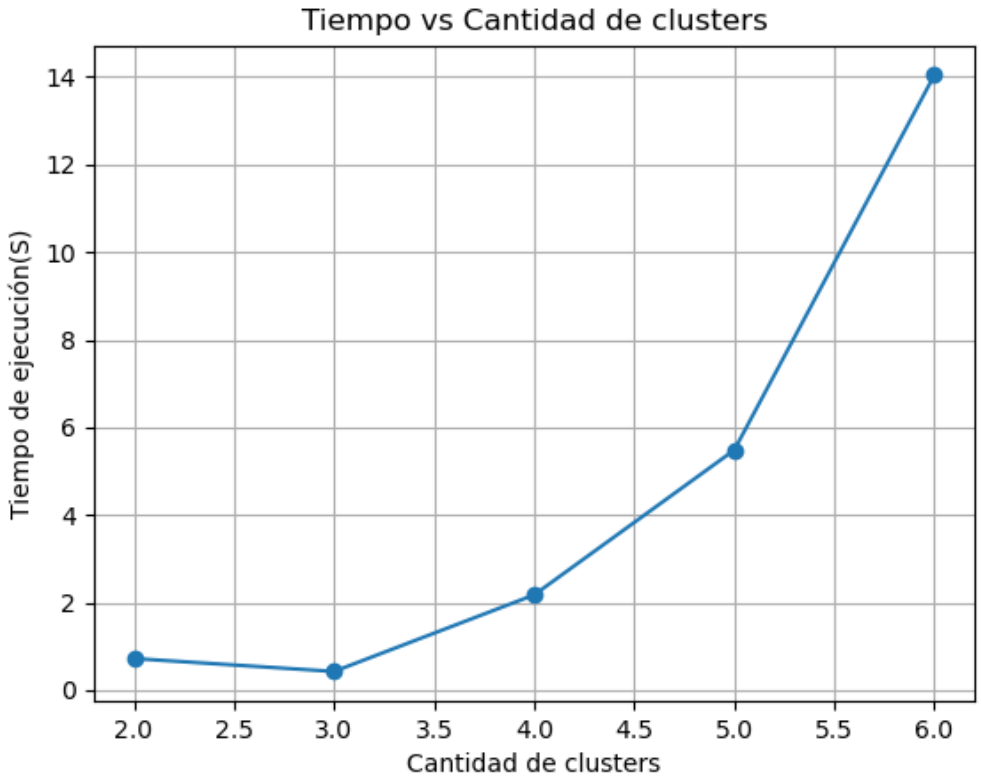

Figura 7: N = 20, K varian

En este gráfico se puede ver que, al incrementar el valor de K, también crece el tiempo de ejecución rapidamente pero en menor medida que cuando se aumenta la cantidad de nodos. Esto puede explicarse por la complejidad del modelo de programación lineal, ya que al aumentar K, también crece la cantidad de variables y restricciones pero aumentar **N** el efecto en el tiempo de ejecución es mayor debido a que el numero de restricciones del algoritmo por programación lineal crece de la siguiente manera $\mathcal{O}(n^2 \cdot k)$.

A diferencia de Backtracking, el algoritmo basado en PL muestra un comportamiento más controlado y escalable, permitiendo ejecutar instancias más grandes en tiempos razonables. Aun así, la cantidad de clusters sigue teniendo un impacto directo en la dificultad del problema.

8.3. Comparativa entre Backtracking y Programación Lineal

En esta sección comparamos los resultados de tiempo obtenidos para los algoritmos de Backtracking y Programación Lineal, con dos sets de prueba distintos de prueba: variando la cantidad de nodos n con un valor fijo de clusters K, y variando el valor de K con una cantidad fija de nodos.

Caso 1: $K = 5$ fijo, n variable

En este caso, el tiempo de ejecución de ambos algoritmos crece a medida que la cantidad de nodos crece. Sin embargo, la diferencia en la escalabilidad es clara:

- El algoritmo de Backtracking crece de forma exponencial y rápidamente se vuelve inviable para grafos grandes. El ajuste a $O(5^n)$ muestra una correspondencia fuerte con el comportamiento empírico.

- El algoritmo de PL, aunque también incrementa su tiempo con el tamaño del grafo, lo hace de manera más controlada. Su crecimiento sigue siendo considerable, pero permite trabajar con instancias más grandes que las posibles con Backtracking.

Esto refleja que la formulación mediante programación lineal ofrece mejor performance para grafos de tamaño moderado, mientras que el enfoque exhaustivo de Backtracking no es tan escalable a medida que aumenta n.

Caso 2: $n = 20$ fijo, K variable

En este segundo caso, el comportamiento vuelve a diferenciarse notablemente:

- El algoritmo de Backtracking presenta un crecimiento abrupto en los tiempos de ejecución a partir de ciertos valores de K. Para valores como $K = 13$, los tiempos superan los 70 segundos, y para valores mayores el algoritmo directamente no logra completar la ejecución mientras que para valores muy bajos el tiempo de ejecución es bajo.
- En contraste, PL presenta un crecimiento más gradual, aunque sí se ve afectado por el aumento de K debido a la mayor complejidad del modelo que se genera. Lo que como se ve en el gráfico, con valores más bajos de k su tiempo de ejecución es mayor que backtracking.

Conclusión general

A partir de ambos escenarios, podemos concluir que:

- **Backtracking**, aunque es óptimo, no escala bien ni con el número de nodos ni con la cantidad de clusters. Su uso queda restringido a instancias pequeñas.
- **Programación Lineal** ofrece una alternativa ligeramente mas performante para instancias de tamaño moderado, aunque también tiene limitaciones en instancias muy grandes o con muchos clusters.

Esto refuerza la necesidad de buscar algoritmos más eficientes o aproximaciones, como podría ser louvain con su complejidad $O(nlogn)$, cuando se requiere escalar a instancias reales o más complejas.