.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo práctico 3: Comunidades NP-Completas

6 de julio de 2025

Integrantes

Alumno	Padrón
Alan Valdevenito	107585
Ignacio Zanoni Mutti	110884
Francisco Gutierrez	103543

Índice

1. Demostración

Problema de **Clustering por bajo diámetro**: Dado un grafo no dirigido y no pesado, un numero entero k y un valor C, ¿es posible separar los vértices en a lo sumo k grupos/clusters disjuntos, de tal forma que todo vértice pertenezca a un cluster, y que la distancia máxima dentro de cada cluster sea a lo sumo C? (Si un cluster queda vacío o con un único elemento, considerar la distancia máxima como 0).

Al calcular las distancias se tienen en cuenta tanto las aristas entre vértices dentro del cluster, como cualquier otra arista dentro del grafo.

Para realizar la demostracion utilizaremos el problema de K-Coloreo el cual vimos en clase que es NP-Completo.

Problema de **K-Coloreo**: Dado un grafo y K colores diferentes, ¿es posible colorear los vertices con a lo sumo k colores de forma tal que los vertices adyacentes no compartan color?.

Para demostrar que el problema de **Clustering por bajo diámetro** es un problema NP-Completo debemos:

- Demostrar que el problema se encuentra en NP
- Reducir el problema de **K-Coloreo** al problema de **Clustering por bajo diámetro**:
  K-Coloreo $\leq_P$ **Clustering por bajo diámetro**

1.1. Clustering por bajo diámetro se encuentra en NP

Para que el problema se encuentre en NP, debe haber un verificador eficiente.

En otras palabras, debe haber un verificador que ejecute en tiempo polinomial.

Verificador: Recibe una instancia del problema y una solucion.

```python
# Al calcular las distancias se tienen en cuenta tanto las aristas entre vertices dentro del cluster, como cualquier otra arista dentro del grafo

def bfs_max_dist_en_grafo_completo(grafo, cluster, origen):
    visitados = set()
    visitados.add(origen)

    distancias = {origen: 0}
    cola = deque([origen])

    max_dist = 0

    while cola:
        v = cola.popleft()

        for w in grafo.adyacentes(v):
            if w not in visitados:
                visitados.add(w)
                distancias[w] = distancias[v] + 1
                cola.append(w)

    # Solo consideramos distancias hacia otros vertices del cluster
    for nodo in cluster:
        if nodo != origen and nodo in distancias:
            max_dist = max(max_dist, distancias[nodo])

    return max_dist

def verificador(grafo, k, C, clusters):

    # Validamos que la solucion tenga a lo sumo k clusters
    if len(clusters) > k: return False

    clusters = [set(c) for c in clusters]

    # Validamos que todo vertice pertenece a un cluster
    for v in grafo.obtener_vertices():
        pertenece = False

        for c in clusters:
            if v in c:
                pertenece = True
                break

        if not pertenece: return False

    # Validamos que la distancia maxima dentro de cada cluster sea a lo sumo C
    for cluster in clusters:
        for v in cluster:
            max_dist = bfs_max_dist(grafo, cluster, v)

            if max_dist > C: return False

    return True
```

Complejidad: O(k x W x (V + E)) con k la cantidad de clusters, W la cantidad maxima de vertices dentro de un cluster, V la cantidad de vertices y E la cantidad de aristas

- Validar que la solucon tenga a lo sumo k clusters tiene una complejidad de O(1)
- Validar que todo vertice pertenece a un cluster tiene una complejidad de O(V x k) con V la cantidad de vertices y k la cantidad de clusters
- Validar que la distancia maxima dentro de cada cluster sea a lo sumo C: Para cada cluster usamos un recorrido BFS desde cada vertice dentro del cluster. El recorrido BFS tiene una complejidad de O(V + E) con V la cantidad de vertices y E la cantidad de aristas. Luego, como se ejecuta un BFS por cada cada cluster y por cada vertice dentro del cluster tiene una complejidad de O(k x W x (V + E)) con k la cantidad de clusters, W la cantidad maxima de vertices dentro de un cluster, V la cantidad de vertices y E la cantidad de aristas.

Luego, se ejecuta en tiempo polinomial, lo cual quiere decir que el problema se encuentra en NP.

1.2. Clustering por bajo diámetro es un problema NP-Completo

Debemos demostrar que **K-Coloreo** $\leq_P$ **Clustering por bajo diámetro**

1.2.1. Reduccion planteada

¿Podemos resolver el problema de **K-Coloreo** utilizando la solucion del problema de **Clustering por bajo diametro**?.

Vamos a utilizar una caja negra que resuelve el problema de **Clustering por bajo diametro** para resolver el problema de **K-Coloreo**.

El problema de **K-Coloreo** recibe un grafo G' y un valor k' que representa la cantidad de colores.

Transformacion del problema: Transformamos la entrada del problema de **K-Coloreo** en una entrada del problema **Clustering por bajo diametro**

1. El grafo G' coincide con el grafo G.
2. El valor de k' coincide con el valor de k.
3. Definimos el valor de C en cero. Es decir que el diametro es cero.

Esta transformacion requiere una cantidad de pasos polinomiales.

A continuacion, para demostrar que la reduccion es correcta, debemos demostrar que:

Hay solucion de **K-Coloreo** con a lo sumo k' colores, si y solo si, hay solucion para el problema de **Clustering por bajo diametro** con a lo sumo k clusters.

1.2.2. Si hay K-Coloreo, entonces hay Clustering por bajo diametro

Hipotesis: Hay solucion de **K-Coloreo** con a lo sumo k' colores.

Tesis: Hay solucion para el problema de **Clustering por bajo diametro** con a lo sumo k clusters.

Dada nuestra hipotesis, existen k' grupos de vertices coloreados con distinto color.

Como no hay aristas entre vertices del mismo color, el subgrafo resultante es un conjunto de vertices aislados con lo cual tienen distancia maxima 0.

Luego, cada cluster tiene diametro 0 y hay a lo sumo k clusters.

1.2.3. Si hay Clustering por bajo diametro, entonces hay K-Coloreo

Hipotesis: Hay solucion para el problema de **Clustering por bajo diametro** con a lo sumo k clusters.

Tesis: Hay solucion de **K-Coloreo** con a lo sumo k' colores.

Dada nuestra hipotesis, existen k clusters disjuntos de tal forma que todo vertice pertenece a un cluster y la distancia maxima dentro de cada cluster es 0.

Luego, que la distancia maxima dentro de cada cluster sea cero quiere decir que no hay aristas entre vertices del mismo cluster. Esto es porque si estuvieran conectados por una arista si distancia seria mayor o igual a 1.

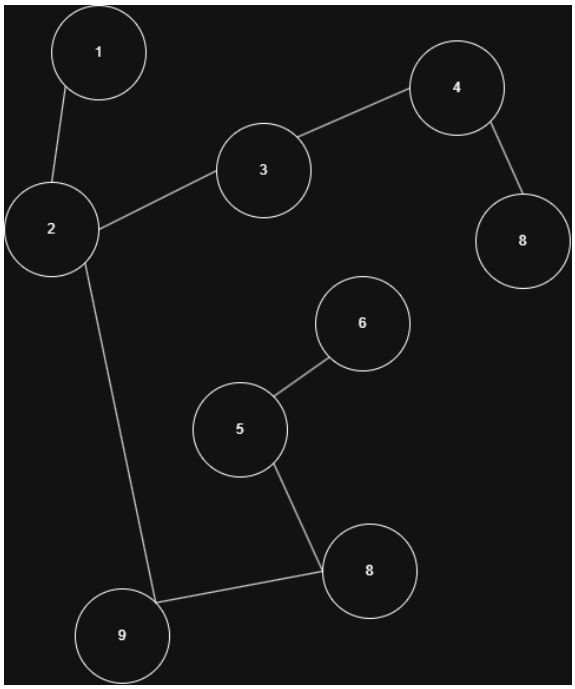
Por lo tanto podemos asignar un color distinto a cada cluster, de tal forma que es posible colorear los vertices con a lo sumo k' colores de tal forma que los vertices adyacentes no compartan color.

1.2.4. Conclusion

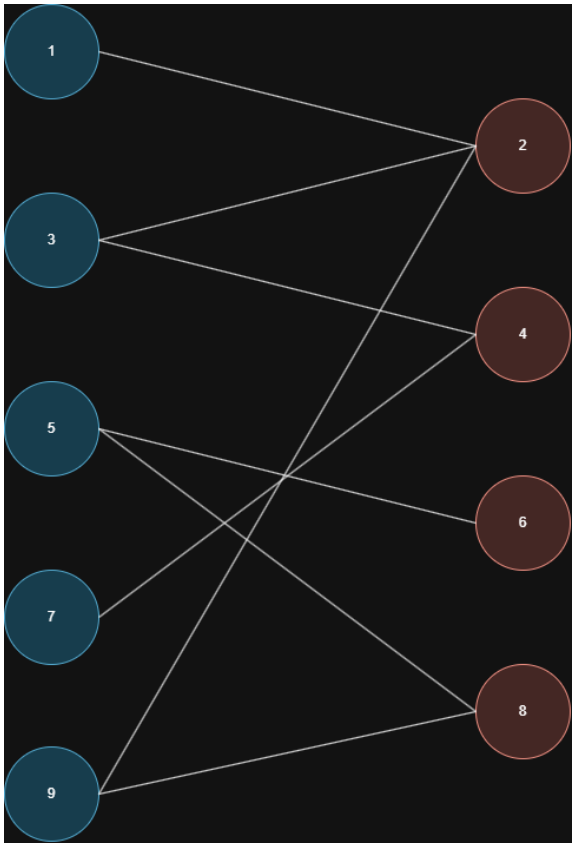
Habiendo demostrado que la reduccion es correcta, queda demostrado tambien que el problema de **Clustering por bajo diametro** es un problema NP-Completo.

1.2.5. Ejemplo

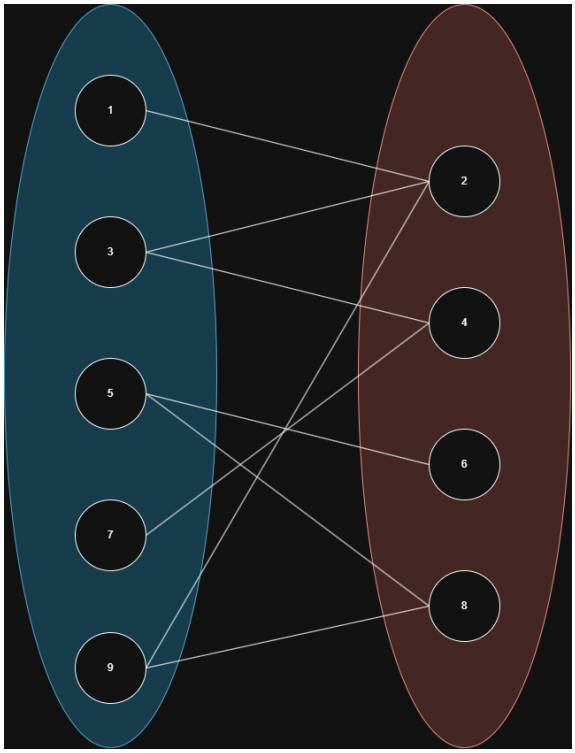
Consideremos el siguiente grafo G.



Si tenemos en cuenta el problema de **K-Coloreo** podemos ver que es posible colorear los vertices con a lo sumo k = 2 colores de forma tal que los adyacentes no compartan color.



Si tenemos en cuenta el problema de **Clustering por bajo diametro** y consideramos C = 0 podemos ver que es posible separar los vertices en a lo sumo k = 2 clusters de tal forma que la distancia maxima dentro de cada cluster es a lo sumo C = 0. Esto ultimo quiere decir que no hay aristas dentro del grafo que unan a los vertices de un mismo cluster.



2. Algoritmo de Backtracking

2.1. Codigo

```python
def bfs_distancias(grafo, origen):
    distancias = {origen: 0}
    visitados = set([origen])
    cola = deque([origen])

    while cola:
        v = cola.popleft()
        for w in grafo.adyacentes(v):
            if w not in visitados:
                visitados.add(w)
                distancias[w] = distancias[v] + 1
                cola.append(w)
    return distancias

def es_valido_extender(cluster, v, distancias_globales, mejor_resultado,
    distancia_actual_cluster, aristas_usadas):
    nueva_distancia = distancia_actual_cluster
    nuevas_aristas = []

    for w in cluster:
        a1 = (v, w)
        a2 = (w, v)

        if a1 in aristas_usadas or a2 in aristas_usadas:
            continue

        d = distancias_globales[v].get(w, math.inf)
        nueva_distancia = max(nueva_distancia, d)

        # Poda: No extender si el nuevo cluster supera el mejor resultado conocido
        if nueva_distancia >= mejor_resultado:
            return False, distancia_actual_cluster, []

        nuevas_aristas.append(a1)
        nuevas_aristas.append(a2)

    return True, nueva_distancia, nuevas_aristas

def clustering_bajo_diametro(grafo, k):

    vertices = grafo.obtener_vertices()

    # Poda: Ordenar los vertices por grado decreciente
    vertices.sort(key=lambda v: len(grafo.adyacentes(v)), reverse=True)

    clusters = [[] for _ in range(k)]
    distancias_globales = {}
    for v in vertices:
        distancias_globales[v] = bfs_distancias(grafo, v)

    distancias_cluster = [0 for _ in range(k)]

    mejor_diametro, mejor_asignacion = backtracking(
        grafo, vertices, k, clusters, 0, math.inf, None,
        distancias_globales, 0, distancias_cluster, set()
    )

    return mejor_asignacion, mejor_diametro
```

```python
def backtracking(grafo, vertices, k, clusters, indice, mejor_resultado, mejor_asignacion,
        distancias_globales, clusters_usados, distancias_cluster, aristas_usadas):

    if indice == len(vertices):
        peor_cluster = max(distancias_cluster[:clusters_usados])
        if peor_cluster < mejor_resultado:
            return peor_cluster, [list(cluster) for cluster in clusters[:clusters_usados]]
        return mejor_resultado, mejor_asignacion

    actual = vertices[indice]

    for i in range(clusters_usados + 1):

        # Poda: No crear mas de k clusters
        if i >= k:
            continue

        # Poda: Evitar crear clusters vac  os innecesariamente
        if len(clusters[i]) == 0 and i != clusters_usados:
            continue

        valido, nueva_distancia, nuevas_aristas = es_valido_extender(
            clusters[i], actual, distancias_globales, mejor_resultado,
            distancias_cluster[i], aristas_usadas
        )
        if not valido:
            continue

        clusters[i].append(actual)
        vieja_distancia = distancias_cluster[i]
        distancias_cluster[i] = nueva_distancia
        for arista in nuevas_aristas:
            aristas_usadas.add(arista)

        nuevo_clusters_usados = clusters_usados
        if i == clusters_usados:
            nuevo_clusters_usados += 1

        # Poda: No continuar si la peor distancia de los clusters actuales ya supera el mejor resultado
        if max(distancias_cluster[:clusters_usados + 1]) >= mejor_resultado:
            clusters[i].pop()
            distancias_cluster[i] = vieja_distancia
            for arista in nuevas_aristas:
                aristas_usadas.remove(arista)
            continue

        mejor_resultado, mejor_asignacion = backtracking(
            grafo, vertices, k, clusters, indice + 1,
            mejor_resultado, mejor_asignacion,
            distancias_globales, nuevo_clusters_usados, distancias_cluster, aristas_usadas
        )

        clusters[i].pop()
        distancias_cluster[i] = vieja_distancia
        for arista in nuevas_aristas:
            aristas_usadas.remove(arista)

    return mejor_resultado, mejor_asignacion
```

2.2. Podas

1. Poda por límite de clusters ($i >= k$): Su objetivo es evitar expandir mas clusetrs de los permitidos (k). Si se permitiera crear mas de k clusters, no se cum0pliria la restriccion delproblema con lo cual esta poda garantiza que no se explore ninguna asignacion que rompa esta condicion te tener a lo sumo k clusters.

2. Poda para evitar clusters vacíos innecesarios: Su objetivo es evitar crear nuevos clusters vacios en posiciones que no corresponden al siguiente cluster nuevo.

3. No extender si la nueva distancia supera el mejor resultado: Cuando se intenta agregar un nuevo nodo a un cluster, se evalúa si eso haría que el diámetro del cluster supere el mejor diámetro encontrado hasta el momento. Si ya sabemos que al agregar un nodo el diámetro local se vuelve peor que el mejor encontrado, no tiene sentido continuar por ese camino.

4. No continuar si el peor cluster actual ya supera el mejor resultado: Antes de continuar con los llamados recursivos, se evalúa el peor diámetro entre los clusters ya formados. Si ese valor ya es mayor o igual al mejor resultado, no vale la pena seguir.

5. Poda de ordenamiento de vértices por grado: Su objetivo es explorar primero los nodos más difíciles o que podrian generar mas conflictos si se dejan para lo ultimo. Es decir, aquellos con más conexiones (mayor grado).

2.3. Resultados de ejecucion y mediciones de tiempo

2.3.1. Resumen

Archivo	**K**	**Tiempo de ejecución (s)**
10_3.txt	2	0.0002
10_3.txt	5	0.0003
22_3.txt	3	0.0076
22_3.txt	4	0.0010
22_3.txt	10	0.0068
22_5.txt	2	0.0010
22_5.txt	7	0.0017
30_3.txt	2	0.0057
30_3.txt	6	0.0047
30_5.txt	5	0.0173
40_5.txt	3	0.0048
45_3.txt	2	0.5131
45_3.txt	3	0.0062
45_3.txt	4	0.7859
45_3.txt	5	859.5344
45_3.txt	6	-
45_3.txt	7	-
50_3.txt	3	0.0089

Cuadro 1: Resumen de tiempos de ejecución por archivo y valor de K

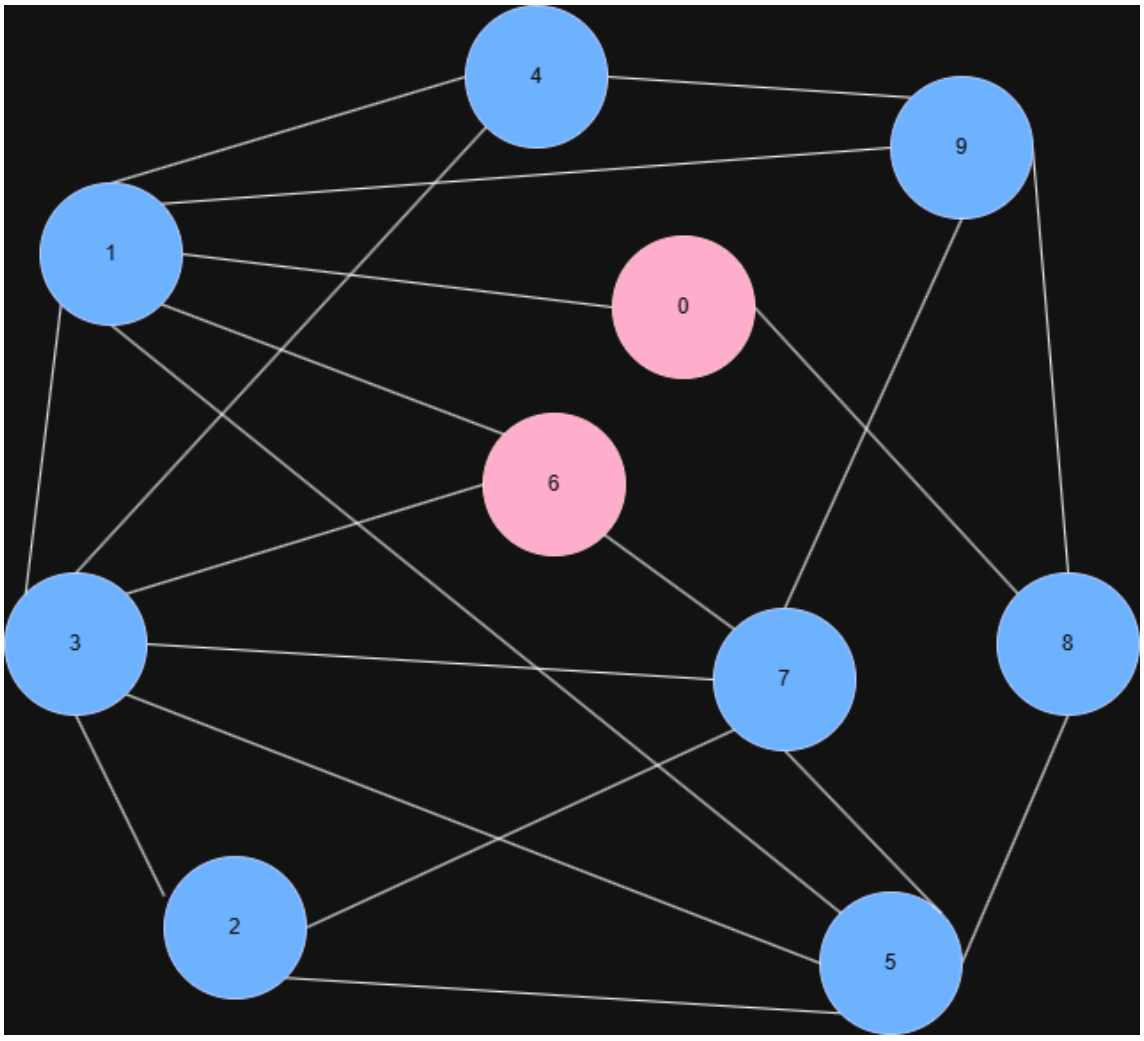
2.3.2. Archivo 10_3.*txt* con a lo sumo 2 clusters

Tiempo de ejecución: 0.0002 segundos
Asignación:

1. Cluster 0 : [1, 3, 5, 7, 9, 8, 4, 2]
2. Cluster 1 : [6, 0]

Máxima distancia dentro del cluster: 2

Grafo y clusters



2.3.3. Archivo 10_3.*txt* con a lo sumo 5 clusters

Tiempo de ejecución: 0.0003 segundos
Asignación:

1. Cluster 0 : [1, 3, 5]
2. Cluster 1 : [7, 6]
3. Cluster 2 : [9, 4]
4. Cluster 3 : [8, 0]
5. Cluster 4 : [2]

Máxima distancia dentro del cluster: 1

2.3.4. Archivo 22_3.*txt* con a lo sumo 3 clusters

Tiempo de ejecución: 0.0076 segundos
Asignación:

1. Cluster 0 : [19, 5, 4, 3, 16, 1, 21, 2, 0, 12, 11, 8, 6]
2. Cluster 1 : [9, 17, 18, 20]
3. Cluster 2 : [7, 10, 13, 15, 14]

Máxima distancia dentro del cluster: 2

2.3.5. Archivo 22_3.*txt* con a lo sumo 4 clusters

Tiempo de ejecución: 0.0010 segundos
Asignación:

1. Cluster 0 : [19, 5, 4, 3, 16, 1, 21, 9, 2, 12, 11, 8, 6]
2. Cluster 1 : [7, 10, 0, 13, 14]
3. Cluster 2 : [17, 18, 20]
4. Cluster 3 : [15]

Máxima distancia dentro del cluster: 2

2.3.6. Archivo 22_3.*txt* con a lo sumo 10 clusters

Tiempo de ejecución: 0.0068 segundos
Asignación:

1. Cluster 0 : [19, 4, 12]
2. Cluster 1 : [5, 16, 13]
3. Cluster 2 : [3, 21]
4. Cluster 3 : [1, 8]

5. Cluster 4 : [9, 20]
6. Cluster 5 : [2, 11]
7. Cluster 6 : [7, 17, 15]
8. Cluster 7 : [10, 14]
9. Cluster 8 : [0]
10. Cluster 9 : [18, 6]

Máxima distancia dentro del cluster: 1

2.3.7. Archivo 22_5.*txt* con a lo sumo 2 clusters

Tiempo de ejecución: 0.0010 segundos
Asignación:

1. Cluster 0 : [10, 11, 14, 19, 4, 21, 1, 2, 16, 18, 17, 12, 13, 8, 7, 20, 9, 0, 5, 15]
2. Cluster 1 : [6, 3]

Máxima distancia dentro del cluster: 2

2.3.8. Archivo 22_5.*txt* con a lo sumo 7 clusters

Tiempo de ejecución: 0.0017 segundos
Asignación:

1. Cluster 0 : [10, 19, 21]
2. Cluster 1 : [11, 14, 1, 20]
3. Cluster 2 : [4, 18, 12, 13]
4. Cluster 3 : [2, 16, 15]
5. Cluster 4 : [17, 5, 3]
6. Cluster 5 : [8, 0]
7. Cluster 6 : [7, 9, 6]

Máxima distancia dentro del cluster: 1

2.3.9. Archivo 30_3.*txt* con a lo sumo 2 clusters

Tiempo de ejecución: 0.0057 segundos
Asignación:

1. Cluster 0 : [4, 3, 13, 9, 5, 12, 26, 0, 2, 14, 23, 22, 24, 20, 16, 6, 19, 15, 21, 17, 7, 10, 28, 1, 25]
2. Cluster 1 : [18, 8, 11, 29, 27]

Máxima distancia dentro del cluster: 3

2.3.10. Archivo 30_3.*txt* con a lo sumo 6 clusters

Tiempo de ejecución: 0.0047 segundos
Asignación:

1. Cluster 0 : [4, 3, 13, 9, 12, 26, 0, 23, 22, 15, 18]
2. Cluster 1 : [5, 2, 14, 20, 6, 19, 21, 1]
3. Cluster 2 : [24, 17, 7, 10, 11]
4. Cluster 3 : [16, 8, 28, 27]
5. Cluster 4 : [29]
6. Cluster 5 : [25]

Máxima distancia dentro del cluster: 2

2.3.11. Archivo 30_5.*txt* con a lo sumo 5 clusters

Tiempo de ejecución: 0.0173 segundos
Asignación:

1. Cluster 0 : [5, 14, 21, 23, 24, 2, 0, 25, 4, 15, 7, 1, 17, 19, 11, 12, 16, 10, 13, 22, 26, 28, 20, 18]
2. Cluster 1 : [6, 3, 8, 9]
3. Cluster 2 : [27, 29]

Máxima distancia dentro del cluster: 2

2.3.12. Archivo 40_5.*txt* con a lo sumo 3 clusters

Tiempo de ejecución: 0.0048 segundos
Asignación:

1. Cluster 0 : [0, 32, 29, 24, 36, 33, 16, 34, 25, 13, 6, 7, 22, 3, 10, 1, 2, 5, 30, 37, 38]
2. Cluster 1 : [27, 14, 28, 26, 9, 23, 17, 12, 8, 31, 35, 39]
3. Cluster 2 : [11, 21, 20, 19, 18, 15, 4]

Máxima distancia dentro del cluster: 2

2.3.13. Archivo 45_3.*txt*

Mencionamos que particularmente para este caso de prueba, con a lo sumo 7 clusters, nuestro algoritmo no logro finalizar tras ejecutarlo por horas.

Debido a esto, probamos ejecutar este mismo caso de prueba con distinta cantidad de clusters para ver como crecia el tiempo de ejecucion.

Nuestro algoritmo finalizo su ejecucion considerando hasta a lo sumo 5 clusters.

A continuacion, presentamos los resultados obtenidos.

Con a lo sumo 2 clusters

Tiempo de ejecución: 0.5131 segundos

Asignación:

1. Cluster 0 : [0, 31, 2, 3, 1, 6, 26, 27, 38, 44, 17, 7, 15, 19, 40, 22, 14, 29, 30, 5, 20, 21, 37, 12, 25, 11, 42, 9, 33, 4, 13, 10, 8, 28, 41, 23, 35, 16]
2. Cluster 1 : [36, 24, 32, 39, 18, 43, 34]

Máxima distancia dentro del cluster: 3

Con a lo sumo 3 clusters

Tiempo de ejecución: 0.0062 segundos

Asignacion:

1. Cluster 0 : [0, 31, 2, 3, 1, 6, 26, 27, 38, 44, 17, 7, 15, 19, 40, 22, 14, 29, 30, 5, 20, 21, 37, 12, 36, 25, 11, 24, 42, 9, 33, 4, 13, 10, 8, 28, 41, 35]
2. Cluster 1 : [32, 23, 39, 43, 34]
3. Cluster 2 : [18, 16]

Máxima distancia dentro del cluster: 3

Con a lo sumo 4 clusters

Tiempo de ejecución: 0.7859 segundos

Asignación:

1. Cluster 0 : [0, 31, 2, 3, 1, 6, 26, 27, 38, 44, 17, 7, 15, 19, 40, 22, 14, 29, 30, 5, 20, 21, 37, 12, 36, 25, 11, 24, 42, 9, 33, 4, 13, 10, 8, 28, 41, 35]
2. Cluster 1 : [32, 23, 39, 43, 34]
3. Cluster 2 : [18, 16]

Máxima distancia dentro del cluster: 3

Con a lo sumo 5 clusters

Tiempo de ejecución: 859.5344 segundos

Asignación:

1. Cluster 0 : [0, 31, 2, 3, 1, 6, 26, 27, 38, 44, 17, 7, 15, 19, 40, 22, 14, 29, 30, 5, 20, 21, 37, 12, 36, 25, 11, 24, 42, 9, 33, 4, 13, 10, 8, 28, 41, 35]
2. Cluster 1 : [32, 23, 39, 43, 34]
3. Cluster 2 : [18, 16]

Máxima distancia dentro del cluster: 3

2.3.14. Archivo 50_3.*txt* con a lo sumo 3 clusters

Tiempo de ejecución: 0.0089 segundos

Asignación:

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

Máxima distancia dentro del cluster: 3

2.4. Conjunto de datos generado

Archivo usado	Valor de k	Tiempo (s)	Distancia máxima dentro del cluster
10_1.txt	1	0.0002	9
10_1.txt	2	0.0018	4
10_1.txt	3	0.0005	3
10_1.txt	4	0.0003	2
10_1.txt	5	0.0004	1
10_1.txt	6	0.0003	1
10_1.txt	7	0.0014	1
10_1.txt	8	0.0003	1
10_1.txt	9	0.0002	1
10_1.txt	10	0.0002	0

Cuadro 2: Comparativa de resultados del algoritmo de Backtracking con distintas particiones k sobre un grafo de 10 vertices con forma de lista

- A medida que k aumenta, la distancia máxima disminuye, lo cual tiene sentido ya que al dividir más, los grupos son más pequeños.
- A partir de $k = 5$, la distancia máxima ya es 1 o menos.
- A $k = 10$, cada vértice está en su propio cluster, por lo que la distancia máxima es 0.
- El tiempo de ejecución no siempre aumenta con k, pero hay pequeñas variaciones por la cantidad de combinaciones exploradas por nuestro algoritmo .

Archivo usado	Valor de k	Tiempo (s)	Distancia máxima dentro del cluster
`100_1.txt`	1	0.0145	99
`100_1.txt`	50	-	-
`100_1.txt`	60	0.1304	1
`100_1.txt`	70	0.1001	1
`100_1.txt`	80	0.0765	1
`100_1.txt`	90	0.0549	1
`100_1.txt`	100	0.0206	0

Cuadro 3: Comparativa de resultados del algoritmo de Backtracking con distintas particiones k sobre un grafo de 100 vertices con forma de lista

- En $k = 1$, todo el grafo entra en un único cluster. La distancia máxima entre extremos (vertices 0 y 99) es 99.
- A partir de $k = 60$, el algoritmo logra una partición donde todos los clusters tienen distancia máxima menor o igual a 1.
- Para $k = 50$, el algoritmo no finalizó su ejecución luego de varias horas, lo que indica un punto crítico de complejidad donde las podas no son suficientes para evitar la explosión combinatoria.
- En $k = 100$, cada vértice está en su propio cluster, por lo tanto, la distancia máxima es 0. Esto refleja la propiedad esperada sobre que a mayor cantidad de clusters, menor será la distancia interna de cada uno, hasta alcanzar el caso donde cada vertice este en su propio cluster.
- El mayor tiempo (0.1304s) ocurre en $k = 60$, donde hay más combinaciones posibles que evaluar para cumplir la condición de diámetro menor o igual a 1. A partir de $k = 70$, el tiempo disminuye porque hay más libertad para dividir el grafo y se aplican más podas efectivas.

Archivo usado	Valor de k	Tiempo (s)	Distancia máxima dentro del cluster
500_1.txt	1	0.3008	499
500_1.txt	250	-	-
500_1.txt	300	22.4992	1
500_1.txt	350	17.7464	1
500_1.txt	400	12.0435	1
500_1.txt	450	5.8424	1
500_1.txt	490	1.6157	1
500_1.txt	500	0.8165	0

Cuadro 4: Comparativa de resultados del algoritmo de Backtracking con distintas particiones k sobre un grafo de 500 vertices con forma de lista

- En $k = 1$, todos los vértices se agrupan en un único cluster, dando una distancia máxima de 499.
- La distancia máxima se reduce abruptamente a 1 al alcanzar $k = 300$, y se mantiene constante hasta $k = 490$.
- Para $k = 250$, el algoritmo no finalizó su ejecución luego de varias horas, lo que indica un punto crítico de complejidad donde las podas no son suficientes para evitar la explosión combinatoria.
- Para $k = 500$, cada vértice se encuentra en su propio cluster, por lo tanto, la distancia máxima es 0.
- El tiempo de ejecución disminuye a medida que k se aproxima a 500, ya que el espacio de búsqueda del algoritmo de backtracking se reduce.
- Este comportamiento refleja cómo el algoritmo explora menos combinaciones cuando los clusters se acercan al caso trivial (cada nodo en su propio grupo).

3. Algoritmo de Programación Lineal

3.1. Codigo

```python
def bfs_distancias(grafo, origen):
    distancias = {origen: 0}
    visitados = set([origen])
    cola = deque([origen])

    while cola:
        v = cola.popleft()
        for w in grafo.adyacentes(v):
            if w not in visitados:
                visitados.add(w)
                distancias[w] = distancias[v] + 1
                cola.append(w)
    return distancias

def programacion_lineal(grafo, vertices, k, clusters, distancias_globales):
    n = len(vertices)
    k = min(k, n)

    # Creo el problema y agrego las variables
    problema = pulp.LpProblem("Clustering_Bajo_Diametro", pulp.LpMinimize)

    x = pulp.LpVariable.dicts("x", [(v, c) for v in vertices for c in range(k)], cat="Binary")
    y = pulp.LpVariable.dicts("y", [c for c in range(k)], cat="Binary")
    D = pulp.LpVariable("D", lowBound=0, cat="Integer")

    problema += D

    # Agrego las restricciones
    for v in vertices:
        problema += pulp.lpSum(x[v, c] for c in range(k)) == 1, f"asignacion_{v}"

    for c in range(k):
        for v in vertices:
            problema += x[v, c] <= y[c]

    problema += pulp.lpSum(y[c] for c in range(k)) <= k, "limite_clusters"

    for u in vertices:
        for v in vertices:
            if u < v:
                d_uv = distancias_globales[u].get(v, math.inf)
                if math.isfinite(d_uv):
                    for c in range(k):
                        problema += d_uv * (x[u, c] + x[v, c] - 1) <= D, f"diametro_{u}_{v}_{c}"

    # Resuelvo el problema y reconstruyo la solucion
    status = problema.solve()

    for v in vertices:
        for c in range(k):
            if pulp.value(x[v, c]) > 0.5:
                clusters[c].append(v)
                break

    return int(pulp.value(D)), clusters
```

```python
def clustering_bajo_diametro(grafo, k):

    vertices = grafo.obtener_vertices()
    vertices.sort(key=lambda v: len(grafo.adyacentes(v)), reverse=True)

    clusters = [[] for _ in range(k)]
    distancias_globales = {}
    for v in vertices:
        distancias_globales[v] = bfs_distancias(grafo, v)

    return grafo, vertices, k, clusters, distancias_globales

def clustering_bajo_diametro_pl(grafo, vertices, k, clusters, distancias_globales):
    mejor_diametro, mejor_asignacion = programacion_lineal(grafo, vertices[:], k,
    clusters[:], distancias_globales)

    return mejor_asignacion, mejor_diametro

def main():
    args = parsear_argumentos()

    grafo = crear_grafo(args.archivo)

    grafo, vertices, k, clusters, distancias_globales = clustering_bajo_diametro(
    grafo, args.clusters)
    return clustering_bajo_diametro_pl(grafo, vertices, k, clusters,
    distancias_globales)
```

3.2. El problema

3.2.1. Definición de variables

Las variables utilizadas para la implementación del problema son 3:

- $X_{u,c}$. La cual es de tipo binario y representa si el vertice v se encuentra dentro del cluster c (1 si v se encuentra en c, 0 si no).
- Y_c. También de tipo binario y representa si el cluster c contiene elementos (1 si el cluster contiene elementos 0 si está vacío).
- D. Por último la variable D, que es de tipo entero y tiene un valor mínimo de 0, la cual representa el diametro máximo de nuestra clausterización.

Para la definición del problema se hizo foco justamente en esta última variable 'D'. La cual buscamos optimizar minimizándola.

3.2.2. Restricciones

- Asignación única de vértices, cada vértice debe pertenecer exactamente a un cluster:

$$\sum_{i=1}^{k} X_{i,c} = 1 \quad \forall i \in V$$

- Límite de clusters, el número de clusters no puede ser superior a k:

$$\sum_{c=1}^{k} y_c = 1$$

- Un cluster está activo si al menos un vértice está asignado a él:

$$x_{i,c} \leq y_c \quad \forall i \in V, \forall c \in \{1, ..., k\}$$

- Límite de diámetro, Para cada par de vertices (i, j) asignados al mismo cluster, la distancia entre ellos debe ser menor o igual al diámetro máximo

$$d_{i,j} \ . \ (\ x_{u,c} \ + \ x_{v,c} \ - \ 1) \ \leq \ D \quad \forall u, v \ \in V, \ \forall c \in \{1, ..., k\}$$

3.2.3. Número total de Restricciones

Si analizamos el último item podemos ver que se agrega una restricción para cada par de vertices u, v en cada cluster. Por ende el número total de restricciónes es equivalente a $O(n * k)$ en notación Big O

3.3. Mediciones de tiempo y comparación con Backtracking

Archivo	K	T.ejecución (s)	diferencia contra backtracking
10_3.txt	2	0.0282	+0.0280
10_3.txt	5	0.0911	+0.0908
22_3.txt	3	0.3671	+0.3595
22_3.txt	4	0.5758	+0.5748
22_3.txt	10	-	-
22_5.txt	2	0.1324	+0.1314
22_5.txt	7	-	-
30_3.txt	2	1.4357	+1.4300
30_3.txt	6	0.0047	=
30_5.txt	5	1.9123	+1.895
40_5.txt	3	4.4245	+4.4197
45_3.txt	2	0.5159	+0.0028
45_3.txt	3	1.6598	+1.6918
45_3.txt	4	5.1351	+4.392
45_3.txt	5	331.7997	-527.7347
45_3.txt	6	-	-
45_3.txt	7	-	-
50_3.txt	3	2.0628	+2.0539

Cuadro 5: Comparación de tiempos de ejecución por archivo y valor de K PL vs backtracking

4. Algoritmo de Aproximacion

Implementaciión del Algoritmo de Louvain para aproximar el problema mediante la maximización de la modularización, buscando K clusters en el grafo. La modularización es un valor que se obtiene con la siguiente fórmula:

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

Y nos indica qué tan dividido está el grafo en comunidades o clusters.

4.1. Codigo

```
def algoritmo_louvain(grafo, K):
    comunidades = {v: v for v in grafo.obtener_vertices()}
    i = 0
    while True:
        i += 1
        comunidades = fase1(grafo, comunidades)
        nuevo_grafo = fase2(grafo, comunidades)
        if len(nuevo_grafo.obtener_vertices()) == len(grafo.obtener_vertices()) or
    len(nuevo_grafo.obtener_vertices()) <= K:
            break

        nuevas_comunidades = {}
        for v in grafo.obtener_vertices():
            nuevas_comunidades[v] = comunidades[v]

        grafo = nuevo_grafo
        comunidades = nuevas_comunidades

    return comunidades


def fase1(grafo, comunidades):
    cambiado = True
    while cambiado:
        cambiado = False
        for nodo in grafo.obtener_vertices():
            mejor, delta = mejor_comunidad(grafo, comunidades, nodo)

            if comunidades[nodo] != mejor and delta > 0:
                comunidades[nodo] = mejor
                cambiado = True
    return comunidades


def fase2(grafo, comunidades):
    nuevo_grafo = Grafo(False)
    clusters = agrupar_por_comunidad(comunidades)
    for c in clusters:
        nuevo_grafo.agregar_vertice(c)s
    for v, w in obtener_aristas(grafo):
        c1, c2 = comunidades[v], comunidades[w]
        peso = grafo.peso_arista(v, w)

        if not nuevo_grafo.pertenece_vertice(c1):
            nuevo_grafo.agregar_vertice(c1)
        if not nuevo_grafo.pertenece_vertice(c2):
            nuevo_grafo.agregar_vertice(c2)
        if nuevo_grafo.estan_unidos(c1, c2):
            nuevo_peso = nuevo_grafo.peso_arista(c1, c2) + peso
            nuevo_grafo.agregar_arista(c1, c2, nuevo_peso)
        else:
            nuevo_grafo.agregar_arista(c1, c2, peso)
    return nuevo_grafo

```

```python
def mejor_comunidad(grafo, comunidades, nodo):
    original = comunidades[nodo]
    vecinos = grafo.adyacentes(nodo)
    comunidades_vecinas = set(comunidades[v] for v in vecinos)
    mejor_delta = 0
    mejor_com = original

    for c in comunidades_vecinas:
        comunidades[nodo] = c
        nueva_mod = modularizacion(grafo, comunidades)
        comunidades[nodo] = original
        actual_mod = modularizacion(grafo, comunidades)
        delta_Q = nueva_mod - actual_mod

        if delta_Q > mejor_delta:
            mejor_delta = delta_Q
            mejor_com = c

    return mejor_com, mejor_delta


def modularizacion(grafo, comunidades):
    Q = 0
    m = len(obtener_aristas(grafo))

    for i in grafo.obtener_vertices():
        for j in grafo.adyacentes(i):
            if comunidades[i] == comunidades[j]:
                peso_ij = grafo.peso_arista(i, j)
                k_i = len(grafo.adyacentes(i))
                k_j = len(grafo.adyacentes(j))

                Q += peso_ij - (k_i * k_j) / (2 * m)

    return Q / (2*m)
```

4.2. Tiempos de ejecución y resultados

Archivo	K	Tiempo de ejecución (s)
10_3.txt	2	0.0165
22_3.txt	3	0.2880
22_5.txt	2	0.2765
30_3.txt	2	0.4159
30_5.txt	5	1.0118
40_5.txt	3	1.0691
45_3.txt	7	2.1744
50_3.txt	3	2.1075

Cuadro 6: Resumen de tiempos de ejecución por archivo y valor de K

4.3. Cota de la aproximación

Para establecer una cota, tomamos los resultados de cualquiera de las pruebas y debemos comparar el resultado óptimo con el aproximado con la siguiente fórmula:

$$\frac{A(I)}{z(I)} \leq r(A)$$

- Siendo r(A) el valor de la cota.
- A(I) el resultado de la aproximación.
- z(I) el resultado óptimo.

Tomando como instancia la prueba 10_3.txt con $K = 2$.

- El resultado óptimo es una distancia máxima dentro del cluster de 2, es decir $z(I) = 2$.
- El resultado aproximado es $A(I) = 3$.
- Lo que resulta en $(1,5 \leq r(A))$

Por lo tanto estamos ante una 1,5 Aproximación.

4.4. Complejidad

La complejidad del algoritmo es $O(E * log(V))$.

4.5. Conjunto de datos generado

Archivo usado	Valor de k	Tiempo (s)	Distancia máxima dentro del cluster
`10_1.txt`	1	0.0030	5
`10_1.txt`	2	0.0030	7
`10_1.txt`	3	0.0023	7
`10_1.txt`	4	0.0040	7

Cuadro 7: Comparativa de resultados del algoritmo de Louvain con distintas particiones k sobre un grafo de 10 vertices con forma de lista

A partir de $K = 2$, la distancia máxima es 7 y no cambia, y encuentra 2 clusters o comunidades.

Archivo usado	Valor de k	Tiempo (s)	Distancia máxima dentro del cluster
`100_1.txt`	1	0.1971	90
`100_1.txt`	50	0.1996	90
`100_1.txt`	100	0.2305	90

Cuadro 8: Comparativa de resultados del algoritmo de Louvain con distintas particiones k sobre un grafo de 100 vertices con forma de lista

Sin importar el valor de K, la distancia máxima es 90 y encuentra 3 clusters o comunidades.

Archivo usado	Valor de k	Tiempo (s)	Distancia máxima dentro del cluster
500_1.txt	1	10.3451	490
500_1.txt	250	10.6549	490
500_1.txt	500	10.6815	490

Cuadro 9: Comparativa de resultados del algoritmo de Louvain con distintas particiones k sobre un grafo de 500 vertices con forma de lista

Ocurre lo mismo que en las pruebas anteriores. Sin importar el valor de K, la distancia máxima es 490 y encuentra 3 clusters o comunidades.

5. Conclusiones

Algoritmo de Backtracking

El archivo de prueba `45_3.txt` representa un caso particularmente ilustrativo para estudiar el comportamiento del algoritmo de *backtracking* ante variaciones del parámetro K. En este caso, observamos una tendencia creciente muy marcada en los tiempos de ejecución:

1. Para $K = 2$, el algoritmo resolvió el problema en aproximadamente **0.5 segundos**.
2. Para $K = 3$, el algoritmo resolvió el problema en aproximadamente **0.0062 segundos**. Notamos que hubo una reduccion en el tiempo de ejecucion.
3. Para $K = 4$, el tiempo aumentó nuevamente a aproximadamente **0.79 segundos**.
4. Para $K = 5$, se produjo una explosión en el tiempo de ejecución, alcanzando más de **859 segundos** (casi 15 minutos).
5. A partir de $K = 6$, el algoritmo no logró finalizar su ejecucion tras varias horas.

Este comportamiento pone en evidencia la sensibilidad del algoritmo ante variaciones en el valor de K, ya que pequeños incrementos pueden generar un crecimiento exponencial en el espacio de soluciones a explorar.

6. Anexo

6.1. Correccion 1: Demostracion

6.1.1. Clustering por bajo diámetro se encuentra en NP

Para que el problema se encuentre en NP, debe haber un verificador eficiente.

En otras palabras, debe haber un verificador que ejecute en tiempo polinomial.

Verificador: Recibe una instancia del problema y una solucion.

```python
# Al calcular las distancias se tienen en cuenta tanto las aristas entre vertices dentro del cluster, como cualquier otra arista dentro del grafo

def bfs_max_dist_en_grafo_completo(grafo, cluster, origen):
    visitados = set()
    visitados.add(origen)

    distancias = {origen: 0}
    cola = deque([origen])

    max_dist = 0

    while cola:
        v = cola.popleft()

        for w in grafo.adyacentes(v):
            if w not in visitados:
                visitados.add(w)
                distancias[w] = distancias[v] + 1
                cola.append(w)

    # Solo consideramos distancias hacia otros vertices del cluster
    for nodo in cluster:
        if nodo != origen and nodo in distancias:
            max_dist = max(max_dist, distancias[nodo])

    return max_dist
```

```python
def verificador(grafo, k, C, clusters):

    # Validamos que la solucion tenga a lo sumo k clusters
    if len(clusters) > k: return False

    clusters = [set(c) for c in clusters]

    # Validamos que los clusters sean disjuntos
    vertices_en_clusters = set()

    for cluster in clusters:
        for v in cluster:

            if v in vertices_en_clusters:
                return False

            vertices_en_clusters.add(v)

    # Validamos que se cubran todos los vertices del grafo
    if set(grafo.obtener_vertices()) != vertices_en_clusters:
        return False

    # Validamos que la distancia maxima dentro de cada cluster sea a lo sumo C
    for cluster in clusters:
        for v in cluster:
            max_dist = bfs_max_dist(grafo, cluster, v)

            if max_dist > C: return False

    return True
```

Complejidad: La complejidad total resulta ser $O(Vx(V+E))$ con V la cantidad de vertices y E la cantidad de aristas.

1. Validar que la solucion tenga a lo sumo k clusters tiene una complejidad de $O(1)$.
2. Validar que los clusters sean disjuntos tiene una complejidad de $O(V)$ con V la cantidad de vertices ya que iteramos cada cluster y por cada cluster iteramos todos sus vertices, lo cual implica que, al ser disjuntos, cada vértice se recorre exactamente una vez. Dado que utilizamos un conjunto (set), las operaciones de agregar y consultar si un vertice se encuentra en dicho conjunto son $O(1)$.
3. Validar que se cubran todos los vertices del grafo tiene una complejidad de $O(V)$ con V la cantidad de vertices ya que se compara el conjunto de vertices del grafo, con el conjunto de vertices en clusters.
4. Validar que la distancia maxima dentro de cada cluster sea a lo sumo C tiene una complejidad de $O(Vx(V+E))$ con V la cantidad de vertices y E la cantidad de aristas ya que iteramos cada cluster y por cada cluster iteramos todos sus vertices, donde para cada vertice se hace un BFS sobre el grafo completo. Recordamos que la complejidad de un recorrido BFS es $O(V+E)$ y mencionamos que como los clusters son disjuntos, al recorrer cada cluster y recorrer sus vertices estamos recorriendo todos los vertices del grafo una unica vez, con lo cual es por esto que por cada vertice del grafo hacemos un recorrido BFS.

Anteriormente mencionamos tanto a k como a W en la complejidad, donde W era la cantidad maxima de vertices en un cluster. Sin embargo, dado que los clusters son disjuntos y cubren todos los vértices del grafo, la suma total de vértices en todos los clusters es exactamente V, con lo cual al validar que la distancia maxima dentro de cada cluster sea a lo sumo C se ejecutan V recorridos BFS. Es por esto que en el nuevo analisis de la complejidad no son mencionados.

Luego, se ejecuta en tiempo polinomial, lo cual quiere decir que el problema se encuentra en NP.

6.2. Correccion 3: Demostracion

6.2.1. Clustering por bajo diámetro es un problema NP-Completo

Para demostrar que el problema de **Clustering por bajo diámetro** es NP-Completo, debemos demostrar que existe una reducción polinomial desde algún problema NP-Completo. En nuestro caso, elegimos realizar la reducción utilizando el problema de **K-Coloreo**.

Luego, debemos demostrar que **K-Coloreo** $\leq_P$ **Clustering por bajo diámetro**

6.4.1.1. Reduccion planteada

¿Podemos resolver el problema de **K-Coloreo** utilizando la solucion del problema de **Clustering por bajo diametro**?.

Vamos a utilizar una caja negra que resuelve el problema de **Clustering por bajo diametro** para resolver el problema de **K-Coloreo**.

El problema de **K-Coloreo** recibe un grafo G y un valor k que representa la cantidad de colores.

Transformacion del problema: Transformamos la entrada del problema de **K-Coloreo** en una entrada del problema **Clustering por bajo diametro**

1. A partir del grafo G construimos el grafo complemento G' donde cada arista $(u, v) \in G'$ si y solo si $(u, v) \notin G$
2. El valor de k del problema de **K-Coloreo** coincide con el valor de k' del problema de **Clustering por bajo diametro**
3. Definimos el valor de $C = 1$ de tal forma que la distancia maxima permitida dentro de cada cluster es 1 teniendo en cuenta que al calcular las distancias se tienen en cuenta tanto las aristas entre vertices dentro del cluster, como cualquier otra arista dentro del grafo

Esta transformacion requiere una cantidad de pasos polinomiales.

A continuacion, para demostrar que la reduccion es correcta, debemos demostrar que:

Hay solucion de **K-Coloreo** con a lo sumo k colores en G, si y solo si, hay solucion para el problema de **Clustering por bajo diametro** con a lo sumo k' clusters y $C = 1$ en G'.

6.4.1.2. Si hay K-Coloreo, entonces hay Clustering por bajo diametro

Hipotesis: Hay solucion de **K-Coloreo** con a lo sumo k colores en G

Tesis: Hay solucion para el problema de **Clustering por bajo diametro** con a lo sumo k' clusters en G'

Dada nuestra hipótesis, existen k grupos de vértices coloreados con distinto color, tales que no hay aristas entre vértices del mismo color en G.

Luego, en el grafo complemento G', todos los vértices que comparten color en G están conectados entre sí, ya que no había aristas entre ellos en G. Esto significa que el conjunto de vértices con el mismo color en G forma un **clique** en G'.

Dado que en un clique toda pareja de vértices está conectada directamente, la distancia entre ellos en el grafo complemento G' es 1.

Por lo tanto, cada conjunto de un mismo color puede utilizarse como un cluster válido en el problema de **Clustering por bajo diametro**, con distancia máxima igual a 1.

6.4.1.3. Si hay Clustering por bajo diametro, entonces hay K-Coloreo

Hipotesis: Hay solución para el problema de **Clustering por bajo diametro** sobre el grafo complemento G' con a lo sumo k' clusters y $C = 1$

Tesis: Hay solucion de **K-Coloreo** con a lo sumo k colores para el grafo original G

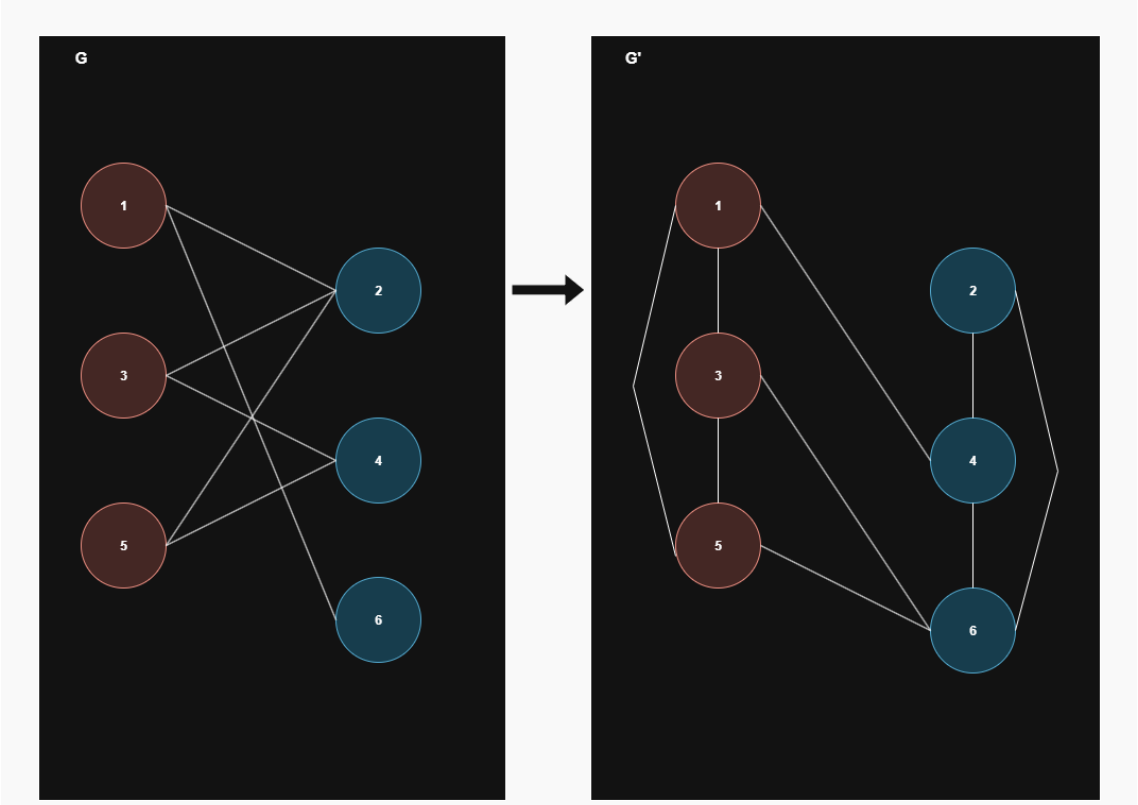
Dada nuestra hipotesis, existen k clusters disjuntos que cubren todos los vertices del grafo, donde en cada cluster la distancia maxima entre pares de vertices (medida en el grafo completo G') es a lo sumo 1.

Eso implica que, en cada cluster, todos los vertices estan conectados entre si en G', es decir, forman un clique.

Luego, en el grafo original G, esos mismos vertices no estan conectados por ninguna arista.

Por lo tanto, podemos asignar un color distinto a cada cluster, y obtenemos una asignación de colores tal que no hay vertices adyacentes con el mismo color.

6.4.1.4. Ejemplo



6.4.1.5. Conclusion

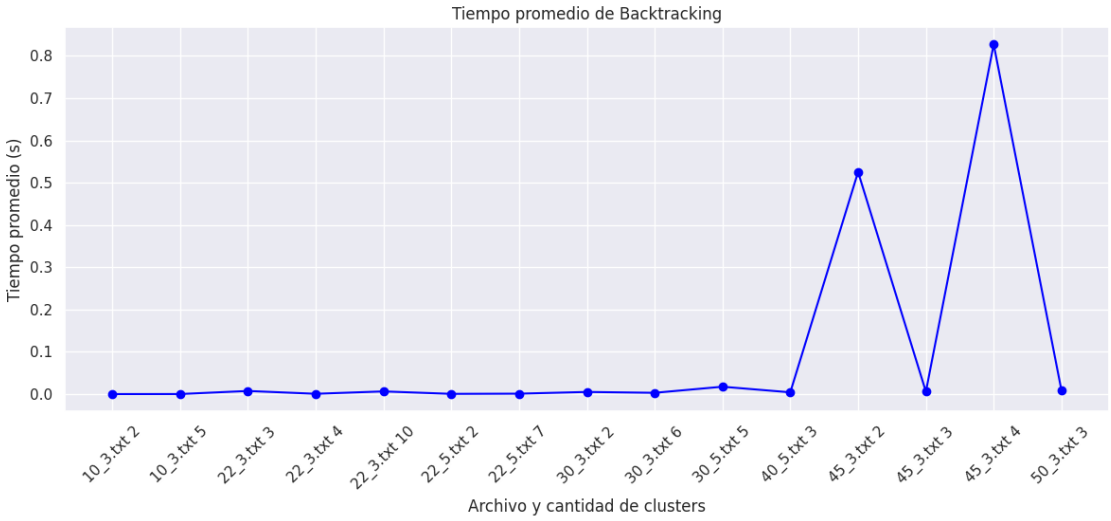
Habiendo demostrado que la reduccion es correcta, queda demostrado tambien que el problema de **Clustering por bajo diametro** es un problema NP-Completo.

6.3. Correccion 4: Algoritmo de Backtracking

6.3.1. Grafico de tiempo de ejecucion

Se realizaron mediciones de tiempo con el objetivo de comparar los tiempos de ejecucion para cada grafo y valor de k de los casos proporcionados por la catedra.

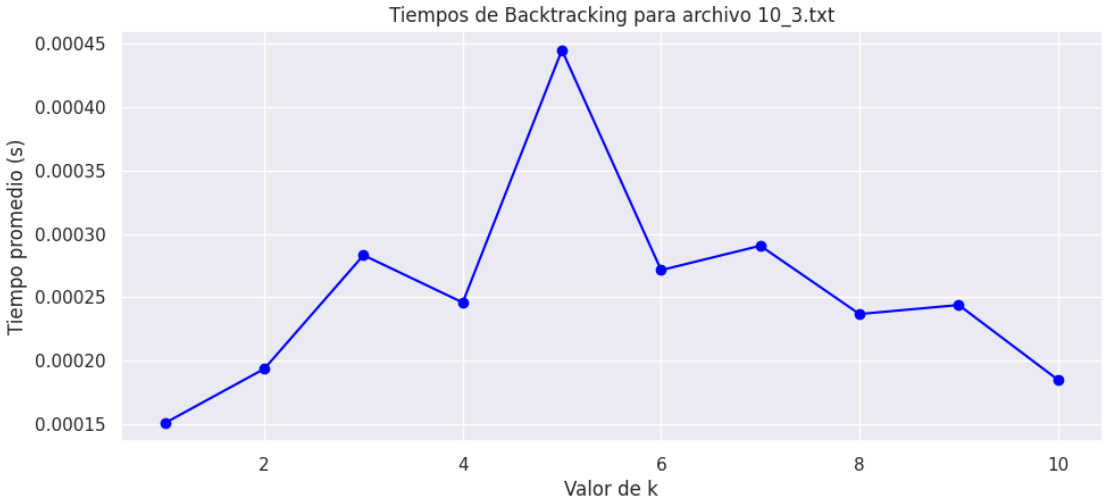
Para ello, se generaron gráficos y se utilizó el codigo proporcionado por la catedra para ejecutar el algoritmo de tal forma que tomamos más de una medición de la misma muestra y nos quedamos con el promedio para reducir el ruido en la medición. Esto utilizando el archivo util.py que nos proporciona la catedra y colocando la constante `RUNS_PER_SIZE` con valor 30.



Podemos ver que el caso 45_3.*txt* con $\mathbf{k = 2}$ y el caso 45_3.*txt* con $\mathbf{k = 4}$ son los casos que mas tiempo tardan.

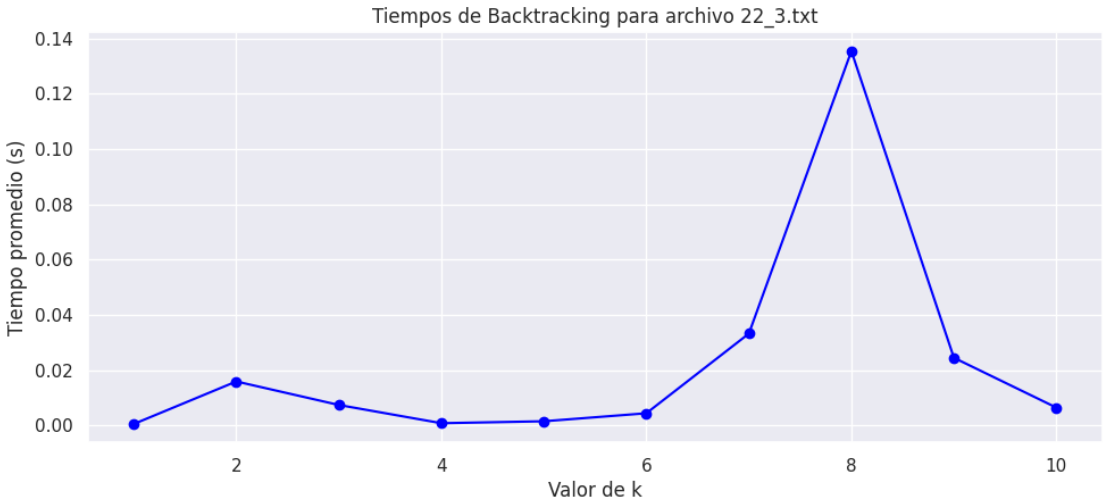
6.3.2. Graficos de variabilidad del valor K

A continuacion, mostramos un grafico de los tiempos de ejecucion para el caso 10_3.*txt* variando el valor de k.



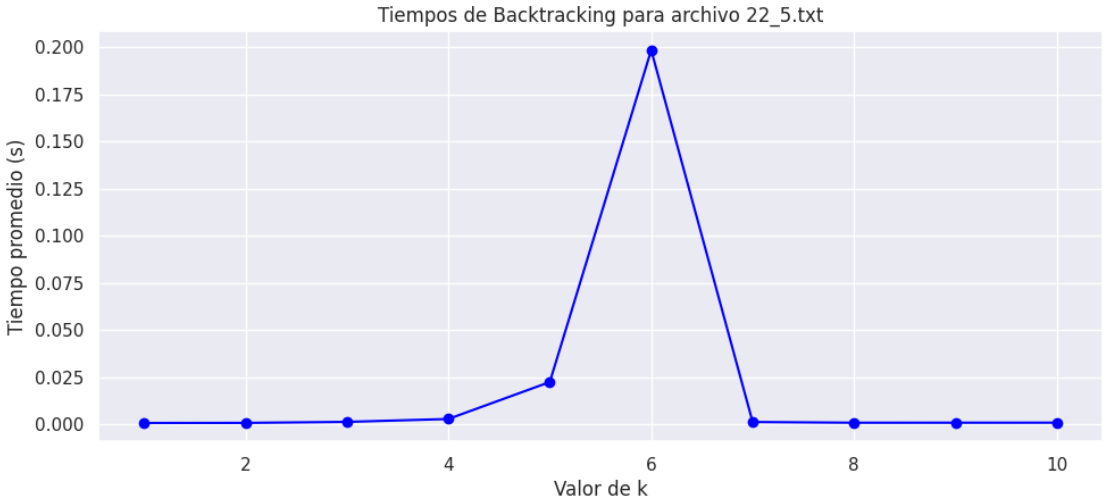
Podemos ver que para $\mathbf{k = 5}$ se alcanza el tiempo mas alto de ejecucion.

A continuacion, mostramos un grafico de los tiempos de ejecucion para el caso 22_3.*txt* variando el valor de k.



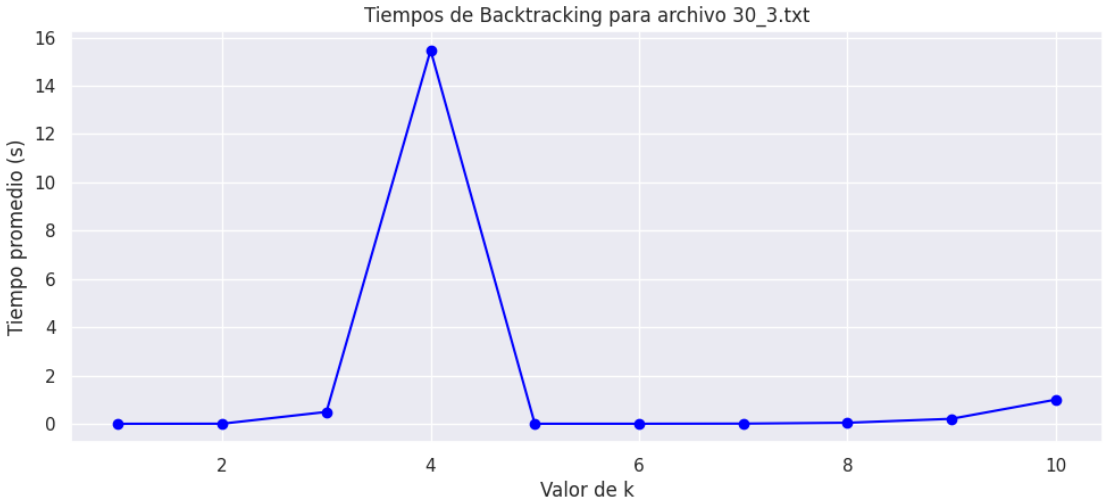
Podemos ver que para $\mathbf{k = 8}$ se alcanza el tiempo mas alto de ejecucion.

A continuacion, mostramos un grafico de los tiempos de ejecucion para el caso 22_5.*txt* variando el valor de k.



Podemos ver que para $\mathbf{k = 6}$ se alcanza el tiempo mas alto de ejecucion.

A continuacion, mostramos un grafico de los tiempos de ejecucion para el caso 30_3.*txt* variando el valor de k.



Podemos ver que para $\mathbf{k = 4}$ se alcanza el tiempo mas alto de ejecucion.

6.4. Correccion 5: Algoritmo de Programacion Lineal

En PL:

(a) Ponen variables con índices y no dicen los rangos de los índices en sí.

(b) La inecuación $\sum y_c = 1$ no debería ser $\sum y_c \leq k$?

(c) Nuevamente, no hay gráficos, difícil comparar con BT.

6.4.1. Corrección a

Las variables y sus rangos son:

- $X_{v,c}$. Es de tipo binario y representa si el vertice v se encuentra dentro del cluster c (1 si v se encuentra en c, 0 si no). Su rango es de 1 a V (cantidad de vertices) para v y de 1 a k (cantidad de clusters) para c
- Y_c. También de tipo binario y representa si el cluster c contiene elementos (1 si el cluster contiene elementos 0 si está vacío). Su rango es [1, k]
- D. Por último la variable D, que es de tipo entero y tiene un valor mínimo de 0, la cual representa el diametro máximo de nuestra clausterización.

6.4.2. Corrección b

Se corrigieron principalmente dos restricciones

- Asignación única de vértices, cada vértice debe pertenecer exactamente a un cluster:

$$\sum_{c=1}^{k} X_{v,c} = 1 \quad \forall v \in V$$

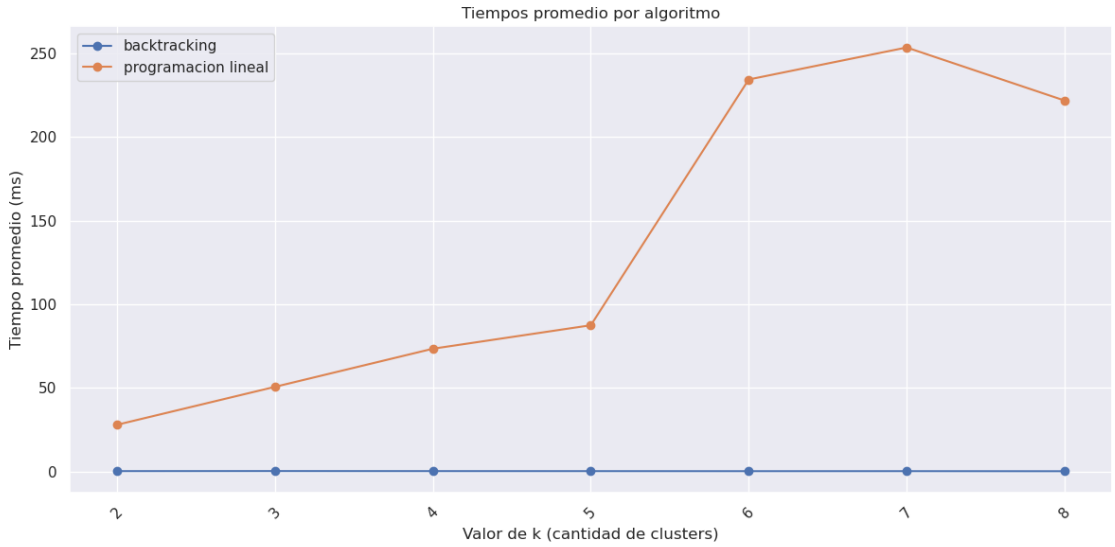
  Aqui estaban mal definidos los subindices ya que hay que iterar sobre los clusters para cada vertice

- Límite de clusters, el número de clusters 'activos' no puede ser superior a k:

$$\sum_{c=1}^{k} Y_c \leq K$$

6.4.3. Corrección c

En la siguiente figura podemos ver la comparación entre los tiempos promedios de ejecución de los algoritmos de programación lineal y backtracking para el archivo 10_3.*txt*. Puede notarse claramente como a medida que aumenta la cantidad de clusters, el tiempo de ejecución del algoritmo de programación lineal se ve seriamente perjudicado respecto al de backtracking. También podemos ver como llega a un máximo en 7 clusters y reduce el tiempo de ejecución para 8. Esto se debe a que el archivo del grafo analizado cuenta unicamente con 10 vertices.



Para la próxima figura se representa el tiempo de ejecución entre ambos algoritmos utilizando el grafo en el archivo 22_3.*txt*. Acá podemos ver, incluso con mayor grado, como se expande el tiempo empleado por el algoritmo de programación lineal. Vemos como hay una diferencia de mas de un minuto en la ejecución aumentando únicamente en uno la cantidad de clusters 7-8. Mientras que el algoritmo de backtracking parece mantenerse en una zona aceptable, la complejidad del algoritmo de programación lineal se dispara.

6.5. Correccion 6: Algoritmo de Aproximacion

6.5.1. Comparación entre el óptimo y la aproximación

A continuación se muestra una tabla con varias pruebas, comparando el diámetro máximo de los clusters generados por el algoritmo exacto (backtracking) con el resultado del algoritmo de Louvain (greedy). También se calcula la razón entre el valor aproximado y el óptimo, así como el error porcentual:

Prueba	**Diámetro máximo (backtracking)**	**Diámetro máximo (greedy)**	$A(I)/z(I)$	**Error porcentual**
10_3_2.txt	2	3	1,5	50 %
10_3_5.txt	1	3	3	200 %
22_3_3.txt	2	3	1,5	50 %
22_3_4.txt	2	3	1,5	50 %
22_3_10.txt	1	3	3	200 %
22_5_2.txt	2	3	1,5	50 %
22_5_7.txt	1	3	3	200 %
30_3_2.txt	3	4	1,33	33 %
30_3_6.txt	2	4	2	100 %
30_5_5.txt	2	3	1,5	50 %
40_5_3.txt	2	3	1,5	50 %
45_3_7.txt	3	4	1,33	33 %
50_3_3.txt	3	4	1,33	33 %

Viendo los resultados de las relacion $A(I)/z(I)$, podemos establecer como una cota máxima:

$$r(A) := \frac{A(I)}{z(I)} \leq 3.$$

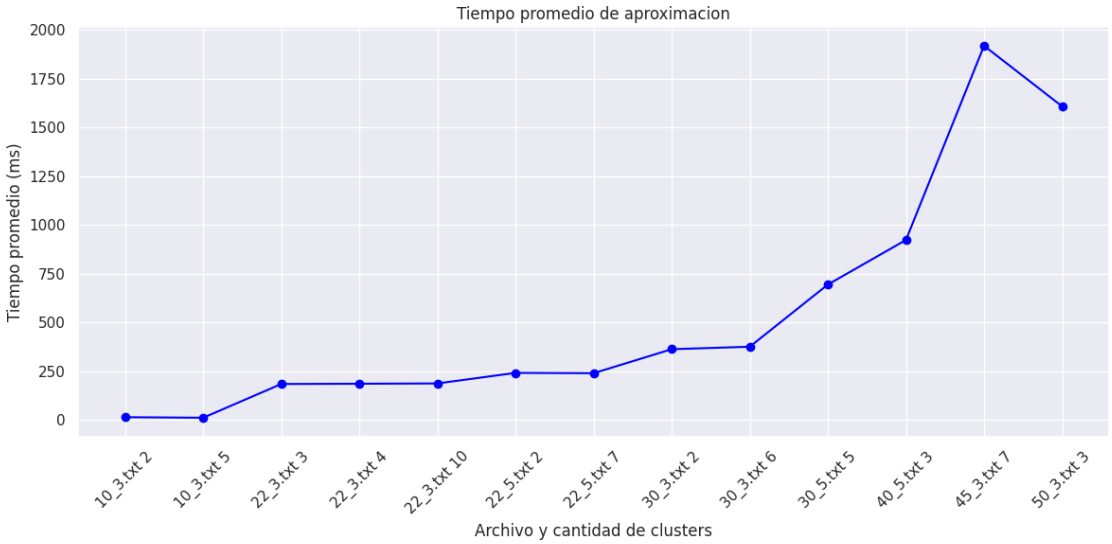
Esto nos da como resultado que, en el peor caso evaluado, el algoritmo de Louvain proporciona una 3-Aproximación respecto al diámetro óptimo.

Las demás cotas encontradas fueron:

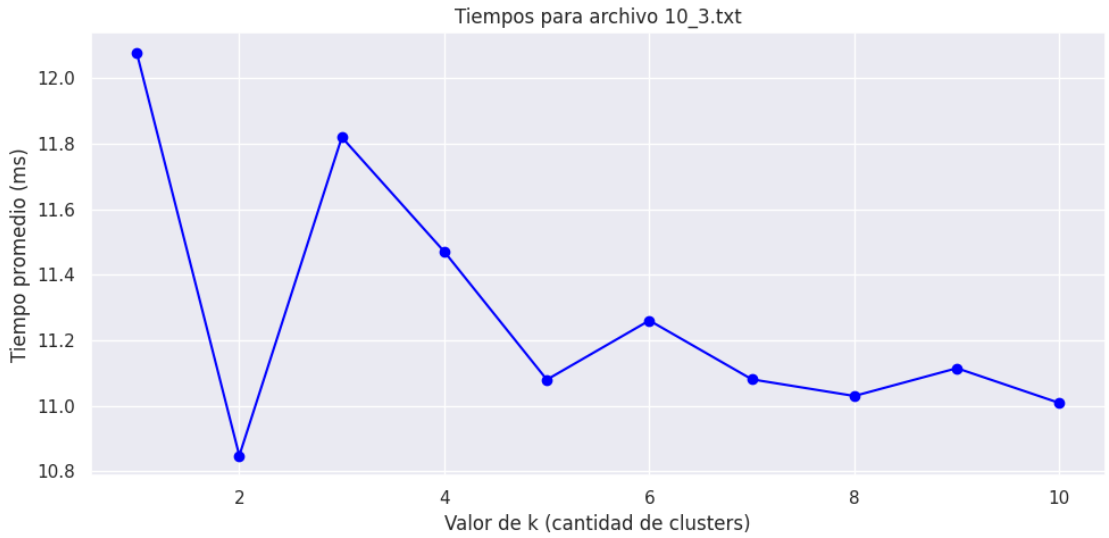
- $A(I)/z(I) \leq 1,33$, para los casos con un error de 33 %, es decir una $\frac{1}{3}$-Aproximación.
- $A(I)/z(I) \leq 1,5$, con un error de 50 %, es decir una $\frac{1}{2}$-Aproximación.
- $A(I)/z(I) \leq 2$, con un error de 100 %, es decir una 2-Aproximación.

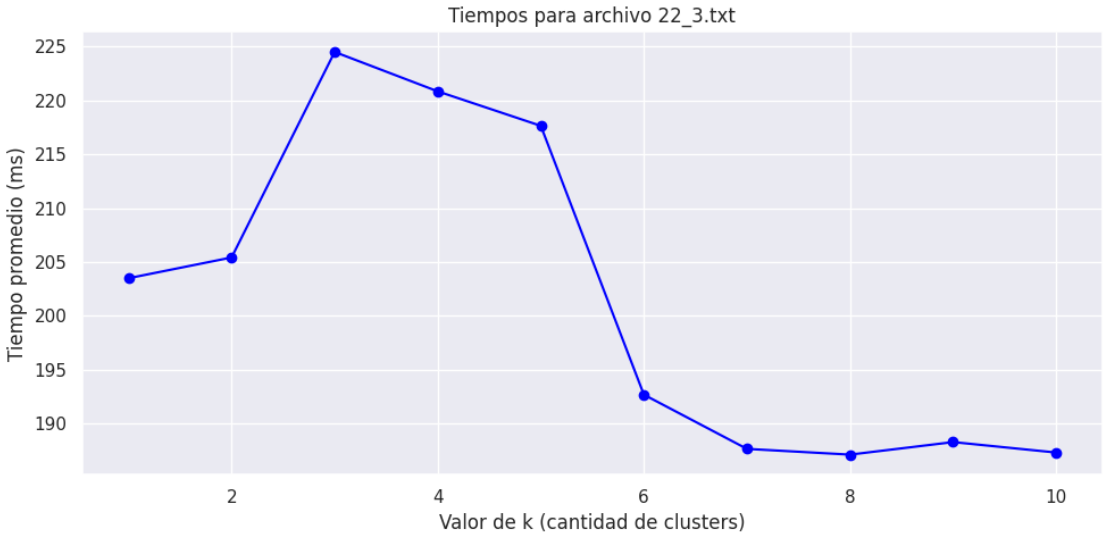
En ninguna de las instancias evaluadas, el algoritmo de Louvain logra alcanzar el resultado óptimo. Sin embargo, se mantiene dentro de un margen razonable de error considerando que se trata de un enfoque heurístico basado en maximización de modularidad y no en minimización directa del diámetro.

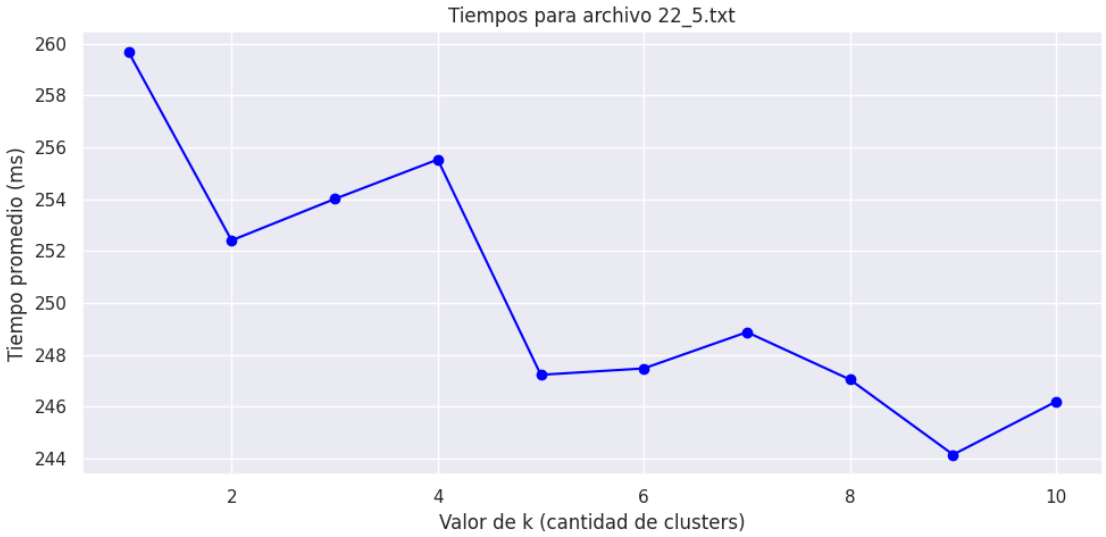
6.5.2. Gráfico de tiempo de ejecución

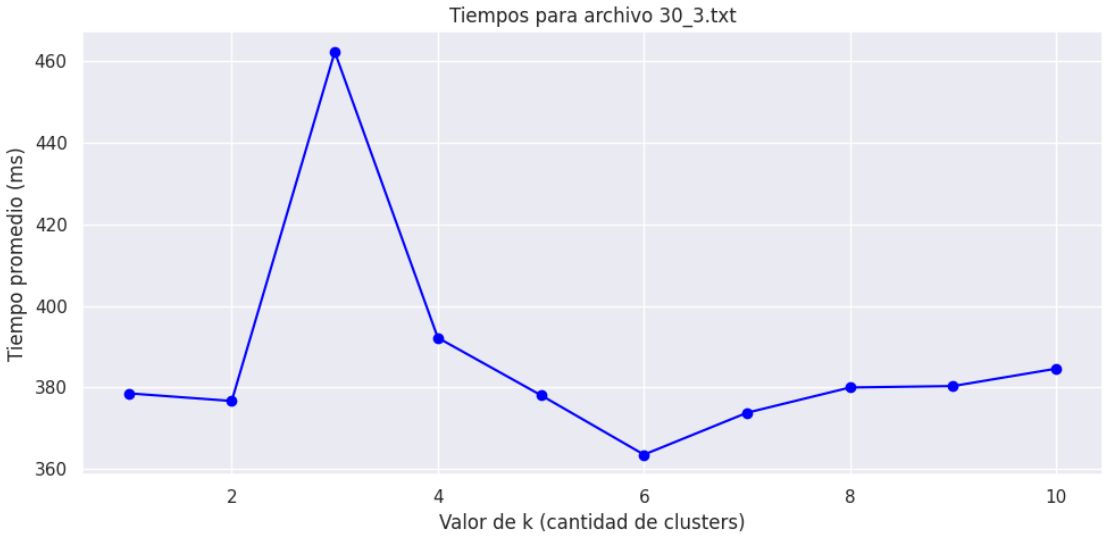


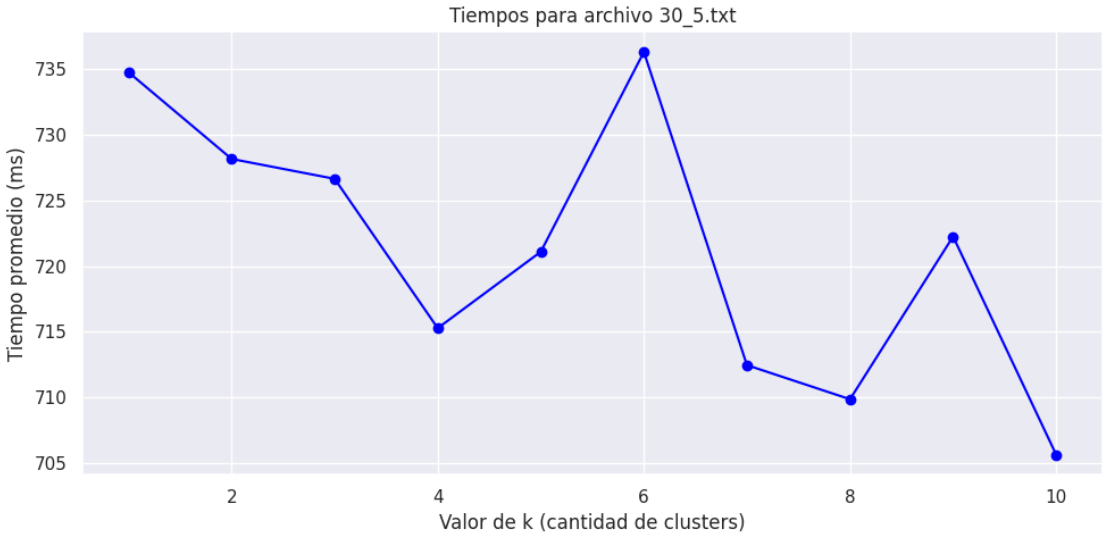
6.5.3. Gráficos de variabilidad del valor K

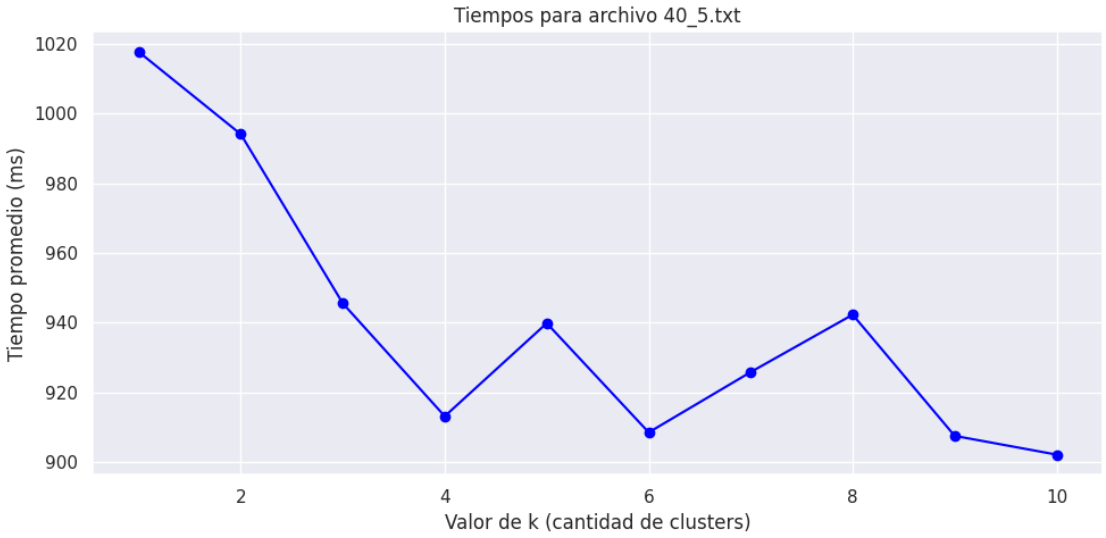



Tiempos para archivo 22_3.txt
Tiempo promedio (ms)
225
220
215
210
205
200
195
190
2
4
6
8
10
Valor de k (cantidad de clusters)

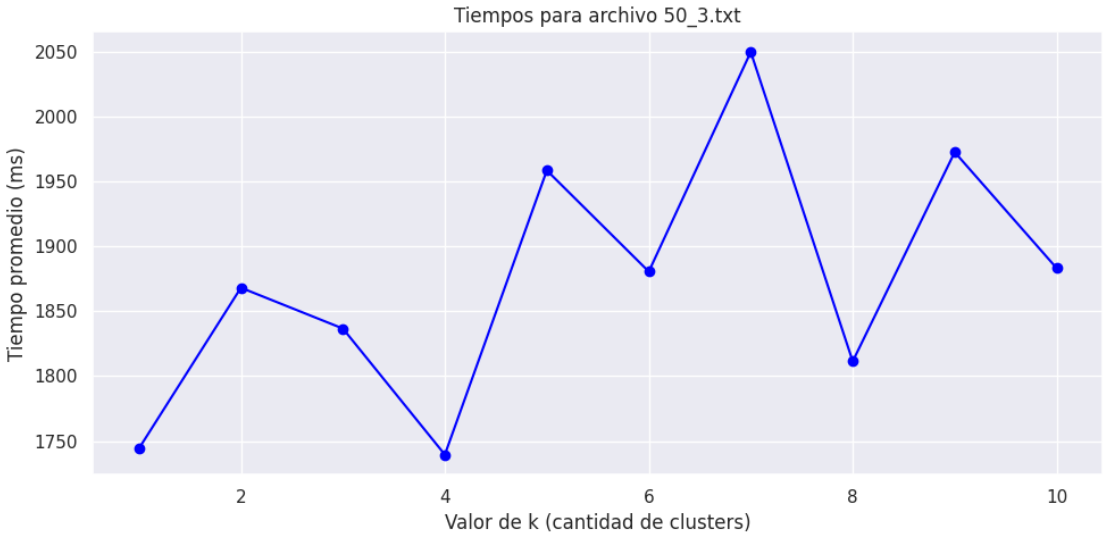

Tiempos para archivo 22_5.txt
Tiempo promedio (ms)
260
258
256
254
252
250
248
246
244
2
4
6
8
10
Valor de k (cantidad de clusters)


Tiempos para archivo 30_3.txt
Tiempo promedio (ms)
460
440
420
400
380
360
2
4
6
8
10
Valor de k (cantidad de clusters)

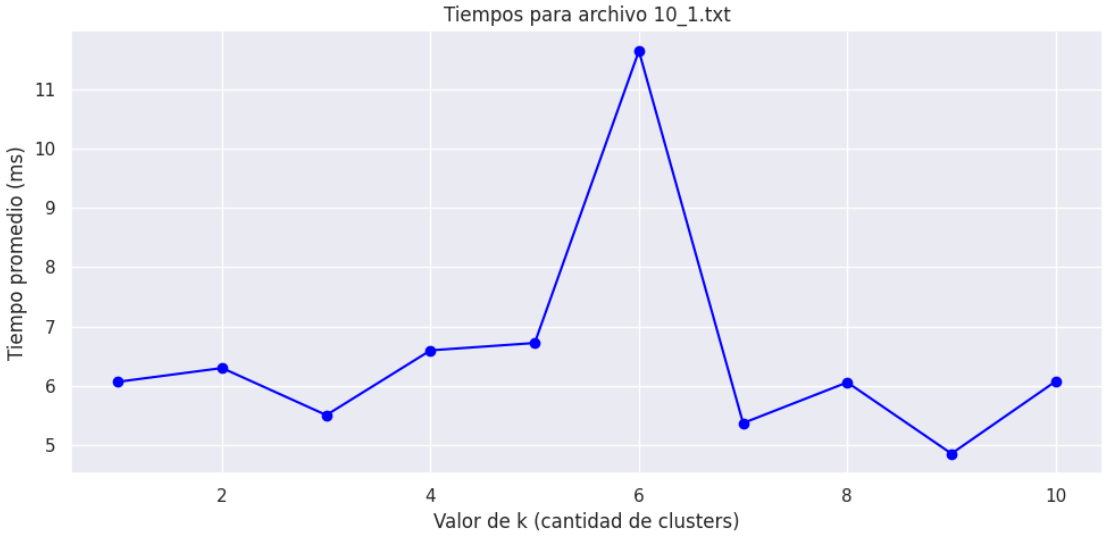

Tiempos para archivo 30_5.txt
Tiempo promedio (ms)
735
730
725
720
715
710
705
2
4
6
8
10
Valor de k (cantidad de clusters)

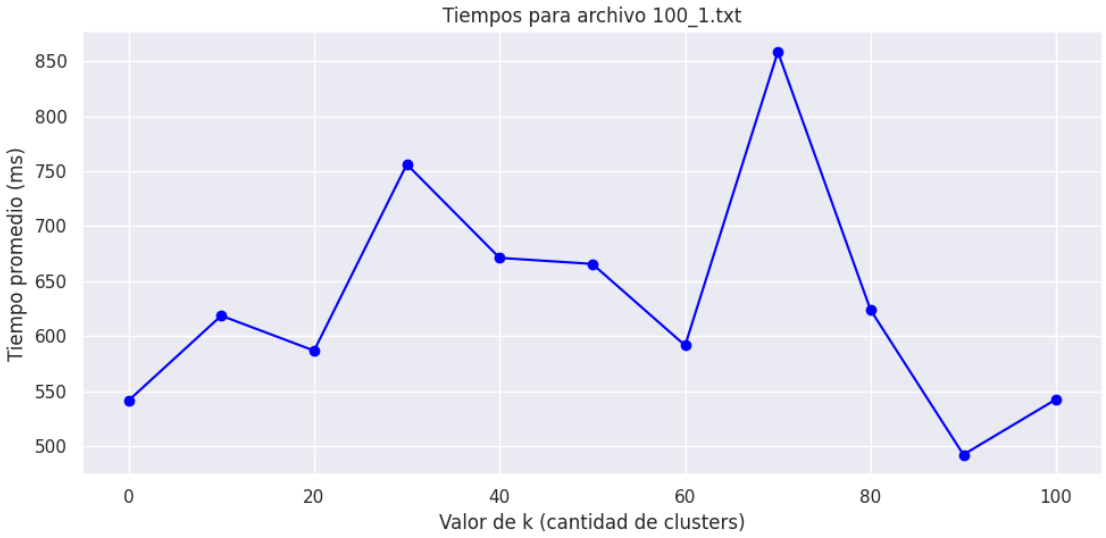

Tiempos para archivo 40_5.txt
Tiempo promedio (ms)
1020
1000
980
960
940
920
900
2
4
6
8
10
Valor de k (cantidad de clusters)


Tiempos para archivo 45_3.txt
Tiempo promedio (ms)
2200
2150
2100
2050
2000
2
4
6
8
10
Valor de k (cantidad de clusters)



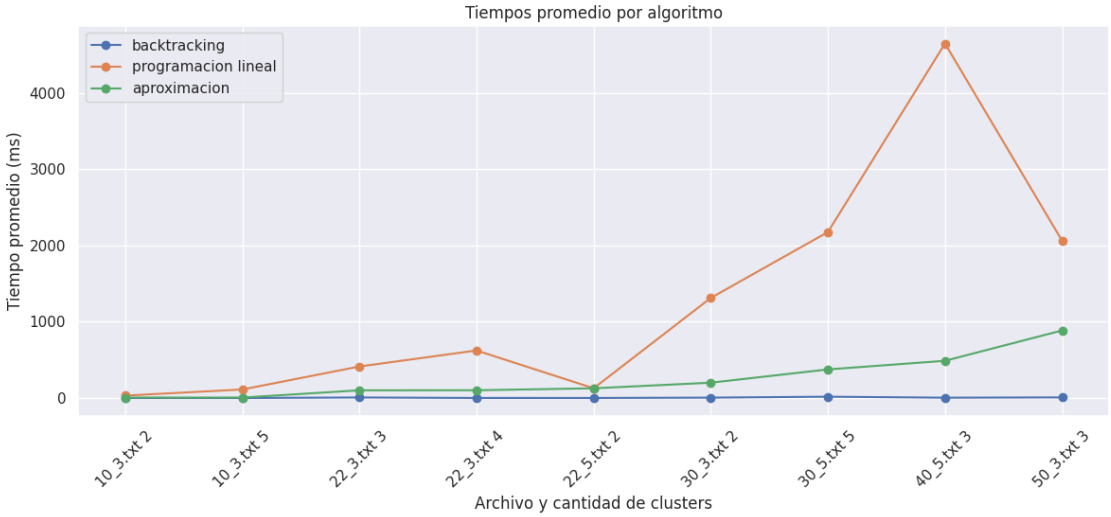
6.5.4. Gráficos de variabilidad del valor K en grafos lista





Se puede notar que a medida que encuentra cierto valor de K que es el máximo posible, el tiempo de ejecución va bajando. Esto se da por cómo funciona el Algoritmo de Louvain, que como busca comunidades, cuando encuentra la mejor, el grafo ya no cambia. Y esto pasa en los ejemplos cuando se da un cierto valor de K y desde ese valor en adelante, el resultado ese el mismo, es decir, las mismas comunidades que ya son las que cumplen la máxima modularización, que sería la distancia máxima dentro del cluster.

6.6. Comparacion



Podemos notar que el algoritmo de backtracking es el más rápido de los tres en los valores pedidos para las pruebas, quitando el caso 45_3.*txt* con $K = 7$, que es donde más afecta la curva exponencial de la complejidad.