.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Comunidades NP-Completas

16 de junio de 2025

Julen Leonel Gaumard
111379

Noelia Salvatierra
100116

Franco Macke
105974

1. Demostración NP

Dado nuestro problema de decisión de Clustering por bajo diámetro: Dado un grafo no dirigido y no pesado, un número 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?

Para demostrar que este problema se encuentra en NP, debo encontrar un verificador polinomial (validador eficiente de tiempo de ejecución polinomial), el cual, dada una solución çlusters", verifique si es una solucion correcta o no.

Siendo la solución una lista de listas de clusters, a continuación se presentará un posible validador polinomial:

```
1
2  def validador_clustering(grafo, clusters, k, C):
3      # Verifico n mero de clusters
4      if len(clusters) > k:
5          return False
6
7      """
8      Convierto cada lista de cluster a set asi saco vertices repetidos
9      """
10
11     clusters_nuevo = []
12     for cluster in clusters:
13         set_cluster = set(cluster)
14         clusters_nuevo.append(set_cluster)
15
16     """
17     Verifico que cada vertice de cada cluster pertenezca al grafo y que la cantidad
        de todos los vertices de todos los clusters sea igual a len(grafo)
18     """
19
20     contador_vertices_clusters = 0
21     for cluster in clusters_nuevo:
22         contador_vertices_clusters = contador_vertices_clusters + len(cluster)
23         for vertice in cluster:
24             if v not in grafo:
25                 return False
26     if contador_vertices_clusters != len(grafo):
27         return False
28
29     # Verifico di metro en cada cluster
30     for cluster in clusters:
31         for i in range(len(cluster)):
32             for j in range(i + 1, len(cluster)):
33                 u, v = cluster[i], cluster[j]
34                 distancia = bfs_distancia_uv(grafo, u, v)
35                 if distancia is None or distancia > C:
36                     return False
37     return True
38
39 def bfs_distancia_uv(grafo, u, v):
40     if u == v:
41         return 0
42     visitado = set()
43     # (nodo, distancia)
44     cola = deque([(u, 0)])
45     while cola:
46         actual, distancia = cola.popleft()
47         if actual == v:
48             return distancia
49         if actual not in visitado:
50             visitado.add(actual)
51             for vecino in grafo.get(actual, []):
52                 if vecino not in visitado:
53                     cola.append((vecino, distancia + 1))
```

```
54      return None
```

Listing 1: Algoritmo principal

La complejidad es O(k + V + E), por el bfs que, en el peor caso, recorre todo el grafo, siendo E el número de aristas totales, V la cantidad de vértices, y k, el tamaño de clusters totales y es polinomial respecto a las variables del problema, por lo tanto, como encontramos un validador polinomial y eficiente, podemos decir que Clustering por bajo diámetro se encuentra en NP.

2. Demostración NP-Completo

Dada la versión de decisión del problema de Clustering por bajo diámetro: Dado un grafo no dirigido y no pesado, un número 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).

Para que este problema sea NP-Completo, se debe demostrar dos cosas:

1) Que K-Clustering se encuentra en NP.

2) Dado el problema de decisión de K-Coloring NP-Completo: Dado un grafo G=(V,E) y un número k, ¿es posible asignar uno de k colores a cada vértice de G, de forma que ningún par de vértices adyacentes tengan el mismo color?, obtener una reducción polinomial K-Coloring

$$\leq p$$

KClustering.

1) La demostración de esto ya se hizo en la sección anterior . 2) Queremos resolver el problema de decisión de KClustering con la caja que resuelve el problema de decisión de K Coloring, esta caja recibe un grafo y los k colores.

Vamos a definir una posible reducción:

1) El valor del k"del problema de KClustering coincide con el valor del k recibido por el problema de K-Coloring.

2) Construimos un nuevo grafo G": (V", E").

3) Para cada arista (u, v) perteneciente a E, reemplazamos la arista por un camino de longitud C+1:

a) Introducimos C nuevos vértices intermedios: a1, a2, ..., ac.

b) Agregamos las aristas: u - a1 - a2- ... - ac - v.

Así: La distancia entre u y v en G.ªhora es C+1, Entonces, no pueden estar en el mismo cluster (porque violarían el límite de diámetro C). Hacemos esto para todas las aristas.

La idea de esta reducción es: Queremos que dos vértices adyacentes en el grafo original no puedan estar en el mismo cluster.

Para eso, hacemos que la distancia entre vértices adyacentes sea mayor que C.

Así, los únicos vértices que pueden compartir cluster son los que no están conectados entre sí en el grafo original.

Luego demostramos que nuestra reducción es correcta demostrando el si y solo si de: G es k-colorable el grafo G sii puede particionarse en a lo sumo k clusters de diámetro a lo sumo C.

Demuestro la ida: Si G es k-coloreable:

1) Asignamos cada color i perteneciente a 1, ..., k a un cluster Vi

2) Ponemos cada vértice original v perteneciente a V en su cluster de acuerdo con su color,

3) Como no hay adyacencias entre vértices del mismo color, los vértices originales en cada cluster están a distancia mayor que C en G"solo si estaban adyacentes en G, pero como no lo están, el diámetro entre ellos en G.ᵉˢ menor o igual a C (o infinito si no hay camino).

4) Para los nodos intermedios, los ubicamos en clusters arbitrarios o solos (no afecta el resultado

5) Luego concluimos que existe un Clustering válido con diámetro menor o igual C y menor o igual k clusters.

Ahora demuestro la vuelta: Si G"tiene clustering válido con menor o igual k clusters y diámetro menor o igual C:

1) Consideramos solo los vértices originales V.

2) Como en G", los vértices que eran adyacentes en G están a distancia C+1, no pueden estar juntos en un cluster.

3) Entonces, cualquier partición válida en k clusters de diámetro menor o igusl a C sobre Gçorresponde a una partición de V donde ningún par de adyacentes están juntos.

4) Luego se puede concluir que existe una k-coloración válida de G.

Como demostramos 1) y 2) podemos concluir que el problema de decisión de K-Clustering bajo diámetro C es NP-Completo.

3. Ejemplos de ejecución

Antes de proveer los ejemplos de ejecución, explicaremos brevemente como funcionan tanto el algoritmo implementado por Backtracking, como el implementado por Programación Lineal.

3.1. Backtracking

Buscamos dividir el conjunto de nodos en **k** clústeres minimizando el diámetro máximo. Lo va a hacer explorando el espacio de soluciones de forma exhaustiva y aplicando las podas que pueda para acelerar la búsqueda.

Como todo algoritmo de backtracking debemos definir su caso base:

Establecemos al mismo, como el caso en el cual, **todos los vertices se encuentran asignados a algún cluster.**

```
if max_diametro < mejor_solucion['max_diametro']:
    mejor_solucion['max_diametro'] = max_diametro
    mejor_solucion['clusters'] = {k: v.copy() for k, v in clusters.items()}
```

Y cuando llegamos a uno, comparamos con el actual y nos quedamos con la mejor solución hasta el momento.

Establecemos al diametro, de cada cluster, como el maximo de las distancias internas.

```
nuevo_diametro = max(diametro_actual, max(distancias[vertice][v] for v in cluster))
```

Cada nodo debe ser asignado a un unico cluster y lo haremos llenando cada cluster al completo antes de comenzar otro para evitar soluciones simétricas.

```
if not esta_lleno_los_clusters_anteriores(clusters, cluster_index):
    continue
```

Sumamente importantes son las podas en estos algoritmos, nosotros realizamos una poda por optimalidad parcial.

```
if max(diametros_clusters) >= mejor_solucion['max_diametro']:
    return
```

Si en algún momento el diámetro actual de algún clúster es peor que la mejor solución conocida, se corta la búsqueda:

De esta forma nuestro algiritmo de backtracking irá recorriendo cada vertice, asignanandolo a un cluster y comprobando si esa asignación lleva a una mejor solución que la previamente encontrada.

3.2. Programación Lineal

Todo algoritmo de programación lineal nos obliga a establecer variables y restricciones y tomar una seria de decisiones.

Primero definimos sus variables, nuestra implementación esta compuesta por:

- **x[v, i]**: Binaria. Y representa que el nodo **v**, se encuentra asignado al cluster **i**.
- **D[i]**: Representa el **diametro** del cluster **i**.
- **D_max**: Representa el **diametro maximo** entre todos.

También debemos plantear las restricciones que tendrá nuestro modelo:

- Cada nodo puede petenecer a un **único clúster**

```
    model += pulp.LpSum(x[v, i] for i in range(k)) == 1
```

- **D[i]** debe ser al menos el diametro del cluster.

```
    model += D[i] >= dist * (x[u, i] + x[v, i] - 1)
```

- **D_max** debe ser al menos tan grande como el mayor de los clusters

```
    model += D_max >= D[i]
```

Por ultimo, como todo, algoritmo de PL, debemos definir la funcion objetivo:

```
model += D_max
```

Buscamos **minimizar el diámetro máximo** entre todos los clústeres.

3.3. Comparación

Usamos un generador random de casos con el siguiente codigo:

```
def generar_casos(self, cantidad, n_min=10, n_max=30, m_min=None, m_max=None,
k_min=2, k_max=5):

    casos = []
    for _ in range(cantidad):
        n = random.randint(n_min, n_max)
        max_aristas = n * (n - 1) // 2
        m_sup = m_max if m_max else max_aristas
        m_inf = max(n - 1, m_min or n)

        m = random.randint(m_inf, min(m_sup, max_aristas))
        k = random.randint(k_min, min(k_max, n))
        grafo = self.generar_grafo(n, m)
        casos.append((grafo, k))
    return casos
```

Listing 2: Generación de ejemplos

En la tabla, los valores **n**, **m** y **k** representan las características principales del grafo y la configuración del problema:

- **n**: Número de vértices del grafo. Indica la cantidad total de nodos o puntos que conforman la estructura del grafo.
- **m**: Número de aristas del grafo. Representa la cantidad de conexiones o enlaces entre los vértices.
- **k**: Cantidad de clusters o grupos en los que se desea dividir el grafo. Este parámetro define cuántas particiones debe tener la solución final.

Estos tres parámetros son esenciales para entender la complejidad y las dimensiones del problema que se está resolviendo, ya que influyen directamente en la dificultad computacional de los algoritmos evaluados (Backtracking y Programación Lineal).

En los resultados se ve que Backtracking con podas es bastante más eficiente que Programación Lineal.

Para visualizar la diferencia de velocidades:

Caso	n	m	k	Tiempo BT (s)	Tiempo PL (s)	Diam. BT	Diam. PL	Ganador
1	20	48	2	0.001386	0.299332	3	3	Backtracking
2	10	17	2	0.001107	0.085534	3	3	Backtracking
3	10	25	4	0.003772	0.083510	2	2	Backtracking
4	13	37	4	0.001404	0.527947	1	1	Backtracking
5	19	163	2	0.001179	0.112392	1	1	Backtracking
6	18	73	3	0.000814	0.183320	2	2	Backtracking
7	15	17	3	0.014322	0.266165	3	3	Backtracking
8	17	46	4	0.003247	0.280822	2	2	Backtracking
9	11	40	2	0.000518	0.598255	2	2	Backtracking
10	11	30	3	0.000785	0.071301	2	2	Backtracking

Cuadro 1: Comparación de tiempos y diámetro máximo entre Backtracking y Programación Lineal

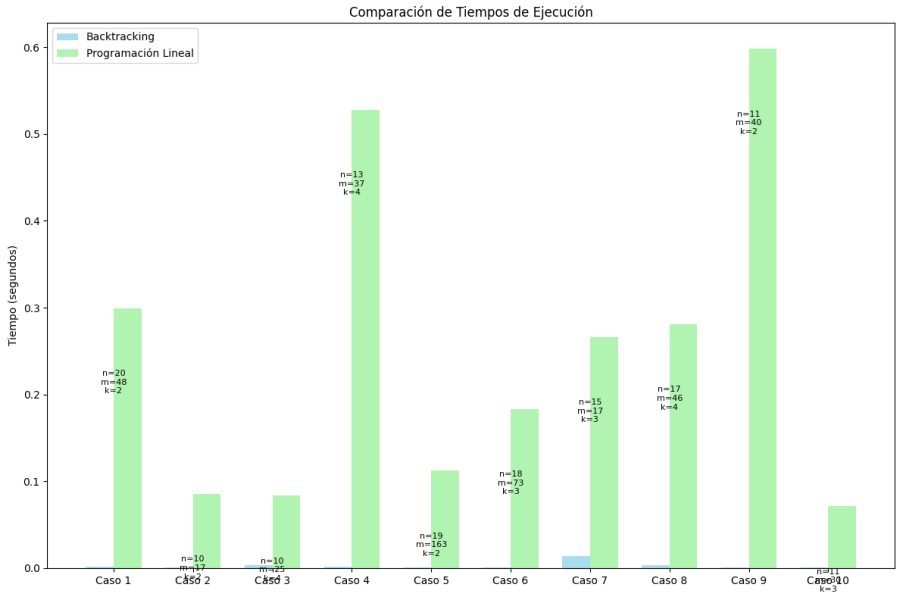

Figura 1: Comparación visual de tiempos entre los algoritmos Backtracking y Programación Lineal.

En conclusión, al comparar la implementación en Programación Lineal y en Backtracking, se observa una diferencia significativa en el rendimiento a favor del Backtracking. Esto se lo podemos atribuir tanto a las podas que decidimos hacer en nuestro algoritmo de Backtracking, como a la alta complejidad que posee el algoritmo encargado de resolver nuestro problema por PL.

3.4. Backtracking

Corrección de cálculo de $nuevo_diametro$

```
def calcular_nuevo_diametro(vertice, cluster, distancias, diametro_actual,
    mejor_solucion):
    nuevo_diametro = diametro_actual

    for v in cluster:
        d = distancias[vertice][v]
        if d == float('inf'):
            break
        nuevo_diametro = max(nuevo_diametro, d)

```

```
10          if nuevo_diametro >= mejor_solucion['max_diametro']:
11              break
12
13      return nuevo_diametro
```

Listing 3: Cálculo del nuevo diámetro al intentar agregar un vértice a un cluster

3.5. Comparación código óptimo vs Louvain

Dada una cota aproximada para estudiar la aproximación por Louvain vs los óptimos:

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

Podemos utilizarla con los set de datos para los cuales conocemos todas las incógnitas y de esta manera calcular la cota inferior referente a la exactitud del algoritmo:

Instancia	K clusters esperados (Óptimo)	K clusters obtenidos (Aproximación)	Razón r(A)
6-1	3	2	0.7
11-1	3	2	0.7
15-8	5	2	0.4
23-21	3	2	0.7

Cuadro 2: Tabla de Cantidad de k clusters

Para complementar, se tomaron en cuenta cuatro ejemplos adicionales cuyas condiciones hacen inmanejables los sets de datos para el algoritmo exacto y calculamos los k clusters con la aproximación de Louvain. Una vez hecho eso, lo arrojamos en la siguiente tabla:

Instancia	K clusters obtenidos (Aproximación)
250-71	5
250-119	9
250-198	6
250-203	7

Cuadro 3: Tabla de cantidad de clusters (set de datos inmanejable)

Es importante aclarar que el criterio utilizado para considerar un set de datos como inmanejable es el tiempo excesivo que requiere el algoritmo exacto para resolver una instancia. Esto se debe al gran número de combinaciones posibles que debe explorar. De esta forma se genera un crecimiento exponencial en la cantidad de posibilidades a medida que aumenta el tamaño del problema, lo que resulta en tiempos de ejecución imprácticos.

A partir de estos datos, obtenemos que el valor mínimo para la cota es 0.40, lo cual consideramos es un valor poco razonable para determinar que es una buena aproximación Louvain (no es una muy buena aproximación).

4. Anexo

4.1. Demostración NP

Agregamos las correcciones del validador polinomial, teniendo en cuenta que:

- La solución es inválida si hay un vértice o más repetidos.
- Usamos una matriz con las distancias precalculadas guardadas dentro de la instancia de Grafo.

```
1
2  def validador_clustering(grafo, clusters, k, C):        # O(k * V^2)
3      if len(clusters) > k:
4          return False
5
6      contador_vertices_clusters = 0
7      for cluster in clusters:
8          contador_vertices_clusters += len(cluster)
9          for vertice in cluster:
10             if vertice not in grafo:
11                 return False
12     if contador_vertices_clusters != len(grafo):
13         return False
14
15     for cluster in clusters:
16         for i in range(len(cluster)):
17             for j in range(i + 1, len(cluster)):
18                 u, v = cluster[i], cluster[j]
19                 distancia = grafo.distancia(u, v)
20                 if distancia is None or distancia > C:
21                     return False
22     return True
```

Listing 4: Algoritmo principal

La complejidad es $O(k \cdot V^2)$, siendo V la cantidad de vértices y k el número de clusters. El validador es polinomial respecto a las variables del problema, por lo tanto, como encontramos un validador eficiente, podemos decir que Clustering por bajo diámetro se encuentra en NP.

4.2. Demostración NP-Completo

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

Para que este problema sea NP-Completo, se debe demostrar:

1. Que K-Clustering se encuentra en NP.
2. Dado el problema de decisión de K-Coloring (NP-Completo), obtener una reducción polinomial:

$$\text{K-Coloring} \leq_p \text{K-Clustering}$$

1) La demostración ya se hizo en la sección anterior.

2) Queremos resolver K-Coloring utilizando una caja negra que resuelve K-Clustering. Esta caja recibe un grafo y el valor de k.

Definimos la siguiente reducción:

- Definimos el diámetro máximo permitido $C' = 1$.
- Para cada arista $(u, v) \in E$ en G, se introduce un camino de longitud 2 en G', añadiendo un vértice intermedio a_{uv}, de modo que la distancia entre u y v en G' sea 2.
- Para cada par de vértices no adyacentes $(u, v) \notin E$, se añade una arista directa (u, v), por lo que su distancia en G' es 1.

Sea $G = (V, E)$ una instancia de $K - Coloring$. Construimos una instancia $(G', k, C = 1)$ de $K - Clustering$ bajo diámetro:

- El conjunto de vértices de G' es:

  $$V' = V \cup \{a_{uv} : (u, v) \in E\}$$

  Es decir, agregamos un nuevo vértice intermedio a_{uv} por cada arista del grafo original.
- Para cada $(u, v) \in E$, agregamos:

  $$(u, a_{uv}) \quad \text{y} \quad (v, a_{uv})$$

  Lo que da como resultado una distancia 2 entre u y v en G'.
- Para cada $(u, v) \notin E$, agregamos:

  $$(u, v)$$

  Por lo tanto, su distancia en G' es 1.

Así, los vértices adyacentes en G quedan a distancia $C' + 1 = 2$, lo que excede el límite de diámetro $C' = 1$, y no pueden estar en el mismo cluster. En cambio, los vértices no adyacentes en G sí pueden agruparse juntos.

La idea de esta reducción es hacer que los únicos vértices que puedan compartir cluster sean los no adyacentes en G, garantizando que los adyacentes queden separados.

Demostramos que la reducción es correcta mediante el siguiente "si y solo si":

G es k-colorable si y solo si puede particionarse en a lo sumo k clusters de diámetro a lo sumo C en G'.

Demostración $\Rightarrow$: Si G es k-coloreable:

1. Asignamos cada color $i \in \{1, \ldots, k\}$ a un cluster V_i.
2. Ponemos cada vértice original $v \in V$ en su cluster correspondiente.
3. Como no hay adyacencias entre vértices del mismo color, su distancia en G' es a lo sumo C.
4. Los nodos intermedios pueden ubicarse en cualquier cluster o quedar solos.
5. Luego, existe un clustering válido en G' con diámetro menor o igual a C y a lo sumo k clusters.

Demostración $\Leftarrow$: Si G' tiene un clustering válido con a lo sumo k clusters y diámetro menor o igual a C:

1. Consideramos solo los vértices originales V.
2. Como en G', los adyacentes en G están a distancia $C+1$, no pueden estar en el mismo cluster.
3. Cualquier clustering válido en G' implica una partición de V donde ningún par adyacente está en el mismo conjunto.
4. Luego se puede construir una k-coloración válida de G.

Como demostramos 1) y 2) podemos concluir que el problema de decisión de K-Clustering por bajo diámetro es NP-Completo.

4.3. Backtracking

Corrección de cálculo de *nuevo_diametro*

```python
def calcular_nuevo_diametro(vertice, cluster, distancias, diametro_actual, mejor_solucion):
    nuevo_diametro = diametro_actual

    for v in cluster:
        d = distancias[vertice][v]
        if d == float('inf'):
            break
        nuevo_diametro = max(nuevo_diametro, d)

        if nuevo_diametro >= mejor_solucion['max_diametro']:
            break

    return nuevo_diametro
```

Listing 5: Cálculo del nuevo diámetro al intentar agregar un vértice a un cluster

Realizamos las correcciones correspondientes en el código de:

- Desanidar las funciones.
- calcular_nuevo_diametro() retorna un valor en vez de hacer un break.
- Refactorizamos backtracking_recursivo() para que devuelva valores.

4.3.1. Validación de correctitud

Generamos varios subsets de datos, para validar manualmente la correctitud del algoritmo.

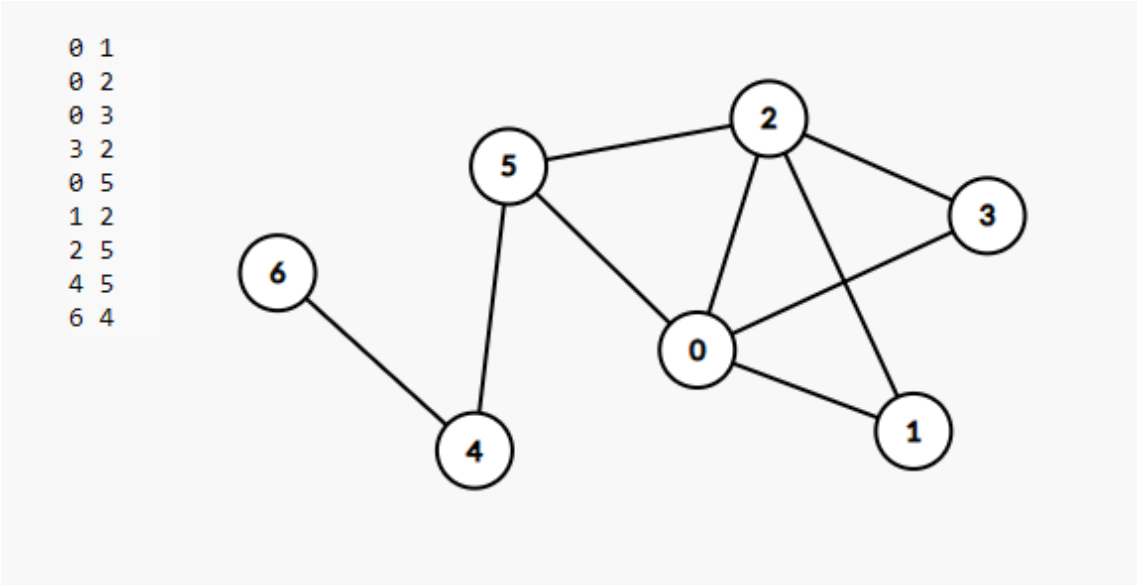

Figura 2: Subset de datos (bt_1.txt) con su respectivo grafo

Ejecutando el algoritmo de backtracking, sobre ese subset, con un $k = 2$ nos devuelve como resultado la siguiente agrupación:

$$0 : [0, 2, 5, 1, 3], 1 : [4, 6]$$

Con una distancia máxima de 2.

Si lo volvemos a calcular con un $k = 4$ obtenemos:

$$0 : [0, 2, 5], 1 : [1], 2 : [3], 3 : [4, 6]$$

Con una distancia máxima de 1.

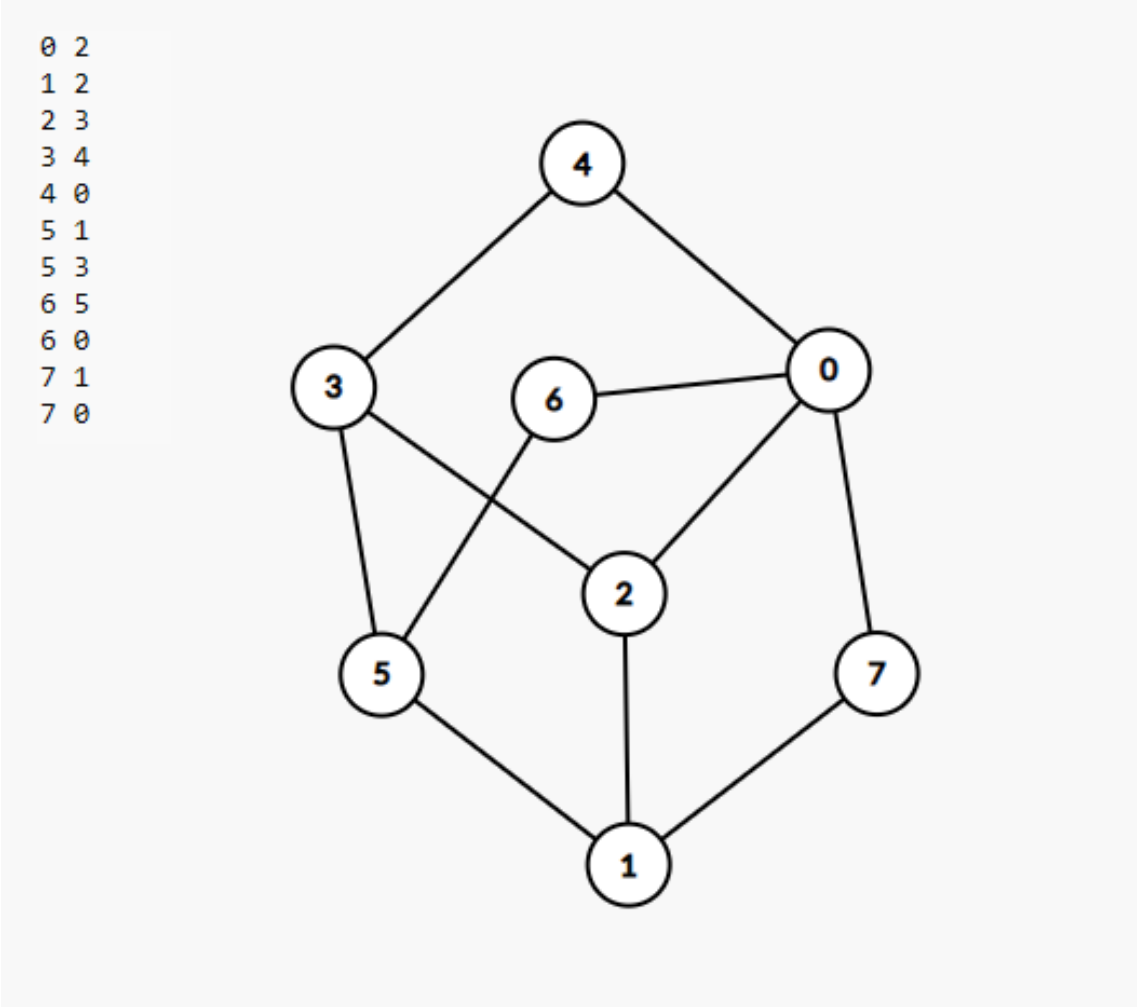

Figura 3: Segundo subset de datos (bt_2.txt) con su respectivo grafo

Ejecutando el algoritmo de backtracking, con un $k = 2$, obtenemos la siguiente agrupación:

$$0 : [0, 2, 1, 3, 5, 6], 1 : [4, 7]$$

Con una distancia máxima de 2.

Si lo volvemos a calcular con un $k = 4$ obtenemos:

$$0 : [0, 2], 1 : [1, 7], 2 : [3, 4], 3 : [5, 6]$$

Con una distancia máxima de 1.

4.4. Programación Lineal

Correcciones:

Todo algoritmo de programación lineal nos obliga a establecer variables, restricciones y tomar una serie de decisiones.

Primero definimos sus variables. Nuestra implementación está compuesta por:

- **x[v, i]**: Variable binaria que vale 1 si el nodo **v** está asignado al clúster **i**, y 0 en caso contrario.
- **D[i]**: Variable continua que representa el **diámetro** del clúster **i**.
- **D_max**: Variable continua que representa el **diámetro máximo** entre todos los clústeres.

También debemos plantear las restricciones que tendrá nuestro modelo:

- **Cada nodo pertenece a un único clúster:**

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

- **El diámetro de un clúster debe ser al menos la distancia entre dos nodos asignados a él:**

$$D_i \geq d(u,v) \cdot (x_{u,i} + x_{v,i} - 1) \quad \forall u, v \in V,\ \forall i \in \{1, \ldots, k\}$$

  Esta restricción solo tiene efecto cuando ambos nodos u y v están asignados al mismo clúster i. En ese caso, $x_{u,i} + x_{v,i} - 1 = 1$, y se exige que $D_i \geq d(u,v)$. Si alguno de los dos no pertenece al clúster i, entonces el lado derecho es ≤ 0, por lo que la restricción se cumple automáticamente. Esto permite que el modelo no fije valores de D_i más grandes de lo necesario, ya que minimizamos $D_{\max}$.

- **El diámetro máximo debe ser al menos el mayor de todos los clústeres:**

$$D_{\max} \geq D_i \quad \forall i \in \{1, \ldots, k\}$$

Por último, como todo algoritmo de programación lineal, debemos definir la función objetivo:

$$\text{mín}\, D_{\max}$$

Es decir, buscamos **minimizar el diámetro máximo** entre todos los clústeres.

4.5. Louvain

El objetivo es encontrar una particion de los vertices que maximice la modularidad. Entendiéndose modularidad como una medida interna que evalua que tan buenas son las comunidades, en base a cuantas conexiones internas tienen frente a las externas.

El algortimo inicia generando una comunidad para cada vertice.

```
1     particion = {v: v for v in grafo.obtener_vertices()}
```

Cada iteracion ejecuta una_iteracion_louvain(), el cual intentara mover cada nodo a la comunidad de sus vecinos si esto mejora la modularidad.

El método iterará sobre cada nodo, de manera repetida hasta que no se logre mejorar la modularidad de las comunidades.

```
1     while mejora:
2         mejora = False
3         random.shuffle(nodos)
4         for nodo in nodos:
```

Se obtendrán los vecinos de la comunidad que se esta evaluando en la iteración y se irá calculando la cantidad de enlaces que posee cada una.

```
    comunidad_actual = particion[nodo]
    vecinos = grafo.vecinos(nodo)

    enlaces_por_comunidad = defaultdict(int)
    for vecino in vecinos:
        comunidad_vecino = particion[vecino]
        enlaces_por_comunidad[comunidad_vecino] += 1
```

La siguiente parte es la principal del algoritmo, irá evaluando cada vecino y decidirá si conviene o no agruparlos. Esto lo hace mediante:

```
    mejor_delta = 0
    mejor_comunidad = comunidad_actual

    particion[nodo] = -1
    for comunidad, enlaces in enlaces_por_comunidad.items():
        suma_grados = sum(grados[n] for n in particion if particion[n] == comunidad
    )
        delta_q = enlaces - grados[nodo] * suma_grados / (2 * m)
        if delta_q > mejor_delta:
            mejor_delta = delta_q
            mejor_comunidad = comunidad
```

Para cada vecino calculara la suma de grados de la comunidad y el cambio de modularidad si decidimos agregar el nodo actual a la comunidad vecina.

```
    suma_grados = sum(grados[n] for n in particion if particion[n] == comunidad)
    delta_q = enlaces - grados[nodo] * suma_grados / (2 * m)
```

Si el delta_q es mejor al mejor actual, va a quedarse con esa comunidad como candidato.

Una vez se terminó de evaluar cada uno de los vecinos de la comunidad se evaluará si la **mejor_comunidad** es mejor que la actual y si lo es se reemplazará.

```
    if mejor_comunidad != comunidad_actual:
        particion[nodo] = mejor_comunidad
        mejora = True
    else:
        particion[nodo] = comunidad_actual
```

Una vez que no se puede mejorar más la modularidad.

Se genera una grafo nuevo, mediante grafo_inducido(grafo, particion), en el cual cada comunidad será un super-nodo. A la vez se almacenará este resultado parcial en la lista **jerarquia**.

Y se ejecutará, con el nuevo grafo con las nuevas comunidades, una_iteracion_louvain().

Se volverá a repetir este proceso hasta que el número de comunidades llega al k buscado o a un valor menor.

Ahi se usará la lista de **jerarquia**, para buscar el resultado parcial más cercano a k. Y se calculará el diametro máximo, con el resultado elegido.

4.5.1. Comparación código óptimo vs Louvain

Instancia	D. máxima [K] (Óptimo)	D. máxima [K] (Aproximación)	Razón r(A)
6-1	1 [3]	2 [2]	2.0
11-1	1 [3]	2 [2]	2.0
15-8	1 [5]	2 [3]	2.0

Cuadro 4: Tabla de distancias máximas y k clusters

Podemos observar que aunque se solicita al algoritmo una solución con k clusters, Louvain no nos devuelve una buena aproximación, ya que su solución tiene una menor cantidad de clusters,

los cuales, a la vez, tienen una distancia máxima superior, mostrando que nos encuentra soluciones que no son óptimas.

Presentamos otros ejemplos con subset de datos inmanejables, creados de maneral manual, lo cual nos permite conocer sus resultados optimos.

Instancia	D. máxima [K] (Óptimo)	D. máxima [K] (Aproximación)	Razón r(A)
200	1 [50]	3 [34]	3.0
250	1 [63]	3 [10]	3.0
300	1 [75]	3 [11]	3.0

Cuadro 5: Tabla de cantidad de clusters (set de datos inmanejable)

4.6. Comparación

Agregamos dos gráficos para ver los tiempos de **Programación Lineal** y **Backtracking**. Cada columna para ambos casos está ejecutado sobre el mismo set de datos.

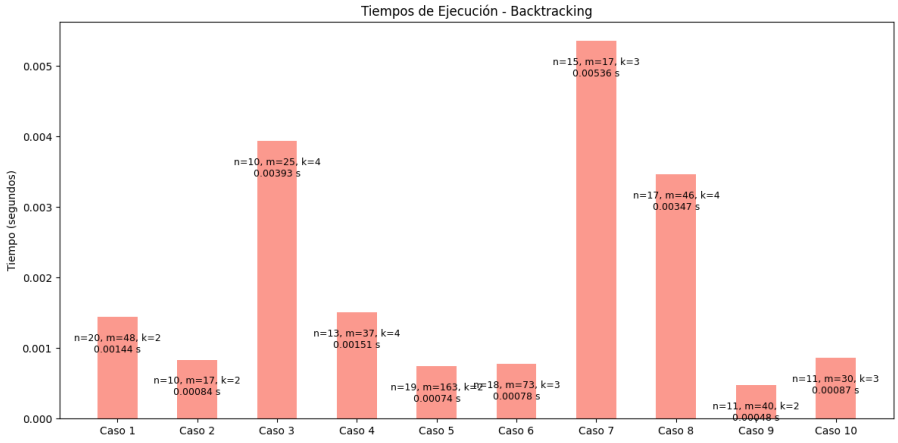

Figura 4: Tiempos de algoritmo Backtracking.

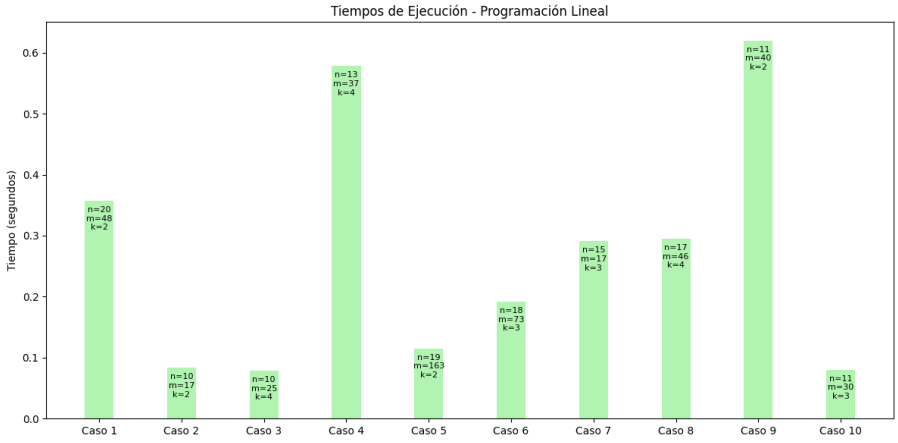

Figura 5: Tiempos de algoritmo Programación Lineal.

5. Anexo 2

5.1. Demostración NP-Completo

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

Para que este problema sea NP-Completo, se debe demostrar:

1. Que K-Clustering se encuentra en NP.
2. Dado el problema de decisión de K-Coloring (NP-Completo), obtener una reducción polinomial:

$$\text{K-Coloring} \leq_p \text{K-Clustering}$$

1) La demostración ya se hizo en la sección anterior.

2) Queremos resolver K-Coloring utilizando una caja negra que resuelve K-Clustering. Esta caja recibe un grafo y el valor de k.

Definimos la siguiente reducción:

- Definimos el grafo $\overline{G}$ complemento del grafo G. Siendo por definición, un grafo que comparte todos los vertices, pero que posee como aristas, todos los pares de vértices que no se encuentran conectados en el grafo original.
- Definimos el diámetro máximo permitido $C = 1$.
- Definimos que existe un k-coloreo de k colores en G, si existe un k-clustering con k clusters en $\overline{G}$.

Sea $G = (V, E)$ una instancia de $K - Coloring$. Construimos una instancia $(\overline{G}, k, C = 1)$ de $K - Clustering$ bajo diámetro.

La idea de esta reducción es generar clusters que representen asignaciones de vertices a un color. Como la instancia de $K - Clustering$ la generamos con el complemento $\overline{G}$ y establecimos la distancia máxima como $C = 1$, en un cluster solo podrán haber vértices que sean adyacentes en el $\overline{G}$, lo que quita la posibilidad de que vértices adyacentes en G compartan clusters.

Si la instancia de $K-Clustering$ llegara a generar una cantidad de clusters menor a k significará que el problema de coloreo puede resolverse con menos colores. Y si llegara a no poder generar clusters, debido a la restriccion de $C = 1$, quiere decir que no se puede resolver el problema de coloreo con al menos k colores.

Demostramos que la reducción es correcta mediante el siguiente "si y solo si":

G es k-colorable si y solo si $\overline{G}$ puede particionarse en a lo sumo k clusters de diámetro a lo sumo C.

Demostración $\Rightarrow$: Si G es k-coloreable, entonces existe un clustering válido en $\overline{G}$:

1. Supongamos que tenemos un G con una coloración propia con k colores.
2. Construimos k clusters agrupando los vértices de un mismo color:

$$C_i = \{v \in V \mid c(v) = i\}$$

3. Observamos que, en ningún conjunto C_i, existen pares de vértices que sean adyacentes en G.

4. Por lo tanto, en $\overline{G}$, todos esos pares están conectados entre si, son adyancetes.
5. De esta forma cada uno de estos clusters tiene un diámetro ≤ 1 en $\overline{G}$.
6. Conclusión: Construimos un clustering válido con k clusters en $\overline{G}$.

Demostración $\Leftarrow$: Si $\overline{G}$ tiene un clustering válido con a lo sumo k clusters y diámetro menor o igual a C, entonces G es k-coloreable:

1. Supongamos que $\overline{G}$ admite partir sus vertices en k clusters, todos con un diametro ≤ 1.
2. Eso implica que, dentro de cada clusters, cualquier par de vértices son adyacentes entre si.
3. Esto significa que, en G, ningun par de vértices dentro de un cluster es adyacente.
4. Si le asignamos un color a los vértices de cada cluster. Tomando un color diferente para cada cluster.
5. Observamos que ningun par de vértices comparte color en G.
6. Conclusion: G es k-coloreable.

Como demostramos 1) y 2) podemos concluir que el problema de decisión de K-Clustering por bajo diámetro es NP-Completo.