.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

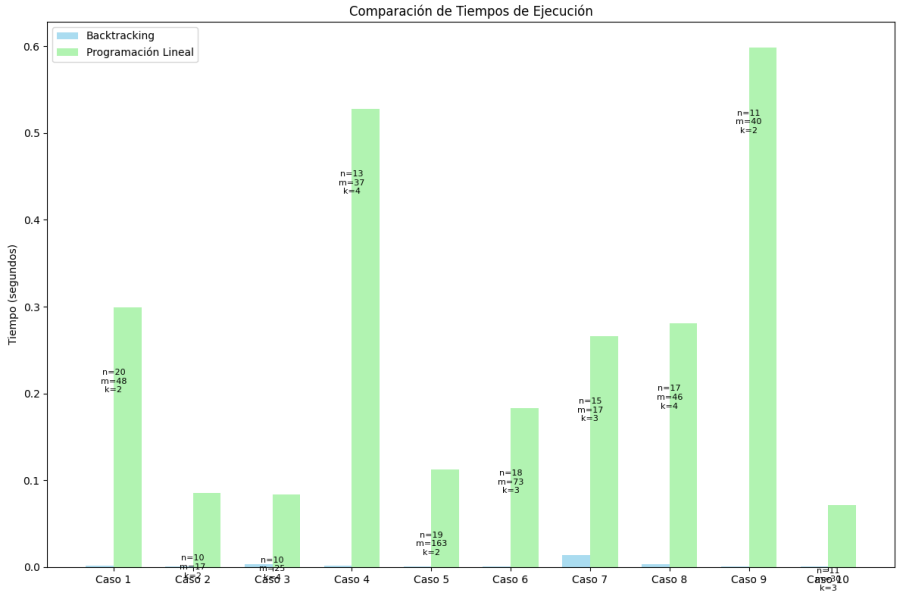

Figura 1: Comparación visual de tiempos entre los algoritmos Backtracking y Programación Lineal.

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. 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:

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 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

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).