Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Comunidades NP-Completas

17 de junio de 2025

M. B. García Lapegna
111848

M. U. Sáenz Valiente
111960

T. Goncalves Rei
111405

Índice

1. Consideraciones

Este informe se distribuirá de la siguiente manera:

- Sección 2: se enunciara el problema a resolver.
- Sección 3: se hara un analisis de la dificultad del problema y se hará la demostración formal para afirmar que es **NP-Completo**.
- Sección 4: se propondrán dos maneras de conseguir una solución optima para el problema propuesto.
- Sección 5: se analizaran distintas maneras de conseguir soluciones aproximadas para el problema en tiempo polinomial.
- Sección 6: desarrollo de las complejidades de los distintos algoritmos propuestos a lo largo del informe, ademas acompañados de sus gráficos correspondientes y comparaciones interesantes entre ellos.
- Sección 7: se explayaran las conclusiones del trabajo presentado.

2. Introducción

En este informe se abordaran distintas opciones que permiten separar un grafo en comunidades, y se expondrán las ventajas y desventajas de cada opción especifica. En particular, se analizara en detalle una opción especifica llamada **Colesterina por bajo diámetro**. Se presentara el problema y se demostrara que este es un problema **NP-Completo**. Sabiendo esto, se implementaran diferentes soluciones para este problema usando varias técnicas de diseño de algoritmos. Las diferentes implementaciones serán comparadas entre si según su eficiencia, lo que permitirá ver las ventajas y desventajas de cada una.

Dada la complejidad del problema presentado, se presentaran algunos algoritmos que permiten obtener soluciones que, aunque no sean optimas, dan muy buenas **aproximaciónes**. Ahondaremos en la técnica de diseño Greedy, que sera útil para este tipo de algoritmos. Se demostrara también que tan buenas o malas pueden ser esas aproximaciones en cada caso.

Por ultimo, se compararan los tiempos de ejecución del total de las implementaciones haciendo uso de varios gráficos comparativos, generados a partir de sets de prueba propios y algunos brindados por la cátedra.

2.1. Enunciado

Dado un grafo no dirigido y no pesado (en el cual los vértices representan personas y las aristas si son cercanas). Separar el grafo en comunidades, es decir, separar los vértices en K clusters(grupos). La manera de hacerlo es minimizando la distancia máxima entre dos vértices dentro de la misma comunidad.

Se pide resolver este problema de forma optima mediante Backtracking y Programacion Lineal, ademas resolverlo mediante el Algoritmo de Louvain y dar un Algoritmo Aproximado con una técnica Greedy.

Ademas, utilizando la versión de decisión del problema, demostrar que es un problema *NP-Completo* y que en efecto pertenece a *NP*.

3. Análisis del problema

3.1. Análisis de dificultad

Se busca demostrar que el problema de decisión propuesto pertenece a NP. Recordemos que un problema pertenece a NP significa que, dada la instancia de un problema y una solución, existe un certificador eficiente, es decir, un verificador que puede validar la solución en tiempo polinomial; en otras palabras, pueden ser resueltos en tiempo polinomial por una maquina de Turing indeterministica.

Nos interesa demostrar mas específicamente que el problema es **NP-Completo**. Decimos que los problemas *NP-Completo* son considerados los *mas difíciles* dentro de *NP*. Ademas, si un problema es *NP-Completo* significa que todo problema que pertenezca a *NP* puede reducirse polinomicamente a el.

En este trabajo se nos pide demostrar que el problema de decisión "*Clustering por bajo diámetro*" es un problema **NP-Completo**. Para esto tendremos que que demostrar tanto que el problema pertenece a *NP*, como que algún problema que pertenezca a *NP-Completo* al que llamaremos $X_{\in NP-C}$ puede reducirse en pasos polinómicos al problema de decisión propuesto. Es decir, se demostrara:

$$X_{\in NP-C} \leq_p \textit{Clustering por bajo diámetro}$$

Para facilitar la demostración, a partir de ahora llamaremos *CD* al problema de *Clustering por bajo diametro*.

3.2. Primera demostración: El problema propuesto pertenece a NP

Como se menciono anteriormente, para que un problema de decisión pertenezca a *NP* debe existir un verificador que se ejecute en tiempo polinomial para este.

A continuación se presentara un verificador propuesto para el problema de *CD*.

```
def validador_cd(grafo:Grafo, k, c, solucion,distancias):
    if len(solucion) > k: return False

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

    for cluster in solucion:
        for v in cluster:
            if v not in vertices or v in visitados : return False
            visitados.add(v)

    if visitados != vertices: return False

    for cluster in solucion:
        for v in cluster:
            for w in cluster:
                if v == w : continue
                if distancias[v][w]>c : return False

    return True
```

Como se puede notar, el verificador corrobora que:

- Verifica que a lo sumo sean k clusters.

  ```
      if len(solucion) > k: return False
  ```

  Esto lo hace en tiempo constante.

- Verifica que todos los vértices del grafo pertenezcan a la solución, sin repetidos y que por supuesto, todos los vértices de la solución sean los del grafo.

```
    vertices = set(grafo.obtener_vertices())
    visitados = set()

    for cluster in solucion:
        for v in cluster:
            if v not in vertices or v in visitados : return False
            visitados.add(v)

    if visitados != vertices: return False
```

Construir el set de vértices es una operación $O(V)$, luego se recorre cada vértice dentro de la solución lo cual también es $O(V)$.

- Verifica que en los clusters la distancia máxima entre los vértices no supere a c.

```
    for cluster in solucion:
        for v in cluster:
            for w in cluster:
                if v == w : continue
                if distancias[v][w]>c : return False
```

Por cada cluster de la solución, que a lo sumo serán k se verifica que cada par de vértices no supere el máximo c, esta operación tiene una complejidad temporal de $O(V^2)$. Y como se dijo, esto es por cada cluster, es decir, $O(k \cdot V^2)$.

Por lo descripto anteriormente, podemos decir que el algoritmo ejecuta en tiempo $O(V+k \cdot V^2)$, siendo V la cantidad de vertices,y k la cantidad clusters. Acotando superiormente a el termino V, podemos concluir que la complejidad de este verificador es $O(k \cdot V^2)$.

Por lo tanto, ejecuta en tiempo polinomial a las variables del problema, lo cual queda demostrado que se trata de un problema que se encuentra en NP.

3.3. Segunda demostración: El problema es NP-Completo

Para demostrar que CD es un problema **NP-Completo**, se haran una serie de demostraciónes consecutivas. Para ello, sera util considerar los siguientes problemas, que seran enunciados en su version de decisión:

- Problema de decisión de **K-Coloreo**: Dado un grafo y un numero natural k, ¿existe alguna forma de *pintar* los vertices del grafo en k colores, de manera que no haya dos vertices adyacentes con el mismo color?
- Problema de decisión de **Separación en R-Cliques (SRC)**: Dado un grafo, ¿existe alguna forma de separar los vertices del grafo en R subconjuntos, de manera que cada subconjunto sea un clique?

Estos dos problemas nos permitiran llevar a cabo dos demostraciónes consecutivas, que concluiran en que el problema de CD es efectivamente **NP-Completo**. En adelante nos refereriremos a ellos como KC y SRC, respectivamente. Partiremos asumiendo que KC es un problema **NP-Completo**. Tambien asumiremos que tanto KC como SRC pertenecen a NP.

Para realizar las demostraciones, definiremos lo siguiente: un problema determinado, llamemoslo X, es *reducible polinomicamente* a otro, al que llamaremos Y, si cumple:

- Cualquier instancia del problema X puede transformarse en una instancia del problema Y en una cantidad polinomica de pasos, respecto del tamaño de la entrada del problema X.
- Luego de la reducción del problema X al problema Y, la solución a la instancia del problema Y determina siempre correctamente el resultado de la instancia del problema X.

Cuando un problema X puede reducirse a un problema Y, usaremos la siguiente notación:

$$X \leq_p Y$$

3.3.1. Demostracion: SRC es un problema NP-Completo

Para demostrar que SRC es un problema **NP-Completo**, reduciremos polinomialmente el problema KC a este. Es decir, desmotraremos que:

$$KC \leq_p SRC$$

Antes, nombramos a los parametros recibidos por cada instancia de los problemas:

- Llamaremos $G(V, E)$ al grafo con los vertices V y las aristas E que recibe la instancia del problema KC. Tambien llamaremos K a la cantidad de colores que se pretenden asignar a los vertices del grafo.
- Llamaremos G' al grafo que recibe nuestra instancia de SRC. Llamaremos R a la cantidad de cliques que en los que se pretende dividir a los vertices del grafo.

Bajo estas condiciones, podemos enunciar lo siguiente: solo existe solución para la instancia del problema KC con sus parametros G y K si existe solución para el problema de SRC siendo G' el grafo complemento de G, y $R = K$. Esto puede escribirse formalmente de la siguiente manera:

$$\exists\ KC\ (G(V, E), K) \iff \exists\ SRC\ (G'(V', E'), R)$$

donde:

- $G' = \overline{G}$: Es decir, el grafo que recibe SRC es el grafo complemento del grafo G que recibio la instancia del problema KC.
- $R = K$: Es decir, buscamos separar G' en la misma cantidad de cliques que los colores que se pretenden asignar en G.

A continuación demostraremos la doble implicancia antes enunciada.

- **Primera implicancia**: $\exists\ KC\ (G(V, E)\ K) \implies \exists\ SRC\ (G(V', E'),\ R)$
  $\exists\ KC\ (G(V, E), K)\ \rightarrow\ \exists\ S = \{s_1, ..., s_k\}$ tal que $S_1 \cup S_2 \cup ... \cup S_k = V$.
  Ademas $\forall_{i<k, j<k}\ S_i \cap S_j = \varnothing$. Es decir, el conjunto S es una partición de V. Esta partición ademas cumple:

  $$\forall S_i = \{v_1, ..., v_m\} : (v_j \in S_i \land v_r \in S_i) \longrightarrow\ v_j \notin Ady(v_r).$$

  Esto ultimo solo sucede si no existe arista entre v_j y v_r en el grafo, es decir, $\nexists e = (v_m, v_r) \in E$.
  Por definición, sabemos que las aristas que pertenecen a G no van a existir en $\overline{G}$. Entonces:

  $$\forall S_i = \{v_1, v_2, ..., v_m\}, (v_m \in S_i \land v_r \in S_i) \rightarrow \nexists e = (v_m, v_r) \in E \rightarrow \exists e = (v_r, v_m) \in E',$$

  donde E' son las aristas de G', que fue definido como $\overline{G}$.
  Por lo tanto, todos los vértices de cada uno de los subconjuntos S_i tendrán aristas entre si. Es decir, habrán K subconjuntos que formaran K cliques en G'. Definimos $K = R$.
  Entonces, habrán R cliques en G'. Es decir, se cumple que $\exists\ SRC\ (G'\ (V', E'\ ),\ R)$.

- **Segunda implicancia**: $\exists\ SRC\ (G'(V', E'), R)\ \implies\ \exists\ KC\ (G(V, E), K)$
  $\exists\ SRC\ (G'(V', E'), R) \rightarrow\ \exists\ S = \{S_1, S_2, ..., S_r\}$, donde $S_i = \{v_1, ..., v_m\}$ tales que
  $S_1 \cup ... \cup S_r = V$ y ademas $\forall_{i<r, j<r}\ Si \cap S_j = \varnothing$.
  Es decir, S forma una partición de los vértices en R subconjuntos. Ademas estos subconjuntos cumplen que:

$$\forall\ S_i, (v_m \in S_i \wedge v_r \in S_i) \rightarrow \exists e = (v_m, v_r) \in G'.$$

Definimos $G' = \overline{G}$.

Por lo tanto:

$$\forall S_i, (v_m \in S_i \wedge v_r \in S_i) \rightarrow \nexists e = (v_i, v_r) \in E.$$

Es decir, los vértices de cada subconjunto S_i no tienen aristas entre si. Sabemos que S tiene R subconjuntos, y definimos $R = K$.

Por lo tanto existen K subconjuntos de vértices que no tienen aristas entre sí. Pintando un subconjunto de cada color, se encuentra una solución para la instancia de KC. Es decir, se cumple que $\exists\ KC\ (G(V, E), K)$.

3.3.2. Demostración: CD es NP-Completo

Habiendo demostrado que SRC es un problema NP-Completo, ahora demostraremos que CD también lo es. Para ello, reduciremos SRC a CD. Es decir, se va a demostrar que:

$$SRC \leq_p CD$$

Antes, para facilitar la demostración, se enuncia el problema CD en su versión de decisión: *Dado un grafo, un número K, y un valor c; ¿es posible separar los vértices en a lo sumo K grupos, de forma tal que todo vértice pertenezca a un cluster, y que la distancia máxima entre cualquier par de vértices dentro de un cluster sea a lo sumo c?*

Haciendo un análisis de cada uno de estos problemas, se puede ver que sus objetivos son similares. Mientras el problema CD busca separar los vertices en grupos que estén a una distancia especifica, el problema SRC busca separarlos en grupos que formen un subgrafo completo; es decir, que estén a distancia uno. Esta similitud va a ser clave para la reducción que se desarrollara a continuación.

En primer lugar, vamos a especificar cada uno de los parámetros que recibe cada problema:

- Una instancia del problema SRC recibe un grafo, al que llamaremos G, que estará formado por los vértices V y las aristas E. Ademas, también recibe un numero R.
- Una instancia del problema CD recibe un grafo, al que llamaremos G', que estará formado por los vértices V' y las aristas E'. Ademas, recibe un numero numero positivo K, y un valor positivo c.

En este caso, el planteo de la reducción sera el siguiente: solo existe una solución para una instancia del problema SRC con el grafo $G(V, E)$ y el valor R, si existe solución para el problema CBD con un grafo idéntico a ese, y con los parámetros $K = R$ y $c = 1$. Es decir:

$$\exists\ SRC\ (G(V, E), R) \iff \exists\ CD\ (G'(V', E'), K, c)$$

donde:

- $G' = G$, y también sus vértices y aristas son los mismos, es decir $V' = V$ y $E' = E$.
- $K = R$, es decir, si buscamos R cliques vamos a transformarlo en buscar K clusters.
- $c = 1$, es decir, buscamos clusters con distancia máxima 1 entre vértices.

Habiendo planteado la transformación necesaria para hacer la reducción, se puede deducir que SRC **es un caso particular de** CD, donde la distancia máxima que se busca entre clusters es exactamente 1. La demostración que se presentara esta guiada por esta idea.

Nuevamente, demostraremos que la doble implicancia es correcta.

- **Primera implicancia:** $\exists\ SRC\ (G(V,E),R) \implies \exists\ CD\ (G'(V',E'),K,c)$

  $\exists\ SRC\ (G(V,E),R) \rightarrow\ \exists\ S = \{S_1,...,S_K\}$ donde S forma una partición del conjunto de vértices V. Ademas, cada subconjunto $S_i = \{v_1,...,v_m\}, v_i \in V$ cumple que :

  $$\forall_{i<m,j<m} : (v_i \in S_i, v_j \in S_i) \rightarrow \exists e = (v_i, v_j)$$

  Es decir, para cada subconjunto S_i, cada vértice que pertenece tiene arista a todos los demás vértices que también pertenecen; cada S_i es un clique.

  Definimos $G'(V',E') = G(V,E)$ y también $K = R$. Por lo tanto, los subconjuntos S_i forman una partición del conjunto de vértices V en K subconjuntos, donde cada uno de ellos esta a distancia máxima 1 (es decir, todos están unidos).

  Por esto ultimo, podemos afirmar que se cumple: $\exists\ CD\ (G'(V',E'),K,c)$.

- **Segunda implicancia:** $\exists\ CBD\ (G'(V',E'),K,c)\ \implies\ \exists\ SRC\ (G(V,E),R)$

  $\exists\ CBD\ (G'(V',E'),K,c)$ donde $K = R$ y $C = 1$. Entonces: $\exists S = \{S_1,...,S_k\}$ tal que S es una partición del conjunto de vértices V' en R subconjuntos. Ademas, sabemos que cada subconjunto S_i cumple:

  $$\forall S_i = \{v_1,...,v_m\} : (v_i \in S_i \land v_k \in S_i) \implies dist(v_i,v_j) <= 1$$

  .

  Definimos $G(V,E) = G'(V',E')$ y ademas $R = K$. Decir que los vértices de los subconjuntos S_i están a distancia máxima 1 es equivalente a decir que todos los pares de vértices pertenecientes a un subconjunto están unidos.

  Por esto ultimo, podemos afirmar que S también forma una partición del conjunto de vértices V que ademas cumple que todos los vértices de cada subconjunto están unidos. Por lo tanto, se cumple: $\exists\ SRC\ (G(V,E),R)$.

Habiendo demostrado que la reducción es correcta y que ademas el problema pertenece a *NP*, queda demostrado también que, en efecto, el problema de *Clustering por bajo diámetro* es un problema **NP-Completo**.

4. Soluciones Optimas

4.1. Solución mediante Backtracking

4.1.1. Algoritmos Backtracking

Cuando se tiene un **problema combinatorio**, en el que se deben explorar todas las soluciones posibles, una opción básica seria utilizar **Fuerza Bruta**, la cual se puede pensar como un *arbol de decicion* donde se prueban de forma implicita todas las combinaciones posibles. Si bien este metodo nos garantiza encontrar una solución optima, lo hace al alto costo de probar absolutamente todas al alternativas,muchas de las cuales podríamos descartar desde un principio ya que no nos llevaran a una solución optima.

Para justamente evitar recorrer combinaciones innecesarias,podemos aplicarle *podas* a nuestro *arbol de decision*. Una *poda* consiste en descartar tempranamente ramas del arbol que ya sabemos que no nos llevaran a una solucion valida o mejor(si es que tenemos una optima memoizada).

Esto anteriormente mencionado es a lo que llamamos **Backtracking**, el cual se basa en retroceder cuando se detecta que una solución parcial ya no nos llevara a una solución optima. Es decir, en lugar de seguir explorando todas las ramas, seguimos recorriendolas pero de manera mas *inteligente*, enfocándonos solamente en las ramas que potencialmente nos pueden llevar a una solución valida.

4.1.2. Algoritmo propuesto

```python
def minimizar_distancia_maxima(grafo:Grafo,k):
    distancias = obtener_distancias(grafo)

    sol_actual = [set() for _ in range(k)]
    dist_max_sol_actual =0

    mejor_sol , dist_mejor_sol = maxima_distancia(grafo,k,distancias)

    vertices = grafo.obtener_vertices()

    return minimizar_distancia_maxima_bt(grafo,k,distancias,vertices,0,sol_actual,
    dist_max_sol_actual,mejor_sol,dist_mejor_sol)


def minimizar_distancia_maxima_bt(grafo,k,distancias,vertices,indice,sol_actual,
    dist_max_sol_actual,mejor_sol,dist_mejor_sol):
    if indice == len(vertices):
        if dist_max_sol_actual<dist_mejor_sol:
            copia_sol_actual = []
            for cluster in sol_actual:
                copia_sol_actual.append(set(cluster))
            return copia_sol_actual, dist_max_sol_actual
        return mejor_sol , dist_mejor_sol

    v = vertices[indice]
    for i in range(k):  #Probamos agregar el vertice en el cluster i
        cluster_actual = sol_actual[i]

        #Poda -> si uno vacio con indice menor ya probe ahi
        if len(cluster_actual)==0:
            hay_cluster_vacio_menor= False
            for j in range(i):
                if len(sol_actual[j]) == 0:
                    hay_cluster_vacio_menor = True
                    break
            if hay_cluster_vacio_menor:
                continue

        condicion , maxima = validacion(cluster_actual,distancias,v,dist_mejor_sol)

        if condicion: #Poda -> si agregarlo a ese cluster empeora mi mejor solucion
            nueva_distancia_max = max(maxima,dist_max_sol_actual)

            if dist_max_sol_actual >= dist_mejor_sol:
                continue

            sol_actual[i].add(v)

            mejor_sol , dist_mejor_sol = minimizar_distancia_maxima_bt(grafo,k,
    distancias,vertices,indice+1,sol_actual,nueva_distancia_max,mejor_sol,
    dist_mejor_sol)

            sol_actual[i].remove(v)

    return mejor_sol , dist_mejor_sol

def validacion(cluster_actual,distancias,v,dist_mejor_sol):
    maxima = 0
    for w in cluster_actual:
        dist = distancias[v][w]
        if dist>=dist_mejor_sol:
            return False , None
        if dist>maxima:
            maxima = dist
    return True , maxima
```

La solución propuesta parte teniendo una *cota superior* la cual se obtiene de un algoritmo Greedy el cual no siempre es optimo (detallado en la **Sección 5.3.**). Este nos devuelve la solución

encontrada junto con la máxima distancia entre los clusters, ademas nos otorga una lista de los vértices ordenados decrecientemente basándose en su *centralidad* respecto a el resto de los vértices del grafo.

La idea de esta *cota superior* es partir de una posible solución para usarla como *poda* para las soluciones parciales que vaya componiendo el algoritmo de Backtracking, y así *podar* o no recorrer las combinaciones que nos lleven a tener una distancia maxima igual o peor a la propuesta por el algoritmo Greedy, ya que en este caso podemos quedarnos con la actual(la mejor hasta el momento, sea la propuesta por el algoritmo Greedy o alguna que lo haya superado, es decir, una que minimice la distancia).

Otra *poda* propuesta es para el caso en el que un vértice se este por agregar a un clúster vació, en este caso podría pasar que ya se haya intentado agregar a uno vació anteriormente y que se haya determinado que no era la mejor solución, por lo que seria redundante volver a probarlo con otro clúster vació sabiendo que no nos llevara a la solución optima.

4.2. Solución mediante Programación Lineal

4.2.1. Modelos en programación Lineal

La **Programación Lineal** es una técnica de diseño matemática que permite resolver problemas de optimización de un sistema de ecuaciones *lineales* en varias variables. Su objetivo es **maximizar o minimizar** una función objetivo especifica, respetando un conjunto de restricciones que limitan el dominio de las variables que involucra.

Estas restricciones son expresadas como ecuaciones e inecuaciones lineales, las cuales definen el conjunto de soluciones validas del problema en cuestión. Cuando se aplican estas restricciones sobre las variables, reducimos su dominio.

Los componentes fundamentales de un modelo de programacion dinamica son:

- **Variables**: pueden ser continuas, enteras o binarias. Estas son la representación de los *elementos* del problema a resolver.
- **Restricciones** : ecuaciones e inecuaciones *lineales* que definen restricciones sobre las variables definidas.
- **Función objetivo**: es una expresión lineal que determina lo que queremos maximizar o minimizar según el problema.
- **Algoritmo de resolucion** : mediante un algortimo que resuelve el modelo lineal (metodo Simplex) obtenemos la solución optima del modelo.

Para implementarlo y resolver el modelo que propondremos en Python, utilizaremos la libreria *PuLP*.

4.2.2. Modelo propuesto

A continuación se propone un posible modelo de programación lineal que permite resolver el problema. Para facilitar la lectura del mismo, consideraremos:

- **Constantes**:
  - $D_{i,j} = D_{j,i}$: 'Distancia entre el vértice i y el vértice j' .
  - V : 'Cantidad de vértices' .

- K : 'Cantidad de clusters' . Con $k \in [0, \ ... \ , |K-1|]$.
- i, j : 'Indices de vértices en el grafo'. Para ambos se cumple $0 < i < |V-1|$, $0 < j < |V-1|$.

- **Variables**:
  - $X_{i,k}$: 'El vértice i pertenece al cluster k' → Variable de tipo *binaria*, es decir $X_{i,k} \in \{0, 1\}$.
  - D_k : 'Distancia máxima entre dos vertices que pertenecen al clúster k"→ Variable de tipo *entera*, y ademas $D_k \in \mathbb{N}_0$.
  - $Y_{i,j}$: 'El vertice i y el vertice j pertenecen al mismo cluster' → Variable de tipo *binaria*, es decir $Y_{i,j} \in \{0, 1\}$.
  - D_{max} : 'Distancia maxima entre dos vertices de un mismo cluster' → Variable de tipo *entera*, y ademas $D_{max} \in \mathbb{N}_0$.
- **Restricciones** :
  - $\forall i : \sum_k X_{i,k} \leq 1$ → Cada vertice estara a lo sumo en un cluster.
  - $\forall k, \forall i, j : Y_{i,j} \geq X_{i,k} + X_{j,k} - 1$ → Si las variables de ambos vertices para ese cluster valen 1 (ambos estan en el mismo cluster), $Y_{i,j}$ valdra como minimo 1.
  - $\forall k, \forall i, j : D_k \geq D_{i,j} * Y_{i,j}$ → La distancia maxima dentro del cluster k (representada por la variable D_k) debe ser como minimo la distancia maxima entre cualquier par de vertices que pertenezca a este.
  - $\forall k : D_{max} \geq D_k$ → Es decir, la variable D_{max} valdra como minimo la maxima distancia entre dos vertices que esten en un mismo cluster.
- **Función objetivo**:
  - $\min(D_{max})$ → Se busca minimizar la distancia máxima.

El modelo propuesto anteriormente tiene la siguiente cantidad de restricciones:

- 1^{er} item : V restricciones, una por cada vertice.
- 2^{do} item : 1 restricción.
- 3^{er} item : $K \cdot V^2$ restricciones.
- 4^{to} item : $K \cdot V^2$ restricciones.
- 5^{to} item : K restricciones, cada una asociada a la variable D_k de cada cluster.

Por lo tanto, el modelo tiene un total de $V + 2K \cdot V^2 + 2$ restricciones.

5. Soluciones Aproximadas

Los **Algoritmos de Aproximación** permiten encontrar soluciones que garantizan ser cercanas a la optima y estan diseñados para ejecutarse en tiempo polinomial. Son especialmente utiles para cuando obtener la solucion optima a un problema no se puede conseguir de manera polinomial, es decir, cuando resolver un problema de forma exacta requiere un tiempo exponencial. En estos casos, se implementan estos algoritmos que aseguran una solución aproximable(la cual es demostrable).

Entre las metodologias mas utilizadas para los **Algoritmos de Aproximacion** se encuentran los **Algoritmos Greedy**, los cuales utilizaremos para conseguir una solución aproximada del problema propuesto.

5.1. Algoritmos Greedy

Un algoritmo **Greedy**, se basa en ir construyendo gradualmente la solución(es decir, "paso a paso"), en cada paso se toma una decisión siguiendo un mismo criterio.

Este criterio permite que la decisión que se toma sea siempre la mejor, es decir, la mas optima. El criterio usado solo contempla el *estado local*.

Definimos *estado local* al estado en el que se encuentra nuestra solución parcial, sin tener en cuenta que decisiones se tomaron anteriormente.

Ademas, llamaremos a una decisión *optimo local* cuando dado el *estado local*, la decisión sea la mejor que se pueda tomar.

La iteración de *óptimos locales* concluye en la obtención del *optimo general*.

5.2. Algoritmo Louvain

El algoritmo de **Louvain** es un método eficiente para detectar comunidades o clusters en un grafo. Un *cluster* es un grupo de nodos que están más densamente conectados entre sí que con el resto del grafo.

Este algoritmo busca maximizar la **modularidad**, que es una métrica que mide la densidad de enlaces dentro de las comunidades comparado con la densidad esperada si los enlaces fueran aleatorios, manteniendo el grado de cada nodo.

El objetivo de este algoritmo se trata de *encontrar una partición de nodos que maximice esta modularidad*.

5.2.1. Algoritmo

El algoritmo de Louvain tiene dos fases que se repiten iterativamente:

Asignación local de nodos a comunidades (optimización local):

- Inicialmente, cada nodo está en su propia comunidad, por lo que tenemos tantas comunidades como cantidad de nodos.
- Para cada nodo, se evalúa el cambio en modularidad si se mueve a la comunidad de uno de sus vecinos.
- Se mueve el nodo a la comunidad que produzca la mayor ganancia en modularidad, siempre que esta ganancia sea positiva. Caso contrario, se deja donde estaba.
- Este proceso se repite para todos los nodos hasta que no se pueden mejorar más las modularidades localmente

Construcción de un nuevo grafo (agregación):

- Se construye un nuevo grafo donde cada comunidad detectada en la fase anterior se convierte en un solo nodo.
- El peso de la arista entre dos nuevos nodos es la suma de pesos de las aristas entre las comunidades originales.
- Este nuevo grafo es más pequeño y representa la estructura comunitaria encontrada.

El algoritmo finaliza una vez que la modularidad no mejora mas o que el grafo no cambie.

5.2.2. Código de Louvain

El siguiente código inicializa el algoritmo con un grafo dado, donde se lo utiliza para obtener la clase de Louvain y ejecutar su algoritmo, del cual obtendremos las comunidades generadas y se podrá calcular la distancia máxima dentro de las mismas.

```python
# Agrupa nodos en comunidades con algoritmo de Louvain
def obtener_comunidades(grafo):
    louvain = Louvain(grafo)
    louvain.ejecutar()
    return louvain.comunidades.items()

# Calcula la distancia maxima dentro de los cluster seleccionados
def calcular_maxima_distancia(grafo, asignaciones):
    distancias = obtener_distancias(grafo)

    comunidades_dict = {}
    for nodo, comunidad in asignaciones:
        comunidades_dict.setdefault(comunidad, set()).add(nodo)

    max_distancias = []

    for nodos in comunidades_dict.values():
        max_dist = 0
        for u in nodos:
            if u not in distancias:
                continue
            for v in nodos:
                if v == u:
                    continue
                if v in distancias[u]:
                    max_dist = max(max_dist, distancias[u][v])
        max_distancias.append(max_dist)

    return max(max_distancias) if max_distancias else None

# Procesamiento principal del algoritmo
def procesar_grafo(nombre_archivo):
    grafo = crear_grafo_desde_archivo(nombre_archivo)
    comunidades = obtener_comunidades(grafo)
    imprimir_comunidades(comunidades)

    distancia_max = calcular_maxima_distancia(grafo, comunidades)
    print(f"Maxima distancia dentro del cluster: {distancia_max}")
```

Para realizar las operaciones del algoritmo de Louvain, se utilizo una clase del mismo, donde se realizan las operaciones particulares del algoritmo.

```python
class Louvain:
    def __init__(self, grafo):
        self.grafo = grafo
        self.comunidades = {v: v for v in self.grafo.obtener_vertices()}
        self.peso_total_aristas = self.calcular_total_peso()
        self.grado_total_nodos = self.calcular_grados()

        self.suma_de_grados_nodos = {}
        self.suma_peso_aristas_internas = {}

```

```
        for v in self.grafo.obtener_vertices():
            c = self.comunidades[v]
            self.suma_de_grados_nodos[c] = self.suma_de_grados_nodos.get(c, 0) +
    self.grado_total_nodos[v]
            self.suma_peso_aristas_internas[c] = self.suma_peso_aristas_internas.
    get(c, 0) + self.grafo.vertices[v].get(v, 0)
```

En esta sección del código se inicializa al grafo, en donde cada nodo pertenece a una comunidad y se realizan cálculos varios para resolver el algoritmo mas adelante.

```

    def calcular_total_peso(self):
        total = 0
        for v in self.grafo.obtener_vertices():
            for w, peso in self.grafo.vertices[v].items():
                total += peso
        return total / 2

    def calcular_grados(self):
        grado = {}
        for v in self.grafo.obtener_vertices():
            grado[v] = sum(self.grafo.vertices[v].values())
        return grado

    def peso_aristas_hacia_comunidad(self, v, comunidad):
        suma = 0
        for w, peso in self.grafo.vertices[v].items():
            if self.comunidades[w] == comunidad:
                suma += peso
        return suma
```

Estas funciones representan cálculos básicos de Louvain, en las cuales se suman todos los pesos de las aristas del grafo, se calcula el grado total de cada nodo y por ultimo, para cada nodo se calcula cuanto pesa su conexión con todos los nodos de una comunidad dada.

```

    def modularidad(self):
        Q = 0
        for c in set(self.comunidades.values()):
            peso_aristas_c = self.suma_peso_aristas_internas.get(c, 0)
            suma_grados_c = self.suma_de_grados_nodos.get(c, 0)
            Q += (peso_aristas_c / (2*self.peso_total_aristas)) - (suma_grados_c /
    (2*self.peso_total_aristas))**2
        return Q
```

Se realiza un calculo de la modularidad actual sumando para cada comunidad.

```

    def mejorar_particion(self):
        mejora = True
        historial_particiones = set()

        while mejora:
            particion_actual = frozenset((v, c) for v, c in self.comunidades.items
    ())
            if particion_actual in historial_particiones:
                # Ciclo repetido
                break
            historial_particiones.add(particion_actual)

            mejora = False
            for v in self.grafo.obtener_vertices():
                comunidad_actual = self.comunidades[v]
                grado_total_v = self.grado_total_nodos[v]

                # Sacar al nodo v de su comunidad actual
                self.suma_de_grados_nodos[comunidad_actual] -= grado_total_v
                grado_total_v_en_comunidad = self.peso_aristas_hacia_comunidad(v,
    comunidad_actual)
                self.suma_peso_aristas_internas[comunidad_actual] -= 2 *
    grado_total_v_en_comunidad + self.grafo.vertices[v].get(v, 0)
```

```
22
23                  # Evaluar a que comunidad adyacente mover el nodo v
24                  comunidades_vecinas = set(self.comunidades[w] for w in self.grafo.
    adyacentes(v))
25                  comunidades_vecinas.add(comunidad_actual)
26
27                  # Calculo de cambio de modularidad
28                  mejor_delta = 0
29                  mejor_comunidad = comunidad_actual
30                  for c in comunidades_vecinas:
31                      if c == comunidad_actual:
32                          continue
33                      suma_pesos_entre_v_c = self.peso_aristas_hacia_comunidad(v, c)
34                      suma_grados_c = self.suma_de_grados_nodos.get(c, 0)
35
36                      # Funcion de modularidad
37                      delta_Q = (suma_pesos_entre_v_c - (grado_total_v *
    suma_grados_c) / (2 * self.peso_total_aristas)) / (2 * self.peso_total_aristas)
38
39                      if delta_Q > mejor_delta:
40                          mejor_delta = delta_Q
41                          mejor_comunidad = c
42
43                  # Decidir si mover el nodo v o dejarlo donde estaba
44                  if mejor_comunidad != comunidad_actual:
45                      self.comunidades[v] = mejor_comunidad
46                      mejora = True
47                      self.suma_de_grados_nodos[mejor_comunidad] = self.
    suma_de_grados_nodos.get(mejor_comunidad, 0) + grado_total_v
48                      suma_pesos_entre_v_c_mejor = self.peso_aristas_hacia_comunidad(
    v, mejor_comunidad)
49                      self.suma_peso_aristas_internas[mejor_comunidad] = self.
    suma_peso_aristas_internas.get(mejor_comunidad, 0) + 2 *
    suma_pesos_entre_v_c_mejor + self.grafo.vertices[v].get(v, 0)
50                  else:
51                      self.suma_de_grados_nodos[comunidad_actual] += grado_total_v
52                      self.suma_peso_aristas_internas[comunidad_actual] += 2 *
    grado_total_v_en_comunidad + self.grafo.vertices[v].get(v, 0)
```

Esta ultima función es la que realiza el trabajo mas pesado, ya que itera mientras haya mejoras posibles. Para cada nodo lo que hace es "sacarlo"temporalmente de su comunidad actual para evaluar moverlo a una comunidad vecina, calculando el cambio de modularidad que produciría. Si la modularidad mejora, mueve el nodo a esa comunidad; si empeora, lo devuelve a su comunidad original. Si en alguna iteración no encuentra un movimiento que mejora la modularidad, se detiene.

5.2.3. Porque es Greedy

El algoritmo de Louvain se considera un *algoritmo Greedy*, debido a que en cada paso realiza una *elección local* que maximiza inmediatamente la mejora de la modularidad, sin considerar posibles efectos futuros o globales más complejos.

De esta forma, en cada iteración, Louvain toma la decisión que parece optima en ese instante para ese nodo, esperando a que eso conduzca a una buena solución global.

Llevándolo al código, para cada nodo evalúa a que comunidad vecina moverlo para obtener el mayor aumento en modularidad. Si lo hace, mueve el nodo a la comunidad, sin revisar o planear movimientos combinados, ni considerar si otro movimiento o no mover a ese nodo podría obtener mejoras a largo plazo. Esto mismo lo realiza iterativamente de forma local hasta llegar a una mejora global de la modularidad.

Este algoritmo no es optimo, ya que la función de modularidad tiene muchos máximos locales, en donde cada decisión local en la cual mueve o no un nodo, puede hacer que no se llegue a la mejor solución global siempre.

5.3. Algoritmo Greedy propuesto

A continuación se enseñara un algoritmo de aproximación propuesto con metodología Greedy.

5.3.1. Algoritmo propuesto

```
def maxima_distancia(grafo:Grafo,k,distancias):
    vertices = grafo.obtener_vertices()
    cant_vertices = len(vertices)

    clusters = []
    dist_max = 0

    #si k>|v| -> OPT
    if k>=cant_vertices:
        for i in range(k):
            if i<cant_vertices:
                clusters.append({vertices[i]})
            else:
                clusters.append(set())
        return clusters , dist_max

    #Obtenemos los k vertices mas centrales del grafo
    mas_centrales = obtener_vertices_centrales_y_distancias(grafo,distancias)

    centros=set()

    #Mas centrales uno en cada cluster
    for i in range(k):
        clusters.append(set())
        clusters[i].add(mas_centrales[i])
        centros.add(mas_centrales[i])

    #Ahora asignamos al resto por cercania
    for v in grafo.obtener_vertices():
        if v in centros : continue

        indice_mejor_cluster = obtener_cluster_mas_cercano(mas_centrales,distancias
    ,k,v)
        clusters[indice_mejor_cluster].add(v)

    #Calculo la distancia maxima de cada cluster
    for i in range(k):
        cluster = clusters[i]
        for v in cluster:
            for w in cluster:
                dist_local = distancias[v][w]
                if dist_local>dist_max:
                    dist_max = dist_local

    return clusters , dist_max
```

El algoritmo propuesto no siempre es optimo, pero un caso en el que se sabe que siempre sera optimo es cuando $k > |v|$ siendo k la cantidad de clusters y $|v|$ la cantidad de vértices en el grafo. Cuando ocurre esto, el algoritmo simplemente asigna un vértice a cada cluster dando la solución optima.

En caso de que esto no se cumpla, el algoritmo obtiene una lista con los vértices ordenados por centralidad de manera descendente (es decir : [mas central , ... , menos central]). Luego asignara los k mas cercanos a un cluster distinto, lo que se busca con esto es minimizar la distancia máxima dentro de un mismo cluster.

Con esta técnica lograríamos a partir de los k vértices mas *centralizados* dentro del grafo cubrir la mayor cantidad de vértices. Lo que resta del algoritmo es recorrer el resto de los vértices y asignarlos al cluster que tenga el *vértice central* que mas minimice la distancia entre ambos. Y al final del algoritmo simplemente se calcula la distancia máxima entre todos los clusters.

5.3.2. Porque es Greedy

El algoritmo enunciado anteriormente cumple con ser un algoritmo Greedy ya que:

Selecciona los k vértices mas centrales como centro para cada cluster, esta decisión no asegura ser la optima pero es la mejor que puede tomar.

Luego de eso asigna el resto de los vértices al cluster con el centro mas cercano a cada uno en particular, considerando unicamente la distancia entre el vértice y el *mas central* de cada cluster.

Notar que una vez que un vértice fue asignado a un cluster ya no se vuelve a cambiar ni cuestionar esa asignación.

5.4. Comparación entre aproximaciones

Para realizar la comparación entre el algoritmo Greedy y el de Louvain, primero describiremos cual es el objetivo de los mismos:

- *Greedy* : El algoritmo Greedy busca formar k clusters que minimicen la distancia máxima interna dentro de cada cluster, es decir, intenta que el diámetro máximo de los clusters sea lo más pequeño posible. Este método no garantiza encontrar la división global óptima que minimice la distancia máxima.
- *Louvain* : Louvain es un algoritmo de optimización heurística para encontrar particiones que maximizan la modularidad, una medida que evalúa qué tan densas son las conexiones dentro de los clusters respecto a conexiones entre clusters. El algoritmo de Louvain no requiere conocer k de antemano, se enfoca en la modularidad,y no en minimizar los diámetros.

La complejidad computacional de estos algoritmos puede compararse en la practica, con grafos densos o grandes, el **Greedy** (aproximado a $O(V^2)$) puede volverse lento debido al cálculo y comparación exhaustiva de distancias; mientras que Louvain (aproximado a $O(m\ log\ n)$ (m = cantidad de aristas)) suele escalar mejor y es mucho más eficiente en grafos grandes.

Para realizar una comparación practica de estos algoritmos, realizaremos una aproximación empírica, la cual es una estimación práctica del factor de aproximación del algoritmo, donde probaremos datos reales para hacer el calculo de la cota empírica, evaluando el rendimiento de los mismos. Los calculos que se realizaran para la comparacion son:

$$\text{Cota Empírica} = \max_{1 \leq i \leq N} \frac{Algoritmo(i)}{OPT(i)}$$

$$\text{Aproximación Empírica Promedio} = \frac{1}{N} \sum_{i=1}^{N} \frac{Algoritmo(i)}{OPT(i)}$$

Instancia	Res. Optimo	Greedy	Louvain	Aprox. Greedy	Aprox. Louvain
Pocos Vertices	10	12	10	1.2	1
Muchos Vertices	50	102	101	2.04	2.02
Pocos Clusteres	50	101	50	2.02	1
Muchos Clusteres	3	23	12	7.6	4

Cuadro 1: Aproximación Empírica

A partir de estos resultados y casos en los que basamos los algoritmos, podemos ver que el algoritmo de Louvain mostró una cota empírica de valor *4* y una aproximación empírica de *2.005*, mientras que el algoritmo Greedy mostró una cota empírica de valor *7.6* y una aproximación empírica de *3.215*.

A partir de estos resultados, podemos observar que el algoritmo de Louvain presentó un mejor desempeño en promedio en relación con el valor óptimo de las instancias evaluadas. Su cota empírica de aproximación fue de 4, lo que indica que, en el peor caso observado, su solución fue a lo sumo 4 veces peor que la óptima.

Además, su aproximación empírica promedio fue de 2.005, lo que refleja un comportamiento generalmente cercano al óptimo en la mayoría de los casos.

En cambio, el algoritmo Greedy obtuvo una cota empírica más alta, de 7.6, lo que evidencia que en ciertas instancias su desempeño se aleja considerablemente del óptimo. Su aproximación empírica también fue mayor, de 3.215, lo que indica que, en promedio, sus soluciones fueron menos precisas que las de Louvain.

Estos resultados sugieren que Louvain ofrece una mejor aproximación general al problema de *clustering por bajo diámetro*, especialmente en escenarios variados.

6. Mediciones

6.1. Complejidad de los distintos algoritmos

Para el desarrollo de esta sección se define:

- V : " Cantidad de vértices en el grafo"
- E : " Cantidad de aristas en el grafo"
- k : " Cantidad de clusters"

Notar ademas, que se supone que el grafo esta implementado con un *diccionario de diccionarios*. Esto es importante para determinar la complejidad de las operaciones que se hagan con dicha estructura.

Para no ser repetitivos durante el desarrollo de las distintas complejidades, el costo de las siguientes primitivas del grafo son:

- $grafo.obtener\_vertices() \rightarrow O(V)$
- $grafo.adyacentes(vertice) \rightarrow O(V)$

6.1.1. Backtracking

El costo de este algoritmo se sabe que finalmente concluirá en **exponencial**.

Pero podemos analizar que ademas de esto, previo al algoritmo de Backtracking como tal, se realiza una llamada a $obtener\_distancias(grafo)$ la cual tiene una complejidad $O(V \cdot (V + E))$ (detallada mas adelante donde toma mas relevancia que aca que es acotada por algo exponencial).

```
def minimizar_distancia_maxima(grafo:Grafo,k):
    distancias = obtener_distancias(grafo)

    sol_actual = [set() for _ in range(k)]
    dist_max_sol_actual =0

    mejor_sol , dist_mejor_sol , mas_centrales = maxima_distancia(grafo,k,
    distancias)

    vertices = grafo.obtener_vertices()

    return minimizar_distancia_maxima_bt(grafo,k,distancias,vertices,0,sol_actual,
    dist_max_sol_actual,mejor_sol,dist_mejor_sol)
```

También se puede observar que se genera una lista de k sets, lo que concluye en $O(k)$ y luego una llamada a un **algoritmo Greedy**(el cual también es detallado mas adelante, mas precisamente la **Sección 6.1.4.** trata específicamente la complejidad de este algoritmo) con complejidad temporal $O(k + V2 + (VE))$.

La lista de vértices que sera pasada a la función que realizara el algoritmo de Backtracking, esta previamente ordenada, para esto utiliza un *sorted* el cual es $O(V)$,y a lo que retorna se le aplica un *list* que también es $O(V)$.

Esto es simplemente un repaso de lo que se hace previamente, pero se sabe que quedara acotado por la **complejidad exponencial** de la naturaleza de un algoritmo de Backtracking.

También podemos analizar la complejidad de alguna de las podas.

```
def validacion(cluster_actual,distancias,v,dist_mejor_sol):
    maxima = 0
    for w in cluster_actual:
        dist = distancias[v][w]
        if dist>=dist_mejor_sol:
            return False , None
        if dist>maxima:
            maxima = dist
    return True , maxima
```

La complejidad de esta es $O(V)$, ya que analiza la distancia del vértice pasado por parámetro contra el resto de los vértices pertenecientes al *cluster_actual*, lo cual, en el peor de los casos podría tratase de analizar la distancia contra $V - 1$ vértices.

```
        if len(cluster_actual)==0:
            hay_cluster_vacio_menor= False
            for j in range(i):
                if len(sol_actual[j]) == 0:
                    hay_cluster_vacio_menor = True
                    break
            if hay_cluster_vacio_menor:
                continue
```

En el fragmento anteriormente mostrado, la peor situación a la que nos podríamos enfrentar es estar en el cluster con mayor indice(es decir en el cluster con indice $= k - 1$), en este caso se podrían llegar a recorrer todos los anteriores, lo cual concluiría en $O(k)$.

6.1.2. Programación Lineal

La complejidad de este algoritmo esta fuertemente ligada a la cantidad de restricciones, las cuales fueron detalladas en la **Sección 4.2.2.** en donde se determino que son $V + 2K \cdot V^2 + 2$ restricciones.

Como se menciono anteriormente, para implementar el modelo de forma algorítmica si utilizo la librería *PuLP* de Python. La complejidad algoritmica de este es $O(V + 2K \cdot V^2 + 2)$ debido a que depende por completo de las restricciones.

6.1.3. Algoritmo Louvain

Se dice que Louvain tiene complejidad $O(m \log n)$, donde n es la cantidad de nodos y m la cantidad de aristas del grafo, porque combina un procesamiento lineal sobre el grafo en cada nivel (fase de mejora local) con un número logarítmico de niveles de compresión. Esta eficiencia lo hace muy adecuado para grafos grandes, a diferencia de métodos exactos como programación lineal o backtracking, que se vuelven ineficaces a gran escala.

Para demostrar esta estimación realizaremos el siguiente análisis:

Fase 1 - Optimización local de la modularidad:

- En esta fase, cada nodo se considera individualmente y se mueve al grupo de algún vecino si eso mejora la modularidad. Cada uno de estos movimientos pueden analizarse en tiempo proporcional al grado del nodo. Si se ejecuta varias pasadas sobre los nodos, el tiempo total es aproximadamente $O(m)$, donde m es el número de aristas.

Fase 2 - Compresión del grafo:

- Se construye un nuevo grafo donde cada comunidad identificada se convierte en un supernodo. Este tiene menos nodos que el original, y el proceso vuelve a comenzar sobre esta versión compactada.

Número de iteraciones (niveles de compresión):

- En cada nivel se reduce significativamente el número de nodos. Por lo que se estima que el número de niveles de compresión es logarítmico en n, es decir, $O(\log n)$.

Complejidad total:

- Como cada fase toma aproximadamente $O(m)$ y se ejecuta durante $O(\log n)$ niveles, la complejidad global es $O(m \log n)$.

En conclusión, se dice que Louvain tiene complejidad $O(m \log n)$ porque combina un procesamiento lineal sobre el grafo en cada nivel con un número logarítmico de niveles de compresión.

6.1.4. Greedy propuesto

Para invocar a la función que realizara el algoritmo propuesto, previamente se deben calcular las distancias entre cada par de vértices, para esto se llama a la funcion $obtener\_distancias(grafo)$.

```
def minimizar_distancia_maxima_greedy(grafo:Grafo,k):
    distancias = obtener_distancias(grafo)
    return maxima_distancia(grafo,k,distancias)
```

A continuación se detallara la complejidad temporal de dicha función.

```
def obtener_distancias(grafo:Grafo):
    distancias = {}

    for v in grafo.obtener_vertices():
        dist_v = bfs(grafo,v)
        distancias[v] = dist_v

    return distancias

def bfs(grafo:Grafo,vertice):
    visitados=set()
    cola = deque()
    dist={}

    cola.append(vertice)
    visitados.add(vertice)
    dist[vertice] = 0

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

    return dist
```

Como se puede observar, dicha función hace un *BFS* por cada vértice, se sabe que la complejidad de este algoritmo es $O(V + E)$ ya que, se visita cada vértice una vez y se visitan todas las aristas desde cada vértice. Por lo tanto, como esto se hace por cada vértice, obtener las distancias de cada vértice hacia el resto cuesta $O(V \cdot (V + E) = O(V^2 + V \cdot E)$.

Ahora si analizaremos la complejidad de la función $maxima\_distancia(grafo, k, distacias)$ como tal.

```
def maxima_distancia(grafo:Grafo,k,distancias):
    vertices = grafo.obtener_vertices()
    cant_vertices = len(vertices)

    clusters = []
    dist_max = 0

    #si k>=|v| -> OPT
    if k>=cant_vertices:
        for i in range(k):
            if i<cant_vertices:
                clusters.append({vertices[i]})
            else:
                clusters.append(set())
        return clusters , dist_max
```

Para el caso en el que la cantidad de clusters sea mayor o igual a la cantidad total de vertices, el algoritmo unicamente asignara un vertice a cada cluster, por lo que el costo de de ese fragmento es $O(k)$.

Luego analizamos la llamada a la función $obtener\_vertices\_centrales\_y\_distancias(grafo, distancias)$

```
#Obtenemos los k vertices mas centrales del grafo
mas_centrales = obtener_vertices_centrales_y_distancias(grafo,distancias)
```

```
def obtener_vertices_centrales_y_distancias(grafo:Grafo,distancias):
    centralidad = {}
    for v in grafo:
        centralidad[v]=0

    for v in grafo:
        suma_centralidad = 0
        for w in grafo:
            if v != w:
                suma_centralidad+= distancias[v][w]
        centralidad[v]=suma_centralidad

    ordenados = sorted(centralidad,key=lambda x: centralidad[x])

    return ordenados
```

Inicializamos un diccionario poniéndole valor de centralidad a cada vértice en 0, esto cuesta $O(V)$.

Luego, por cada vertice vemos el resto de los vertices, lo cual tiene una complejidad temporal $O(V^2)$.

Finalmente se ordenan a los vertices por centralidad, el costo de hacer este *sorted* es $O(V\ log\ V)$.

Por lo que, concluimos que la complejidad temporal total de esta función es $O(V+V^2+V\ log\ V)$, luego podemos acotar superiormente a V y a $V\ log\ V$ con V^2,por lo que, finalmente queda $O(V^2)$.

Una vez obtenidos los vértices con *mayor centralidad* en la función anterior, hay que asignar uno en cada cluster.

```
centros=set()

    for i in range(k):
        clusters.append(set())
        clusters[i].add(mas_centrales[i])
        centros.add(mas_centrales[i])
```

Este ciclo es $O(k)$ ya que recorremos los k clusters y realizamos la operación anteriormente propuesta que es constante.

Luego, para asignar el resto de los vertices en el resto de los clusters, se invoca a la función $obtener\_cluster\_mas\_cercano(mas\_centrales, distancias, k, v)$ por cada vertice.

```
    for v in grafo.obtener_vertices():
        if v in centros : continue

        indice_mejor_cluster = obtener_cluster_mas_cercano(mas_centrales,distancias
    ,k,v)
        clusters[indice_mejor_cluster].add(v)
```

```
def obtener_cluster_mas_cercano(mas_centrales,distancias,k,v):
    min_dist = float("inf")
    indice_mejor_cluster = None
    for i in range(k):
        c = mas_centrales[i]
        dist = distancias[c][v]
        if dist<min_dist:
            min_dist = dist
            indice_mejor_cluster = i

    return indice_mejor_cluster
```

Dentro de esta función, se recorre cada cluster, lo cual es $O(k)$ y por cada uno de estos se analiza la distancia entre el vértice y el *vértice central* que fue asignado previamente para este cluster. Luego el algoritmo devuelve el indice del cluster con el que el vértice tenga menor distancia.

Esto se realiza por cada vértice, por lo que es $O(V \cdot k)$.

Finalmente, para poder retornar cual fue la mayor distancia entre los vértices de los distintos clusters:

```
    for i in range(k):
        cluster = clusters[i]
        for v in cluster:
            for w in cluster:
                dist_local = distancias[v][w]
                if dist_local>dist_max:
                    dist_max = dist_local
```

Por cada cluster se analiza la distancia de cada vértice hacia todo el resto de vértices que pertenecen a ese mismo cluster. Esto es un poco mas complejo de analizar, ya que no podemos saber con exactitud la distribución entre clusters.

Consideramos como el peor caso tener, por ejemplo, dos clusters ($k = 2$) y que en un cluster se asigne un solo vértice y en el otro el resto. Entonces la complejidad de este fragmente de codigo para ese cluster que contiene $V - 1$ vertices seria, $O(k \cdot V \cdot V) = O(k \cdot V^2)$.

Complejidad total del algoritmo:

- Si $k >= V$:
  - Obtener distancias : $O(V^2 + (V \cdot E))$
  - Obtener vértices : $O(V)$
  - Asignar cada vértice a un cluster : $O(k)$

  Por lo que la complejidad temporal total del algoritmo seria : $O((V+k)+V^2+(V \cdot E))$, y como sabemos que $k >= V$ podemos acotar superiormente a V y por lo tanto, $O(2k) = O(k)$. Luego quedaría $O(k + V^2 + (V \cdot E))$.
- Si $k < V$:

- Obtener distancias : $O(V^2 + (V \cdot E))$
- Obtener una lista con los vértices mas *centrales* : $O(V^2)$
- Asignar un vértice *mas central* a cada cluster : $O(k)$
- Asignar el resto de los vértices en el cluster con el *vértice central* con menor distancia entre ambos : $O(V \cdot k)$.
- Hallar la maxima distancia entre los clusters : $O(k \cdot V^2)$

  Por lo tanto, la complejidad final del algoritmo es, $O(V^2+(V\cdot E)+V^2+k+(V\cdot k)+(V^2\cdot k)$ el cual acotando lo necesario queda en $O((V \cdot E) + (V^2 \cdot k))$. La mayoria de los terminos fueron acotados superiormente por $V^2 + k$.

6.2. Gráficos de comportamiento temporal

A continuación se hará un análisis acompañado de gráficos que mostraran el comportamiento de los distintos algoritmos desarrollados durante el trabajo. Se indagara también en la comparación entre algunos frente a distintos escenarios propuestos.

6.2.1. Backtracking

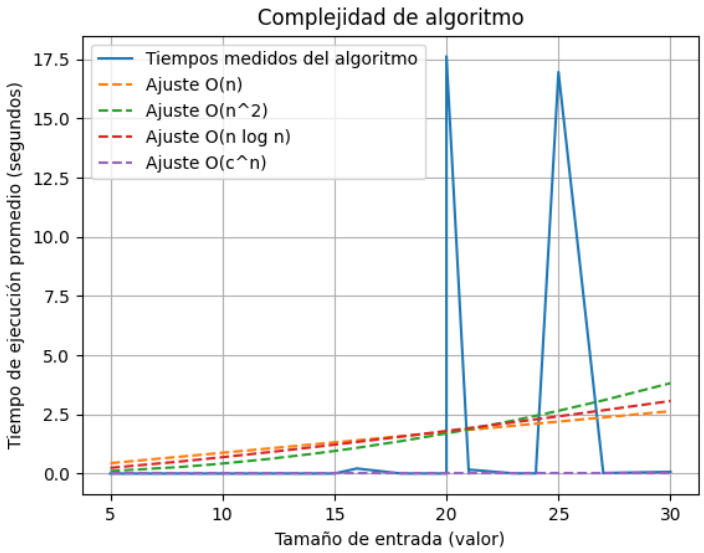

Figura 1: Complejidad Backtracking

En el gráfico se puede apreciar como el algoritmo tiene un comportamiento exponencial pero, al tener valores de entrada bajos, la gráfica " crece lentamente" al principio y recién explota en valores mayores de entrada.

El algoritmo de backtracking depende fuertemente de la calidad de la cota superior inicial para podar el espacio de búsqueda. En este caso, esa cota suele estar dada por la solución obtenida con un algoritmo Greedy.

Cuando el algoritmo Greedy encuentra una solución óptima o muy cercana al óptimo, esa solución permite podar gran parte del árbol de decisiones desde el inicio, lo que reduce drásticamente el tiempo de ejecución. Sin embargo, si el algoritmo Greedy no acierta con una buena solución, el algoritmo de backtracking no puede realizar tantas podas tempranas y debe explorar una parte mucho más grande del espacio de soluciones. Esto causa una explosión y tiempos de ejecución mucho mayores en algunos casos puntuales, generando los picos observados en el gráfico.

6.2.2. Programación Lineal

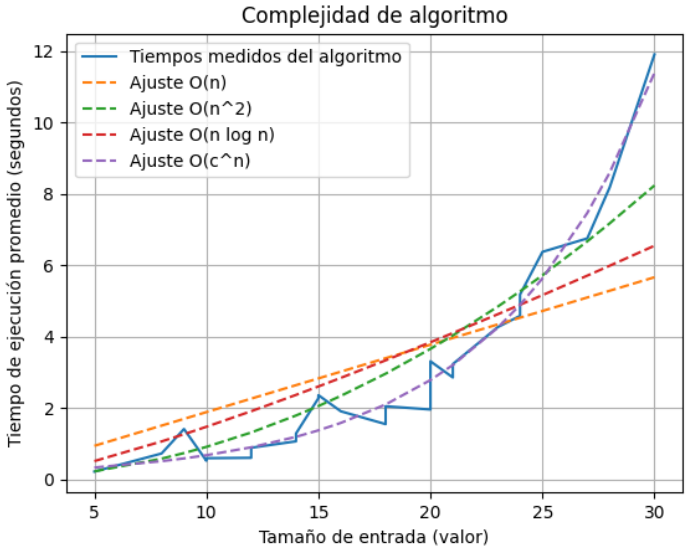

Figura 2: Complejidad Programación Lineal

Aunque el modelo planteado se basa en programación lineal, el uso de variables binarias y restricciones enteras lo convierte en un problema de programación lineal entera.

En consecuencia, el número de combinaciones posibles crece exponencialmente con el tamaño del grafo, lo que genera un comportamiento de tiempo de ejecución exponencial en la práctica.

6.2.3. Comportamiento Backtracking vs Programación Lineal

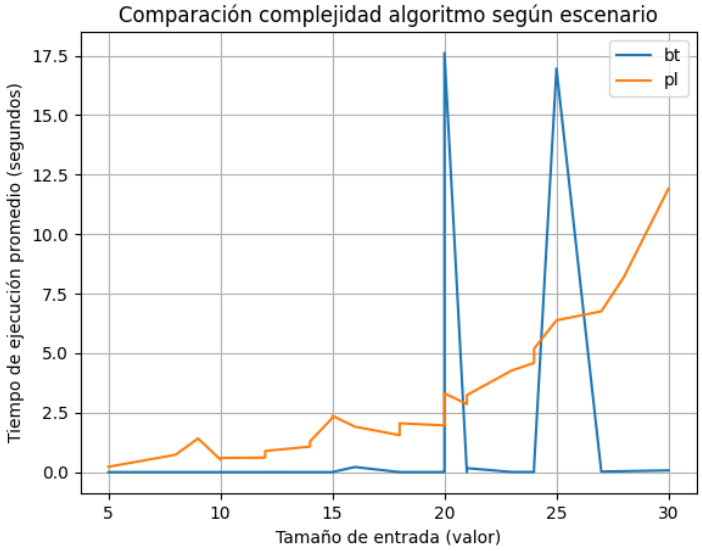

Figura 3: Backtracking vs Programación Lineal

El algoritmo de backtracking muestra una gran variabilidad dependiendo del caso, con tiempos que oscilan entre extremadamente bajos y muy altos. Al contrario, la programación lineal tiene una complejidad más estable, aunque con crecimiento exponencial suave y sin mejoras espectaculares en casos favorables.

Esta diferencia se explica porque el backtracking depende críticamente de la calidad del Greedy para poder podar, mientras que la programación lineal resuelve el problema completo sin atajos, independientemente del caso.

6.2.4. Algoritmo Louvain

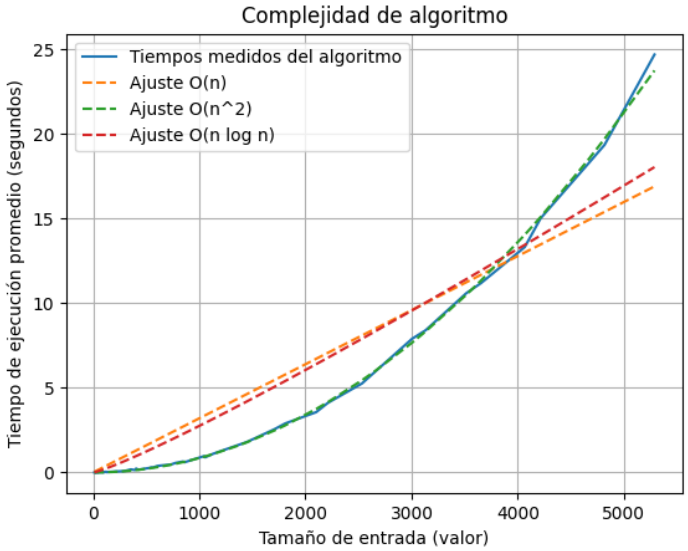

Figura 4: Complejidad Louvain

Si bien el algoritmo de Louvain no tiene una cota teórica clara en el peor caso, en la práctica suele mostrar un comportamiento próximo a $O(n^2)$. Esto se debe a que el proceso de optimización implica múltiples pasadas sobre los nodos del grafo, y cada iteración evalúa el beneficio de mover cada nodo a diferentes comunidades.

En grafos densos o de tamaño medio a grande, este procedimiento genera un costo computacional que crece cuadráticamente con la cantidad de nodos, especialmente cuando el número de comunidades posibles también crece. Por eso, aunque Louvain es eficiente y escalable en muchos casos reales, su rendimiento empírico puede alinearse con una curva cuadrática, como se ve reflejado en la imagen.

6.2.5. Greedy propuesto

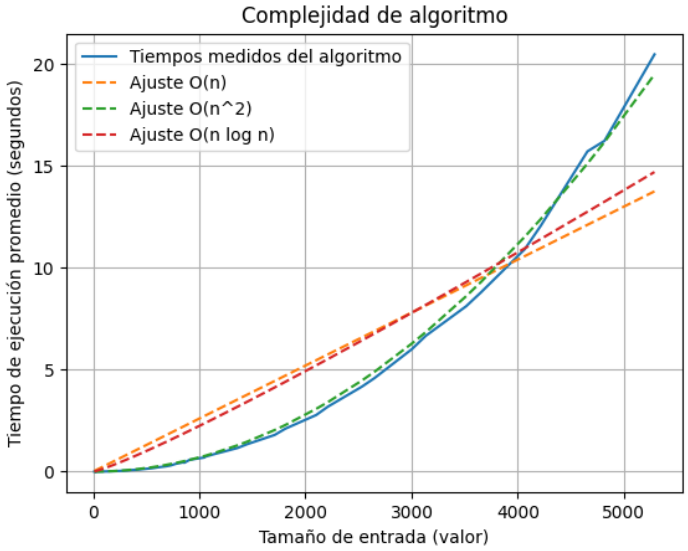

Figura 5: Complejidad Greedy

Como se puede ver en el gráfico el algoritmo tiene un comportamiento cuadrático, esto lo habíamos adelantado cuando explicamos la complejidad de este. También se puede notar como el k que también estaba ligado a la complejidad cuadrática pasa desapercibido, es decir, queda totalmente acotado por V^2, esto mismo sucede también con el otro termino en la notación O de la complejidad.

6.2.6. Comportamiento algoritmo Louvain vs Greedy

En esta sección se compararan los dos algoritmos de aproximación que se trataron a lo largo del informe.

Analizamos y comparamos el comportamiento de ambos frente a un pedido con pocos clusters y en donde se sabe que la distancia máxima optima dentro de cada cluster es un numero grande.

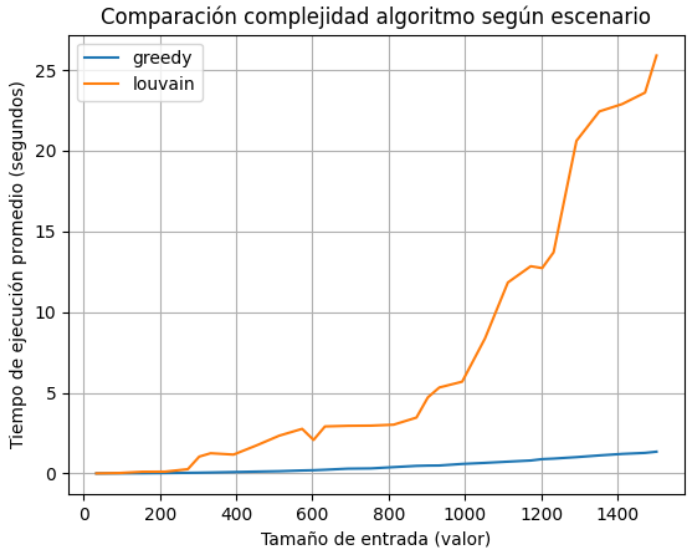

Figura 6: Pocos Clusters y Mucha Distancia Maxima

Como se puede notar en el gráfico, aunque anteriormente afirmamos que el comportamiento de nuestro algoritmo Greedy tenia un comportamiento cuadrático, frente a compararlo con el algoritmo de Louvain queda *"planchado"*, mientras el algoritmo de Louvain crece a pasos agigantados. Sin tratar temas de mejor o peor aproximación, en cuanto tiempos de ejecución se puede notar que nuestro Greedy es mucho mas eficiente en este escenario.

Veamos ahora que sucede si crece la cantidad de clusters que podemos utilizar para hacer las asignaciones mientras la distancia máxima optima es mucho mas chica. Es decir, es el caso antagónico al anterior.

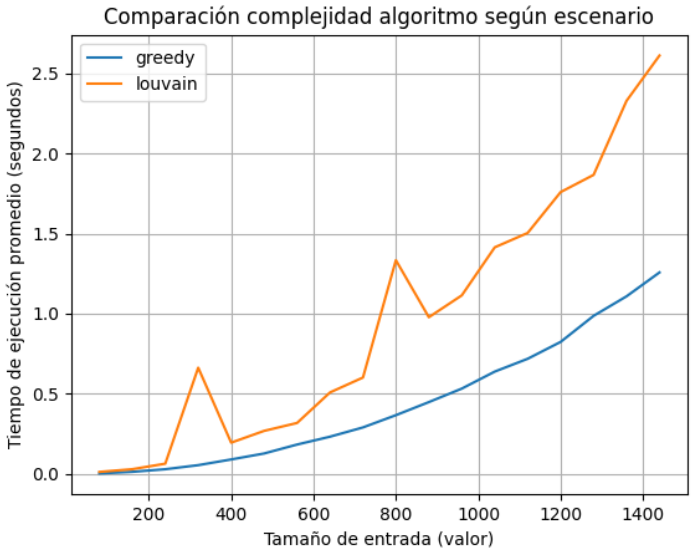

Figura 7: Varios Clusters y Poca Distancia Maxima

Nuevamente, notamos como nuestro algoritmo muestra tener un mejor comportamiento en cuento a complejidad temporal en frente al algoritmo de Louvain. De igual manera podemos observar que en este escenario hay una brecha menor entre ambos.

7. Conclusión

Durante el desarrollo del trabajo, indagamos sobre distintas técnicas para resolver un mismo problema planteado.

Comenzamos abordando una versión del problema formulada como un *"problema de decisión"*. Se realizo un breve análisis sobre la dificultad computacional de los problemas y demostramos que el problema en cuestión era pertenece a la clase de los *NP-Completo*. Para ello, analizamos y utilizamos dos problemas conocidos que también pertenecen a *NP-Completo*.

Luego, exploramos dos métodos para resolver el problema original. Observamos que, si bien ambas técnicas (Backtracking y Programación Lineal) proporcionaban soluciones exactas y optimas, presentaban tiempos de ejecución muy elevados, lo cual se vuelve un problema para instancias grandes del problema.

Frente a esto, se plantea aplicar algoritmos de aproximación. Aunque como se explico en el informe, estos no garantizan encontrar la solución optima, logran resultados muy cercanos y que, a diferencia de los métodos anteriormente mencionados, tienen tiempo de ejecución polinomiales, lo que los hace considerablemente mas eficientes. Ademas, en muchos casos, las soluciones obtenidas eran optimas o muy cercanas a las optimas.

Finalmente, se mostró gráficamente los resultados obtenidos, destacando las diferencias en los tiempos de ejecución en distintos enfoques o escenarios.