Teoría de Algoritmos

(TB024) Curso Buchwald - Genender

Trabajo Práctico 3
Comunidades NP-Completas

16 de junio de 2025

Conde Cardó,	Colombo Farre,	Willson,
Lucas Ariel	Iván Joel	Marina
112201	111671	103532

Índice

1. Introducción

Luego de los grandes resultados obtenidos en los trabajos anteriores, fuimos nuevamente ascendidos. Ahora somos la mano derecha del Gringo (que dejó de decirnos cada 3 palabras cosas de desayunar con peces).

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

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

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

1.1. Consigna

Para los primeros dos puntos considerar 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).

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

- Demostrar que el Problema de Clustering por bajo diámetro se encuentra en NP.
- Demostrar que el Problema de Clustering por bajo diámetro es, en efecto, un problema NP-Completo. Si se hace una reducción involucrando un problema no visto en clase, agregar una (al menos resumida) demostración que dicho problema es NP-Completo.
- Escribir un algoritmo que, por backtracking, obtenga la solución óptima al problema (valga la redundancia) en la versión de optimización: Dado un grafo no dirigido y no pesado, y un valor k, determinar los k clusters para que la distancia máxima de cada cluster sea mínima. Para esto, considerar minimizar el máximo de las distancias máximas (es decir, de las distancias máximas de cada cluster, nos quedamos con la mayor, y ese valor es el que queremos minimizar).

  Generar sets de datos para corroborar su correctitud, así como tomar mediciones de tiempos.
- Escribir un modelo de programación lineal que resuelva el problema de forma óptima. Recordar que el cálculo de las distancias mínimas se puede hacer previamente, y no ser parte del modelo de programación lineal (problema muy sencillo de resolver con un algoritmo BFS, pero no así con programación lineal).

  Implementar dicho modelo, y ejecutarlo para los mismos sets de datos para corroborar su correctitud. Tomar mediciones de tiempos y compararlas con las del algoritmo que implementa Backtracking.
- Ya pudiendo mostrar que no es una buena idea trabajar con estas métricas para obtener las comunidades de una organización tan grande, la agente T nos recomendó analizar la posibilidad de no clusterizar por bajo diámetro, sino maximizando la Modularidad.

¿La qué?

La modularización es una métrica para definir la densidad de aristas dentro de una comunidad. En una comunidad real, hay un alto coeficiente de clustering. Básicamente, hay muchos triángulos; es decir, amigos en común. Esto se manifiesta en tener que una comunidad tiene necesariamente muchas aristas dentro de la misma, y al mismo tiempo pocas *hacia afuera*. Podemos definir la modularidad como:

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

Donde:

- $peso(v_i, v_j)$: el peso de la arista entre i y j (0 si no están unidos, y en nuestro caso 1 sí si lo están).

- k_i: es la suma de las aristas del vértice i (en nuestro caso, su grado).

- $2m$: es la suma de todos los pesos de las aristas.

- c_i: es la comunidad del vértice i.

- δ: función delta de Kronecker, que es básicamente 1 si ambas comunidades son iguales, 0 si son diferentes.

¿El problema? Maximizar la modularización es también un problema NP-Completo. ¿Lo bueno? Es conocido un algoritmo greedy que funciona muy bien para esto: El Algoritmo de Louvain. Obviamos transcribir la descripción del algoritmo, que pueden leer allí, pero la agente T logró obtener una grabación de una clase de la facultad de ingeniería donde explicaban este algoritmo. Por supuesto, si les interesa el tema pueden revisar el video entero. Dejamos también un link a las diapositivas de dicha clase.

Implementar el algoritmo de Louvain para obtener una separación en K clusters. Realizar mediciones para determinar una cota empírica de aproximación al utilizar dicho algoritmo para aproximar al problema de Clustering por bajo diámetro. Realizar esto con datos generados por ustedes, incluyendo potencialmente set de datos de volúmenes inmanejable para los algoritmos antes implementados. Recomendamos leer la explicación de la complejidad del algoritmo, la cual no es sencilla (y, por lo tanto, no la pedimos aquí tampoco).

- **Opcional**: Implementar alguna otra aproximación (o algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto anterior. Indicar y justificar su complejidad, en lo posible. No es obligatorio hacer este punto para aprobar el trabajo práctico (pero si resta un punto no hacerlo).

- Agregar cualquier conclusión que parezca relevante.

2. Análisis del problema

En el presente trabajo práctico se plantea el problema de *Clustering por bajo diámetro*. Dado un grafo no dirigido y no pesado y un valor k, determinar los k clústeres tal que la distancia máxima de cada grupo sea mínima.

Entonces, el desafío en este será hallar una combinación de vértices que, agrupados en como mucho k vértices, minimice la máxima distancia de las distancias máximas de cada clúster, es decir, que minimice el diámetro máximo. El diámetro de un clúster se define como el camino más corto sobre el grafo original entre los vértices más alejados del grupo en cuestión. Dado que el grafo es no pesado, la distancia se medirá en función de la cantidad de aristas que atraviese.

Además, una condición necesaria es que los grupos sean disjuntos por lo que habrá que tener especial cuidado de no repetir elementos y no olvidar ninguno, ya que todos deben estar asignados a un clúster.

Sabiendo que presuntamente el problema es NP-Completo, se probarán y compararán cuatro algoritmos con enfoques diferentes: Backtracking, Programación lineal, una implementación del *Algoritmo de Louvain* y un algoritmo greedy ideado por nosotros. Esto con el fin de encontrar el equilibrio ideal entre optimalidad y bajos tiempos de ejecución.

Los dos primeros algoritmos permitirán encontrar soluciones óptimas, aunque probablemente con un alto costo computacional por su conocida complejidad exponencial. Los algoritmos restantes nos permitirán obtener aproximaciones, en el caso de Louvain, basadas en la optimización de la modularidad y, en el caso de nuestro algoritmo greedy, en el bajo diámetro, tal como lo hacen las dos primeras soluciones. Hipotéticamente, estos nos brindarán soluciones velozmente que, aún sin ser óptimas, nos servirán como punto de partida para otros algoritmos, como los dos primeros.

En este sentido, otro gran desafío del trabajo será encontrar optimizaciones, ya sea mediante podas en Backtracking o minimizando la cantidad de restricciones en Programación Lineal, para que la ejecución sea lo más rápida posible mientras se garantice la optimalidad del resultado en dichos algoritmos.

3. Clases de complejidad

Para esta parte del trabajo, se evaluará el problema en su versión de decisión: dado un grafo no dirigido y no pesado, y dos valores k y C, ¿es posible separar los vértices en *a lo sumo* k clústeres disjuntos, de tal forma que todo vértice pertenezca a un clúster, y que la distancia máxima dentro de cada clúster sea *a lo sumo* C?

Como enuncia la consigna, este problema es, aparentemente, NP-Completo. Para poder verificarlo se probará su pertenencia a NP, el conjunto de problemas que pueden ser resueltos en tiempo polinómico por una máquina de *Turing* no determinista. Asimismo, dado que, por definición, todo problema que esté en NP puede reducirse polinómicamente a un problema NP-Completo y que toda reducción de un problema NP-Completo a otro implica que ese otro también sea NP-Completo, se planteará una reducción de un problema que sabemos que es NP-Completo, al nuestro.

De esta forma, podremos demostrar si el problema de Clusterización por Bajo Diámetro se trata efecticamente de un problema perteneciente a esta categorización o no.

3.1. Inclusión en NP

Para poder decir que el problema pertenece a NP, es necesario hallar un verificador eficiente que, dado un grafo no dirigido y no pesado ($G = (V, E)$), un número entero que represente la cantidad máxima de clústeres (k), un valor que indique el diámetro máximo válido (C) y una supuesta solución a esa instancia del problema, permita corroborar en tiempo polinomial que efectivamente esa solución es válida.

En el caso de la versión de decisión del problema de *Clustering por bajo diámetro*, este certificador debe probar que:

- El tamaño de la solución sea a lo sumo igual a k.
- Todos los elementos de la solución sean vértices del grafo.
- Todos los elementos del grafo estén presentes una única vez.
- El diámetro de cada clúster sea a lo sumo C.

```python
def validador_solucion_clustering(grafo, k, c, solucion):
    if len(solucion) > k:
        return False

    distancias = calcular_distancias(grafo)

    visitados = set()
    for cluster in solucion:
        for v in cluster:
            if v not in grafo or v in visitados:
                return False
            visitados.add(v)
            for a in cluster:
                if a not in grafo or distancias[v][a] > c:
                    return False
    return len(visitados) == len(grafo)
```

El algoritmo se compone de:

- Operaciones constantes, principalmente de comparación, tanto en el inicio como en el fin del algoritmo.
- El cálculo de distancias, basado en BFS, cuya complejidad temporal resulta $\mathcal{O}(V + E)$, con V igual a la cantidad de vértices y E a la cantidad de aristas, que, como se realiza para cada vértice del grafo presenta una complejidad total de $\mathcal{O}(V \cdot (V + E))$. Omitimos el código de la función debido a su extrema sencillez. El mismo puede verse deternidamente en el repositorio del proyecto.

- Un ciclo `for` que recorre todos los clústeres comparando, para cada uno de ellos, la distancia entre cada par de vértices perteneciente. Por lo tanto, como en el peor de los casos se comparan ente sí todos los pares de vértices, la complejidad resultante es cuadrática respecto a V. Adicionalmente, se llevan a cabo otras operaciones constantes, que no influyen en el resultado final de esta.

Entonces, la complejidad temporal total es $\mathcal{O}(V^2 + V \cdot (V + E))$ más una cantidad constante de operaciones $\mathcal{O}(1)$. Dado que el término $V \cdot (V + E)$ abarca a V^2 y a las operaciones constantes, se puede definir más precisamente a la cota superior como $\mathcal{O}(V \cdot (V + E))$.

Efectivamente es polinomial respecto de las variables del problema, por lo que se puede afirmar que se encontró un verificador eficiente de complejidad polimonial y, por ende, el problema de Clusterización por bajo diámetro en su versión de decisión está en NP.

3.2. Demostración del problema NP-Completo

Habiendo demostrado que el problema forma parte de NP, para poder decir que es NP-Completo resta plantear una reducción polinomial de un problema NP-Completo conocido a éste.

Para ello, elegimos el problema de K-Coloreo, cuya versión de decisión plantea: "Dado un grafo no dirigido y no pesado $G = (V, E)$ y un entero k, ¿Es posible asignar a cada vértice de G un color entre k colores diferentes, de modo que no haya ningún par de adyacentes con el mismo color?"

Es necesario plantear la transformación de una instancia de K-Coloreo a otra de nuestro problema de Clusterización por Bajo Diámetro para comenzar con la demostración.

Como ya se mencionó, K-coloreo tiene como parámetros un grafo no dirigido G y un entero k. Mientras que nuestro problema tiene como parámetros de entrada un grafo no dirigido y no pesado G', y dos valores enteros k' y C.

La transformación planteada se basa en complementar el grafo G y que ese sea el parámetro G' en la *caja negra* resolvedora del problema de Clusterización. El sentido de esta conversión radica en que en un grafo K-coloreado, dos vértices del mismo color no pueden estar conectados entre ellos por definición. En consecuencia, este mismo par de vértices en el grafo complemento estará indefectiblemente unido por una arista. Entonces, todos los vértices del mismo color del grafo G, que en él forman un Independent Set, formarán un clique en el grafo $G' = \overline{G}$.

Un clique es un grafo cuyos vértices están todos interconectados, esto quiere decir que el diámetro máximo dentro de un clique es 1. Entonces, si se tomara un $C = 1$ y se indicara un $k' = k$, se podría resolver el problema de K-Coloreo con la Clusterización por Bajo Diámetro.

Dado que $C = 1$ es un caso especial de nuestro problema, (y en particular, equivalente al problema de Separación en R-Cliques), es necesario aclarar que la reducción también es válida para el caso general porque éste es al menos tan difícil como cualquier caso particular del mismo problema.

Todos los pasos de la transformación fueron polinomiales:

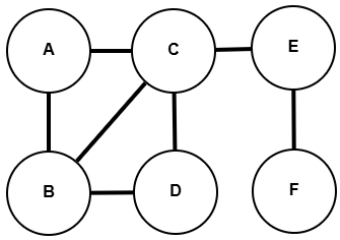
- Copiar el grafo complementado: $\mathcal{O}(V^2)$.
- Asignar $k = k'$ y $C = 1$: $\mathcal{O}(1)$.
- Realizar un único llamado a la función que resuelve el problema de Clusterización.
- La salida del problema anterior es exactamente la salida de K-coloreo, no requiere ningún tipo de manipulación. En este caso, cada clúster representaría un conjunto de vértices coloreados de un mismo color.

En código, esta transformación resultaría:

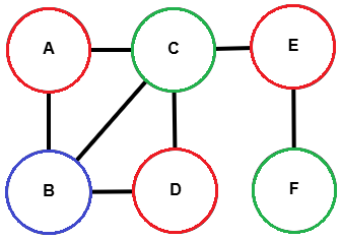
```
1 def k_coloreo(grafo, k):
2     grafo_complemento = Grafo(False, grafo.obtener_vertices())
```

```
3
4       for v in grafo:
5           for w in grafo:
6               if v == w:
7                   continue
8               if not grafo.estan_unidos(v, w):
9                   grafo_complemento.agregar_arista(v, w)
10
11      return clusterizar_bajo_diametro(grafo_complemento, k, 1)
```

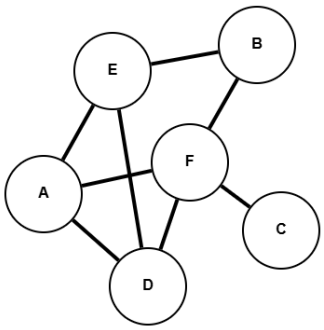
Aplicando esta reducción en un ejemplo concreto, supongamos que queremos conocer si es posible colorear el siguiente grafo G utilizando, a lo sumo, 3 colores ($k = 3$):



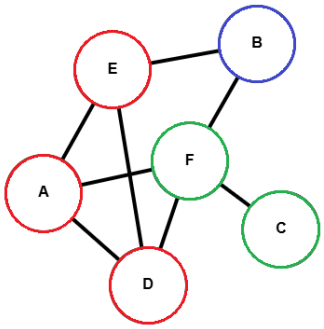
Una posible solución, resolviendo el problema con un algoritmo que resuelva, de forma directa, el problema de K-Coloreo sería:



Sin embargo, aplicando nuestra reducción, primero hallaríamos el grafo complemento $\overline{G}$,



Y, finalmente, realizaríamos el llamado al algoritmo resolvedor del problema de Clusterización, pasándole $G' = \overline{G}$, $k' = 3$ y $C' = 1$ como parámetros. Una posible solución (considerando cada color como un clúster), resultaría:



De esta forma, tal como se puede observar, el algoritmo de clusterización no solo encuentra solución siempre que exista una, sino que incluso es capaz de encontrar soluciones que sean exactas a una encontrada por un algoritmo resolvedor de K-coloreo directamente.

El paso final es demostrar formalmente que G *se puede pintar en* k *colores* $\iff$ $\overline{G}$ *puede dividirse en* k *clústeres de diámetro* $C = 1$. Para esto es necesario probar que se cumple la doble implicancia, lo cual haremos mediante método directo.

- G se puede pintar en k colores $\implies$ $\overline{G}$ puede dividirse en k clústeres de diámetro $C = 1$

  Se asume que G se puede pintar de k colores, esto quiere decir que no hay dos vértices adyacentes del mismo color. Para todos los pares de vértices (u, v) de G, por definición de complemento, si hay una arista entre u y v, en $\overline{G}$ no habrá arista, en cambio si no la hay en G, en $\overline{G}$ existirá.

  De esta forma, queda conformado $\overline{G}$ por todos los vértices de G y sus aristas complementarias. Entonces, si G se puede dividir en k grupos disjuntos, uno por color, esos mismos grupos de vértices en $\overline{G}$ conformarán un clique. Por definición de clique, cada uno de esos grupos tiene diámetro 1, es decir, se conforma un clúster con $C = 1$.

- $\overline{G}$ puede dividirse en k clústeres de diámetro $C = 1$ $\implies$ G se puede pintar en k colores

  Se asume que $\overline{G}$ puede dividirse en k clústeres de diámetro $C = 1$. Esto es equivalente a decir que $\overline{G}$ puede dividirse en k cliques. Estos subconjuntos de vértices tienen aristas entre todos los que los componen, en un grafo complementario, dichas aristas desaparecerían y los vértices de cada una no serían adyacentes, por lo tanto, podrían compartir color en ese nuevo grafo. De esta manera, se podría interpretar a cada clúster como un color distinto del grafo original, como son k los clústeres, serán k los colores también en G y todos los vértices podrán ser pintados sin repetir colores entre adyacentes.

De esta forma, queda demostrado que

$$\text{K-Coloreo} \leq_P \text{Clusterización por bajo diámetro}$$

3.3. Conclusión

Habiendo hallado un verificador eficiente y demostrado que la reducción de un problema NP-Completo al nuestro también se realiza en pasos polinómicos, se puede afirmar que el problema de Clusterización por Bajo Diámetro en su versión de **decisión** es, en efecto, un problema NP-Completo. Teniendo esta información, también se puede decir que el mismo problema pero en su versión de optimización es NP-Difícil. Esto significa que cualquier algoritmo que dé la solución **óptima** no podrá tener una complejidad mejor que la exponencial y ese es el caso de los algoritmos que se presentarán en la próxima sección.

4. Soluciones implementadas

Tal como se expuso en la sección de *Análisis del problema*, se desarrollarán cuatro algoritmos que, mediante la aplicación de distintos enfoques y técnicas, permitirán abordar el problema en su versión de **optimización**, obteniendo soluciones robustas y óptimas en algunos casos, y aproximadas pero eficientes en otros.

4.1. Backtracking

En esta sección se presentará el algoritmo Backtracking, capaz de encontrar soluciones óptimas y, como se mencionó recientemente, tendrá una complejidad exponencial por tratarse de un problema NP-Difícil.

Nuestra función `clusterizar_bajo_diámetro` recibe el grafo y la cantidad de clústeres k, toma los vértices y calcula la distancia de cada uno al resto recorriendo el grafo por BFS. Luego busca una solución inicial con la ayuda del algoritmo de aproximación presentado al final de esta sección, para luego decidir si está ante un caso trivial que permitiría terminar la ejecución inmediatamente. De no ser así, se procederá a hacer la asignación de una solución parcial inicial y luego se llamará a la función de Backtracking.

```
def clusterizar_bajo_diametro(grafo, k):
    # Ordena por grado ascendente:
    vertices = sorted(grafo.obtener_vertices(), key=lambda v: grafo.grado(v))
    distancias = calcular_distancias(grafo)

    sol_inicial, diam_max_inicial = hallar_aproximacion_inicial(vertices, k,
    distancias)
    # Casos en los que la aproximacion siempre es correcta:
    if diam_max_inicial == 0 or (diam_max_inicial == 1 and k <= len(vertices)):
        return sol_inicial, diam_max_inicial

    sol_par, seteados_inicialmente = hallar_clusters_iniciales(vertices, k, grafo)

    return clusterizar_bajo_diametro_bt(1, vertices, k, distancias, sol_par,
    seteados_inicialmente, 0, sol_inicial, diam_max_inicial)
```

En `clusterizar_bajo_diametro`, se aplican algunas heurísticas como:

- **Ordenar por grado** ascendente los vértices: en teoría, los vértices con menor grado, los "menos conectados", podrían resultar más restrictivos que los de mayor grado y, por lo tanto, permitirían arribar a soluciones de menor diámetro con anterioridad.
- Dar una **aproximación** como **solución** inicial: mediante un algoritmo greedy se busca rápidamente una solución cuyo diámetro sirve como cota inicial y punto de partida para el resto de soluciones.
- Dar una **solución parcial** inicial: a cada clúster se le asigna un vértice de una componente conexa diferente, son vértices que siempre tendrán entre ellos una distancia infinita por ser inalcanzables desde otra componente conexa, colocarlos en distintos clústeres permite asignarlos correctamente de antemano.

El algoritmo de Backtracking propiamente dicho inicia verificando si ya se exploraron todos los vértices del grafo, en caso positivo se devuelve una copia de la solución parcial actual (y su diámetro) ya que si "sobrevivió" hasta ese punto entonces tiene un diámetro menor que la considerada, hasta el momento, óptima.

Luego, esencialmente, en cada llamada a la función recursiva con un vértice nuevo, se recorren todos los clústeres explorando el árbol de soluciones, agregando el vértice y quitándolo de cada clúster. Para evitar examinar caminos que pueden ser descartados tempranamente y encontrar combinaciones que funcionen como cotas de otras, se aplican algunas podas:

- Chequear si agregando un nuevo vértice el diámetro óptimo empeoraría antes de asignarlo: evita las operaciones de agregado y borrado de vértices innecesarias, como así también la exploración de ramas que ya se sabe que no podrán llegar a una solución mejor a la que se tiene.

  Pedir que sea mejor es que sea $<$ `diam_max_opt`, en este caso, descartar las opciones mayores o iguales ($\geq$) también funciona como una poda poderosa versus solamente las mayores ($>$).

- Evitar probar soluciones iguales: si al sacar un elemento de un clúster éste queda vacío, no vuelve a probarlo en otro clúster sin elementos ya que el resultado sería el mismo. Por esto mismo los clústeres vacíos se ubican al final del arreglo de solución ya que, de esta manera, luego de probar en el primer vacío, se puede cortar la iteración del ciclo.

```
def clusterizar_bajo_diametro_bt(act, vertices, k, distancias, sol_par,
    seteados_inicialmente, diam_max_par, sol_opt, diam_max_opt):
    if act == len(vertices):
        return [cluster[:] for cluster in sol_par], diam_max_par

    if vertices[act] in seteados_inicialmente:
        return clusterizar_bajo_diametro_bt(act + 1, vertices, k, distancias,
    sol_par, seteados_inicialmente, diam_max_par, sol_opt, diam_max_opt)


    for i in range(k):
        nuevo_diam = calcular_nuevo_diam(vertices[act], sol_par[i], distancias)

        if nuevo_diam >= diam_max_opt:
            continue

        sol_par[i].append(vertices[act])
        sol_opt, diam_max_opt = clusterizar_bajo_diametro_bt(act + 1, vertices, k,
    distancias, sol_par, seteados_inicialmente, max(diam_max_par, nuevo_diam),
    sol_opt, diam_max_opt)
        sol_par[i].pop()

        if len(sol_par[i]) == 0:
            break

    return sol_opt, diam_max_opt
```

Entre las funciones "auxiliares" usadas, cuyo código todavía no incluimos, se encuentran:

`hallar_clusters_iniciales`, si bien su funcionamiento como poda fue explicado, puede verse su código a continuación:

```
def hallar_clusters_iniciales(vertices, k, grafo):
    seteados_inicialmente = set()
    clusters = [[] for _ in range(k)]
    ult = 0
    visitados = set()
    for v in vertices:
        if v not in visitados:
            seteados_inicialmente.add(v)
            clusters[ult].append(v)
            ult += 1
            if ult == k:
                break

            visitados.add(v)
            cola = deque([v])
            while cola:
                w = cola.popleft()
                for a in grafo.adyacentes(w):
                    if a not in visitados:
                        visitados.add(a)
                        cola.append(a)
    return clusters, seteados_inicialmente
```

Se basa simplemente en iterar sobre todos los vértices donde cada vez que encuentra uno no visitado, lo agrega al conjunto inicial y realiza un recorrido BFS desde ese vértice, marcando como visitados todos los alcanzables desde él. De esta forma, se evita agregar vértices que pertenezcan a la misma componente conexa.

Por otro lado, `calcular_nuevo_diam` puede verse a seguidamente,

```
def calcular_nuevo_diam(v, cluster, distancias):
    max = 0
    for a in cluster:
        max = distancias[v][a] if distancias[v][a] > max else max
    return max
```

En ésta, solamente se busca la distancia máxima desde un vértice específico a todos los pertenecientes a un clúster determinado.

En la próxima sección se medirá el rendimiento de este algoritmo frente a distintos sets de prueba.

4.2. Programación Lineal

Para esta sección no desarrollaremos un algoritmo en sí, sino que presentaremos un modelo que permite resolver el problema de forma óptima y puede ser implementado, como se verá más adelante, con herramientas capaces de resolver problemas de Programación Lineal Entera, tales como PuLP, SciPy, CPLEX, entre otros.

Un dato a tener en cuenta es que la Programación Lineal Entera, a diferencia de la continua que pertenece a P y, por lo tanto, puede resolverse en tiempo polinomial, también se trata de un problema NP-Completo. Por lo tanto, con el conocimiento que se dispone en la actualidad, resolver este problema presenta una complejidad temporal exponencial y la mayoría de los *solvers* implementan una metodología de resolución de problemas conocida como *Branch and Bounds*, que, en esencia, se trata de un "símil" backtracking que aplica ramificaciones para probar diferentes alternativas y utiliza las restricciones como podas.

Dicho esto, ya estamos en condiciones de presentar el modelo en cuestión:

> Dados un grafo no dirigido y no pesado $G = (V, E)$, donde V indica los vértices en él y E las aristas que conectan dichos vértices, y un número entero positivo K, que indica la cantidad máxima de clústeres, definimos:

Constantes

- $\forall i, j \in V,\ \exists D_{ij} \in \mathbb{R}_0^+$ tal que D_{ij} es la menor distancia entre los vértices i y j en el grafo.
- $D_{\text{aprox}}^{\max}$ es la distancia máxima encontrada por el algoritmo de aproximación.

Como puede notarse, previamente se debe realizar un recorrido BFS por cada vértice para determinar las distancias mínimas entre todos ellos, en tiempo $\mathcal{O}(V \cdot (V + E))$, y calcularse una solución aproximada (con el algoritmo greedy que se presenta más adelante, por ejemplo), que nos proveerá una cota inicial del diámetro máximo dentro de los clústeres.

Variables

- $\forall i \in V,\ \forall k \in K,\ \exists v_{ik} \in \{0, 1\},$ donde $v_{ik} = 1$ si el vértice i pertenece al clúster k.
- $d_{\max} \in \mathbb{R}_0^+$ representa la distancia máxima entre vértices de un mismo clúster.

Es de importancia mencionar que $d_{\max}$ es definida como una variable continua únicamente para que los *solvers*, como PuLP, puedan aplicar metodologías de optimización extra que permiten

obtener resultados con mayor rapidez. Esta puede definirse como una variable entera con valor mayor o igual a 0 sin problema alguno (de hecho, sería más acorde dado que las distancias son valores enteros).

Adicionalmente, debe aclararse que definir el dominio de la variable como $\mathbb{R}_0^+$ se trata, técnicamente, de una restricción impuesta sobre el valor mínimo que ésta puede tomar. En este caso, 0.

Restricciones

- $\sum_{k \in K} v_{ik} = 1, \quad \forall i \in V$

  Así, se garantiza la pertenencia de cada vértice a exactamente un único clúster.

- $\forall i, j \in V,\ i \neq j, \quad \forall k \in K, \quad D_{ij} - D_{ij} \cdot (2 - v_{ik} - v_{jk}) \leq d_{\max}$

  Por lo tanto, si dos vértices pertenecen a un mismo clúster, $d_{\max}$ deberá ser mayor o igual a la distancia entre ellos y, dado que estamos trabajando con un problema de minimización, no será posible que tome un valor mayor a la distancia máxima entre dos vértices pertenecientes a un mismo clúster, que es exactamente lo que estamos buscando.

  Cuando uno de los vértices pertenezca al clúster y el otro no, $d_{\max}$ deberá ser mayor o igual a 0. De esta forma, vértices *disjuntos* no condicionan el valor de ésta.

- $d_{\max} \leq D_{\text{aprox}}^{\max}$

  Se impone una cota máxima del valor que $d_{\max}$ puede tomar en función de la distancia máxima encontrada por el algoritmo de aproximación.

  Esta restricción es exclusivamente opcional y permite a algoritmos como el ya mencionado *Branch and Bounds* encontrar la mejor solución en menor tiempo. Sin embargo, podría omitirse completamente del modelo. En casos donde la cantidad de vértices del grafo es baja y, por ende, una restricción extra genera una importante diferencia, notamos un mejor desempeño si no se la incluye.

  De todas formas, en casos generales, suele implicar una gran mejora y por eso decidimos dejarla.

- La siguiente restricción consiste en un seteo inicial de vértices pertenecientes a diferentes componentes conexas, tal como se vio en el algoritmo anterior.

  Sea $S = \{i_1, i_2, \ldots, i_m\} \subseteq V$, con $m \leq K$, el conjunto de vértices ubicables en los primeros m clústeres. Entonces, se impone la restricción: $v_{i_r,\, r} = 1, \quad \forall r \in \{1, \ldots, m\}$

  Nuevamente, al igual que en el caso anterior, esta condición puede ser removida del modelo sin consecuencia alguna. No obstante, esta sí lleva a una mejora sustancial en los tiempos de ejecución al implementar el modelo.

  Nota: Definimos a S en este apartado únicamente con fines expositivos. En la práctica, se trata de un conjunto de constantes, por lo que debería haberse introducido previamente.

Función objetivo

Como el propósito del problema es que la distancia sea mínima, la función objetivo resulta simplemente,

$$\text{mín}(d_{\max})$$

De esta forma, quedó definido el modelo de Programación Lineal que permite resolver de forma óptima el problema planteado. En total se definen alrededor de $V \cdot k$ variables y $V^2 \cdot k$ restricciones, lo que lo convierte en un modelo bastante ineficiente que, a pesar de nuestros esfuerzos, no pudimos mejorar más.

Por último, proseguimos con su implementación utilizando una de las herramientas de Python que mencionamos al comienzo de esta subsección: PuLP.

A continuación, el código:

```
def clusterizar_bajo_diametro(grafo, cant_clusters):
    vertices = grafo.obtener_vertices()
    distancias = calcular_distancias(grafo)
    indices_seteo_inicial = hallar_seteo_inicial(vertices, cant_clusters, grafo)
    _, dist_max_aprox = hallar_aproximacion_inicial(vertices, cant_clusters,
    distancias)

    # Definicion del problema:
    problema = LpProblem("determinar_clusters", LpMinimize)

    variables = [] # Fila indica cluster, columna indica vertice
    # Cada variable indica, El i-esimo vertice esta en el k-esimo cluster?:
    for cluster in range(cant_clusters):
        variables.append([LpVariable(f'{v}_{cluster}', cat = "Binary") for v in
    vertices])

    # Maxima distancia:
    dist_max = LpVariable(f'dist_max', lowBound = 0, cat = "Continuous")

    problema += dist_max <= dist_max_aprox

    # Seteo inicial de vertices que no podrian estar en un mismo cluster:
    for cluster, i in enumerate(indices_seteo_inicial):
        problema += variables[i][cluster] == 1

    for i, v in enumerate(vertices):
        # Cada vertice debe estar en un unico cluster:
        problema += lpSum(variables[cluster][i] for cluster in range(cant_clusters)
    ) == 1

        for j in range(i + 1, len(vertices)):
            w = vertices[j]

            for cluster in range(cant_clusters):
                # Si los dos vertices pertenecen al mismo cluster, dist_max debera
    ser >= a la distancia entre ellos:
                problema += distancias[v][w] - distancias[v][w] * (2 - variables[
    cluster][i] - variables[cluster][j]) <= dist_max

    # Funcion objetivo (se desea minimizar la distancia maxima):
    problema += dist_max
    problema.solve(PULP_CBC_CMD(msg = False))

    # Interpretacion de los resultados:
    clusters = []
    for cluster in range(cant_clusters):
        clusters.append([vertices[i] for i in range(len(vertices)) if value(
    variables[cluster][i]) == 1])
    return formatear_salida(clusters, int(value(dist_max)))
```

Una mejora extra que consideramos y creímos que podría funcionar fue ordenar previamente los vértices ascendentemente según su grado, como realizamos con el algoritmo anterior. Sin embargo, no reflejó una baja en los tiempos de ejecución. Probablemente PuLP no procese las variables en el orden que se setean, por lo que no puede aprovechar el potencial de éste como lo hicimos anteriormente.

En la sección 5 se medirán los tiempos de ejecución de esta implementación y compararán con los obtenidos con el algoritmo por Backtracking. Mientras tanto, continuamos desarrollando los dos algoritmos de aproximación.

4.3. Aproximaciones

4.3.1. Algoritmo de Louvain

Este punto resultó en un gran desafío al que le dedicamos mucho tiempo y esfuerzo realizando una amplia investigación inicial, en la que debatimos sobre cómo deberíamos implementarlo, leímos diversos recursos y recolectamos toda la información que nos pareció de utilidad. Luego, pudimos comenzar con su desarrollo.

No obstante, queremos comenzar introduciendo el algoritmo y explicando por qué consideramos que, a pesar de no minimizar distancias, se trata una metodología perfecta para el problema desde el punto de vista planteado inicialmente en la introducción, el de separar un grafo en comunidades.

Este debe su creación a Vincent Daniel Blondel, profesor de la Univerisdad Católica de Louvain, Bélgica, quién, junto con otros investigadores, publicó en el año 2008 un paper titulado *Fast unfolding of communities in large networks*. En él se presentó este método que, basándose en la optimización de la modularidad, permite extraer de forma casi certera la estructura de comunidades de una red.

La modularidad es un valor que oscila entre -1 y 1 y mide la densidad relativa de las aristas dentro de las comunidades con respecto a las aristas fuera de ellas. Optimizar este valor permite obtener la mejor agrupación de vértices en una red. Sin embargo, dado que, como bien es sabido, se trata de un problema NP-Completo, analizar todas las posibles soluciones puede resultar muy ineficiente. En consecuencia, debemos utilizar algoritmos heurísticos o greedy, tal como este: el **Algoritmo de Louvain**.

Su implementación cuenta con dos fases principales:

1. Cada vértice es asignado inicialmente en su propia comunidad. En esta fase se iteran todos ellos sucesivas veces, calculando la ganancia de modularidad que implicaría moverlos hacia cada una de sus comunidades vecinas y, por ende, asignándolos a la que el aumento sea máximo. De manera que esta fase finaliza cuando ningún vértice puede incrementarla más.
2. Se genera un nuevo grafo pesado. En él, todos los vértices originales son reemplazados por supervértices que engloban a todos los pertenecientes a una misma comunidad. Las conexiones entre ellos están dadas por bucles, que representan las conexiones originales entre vértices de una misma comunidad, y aristas dadas por las aristas originales que unían previamente a los vértices de una comunidad con los de otra.

Estas dos fases se ejecutan repetidamente hasta que no se realice ninguna modificación, consecuencia de haber logrado la *convergencia*. En nuestra implementación, el algoritmo recibe un número k, que representa la cantidad máxima de comunidades que queremos hallar, y puede finalizar por dos motivos:

- Se logró la ya mencionada convergencia.
- La cantidad de comunidades actuales es menor o igual a k.

Para el primer caso, donde la cantidad de comunidades resultará mayor a k, debemos realizar un *mergeo* entre las cuales el peso entre aristas en el grafo resultante sea mayor. De esta forma, a pesar de ya no poder mejorar la modularidad, conectamos comunidades cuya densidad era alta en el grafo original y podemos llegar al número máximo de comunidades requerido.

Por otro lado, es de gran importancia detallar cómo se realiza el cálculo de modularidad. Como se vio al inicio de este informe, su cálculo es bastante extenso e involucra diversos factores dependientes de las conexiones en el grafo y la asignación de comunidades. Sin embargo, al implementar este algoritmo no es realmente necesario aplicar exactamente ese cómputo. De hacerlo así, deberíamos recalcular la modularidad del grafo entero en cada posible asignación e implicaría un enorme costo computacional que no vale la pena disponer. Es por eso que, para evitarlo, decidimos calcular el incremento en la modularidad que implicaría hacer cada pasaje de un vértice desde una comunidad hacia otra. Para esto, calculamos

- La ganancia en modularidad dada por la inclusión de un vértice (v) en una determinada comunidad (C): $\Delta Q(v \to C)$
- La ganancia en modularidad dada por la inclusión de un vértice en su comunidad actual (D). Para esto, debemos realizar los cálculos como si no fuese parte de ella (es decir, *excluirlo*): $\Delta Q(v \to D)$

Luego, como lo que realmente nos interesa es la ganancia que obtendríamos al sacar a v de D y ponerlo en C, se debe resolver:

$$\Delta Q(D \to v \to C) = \Delta Q(v \to C) \text{ - } \Delta Q(v \to D)$$

donde, $\Delta Q(v \to X)$, X una comunidad cualquiera, equivale a:

$$\Delta Q(v \to X) = \left[\frac{\sum_{in} + k_{v,in}}{2m} - \left(\frac{\sum_{tot} + k_v}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_v}{2m}\right)^2\right] \quad (1)$$

siendo,

- $\sum_{in}$ la suma de pesos entre los vértices de X.
- $\sum_{tot}$ la suma de pesos totales de los vértices de X (incluyendo aristas hacia fuera de la comunidad).
- $k_{v,in}$ la suma de pesos entre v y los vértices de X.
- k_v la suma de pesos de v.
- $2m$ la suma de pesos de todas las aristas del grafo (sumando ambos extremos).

A vista de ello, consideramos que ya estamos en condiciones de introducir su implementación. Puesto que el código entero es considerablemente extenso (alrededor de 250 líneas), solo presentaremos las funciones principales. La implementación completa puede verse en el repositorio de este trabajo.

La función principal, `louvain`, engloba las fases 1 y 2, así como también todos los llamados a la funciones necesarias para realizar los procesamientos extras requeridos (como el *mergeo* final y el correcto formateo de los resultados, por ejemplo):

```
def louvain(grafo, k):
    """
    Obtiene k comunidades disjuntas de vertices pertenecientes a
    un grafo.
    """
    # Asigna cada vertice a una comunidad inicial:
    comunidad = _asignar_comunidades(grafo)

    # Crea un diccionario para guardar las referencias originales de las
    # comunidades de los vertices:
    comunidades_originales = _asignar_comunidades_originales_inicial(grafo)

    deberia_seguir = True
    # Corta cuando:
    # 'deberia seguir' es False, porque las comunidades ya convergieron.
    # Ya se tiene la cantidad de comunidades buscada.
    while len(comunidades_originales) > k and deberia_seguir:
        # FASE 1:
        # Agrupa los vertices en comunidades que maximicen la modularidad:
        deberia_seguir = _agrupar_nodos(grafo, comunidad)

        # FASE 2:
        # Reemplaza los vertices pertenecientes a mismas comunidades con
        # nuevos "supervertices" y sus respectivas conexiones:
```

```
        grafo, comunidades_originales = _crear_super_comunidades(grafo, comunidad, comunidades_originales)
        # Asigna comunidades a estos nuevos vertices:
        comunidad = _asignar_comunidades(grafo)

    # Si las comunidades convergieron antes de llegar a k, se mergean
    # las mas cercanas hasta llegar:
    if len(comunidades_originales) > k:
        _mergear_comunidades(grafo, k, comunidades_originales)

    # Formatea correctamente la solucion y la retorna:
    return _transformar_comunidades_a_lista(comunidades_originales, k)
```

Luego, es la función `_agrupar_nodos` la encargada de realizar todos los pasos pertenecientes a la fase 1. Esta presenta un ciclo `while` que se ejecuta hasta que no haya más cambios de modularidad. Internamente a este, se ejecuta un ciclo `for` que itera todos los vértices y para cada uno de ellos calcula $\Delta Q(D \to v \to C)$, siendo C cada una de las comunidades vecinas, y los mueve a la comunidad que sea conveniente.

Para omitir la inclusión de las funciones que únicamente realizan los cálculos descritos anteriormente, cada línea se encuentra comentada describiendo la tarea exacta que realiza internamente:

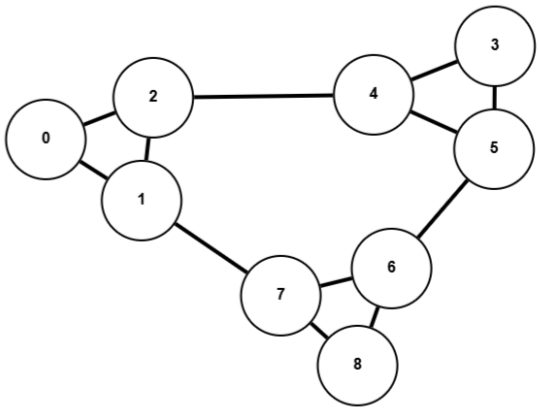
```
def _agrupar_nodos(grafo, comunidad):
    """
    Agrupa los vertices de manera que se maximice la modularidad.
    Corta cuando ya no puede ser mejorada.
    """

    # Suma de pesos de las aristas:
    dos_m = _calcular_dos_m(grafo)

    deberia_seguir = False
    hubo_cambios = True

    while hubo_cambios:
        hubo_cambios = False
        vertices = grafo.obtener_vertices()
        # Evita iterar los vertices siempre en el mismo orden:
        random.shuffle(vertices)

        for v in vertices:
            # Primero se calcula delta_Q(v -> D) como si el vertice no estuviese alli.

            # Calcula la suma de pesos internos de cada comunidad:
            sum_in = _calcular_suma_pesos_internos_comunidades(grafo, comunidad, v)

            # Calcula la suma de pesos totales de cada comunidad:
            sum_tot = _calcular_suma_pesos_comunidades(grafo, comunidad, v)

            # Suma de pesos entre el vertice y su comunidad (haciendo como si no
            # estuviese en ella):
            k_i_in = _calcular_suma_pesos_v_y_comu(grafo, v, comunidad[v], comunidad)

            # Suma de pesos del vertice:
            k_i = _calcular_suma_pesos_v(grafo, v)

            # Si era el unico vertice en su comunidad, no se agrego esta previamente
            # al diccionario. Entonces:
            sum_in[comunidad[v]] = sum_in.get(comunidad[v], 0)
            sum_tot[comunidad[v]] = sum_tot.get(comunidad[v], 0)

            # Calculo de delta_Q(v -> D)
            delta_q_d = (((sum_in[comunidad[v]] + k_i_in) / dos_m) - (((sum_tot[comunidad[v]] + k_i) / dos_m)**2)) - ((sum_in[comunidad[v]] / dos_m) - ((sum_tot[comunidad[v]] / dos_m)**2) - ((k_i / dos_m)**2))

            visitadas = {comunidad[v]} # La modularidad en la comunidad donde ya
```

```
    esta interesa.
            mejor = 0
            comu_mejor = None
            for a in grafo.adyacentes(v):
                comu_vecina = comunidad[a]
                if comu_vecina in visitadas:
                    continue
                else:
                    visitadas.add(comu_vecina)

                k_i_in = _calcular_suma_pesos_v_y_comu(grafo, v, comu_vecina,
    comunidad)
                # Calcula delta_Q(v -> C):
                delta_q_c = (((sum_in[comu_vecina] + k_i_in) / dos_m) - (((sum_tot[
    comu_vecina] + k_i) / dos_m)**2)) - ((sum_in[comu_vecina] / dos_m) - ((sum_tot[
    comu_vecina] / dos_m)**2) - ((k_i / dos_m)**2))
                # Calcula el incremento de modularidad, delta_Q(v -> C) - delta_Q(v
     -> D):
                incremento = delta_q_c - delta_q_d

                if incremento > mejor:
                    mejor = incremento
                    comu_mejor = comu_vecina
                    hubo_cambios = True
                    deberia_seguir = True

            if comu_mejor is not None:
                comunidad[v] = comu_mejor

    return deberia_seguir
```

La fase 2 se lleva a cabo casi en su totalidad por la función `_crear_super_comunidades`. Allí se realiza todo el procesamiento necesario para la creación de un nuevo *súper* grafo en el que se incluye todo lo mencionado con anterioridad:

```
def _crear_super_comunidades(grafo, comunidad, comunidades_originales):
    """
    Se crea un nuevo grafo donde cada comunidad resultando ahora es un
    nuevo supervertice. Los pesos de las aristas estan dados: en el caso
    de los bucles, por la suma de pesos entre aristas de los vertices
    anteriores de una misma comunidad, y, en el caso de las aristas entre
    supervertices, por los pesos de las aristas que conectaban
    anteriomente vertices entre ellas.
    """
    nuevo_grafo = Grafo(False)
    visitados = set()
    for v in grafo:
        visitados.add(v)
        for a in grafo.adyacentes(v):
            x = comunidad[v]
            y = comunidad[a]

             # Las aristas entre distintas comunidades solo se suman una vez:
            if a in visitados and x != y:
                continue

            if x not in nuevo_grafo:
                nuevo_grafo.agregar_vertice(x)
            if not y in nuevo_grafo:
                nuevo_grafo.agregar_vertice(y)

            if not nuevo_grafo.estan_unidos(x, y):
                nuevo_grafo.agregar_arista(x, y, grafo.peso_arista(v, a))
            else:
                nuevo_grafo.actualizar_peso(x, y, nuevo_grafo.peso_arista(x, y) +
    grafo.peso_arista(v, a))

    return nuevo_grafo, _generar_nuevas_comunidades(comunidad,
    comunidades_originales)
```

Para verificar su correcto funcionamiento, diseñamos algunos grafos de prueba sencillos, con comunidades claramente diferenciadas. Por ejemplo, en el siguiente grafo:



Es evidente la existencia de tres comunidades: **(0, 1, 2)**, **(3, 4, 5)** y **(6, 7, 8)**, cada una formando un clique, con muy pocas aristas conectándolas con otras comunidades. Al ejecutar la prueba con nuestra implementación, el resultado obtenido fue el siguiente:

```
Comunidad 1 : ['1', '2', '0']
Comunidad 2 : ['6', '8', '7']
Comunidad 3 : ['4', '3', '5']
```

Claramente, es el esperado (el orden de los vértices dentro de la comunidad es indistinto). Los resultados obtenidos con esta y otras pruebas, incluyendo la presentada en al inicio del artículo previamente mencionado (también utilizada en las diapositivas de la clase de **Redes Complejas**), nos permitieron confirmar que la implementación en correcta.

Finalmente, para aplicar esta implementación al problema de Clusterización por Bajo Diámetro, interpretamos las comunidades detectadas como clústeres. Luego, calculamos la distancia máxima entre pares de vértices dentro de cada clúster y presentamos toda la información siguiendo el formato predefinido por la cátedra.

4.3.2. Nuestra propuesta

Nuestra primera idea al considerar una solución de aproximación fue desarrollar un algoritmo greedy en el que se crearan los k clústeres y se iteraran los vértices agregando cada uno al clúster en el cual la distancia máxima fuera mínima. De esta forma, con una heurística muy sencilla y muy poco código, logramos *acercarnos* a las soluciones óptimas en tiempo polinomial.

```python
def hallar_aproximacion_greedy(grafo, k):
    vertices = sorted(grafo.obtener_vertices(), key=lambda v: grafo.grado(v))
    distancias = calcular_distancias(grafo)
    solucion, max_diam = [[] for _ in range(k)], 0

    for v in vertices:
        min_diam, min_i = float('inf'), 0

        for i in range(k):
            # calcular_nuevo_diam halla el nuevo diametro max. si se ubica el vertice en ese cluster
            nuevo_diam = calcular_nuevo_diam(v, solucion[i], distancias)
            if nuevo_diam < min_diam:
                min_diam, min_i = nuevo_diam, i

        max_diam = min_diam if min_diam > max_diam else max_diam
        solucion[min_i].append(v)
```

```
17
18      return solucion, max_diam
```

Como se aprecia, incluso añadimos optimizaciones como el ordenamiento por grado de adyacencias. De esta forma, al igual que en los algoritmos anteriores, se setean inicialmente los vértices "más lejanos".

La complejidad del mismo, siendo V la cantidad de vértices y E la cantidad de aristas, se debe predominantemente a la búsqueda de distancias que, como ya indicamos repetidas veces, tiene un costo de $\mathcal{O}(V \cdot (V + E))$. El ciclo principal, que itera todos los vértices e internamente a todos los clústeres y sus vértices (tarea que, en consecuencia, presenta una complejidad de $\mathcal{O}(V)$), resulta en un orden de complejidad de $\mathcal{O}(V^2)$, término que ya se encuentra abarcado por la complejidad previa. En consecuencia, la complejidad final resultante es $\mathcal{O}(V^2 + V \cdot E)$.

Pero, luego de un arduo análisis, no pudimos encontrar ninguna cota de factor constante para nuestra aproximación. Si bien el algoritmo se comporta muy bien en la práctica (mejor que nuestra próxima solución, podrá comprobarse en la próxima sección), podrían darse situaciones en las que la solución brindada sea sumamente peor que la provista por un algoritmo óptimo.

Fue por esto que comenzamos con la investigación de posibles aplicaciones para aproximar soluciones con cotas de factor contante. Hallamos una familia de problemas, también NP-Completos en su versión de decisión y NP-Difíciles en su versión de optimización, cuya similitud con la Clusterización por Bajo Diámetro es muy grande, entre ellos:

- K-Median: Dado un entero K y un conjunto de vértices V de un grafo, de los cuales conocemos la distancia entre ellos, se deben seleccionar K centros (vértices) de forma tal que la **suma** de las **distancias mínimas** de cualquier vértice perteneciente a V al centro más cercano sea mínima.
- K-Means: Dado un entero K y un conjunto de vértices V de un grafo, de los cuales conocemos la distancia entre ellos, se deben seleccionar K centros (vértices) de forma tal que la **suma** de las **distancias mínimas al cuadrado** de cualquier vértice perteneciente a V al centro más cercano sea mínima.
- K-Center: Dado un entero K y un conjunto de vértices V de un grafo, de los cuales conocemos la distancia entre ellos, se deben seleccionar K centros (vértices) de tal forma que la **distancia máxima** de los vértices de V al centro más cercano sea mínima.

Claramente el problema de K-Center es muy similar a lo que estamos buscando resolver. Por lo tanto, buscando un algoritmo de aproximación que pueda resolverlo, nos topamos con el **Algoritmo de González**. Este es un sencillo algoritmo Greedy desarrollado por Teófilo González, investigador y profesor de la UCSB, que permite obtener aproximaciones a lo sumo tan grandes como el **doble** de la solución óptima.

Usando *Farthest-first traversal* en K iteraciones, este algoritmo simplemente elige el vértice más lejano de los centros actuales en cada iteración como un nuevo centro. A continuación se describe el ciclo que realiza:

1. Comienza seleccionando un $v \in V$ cualquiera y lo asigna como centro.
2. Itera cada v calculando la distancia mínima de cada uno de ellos a cada centro ya asignado.
3. Selecciona el v cuya distancia mínima a los centros ya asignados sea máxima y lo agrega como centro.
4. Repite los pasos 2 y 3 hasta que se hayan seleccionado K centros.

A pesar de que la demostración de por qué es una 2-Aproximación no presenta una gran complejidad, obviamos transcribirla pero dejamos el link a un apunte perteneciente a la Universidad de Maryland en el que se explica todo acerca del problema y el algoritmo con el que estamos tratando.

Ahora bien, las preguntas que surgen son: ¿Podemos adaptar este algoritmo para resolver, de forma aproximada, el problema de Clusterización por Bajo Diámetro? ¿Qué tan cercana será la solución obtenida a la óptima?

La respuesta es sí. Dada la entrada del problema de Clusterización por Bajo Diámetro, podemos calcular las distancias entre todos los pares de vértices y, utilizando el valor K recibido (el número de clústers máximo), aplicar el Algoritmo de González para obtener una 2-Aproximación del problema de K-Center. Como sabemos, este algoritmo garantiza que la distancia entre cualquier vértice y su centro más cercano es como máximo r, donde $r \leq 2r^*$, siendo r^* el radio óptimo.

Como nuestro objetivo es minimizar el diámetro entre cualquier par de vértices dentro de un clúster, asignamos cada vértice al clúster cuyo centro esté más cercano. En el peor de los casos, podría haber dos vértices a distancia r del centro, lo que implicaría que la distancia entre ellos pueda ser $2r$ (si ambos están en extremos opuestos). Dado que $r \leq 2r^*$, se sigue que $2r \leq 4r^*$.

Recordando que el diámetro de un clúster es a lo sumo el doble del radio ($d = 2r$), entonces $d \leq 4r^*$. Como el diámetro óptimo d^* es $2r^*$, concluimos que el diámetro obtenido está acotado por $2d^*$. Por lo tanto, el algoritmo implementado proporciona una 2-aproximación para el problema de Clusterización por Bajo Diámetro.

A continuación, la implementación recién explicada:

```
def clusterizar_bajo_diametro(grafo, k):
    distancias = calcular_distancias(grafo)

    # Se crea un diccionario en el que cada centro es la clave y sus valores
    # son los vertices mas cercanos. Luego, estos formaran clusters.
    centros = {c: set() for c in k_center(distancias, grafo, k)}

    # Se itera cada vertice y se los asigna al centro con < distancia.
    for v in grafo:
        if v in centros:
            continue

        dist_min = float('inf')
        asignacion = None
        for c in centros.keys():
            if distancias[v][c] < dist_min:
                dist_min = distancias[v][c]
                asignacion = c

        centros[asignacion].add(v)

    # Se le da el formato correcto a los clusters y se calcula la distancia
    # maxima entre dos vertices.
    solucion = [[] for _ in range(k)]
    dist_max = 0
    for i, (c, asignados) in enumerate(centros.items()):
        solucion[i].append(c)
        for v in asignados:
            for a in solucion[i]:
                dist_max = distancias[v][a] if distancias[v][a] > dist_max else
    dist_max
            solucion[i].append(v)

    return solucion, dist_max

def k_center(distancias, grafo, k):
    vertices = grafo.obtener_vertices()

    # Asigna inicialmente un vertice aleatorio como centro inicial.
    # Podria asignarse arbitrariamente si se quiere determinismo.
    centros = {grafo.vertice_aleatorio()}

    # Itera todos los vertices y asigna como centros a todos aquellos
    # cuya distancia minima sea maxima, hasta llega a tener K centros.
    while len(centros) < k:
        dist_min_max, min_max = -1, None

```

```
47          for v in vertices:
48              if v in centros:
49                  continue
50
51              min_dist_a_centros = min(distancias[v][c] for c in centros)
52              if min_dist_a_centros > dist_min_max:
53                  dist_min_max = min_dist_a_centros
54                  min_max = v
55
56          if min_max is None: # Si no llega a poder asignar K centros
57              break
58          centros.add(min_max)
59
60      return centros
```

La complejidad temporal de todo el algoritmo está dada por el cálculo de distancias, $\mathcal{O}(V \cdot (V + E))$, y la función `k_center`. En ésta se iteran K veces todos los V vértices, para los cuales se verifica la distancia a cada centro (K), por lo que la complejidad resulta $\mathcal{O}(V \cdot K^2)$ y, en consecuencia, la complejidad temporal máxima resultante de todo el algoritmo es $\mathcal{O}(V \cdot (V + E) + V \cdot K^2)$ o, si se prefiere, $\mathcal{O}(max(V^2, V \cdot E, V \cdot K^2))$.

Al igual que con todos los algoritmos presentados, se realizarán mediciones y comparaciones a continuación.

5. Mediciones y comparativa de algoritmos

Llegado el punto de tener que realizar mediciones y comparar los algoritmos nos topamos con un gran problema: ¿Cómo deberíamos generar los grafos de prueba?

Si bien es un tema que excede los contenidos vistos en la materia, consultando diversos recursos descubrimos la existencia de **modelos de generación de grafos aleatorios**. Estos permiten generar redes con ciertas características que emulan a sistemas reales empleados por el humano y la naturaleza, como podrían ser las redes sociales, redes de tendido eléctrico o la distribución de idiomas hablados por una sociedad.

Estos presentan distribuciones de grado regulables que permiten variar el número de conexiones asociadas a ciertos vértices y, por ende, obtener grafos de diversas densidades. Otros también se centran en la aplicación de leyes de potencia (o libres de escala), que permiten ajustar de forma proporcional este último factor.

Algunos de los modelos que nos resultaron relevantes para la generación de estas pruebas fueron **Barabási–Albert** (BA), para generar redes donde algunos pocos nodos están altamente conectados, mientras que la mayoría tiene un grado muy bajo; **Watts-Strogatz** (WS), donde la mayoría de nodos no están conectados entre sí pero son alcanzables desde cualquier otro; y **Erdös–Rényi** (ER), bastante más antiguo y centrado principalmente en la conexión basándose, casi únicamente, en una probabilidad fija, donde todos los nodos presentan una conectividad similar.

Puesto que su implementación es realmente compleja, utilizamos la librería de python `NetworkX` (diseñada para la creación, estudio y análisis de grafos) para desarrollar un generador de pruebas que, siguiendo el mismo formato que las pruebas provistas por la cátedra, permite producir redes con los modelos mencionados en el anterior párrafo.

Al realizar un análisis sobre las pruebas provistas por la cátedra notamos la presencia de vértices centrales, con grado muy alto, y otros con grados más moderados y similares entre sí. Además de ser grafos conexos. Es por ello que decidimos generar redes utilizando el modelo de BA, que genera redes con una estructura muy similar. Si bien no tenemos certeza alguna de si las pruebas de la cátedra fueron creadas con la utilización de este modelo, o si se utilizó siquiera algún modelo conocido, sospechamos que podría haber sido este u otro cuya estrategia resulte parecida.

Los parámetros utilizados para la creación de estas primeras pruebas fueron el **número de vértices** (V) y el **número de aristas que unen a cada nuevo vértice con los anteriores** (m), el cual seteamos en **3** para obtener resultados próximos a las pruebas de cátedra cuyo nombre está dado por el formato "`V_3.txt`". Los tiempos que se presentarán son el promedio de cinco ejecuciones de cada prueba.

Durante toda esta sección k será la cantidad de clústeres.

5.1. Backtracking

Al correr los ejemplos con este algoritmo y $k = 7$, los resultados fueron los siguientes:

V	Tiempo (s)	Dist. máx.
5	6.977e-05	0
10	0.0005990	1
15	0.0003238	1
20	0.0019908	2
25	0.0037496	2
30	0.0056796	2
35	0.0038654	2
40	0.0031046	2
45	0.2388005	2
50	0.1587331	3
55	1.1580761	3
60	0.1399792	3
70	0.5514237	3
80	4.3688325	3
100	0.1594807	3

Cuadro 1: Algunos tiempos de ejecución. Algoritmo BT con V variable, $m = 3$ y $k = 7$

Figura 1: Tiempo de ejecución vs cantidad de vértices. Algoritmo BT con $m = 3$ y $k = 7$

Se observa un fenómeno, cuanto menos, curioso en el comportamiento de este algoritmo. En todos los grafos analizados, cuando el valor de **k es muy pequeño**, asignar un vértice a un clúster "incorrecto" tiende a aumentar significativamente el diámetro. En estos casos, la poda asociada a fijar un diámetro máximo inicial permite realizar cortes de ramas desde etapas muy tempranas, lo que reduce notablemente el espacio de búsqueda. En el otro extremo, cuando **k es muy grande**, los clústeres resultantes son más pequeños y, por lo tanto, también lo son sus diámetros máximos. Dado que en estos casos la solución es trivial (o muy cercana a esta), el algoritmo vuelve a ejecutarse en tiempos muy breves.

Sin embargo, existe un punto crítico para ciertos valores intermedios de k en el que los tiempos de ejecución crecen exponencialmente. En este escenario, las técnicas de poda pierden gran parte de su efectividad y el algoritmo se aproxima a un enfoque de fuerza bruta, necesitando explorar muchas más combinaciones. Este comportamiento se presenta en prácticamente cualquier grafo, aunque el valor específico de k en el que ocurre varía según la estructura del mismo, su cantidad de vértices y aristas, y otras características topológicas. Naturalmente, dichas propiedades también influyen en la magnitud del crecimiento temporal observado.

Esta última situación pudo observarse en las mediciones realizadas para el grafo de 80 vértices, cuyo tiempo de ejecución resultó 27 veces mayor al observado con el grafo de 100 vértices, siendo éste un 25 % más grande. Para mostrarlo de una forma un poco más formal, realizamos mediciones

para el grafo de 55 vértices variando el valor de k entre 1 y 55, los resultados fueron los siguientes (se omiten algunos valores intermedios cuyos tiempos no eran de gran relevancia):

k	Tiempo (s)	Dist. máx.
1	0.0244545	4
4	0.0270152	3
5	0.0812611	3
6	0.2821773	3
7	1.6592603	3
8	20.099313	3
9	0.0055314	2
10	0.0048707	2
15	0.2722041	2
16	1.0092414	2
17	3.6662062	2
18	16.362976	2
19	0.0092642	2
20	0.0067912	2
30	0.0100477	1
40	0.0043908	1
55	0.0042057	0

Cuadro 2: Algunos tiempos de ejecución. Algoritmo BT, grafo con $V = 55$, $m = 3$ y k variable

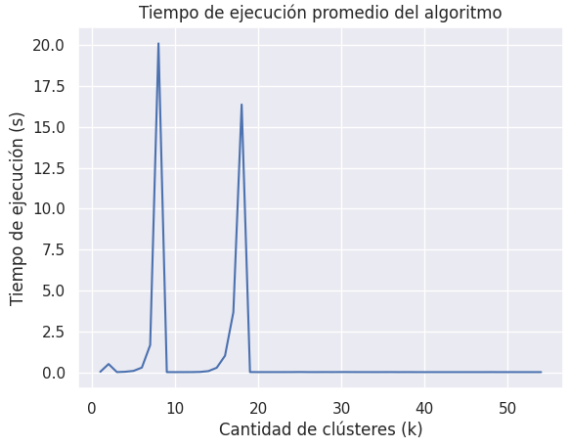


Figura 2: Tiempo de ejecución vs valor de k. Algoritmo BT, grafo con $V = 55$ y $m = 3$

En estas puede notarse lo observado en el ejemplo anterior. Cuando k es igual a 8 o 18, el algoritmo tarda exponencialmente más en encontrar una solución que con cualquier otro valor.

Por último, mostramos los tiempos de ejecución de las ya mencionadas pruebas de la cátedra. Cuando comenzamos con el desarrollo de este algoritmo, muchas de ellas tardaban horas en correr, pero a medida que incluimos las diversas podas, pudimos gradualmente reducir sus tiempos para que finalmente su ejecución dure tan solo unas pocas fracciones de segundo:

Archivo	k	Tiempo (s)	Dist. máx.
10_3	2	0.0002215	2
10_3	5	0.0001937	1
22_3	3	0.0009834	2
22_3	4	0.0010463	2
22_3	10	0.0012279	1
22_5	2	0.0009724	2
22_5	7	0.0011678	1
30_3	2	0.0015483	3
30_3	6	0.0016413	2
30_5	5	0.0018274	2
40_5	3	0.0034914	2
45_3	7	0.1225371	2
50_3	3	0.0042355	3

Cuadro 3: Pruebas de cátedra. Algoritmo BT

Efectivamente, las distancias máximas obtenidas son las especificadas en los resultados esperados, por lo que podemos confirmar la correcta optimalidad del algoritmo.

5.2. Programación Lineal

Corroboramos los tiempos de ejecución de la implementación de este modelo repitiendo las mismas mediciones que para Backtracking. Sin embargo, debimos omitir algunas debido a la gran cantidad de restricciones involucradas que provocaban tiempos de ejecución que, de esperar a su finalización, nos imposibilitarían entregar este trabajo a tiempo.

Para las pruebas con cantidad variable de vértices (entre 5 y 100, como se probó para el algoritmo anterior) decidimos ejecutar únicamente las primeras seis instancias, correspondientes a 5, 10, 15, 20, 25 y 30 vértices, con $k = 7$. Se recuerda que el modelo presenta $V^2 \cdot k$ restricciones, lo que implica que, en el caso de $V = 30$, se generen alrededor de 6300 restricciones, volviendo inviables las ejecuciones para valores mayores.

Seguidamente, los resultados obtenidos:

V	Tiempo (s)	Dist. máx.
5	0.0437557	0
10	0.1712076	1
15	0.7990758	1
20	1.5309391	2
25	2.5101041	2
30	313.21220	2

Cuadro 4: Algunos tiempos de ejecución. Algoritmo PL con V variable, $m = 3$ y $k = 7$

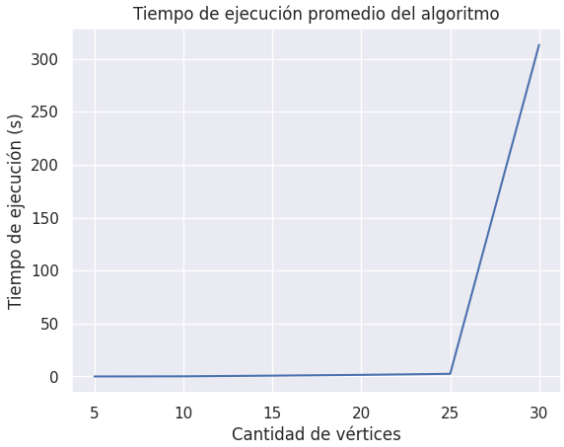


Figura 3: Tiempo de ejecución vs cantidad de vértices. Algoritmo PL con $m = 3$ y $k = 7$

Es evidente que, en este caso, el crecimiento en los tiempos de ejecución es exponencial respecto al tamaño de entrada. Con tan solo 5 vértices extra, la última prueba tarda 125 veces más que la realizada con el grafo de 25 vértices. Un resultado impresionante que demuestra la ineficiencia de esta implementación y nos confirma que las dimensiones (vértices y aristas) del grafo, a diferencia de en el algoritmo anterior, influyen de forma mucho más directa que su estructura.

Por otro lado, puede observarse que todos los diámetros obtenidos son iguales a los calculados anteriormente.

Para probar cómo se comporta el algoritmo variando el valor de k no pudimos utilizar el grafo con 55 vértices, ya que los tiempos de ejecución se volvían inmanejables. En cambio, decidimos utilizar el de 15 vértices y oscilar el valor de k entre 1 y 15:

k	Tiempo (s)	Dist. máx.
1	0.0783971	3
2	0.1401206	2
3	0.4366541	2
4	0.6042876	2
5	0.9748828	2
6	1.7984582	1
7	1.3968308	1
8	1.5335065	1
9	1.7733304	1
10	1.9600728	1
11	2.0600506	1
12	2.2034268	1
13	2.7742329	1
14	3.4008234	1
15	1.3600648	0

Cuadro 5: Algunos tiempos de ejecución. Algoritmo PL, grafo con $V = 15$, $m = 3$ y k variable

Figura 4: Tiempo de ejecución vs valor de k. Algoritmo PL, grafo con $V = 15$ y $m = 3$

En este caso no se da la situación del "k crítico" como pasa con el algoritmo anterior. Si bien es posible que algunas optimizaciones como el seteo de un diámetro máximo como restricción generen esos picos que pueden observarse para algunos valores, el crecimiento de los tiempos de ejecución es mucho más dependiente del valor de k, que hace crecer linealmente la cantidad de restricciones. Sin embargo, en casos triviales como el dado cuando $k = 15$, *Branch and Bounds* encuentra la solución con muchísima rapidez. Creemos que esto se debe más a optimizaciones propias de la metodología que a las implementadas por nosotros. No obstante, su funcionamiento es pésimo en situaciones "casi triviales", como la dada cuando $k = 14$, donde nuestro algoritmo por Backtracking encuentra soluciones prácticamente instantáneas.

Luego realizamos las mediciones de tiempos y optimalidad de las pruebas dadas por la cátedra. Los resultados de ellas fueron:

Archivo	k	Tiempo (s)	Dist. máx.
10_3	2	0.0590733	2
10_3	5	0.2693532	1
22_3	3	0.7095135	2
22_3	4	1.0887045	2
22_3	10	-	-
22_5	2	0.1264466	2
22_5	7	4.4743445	1
30_3	2	0.2977084	3
30_3	6	63.489293	2
30_5	5	3.1778523	2
40_5	3	2.1693797	2
45_3	7	2006.0416	2
50_3	3	2.3678537	3

Cuadro 6: Pruebas de cátedra. Algoritmo PL

Los tiempos de ejecución resultaron notoriamente elevados, por tal razón dedicaremos el próximo apartado a comparar empíricamente los resultados obtenidos con esta implementación frente a los del algoritmo previo. Sin embargo, y diferenciándose ligeramente de lo que habíamos observado con nuestras pruebas anteriores, el valor de k parece influir en mayor magnitud de lo pensado, generando que la prueba 45_3 con $k = 7$ haya tardado alrededor de media hora y no pudiésemos terminar de correr la asociada al archivo 22_3 con $k = 10$, por la cual esperamos durante más de una hora sin obtener resultados. Además, pudimos comprobar que la distancia máxima encontrada dentro de los clústeres es la esperada, por lo que podemos inferir que se cumple la optimalidad requerida.

Finalmente, para probar la eficiencia de las dos optimizaciones (seteo inicial de vértices y cota máxima inicial), bastante inusuales de poner en práctica a la hora de modelar utilizando Programación Lineal, decidimos correr algunas de estas últimas pruebas sin implementar ninguna de ellas y comparamos los tiempos obtenidos:

Archivo	k	Tiempo c/ opts. (s)	Tiempo s/ opts. (s)	Ganancia (%)
10_3	2	0.0590733	0.1291751	54.3
10_3	5	0.2693532	0.3959841	31.9
22_3	3	0.7095135	2.4074191	70.5
22_3	4	1.0887045	0.8583910	-21.2
22_5	2	0.1264466	0.2559329	50.6
30_3	2	0.2977084	1.0820386	72.5
30_5	5	3.1778523	3.2218072	1.4
40_5	3	2.1693797	11.982954	81.9
50_3	3	2.3678537	15.485355	84.7

Cuadro 7: Comparativa de tiempos con y sin optimizaciones en PL

Nuevamente, omitimos pruebas cuya ejecución sin ninguna optimización tarda tanto que no presenta sentido aguardar. A pesar de esto, las pruebas corridas son suficientes para demostrar que las dos optimizaciones representan una mejora decisiva, tal como era esperado. Raramente con el archivo 22_3 obtuvimos un rendimiento 21,2 % peor. No sabemos a ciencia exacta a qué podría deberse, pero es probable que, como mencionamos durante la explicación del modelo, la cantidad extra de restricciones sea perjudicial en casos como este.

5.3. Backtracking vs. Programación Lineal

A pesar de ser evidente la ganancia de rendimiento al elegir el algoritmo diseñado con Backtracking por sobre la implementación de Programación Lineal, decidimos contrastarlos lado a lado realizando cuadros comparativos en donde se especifica la ganancia porcentual de rendimiento recién mencionada.

A continuación, las diferencias observadas al correr las pruebas desarrolladas por nosotros:

V	Tiempo c/ BT (s)	Tiempo c/ PL (s)	Ganancia (%)
5	6.977e-05	0.1712076	99.96
10	0.0005990	0.7990758	99.92
15	0.0003238	1.5309391	99.98
20	0.0019908	0.8583910	99.77
25	0.0056796	2.5101041	99.77
30	0.0038654	313.21220	100.0

Cuadro 8: Comparativa de tiempos BT vs. PL con pruebas propias

Por otro lado, estas fueron las diferencias al comparar las pruebas de la cátedra:

Archivo	k	Tiempo c/ BT (s)	Tiempo c/ PL (s)	Ganancia (%)
10_3	2	0.0002215	0.0590733	99.63
10_3	5	0.0001937	0.2693532	99.93
22_3	3	0.0009834	0.7095135	99.86
22_3	4	0.0010463	1.0887045	99.90
22_5	2	0.0009724	0.1264466	99.23
22_5	7	0.0011678	4.4743445	99.97
30_3	2	0.0015483	0.2977084	99.48
30_3	6	0.0016413	63.489293	100.00
30_5	5	0.0018274	3.1778523	99.94
40_5	3	0.0034914	2.1693797	99.84
45_3	7	0.1225371	2006.0416	99.99
50_3	3	0.0042355	2.3678537	99.82

Cuadro 9: Comparativa de tiempos BT vs. PL con pruebas de la cátedra

Se pueden apreciar ganancias de prácticamente un 100 % de rendimiento en todas las pruebas. Esto demuestra que, a pesar de no ser muy eficiente, el algoritmo de Backtracking puede utilizarse sin mayores problemas en muchas situaciones realistas, mientras que la implementación de Programación Lineal es ineficiente en cualquier situación que se presente, incluso con grafos pequeños y cantidades de clústeres que, en otros casos, no generan grandes sobresaltos.

5.4. Aproximaciones

Como fue mencionado con anterioridad, el comportamiento (en el peor de los casos) exponencial de los algoritmos anteriores, hace que, para tamaños de entrada grandes, no sea posible ejecutarlos en un tiempo razonable. En cambio, los algoritmos de aproximación podrían encontrar buenas soluciones en tiempos muchísimo menores.

En este apartado se medirán los tiempos de ejecución para distintos tamaños de grafos y con distintos k, se comparará el rendimiento entre ellos y con los algoritmos óptimos en pos de calificar las aproximaciones.

5.4.1. Louvain

A continuación, se presentan los tiempos de ejecución para grafos generados con BA con valores chicos de V con un $m = 3$ y $k = 7$.

V	Tiempo (s)	Dist. máx.
50	0.742994	3
55	0.404340	4
60	0.208958	4
65	2.998783	4
70	0.441758	4
75	4.364293	4
80	0.849628	4

Cuadro 10: Tiempos de ejecución y distancia máxima para Louvain con V chico variable, $m = 3$ y $k = 7$

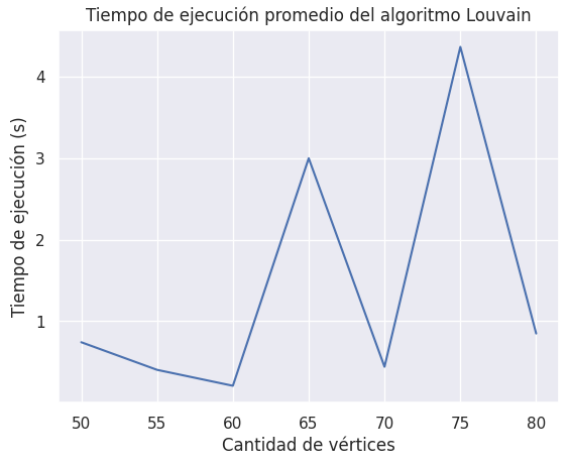


Figura 5: Tiempo de ejecución vs cantidad de vértices para aproximación Louvain

En estos ejemplos, nuevamente los tiempos de ejecución son oscilantes para distintos tamaños de grafo. Pero como se viene explicando a lo largo del informe, para algoritmos como el de Louvain es importante no sólo comparar cuánto demoran sino también qué tan buena es su resolución.

Pero antes, observemos el comportamiento de Louvain para $V = 15$ y k variable.

k	Tiempo (s)	Dist. máx.
1	0.0085756	3
2	0.0136570	3
3	0.0221817	3
4	0.0097763	2
5	0.0056495	2
6	0.0145018	2
7	0.0077626	2
8	0.0057671	2
9	0.0130598	2
10	0.0245866	2
11	0.0203964	2
12	0.0170364	2
13	0.0227410	2
14	0.0111072	2
15	0.0005130	0

Cuadro 11: Tiempos de ejecución y distancia máxima para Louvain con k variable, $V = 15$ y $m = 3$

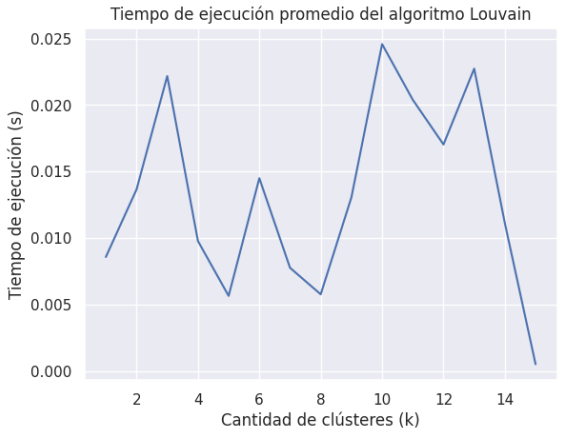


Figura 6: Tiempo de ejecución vs cantidad de clústeres para aproximación Louvain

Aquí nuevamente al variar k nos encontramos con tiempos oscilantes y bajos.

Al final de la sección de mediciones se compararán todos los resultados de este algoritmo con los de Backtracking para discutir su desempeño al aproximar la solución.

5.4.2. Nuestra propuesta

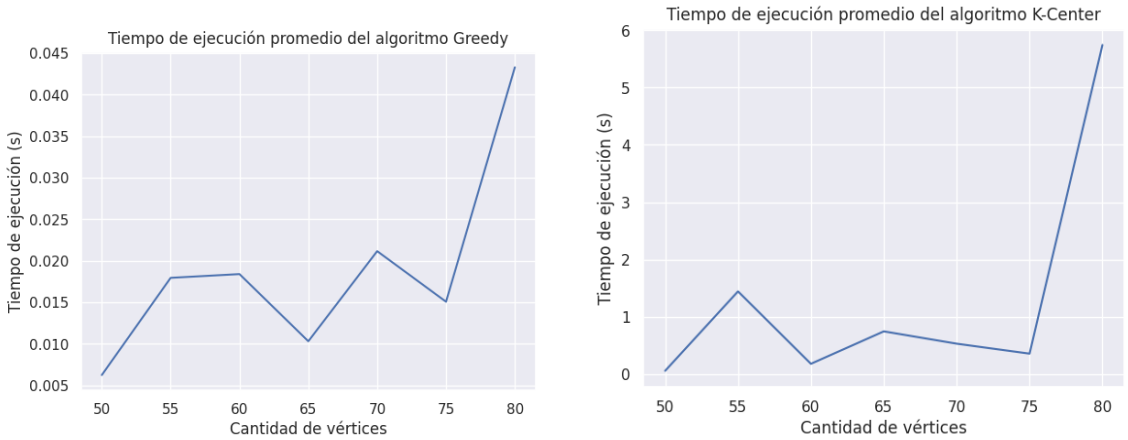
Las dos implementaciones expuestas tienen una base Greedy por lo que se esperan resultados de forma bastante veloz, incluso para instancias con grafos grandes.

A continuación se presentan los resultados obtenidos para ambas alternativas variando V entre 50 y 80 vértices.

V	Implementación 1		Implementación 2	
	Tiempo (s)	Dist. máx.	Tiempo (s)	Dist. máx.
50	0.008453	3	0.006251	4
55	0.005979	3	0.017956	4
60	0.010142	3	0.018405	4
65	0.014321	3	0.010321	4
70	0.014912	3	0.021169	4
75	0.014900	3	0.015079	4
80	0.024172	3	0.043306	4

Cuadro 12: Tiempos de ejecución y distancia máxima para las dos implementaciones con V chico variable, $k = 7$ y $m = 3$

A priori, la segunda implementación sería peor que la primera en términos de optimalidad mientras que temporalmente son semejantes.



(a) Tiempo de ejecución vs cantidad de vértices para aproximación Greedy

(b) Tiempo de ejecución vs cantidad de vértices para aproximación K-Center

Figura 7: Comparativa de tiempos de ejecución al variar V para Greedy y K-Center.

Para grafos más grandes, entre 100 y 500 vértices, los resultados fueron los siguientes

V	Implementación 1		Implementación 2	
	Tiempo (s)	Dist. máx.	Tiempo (s)	Dist. máx.
100	0.040834	4	0.026683	4
150	0.066669	4	0.047812	4
200	0.143488	4	0.173509	4
250	0.142658	4	0.123741	5
300	0.182546	4	0.186499	5
350	0.387798	5	0.308825	5
400	0.399669	5	0.341646	5
450	0.410011	5	0.422365	5
500	0.532024	5	0.524006	5

Cuadro 13: Tiempos de ejecución y distancia máxima para las dos implementaciones con V grande variable, $k = 7$ y $m = 3$

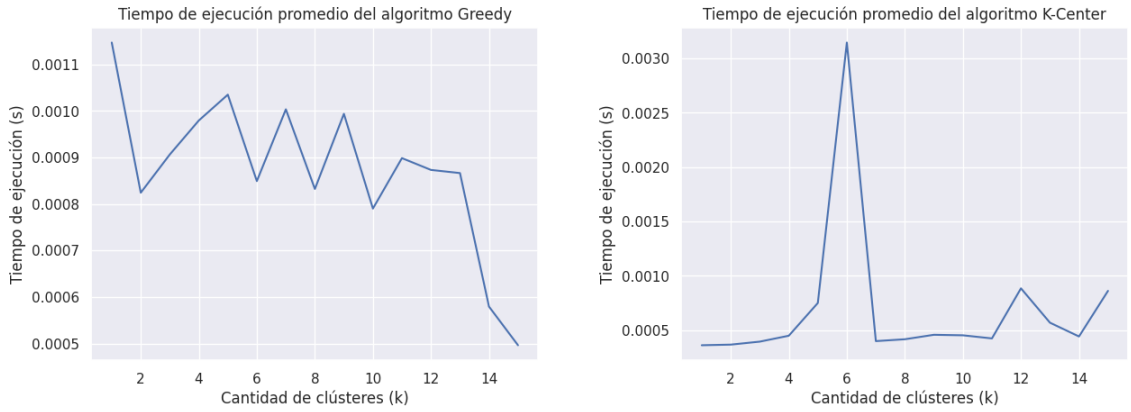
En estos casos, nuevamente los tiempos fueron similares pero los diámetros ya no son diferentes en todos los casos.

Cabe destacar que los tamaños medidos en esta prueba son completamente inmanejables para Backtracking y Programación Lineal y estas aproximaciones pudieron brindar resultados en fracciones de segundos.

A continuación, evaluamos el comportamiento de estas aproximaciones fijando V en 15 vértices como se ha hecho con los otros algoritmos y variando k.

k	Implementación 1		Implementación 2	
	Tiempo (s)	Dist. máx.	Tiempo (s)	Dist. máx.
1	0.0011470	3	0.0003609	3
2	0.0008242	2	0.0003668	3
3	0.0009063	2	0.0003941	3
4	0.0009794	2	0.0004484	2
5	0.0010350	2	0.0007495	2
6	0.0008491	2	0.0031454	2
7	0.0010035	1	0.0003988	2
8	0.0008323	1	0.0004160	2
9	0.0009938	1	0.0004570	2
10	0.0007901	1	0.0004522	2
11	0.0008985	1	0.0004236	1
12	0.0008732	1	0.0008842	1
13	0.0008664	1	0.0005683	1
14	0.0005794	1	0.0004415	1
15	0.0004963	0	0.0008611	0

Cuadro 14: Tiempos de ejecución y distancia máxima con k variable, $V = 15$ y $m = 3$



(a) Tiempo de ejecución vs cantidad de clústeres con aproximación Greedy

(b) Tiempo de ejecución vs cantidad de clústeres con aproximación K-Center

Figura 8: Comparativa de tiempos de ejecución al variar k para Greedy y K-Center.

Para esta prueba, notamos un comportamiento similar al que se observa variando V: los tiempos de ejecución son semejantes entre ambas implementaciones y, si bien en algunos casos las dos arrojan un diámetro igual, en los casos que difieren siempre el "peor" es la segunda implementación, basada en el problema de K-Center.

5.4.3. Louvain vs nuestra propuesta

A continuación, compararemos el rendimiento de las tres implementaciones de aproximaciones para un tamaño variable de vértices.

V	Louvain		Implementación 1		Implementación 2	
	Tiempo (s)	Dist. máx.	Tiempo (s)	Dist. máx.	Tiempo (s)	Dist. máx.
50	0.742994	3	0.008453	3	0.006251	4
55	0.404340	4	0.005979	3	0.017956	4
60	0.208958	4	0.010142	3	0.018405	4
65	2.998783	4	0.014321	3	0.010321	4
70	0.441758	4	0.014912	3	0.021169	4
75	4.364293	4	0.014900	3	0.015079	4
80	0.849628	4	0.024172	3	0.043306	4

Cuadro 15: Tiempos de ejecución y distancia máxima para Louvain y nuestra propuesta con V chico variable, $k = 7$ y $m = 3$

Es notable que los resultados de la Implementación 1 son los mejores. Como fue destacado antes, en tiempo los valores son similares a los de la Implementación 2, pero al comparar con Louvain son insignificantes. En cuanto al diámetro, Louvain y la Implementación 2 tuvieron casi el mismo desempeño.

Ahora, se evaluará comparativamente qué sucede al variar k y dejando V fijo en 55 vértices.

k	Louvain		Implementación 1		Implementación 2	
	Tiempo (s)	Dist. máx.	Tiempo (s)	Dist. máx.	Tiempo (s)	Dist. máx.
1	0.458959	4	0.010035	4	0.007368	4
2	0.406066	4	0.008740	4	0.007618	4
3	0.760219	4	0.006824	3	0.006530	4
4	0.415081	4	0.004357	3	0.007909	4
5	0.281336	4	0.005213	3	0.008987	4

Cuadro 16: Tiempos de ejecución y distancia máxima para Louvain y nuestra propuesta con k variable, $V = 55$ y $m = 3$

El resultado es análogo al de la tabla anterior, Louvain demora considerablemente más que los otros dos y los mejores diámetros los brinda la Implementación 1 aunque para k chico los tres devuelven el mismo resultado.

5.5. Backtracking vs aproximaciones

Aquí, compararemos directamente el diámetro óptimo obtenido mediante Backtracking y el de las aproximaciones, teniendo en cuenta también los tiempos de ejecución con el fin de determinar cuál sería más confiable y, en lo posible, rápido.

V	Backtracking		Louvain		Implementación 1		Implementación 2	
	Tiempo (s)	D. máx.	Tiempo (s)	D. máx.	Tiempo (s)	D. máx.	Tiempo (s)	D. máx.
50	0.054393	2	0.742994	3	0.008453	3	0.006251	4
55	1.441666	3	0.404340	4	0.005979	3	0.017956	4
60	0.176751	3	0.208958	4	0.010142	3	0.018405	4
65	0.742329	3	2.998783	4	0.014321	3	0.010321	4
70	0.528502	3	0.441758	4	0.014912	3	0.021169	4
75	0.353870	3	4.364293	4	0.014900	3	0.015079	4
80	5.744360	3	0.849628	4	0.024172	3	0.043306	4

Cuadro 17: Tiempos de ejecución y distancia máxima para Backtracking, Louvain y nuestra propuesta con V chico variable, $k = 7$ y $m = 3$

Para estos ejemplos, en todos los casos Louvain obtuvo un diámetro mayor en una unidad al óptimo con tiempos de ejecución aceptables en comparación a los de Backtracking. Dado que a

partir de $V > 80$, los algoritmos óptimos en algunas instancias se vuelven inmanejables temporalmente, consideramos que el *trade-off* ofrecido por Louvain es bastante tolerable. De todas formas, este algoritmo de aproximación tampoco llega a manejar valores mucho más grandes de V.

Por su parte, Greedy y K-Center demoran muchísimo menos. En particular, el primero obtiene el resultado óptimo en 6 de 7 pruebas, mientras que la segunda implementación brinda resultados muy parecidos a los de Louvain, aunque en un mejor tiempo.

Queda claro que entonces, el *trade-off* ofrecido por la Implementación 1 de nuestra propuesta de aproximación es altamente superior a las otras dos, lo que sería un buen augurio para las instancias con grafos más grandes que los algoritmos óptimos no podrían manejar.

Queda analizar qué sucede cuando el que varía es el k.

k	BT		Louvain		Implementación 1		Implementación 2	
	Tiempo (s)	D. máx.	Tiempo (s)	D. máx.	Tiempo (s)	D. máx.	Tiempo (s)	D. máx.
1	0.0783971	3	0.0085756	3	0.0011470	3	0.0003609	3
2	0.1401206	2	0.0136570	3	0.0008242	2	0.0003668	3
3	0.4366541	2	0.0221817	3	0.0009063	2	0.0003941	3
4	0.6042876	2	0.0097763	2	0.0009794	2	0.0004484	2
5	0.9748828	2	0.0056495	2	0.0010350	2	0.0007495	2
6	1.7984582	1	0.0145018	2	0.0008491	2	0.0031454	2
7	1.3968308	1	0.0077626	2	0.0010035	1	0.0003988	2
8	1.5335065	1	0.0057671	2	0.0008323	1	0.0004160	2
9	1.7733304	1	0.0130598	2	0.0009938	1	0.0004570	2
10	1.9600728	1	0.0245866	2	0.0007901	1	0.0004522	2
11	2.0600506	1	0.0203964	2	0.0008985	1	0.0004236	1
12	2.2034268	1	0.0170364	2	0.0008732	1	0.0008842	1
13	2.7742329	1	0.0227410	2	0.0008664	1	0.0005683	1
14	3.4008234	1	0.0111072	2	0.0005794	1	0.0004415	1
15	1.3600648	0	0.0005130	0	0.0004963	0	0.0008611	0

Cuadro 18: Tiempos de ejecución y distancia máxima para todos los algoritmos con k variable, $V = 15$ y $m = 3$

En este caso, a medida que aumenta la cantidad de clústeres k, los tiempos en Backtracking aumentan, aún con picos, pero pasando $k = 5$ son estrictamente mayores a 1 segundo, mientras que en Louvain se mantienen bajos. Nuevamente, vale analizar la relación entre la duración de la ejecución y su resultado. Variando k sigue sucediendo que el diámetro obtenido por Louvain es lo suficientemente aceptable teniendo en cuenta que para valores grandes de V y de k, Backtracking no podrá ejecutar con rapidez.

De todas formas, una vez más los algoritmos de aproximación que propusimos son notablemente mejores, tanto en velocidad como en aproximación a la solución en la mayoría de los casos. En particular, la Implementación 1 brinda el resultado correcto en 14 de las 15 pruebas.

Con estos resultados, se puede calcular la cota empírica de aproximación.

Con Louvain se obtuvo:

- Para $V = 15$ y $k = 2$ un diámetro igual a 3 siendo el óptimo igual a 2 $\Rightarrow \frac{D_{Louvain}}{D_{opt}} = 1{,}5$
- Para $V = 15$ y k entre 6 y 14 un diámetro igual a 2 siendo el óptimo igual a 1 $\Rightarrow \frac{D_{Louvain}}{D_{opt}} = 2$

Con la Implementación 1:

- Para $V = 15$ y $k = 6$ un diámetro igual a 2 siendo el óptimo igual a 1 $\Rightarrow \frac{D_{I_1}}{D_{opt}} = 2$

Con la Implementación 2:

- Para $V = 15$, $k = 2$ y $k = 3$ un diámetro igual a 3 siendo el óptimo igual a 2 $\Rightarrow \frac{D_{I_2}}{D_{opt}} = 1{,}5$

- Para $V = 15$ y k entre 6 y 10 un diámetro igual a 2 siendo el óptimo igual a 1 $\Rightarrow \frac{D_{Louvain}}{D_{opt}} = 2$

Es decir, para las tres aproximaciones obtuvimos una cota empírica igual a 2. En el caso particular de la segunda implementación basada en K-Center, ese resultado es consistente con el presentado en la parte teórica.

Lo único que resta entonces es comparar el desempeño de todos los algoritmos para las pruebas de la cátedra.

Archivo	k	Backtracking		Louvain		Implementación 1		Implementación 2	
		Tiempo (s)	D	Tiempo (s)	D	Tiempo (s)	D	Tiempo (s)	D
10_3	2	0.0002215	2	0.005502	2	0.000160	2	0.000396	2
10_3	5	0.0001937	1	0.005207	2	0.000245	1	0.000297	2
22_3	3	0.0009834	2	0.031236	3	0.000937	3	0.000796	3
22_3	4	0.0010463	2	0.021638	2	0.001386	3	0.000780	2
22_3	10	0.0012279	1	0.031086	2	0.001610	2	0.000875	2
22_5	2	0.0009724	2	0.033864	3	0.000795	2	0.000879	3
22_5	7	0.0011678	1	0.030728	2	0.001811	2	0.001005	2
30_3	2	0.0015483	3	0.074179	4	0.001323	3	0.001917	3
30_3	6	0.0016413	2	0.057971	2	0.002553	2	0.002299	2
30_5	5	0.0018274	2	0.142246	3	0.001748	2	0.002860	3
40_5	3	0.0034914	2	0.336473	3	0.003884	3	0.005283	3
45_3	7	0.1225371	2	0.101074	3	0.002789	3	0.005528	3
50_3	3	0.0042355	3	0.332877	4	0.003350	3	0.007576	4

Cuadro 19: Tiempos de ejecución y distancia máxima para Backtracking, Louvain y nuestra propuesta con los ejemplos de la cátedra

En este último cuadro parecería que para esos grafos y esos k en particular, todas las aproximaciones fueron bastante buenas. Si bien los tiempos de las aproximaciones Greedy fueron menores, los resultados de los diámetros no fueron tanto más precisos que los de Louvain. Exactamente, la Implementación 1 obtuvo el resultado óptimo en 7 de 13 pruebas, la Implementación 2 en 4 y Louvain en 3.

En todos los casos, los resultados son consistentes con las cotas empíricas calculadas previamente.

Basándonos en estos últimos resultados, podemos decir que quizás las pruebas emuladas con BA para $m = 3$ favorecían al algoritmo que propusimos como aproximación en primera instancia. Pero, en topologías más diversas, si bien sigue dando mejores resultados (más cercanos al óptimo o en mayor cantidad de casos igual al óptimo) con mejores tiempos ese algoritmo, las otras dos aproximaciones son competentes. Incluso, quizás haya configuraciones de grafos en los que alguno de ellos sea mejor que la Implementación 1.

6. Conclusiones

A lo largo de este trabajo hemos estudiado en profundidad el problema de *Clusterización por bajo diámetro*, tanto en su versión de optimización como en su versión de decisión.

Pudimos anticipar que nos encontraríamos con desafíos bastante grandes dado que el problema tenía aspecto de NP-Difícil/NP-Completo, lo que obligaba a resolver su versión de optimización en tiempo exponencial.

Pero antes de codear soluciones, tuvimos que demostrar su pertenencia a NP y que era NP-Completo en su versión de decisión. Gracias a ejemplos vistos con anterioridad, dicha demostración resultó bastante directa una vez que hallamos el problema NP-Completo indicado para reducirlo al nuestro.

Logrado ese primer objetivo, procedimos a plantear los algoritmos que dan soluciones óptimas. En primer lugar, desarrollamos Backtracking. Su implementación fue fundamental para aplicar lo aprendido, las heurísticas y podas halladas en los algoritmos subsiguientes. Luego, pudimos plantear el modelo de Programación Lineal y las aproximaciones. Cada uno de los algoritmos presentó su propio desafío como fue comentado en cada sección.

Ya con todos los programas listos, procedimos a hacer las mediciones.

Pudimos corroborar que los **algoritmos óptimos**:

- no son escalables por su naturaleza exponencial.

  Si bien es difícil precisar la combinación de valores de entrada para los que se vuelven inmanejables, fue claro que la tendencia temporal global era creciente para una cantidad de vértices mayor en ambos casos y, particularmente en Programación Lineal, al aumentar k también era clara la tendencia creciente.

- Las optimizaciones aplicadas generaron un impacto considerable en los tiempos de ejecución, aunque esto dista de decir que su complejidad temporal mejoró.

En cuanto a los **algoritmos de aproximación**:

- en detrimento de su optimalidad, son altamente escalables.

  Si bien Louvain para algunos casos también se vio limitado ante un tamaño creciente de vértices, su desempeño fue competitivo en los ejemplos de la cátedra que consideramos que fueron más variados.

  En cuanto a la aproximación Greedy, en algunos casos fue muy buena, coincidiendo con los resultados de los algoritmos óptimos

- las cotas empíricas halladas para las tres implementaciones fueron de la clase 2-aproximación aunque en la mayoría de los ejemplos resultaron mejores.

En resumidas cuentas:

- los algoritmos óptimos resultan adecuados para grafos pequeños y así aprovechar su garantía de optimalidad. En particular, sugerimos Backtracking porque como se vio, el modelo de Programación Lineal es incluso más limitado en el tamaño de entrada soportado.

- para grafos más grandes, es necesario optar por algoritmos de aproximación como los estudiados ya que ofrecen un *trade-off* aceptable entre la calidad de la solución y el tiempo que demoran en obtenerla. Dependiendo de la topología de la red, alguno puede resultar mejor que otro.

Por la naturaleza de este problema, es muy difícil dar complejidades exactas y a partir de ellas inferir el comportamiento que tendrán en la práctica los algoritmos. Hay muchas variables que entran en juego y determinar cotas precisas y universales escapa completamente al alcance de la materia.

Lo que sí podemos afirmar es que, una vez más, poner en práctica los temas estudiados en problemas más complejos, hacer un análisis diverso, con diferentes enfoques y, sobre todo, ver resultados numéricos de su desempeño, nos permitió lograr un entendimiento más profundo del alcance de los algoritmos óptimos, las ventajas de los aproximados y a estudiar con cuidado sus limitaciones.