NP-COMPLETOS

Autores
Cledy Elizabeth Becerra Sipiran
Massiel Oviedo Sivincha
Harold Alejandro Villanueva Borda

Tutor
Rensso Victor Hugo Mora Colque

Universidad Católica San Pablo
Análisis y Diseño de Algoritmos
Arequipa, Perú
Junio de 2022

ÍNDICE

1. Introducción

Los llamados problemas fáciles o tratables pueden resolverse mediante algoritmos informáticos que se ejecutan en tiempo polinomial. Los algoritmos para resolver problemas difíciles o intratables, por otro lado, requieren tiempos que son funciones exponenciales del tamaño del problema n. Los algoritmos de tiempo polinomial se consideran eficientes, mientras que los algoritmos de tiempo exponencial se consideran ineficientes, porque los tiempos de ejecución de estos últimos crecen mucho más rápidamente a medida que aumenta el tamaño del problema.

Si un problema es NP y todos los demás problemas NP son reducibles a él en tiempo polinomial, el problema es NP-completo. Por lo tanto, encontrar un algoritmo eficiente para cualquier problema NP-completo implica que se puede encontrar un algoritmo eficiente para todos esos problemas, ya que cualquier problema que pertenezca a esta clase se puede reformular en cualquier otro miembro de la clase. No se sabe si alguna vez se encontrarán algoritmos de tiempo polinomial para problemas NP-completos, y determinar si estos problemas son tratables o intratables sigue siendo una de las cuestiones más importantes de la informática teórica. Cuando se debe resolver un problema NP-completo, un enfoque es usar un algoritmo polinomial para aproximar la solución; la respuesta así obtenida no será necesariamente óptima pero será razonablemente cercana.[**1**]

2. Marco Teórico

2.1. Problemas Computacionales

Todos los algoritmos que conocemos son los algoritmos que toman tiempo polinomial y los algoritmos que toman tiempo exponencial.

TIEMPO POLINOMIAL	TIEMPO EXPONENCIAL
Búsqueda Lineal - n	O/I KnapSack - 2^n
Búsqueda Binaria - $\log n$	Agente Viajero - 2^n
Insertion Sort - n^2	Suma de SubConjuntos - 2^n
Merge Sort - $n \log n$	Coloreo de Grafos - 2^n
Multiplicación Matricial - n^3	Circuito Hamiltoniano - 2^n

2.1.1. Alcance a investigar

Veamos un ejemplo. Tenemos un algoritmo de búsqueda lineal que toma $\mathcal{O}(n)$ y más rápido que eso es una búsqueda binaria que $\mathcal{O}(\log n)$. Estamos en busca de un algoritmo que es mucho más rápido que este que es del orden $\mathcal{O}(1)$. De manera similar para estos problemas de tiempo exponencial queremos un método más rápido y fácil para resolverlos, como algoritmos de tiempo polinomial porque 2^n o n^n es mucho mayor que el tiempo polinomial, incluso cualquier valor de n en n^3, aunque n sea muy grande, siempre es menor que 2^n.

Debemos hacer un trabajo de tal manera que intente mostrar las similitudes entre los problemas, de modo que dado un algoritmo para un problema, todos los demás problemas también deberían resolverse, no estaremos haciendo un trabajo de investigación individualmente en todos y cada uno. Para eso nosotros debe existir una relación entre ellos asociación, así que tenemos que mostrar que las propiedades que están teniendo son similares. Una segunda cosa es cuando no puede escribir algoritmos determinista para ellos, pero ¿Por qué no escribir algoritmos no deterministas para ellos?, ¿Qué son estos? Veamos.

2.2. Algoritmos No Deterministas

Vea que los algoritmos que escribimos generalmente son deterministas, entonces, ¿Qué significan todas y cada una de las declaraciones? ¿Cómo funcionan? Lo sabemos claramente. Escribimos las declaraciones que estamos seguros de cómo funcionan para que sepamos cómo funciona el algoritmo, por lo que es determinista. Si queremos escribir un algoritmo no determinista, eso significa que no sabemos cómo funcionan, pero, ¿cómo podemos escribir un algoritmo sin saber cómo funciona? Puedo saber la mayoría de las declaraciones pero algunas de las declaraciones para las que aún estamos investigando cómo hacerlas polinómicas, las dejamos en blanco y decimos que esto es no determinista.

Así que cuando llegue a saber cómo se debe llenar, llenaré esas declaraciones para en un algoritmo no determinista, también la declaración puede ser determinista, de esta manera podemos preservar nuestro trabajo de investigación para que en el futuro alguien más esté trabajando en esto el mismo problema puede hacer que esa parte no determinista sea determinista.

2.2.1. Escritura de un Algoritmo No Determinista

Algoritmo `NSearch`(A, n, key)**:**
- j = choice();
- **if** *key = A[j]* **then**
  - write(j) success();
- write(0) failure();

2.2.2. Declaraciones conocidas

choose(), success() y failure() son las declaraciones conocidas que usamos para escribir algoritmos no deterministas, por lo que estas declaraciones no son deterministas y asumimos que todas estas declaraciones toman solo una unidad de tiempo $\mathcal{O}(1)$. Ahora la pregunta es ¿Cómo se llegó a saber que el elemento clave está presente en el índice j?, eso es lo que es no determinismo. Así podemos escribir algoritmo no determinista para estos algoritmos de toma de tiempo exponencial de tiempo polinomial que es lo que escribiremos en algoritmos de tiempo polinomial. Entonces con esto definimos dos clases aquí: P y NP.

2.3. Clases P y NP

2.3.1. Clase P

Informalmente, la clase P es la clase de problemas de decisión que se pueden resolver mediante algún algoritmo dentro de un número de pasos acotados por algún polinomio fijo en la longitud del aporte. A Turing no le preocupaba la eficiencia de sus máquinas, sino su preocupación era si podían simular algoritmos arbitrarios con el tiempo suficiente. Sin embargo, resulta que las máquinas de Turing generalmente pueden simular modelos de computadoras más eficientes (por ejemplo, máquinas equipadas con muchas cintas o un aleatorio ilimitado acceso a la memoria) elevando al cuadrado o al cubo el tiempo de cálculo como máximo.

Así P es un clase robusta y tiene definiciones equivalentes sobre una gran clase de modelos de computadora. Aquí seguimos la práctica estándar y definimos la clase P en términos de máquinas de Turing. Formalmente los elementos de la clase P son lenguajes. Sea $\sum$ un alfabeto finito (es decir, un conjunto finito no vacío) con al menos dos elementos, y sea $\sum *$, el conjunto de cuerdas finitas sobre $\sum$. Entonces un lenguaje sobre $\sum$ es un subconjunto L de $\sum *$. Cada Máquina de Turing M tiene un alfabeto de entrada asociado $\sum$. Para cada cadena w en $\sum *$ hay un cálculo asociado con M con entrada w. Decimos que M acepta w si este cálculo termina en el estado de aceptación. Tenga en cuenta que M no acepta w si este cálculo termina en el estado de rechazo, o si el cálculo falla para terminar. El lenguaje aceptado por M, denotado $L(M)$, tiene un alfabeto asociado $\sum$ y está definido por [**2**]:

$$L(M) = \{\ w\ \in \sum * \mid\ M \text{ acepta } w\}. \tag{1}$$

Denotamos por $t_M(w)$ el número de pasos en el cálculo de M en la entrada w. Si este cálculo nunca se detiene, entonces $t_M(w) = \infty$. Para $n \in N$ nosotros indicamos por $TM(n)$ el tiempo de ejecución del caso más desfavorable de M; eso es

$$TM(n) = max\{t_M(w) \mid\ w\ \in \sum{}^n\} \tag{2}$$

Donde $\sum^n$ es el conjunto de todas las cadenas sobre $\sum$ de longitud n. Decimos que M corre en tiempo polinomial si existe k tal que para todo n, $TM(n) \leq nk + k$. Ahora definimos la clase P de lenguajes por

$$P = \{L \mid L = L(M) \text{ para alguna Maquina de Turing que corre en tiempo polinomial}\}. \quad (3)$$

2.3.2. Clase NP

La notación NP significa "tiempo polinomial no determinista", ya que originalmente NP se definió en términos de máquinas no deterministas (es decir, máquinas que tienen más de un movimiento posible desde una configuración dada). Sin embargo, ahora es común dar una definición equivalente usando la noción de una relación de verificación, que es simplemente una relación binaria $R \subseteq \sum * \sum *$ para algunos alfabetos finitos $\sum$ y $\sum 1$. Asociamos con cada una de estas relaciones R un lenguaje L_R sobre $\sum \cup \sum 1 \cup \{\#\}$ definido por [**2**]:

$$L_R = \{w\#y \mid R(w, y)\} \quad (4)$$

Donde el símbolo no está en $\sum$. Decimos que R es polinomial en tiempo si y solo si $L_R \in P$.

Ahora definimos la clase NP de lenguajes por la condición de que un lenguaje L sobre $\sum$ está en NP si y sólo si hay un $k \in N$ y una relación de verificación R en tiempo polinomial tal que para todo $w \in \sum *$,

$$w \in L \iff \exists y(|y| \leq |w|^k \text{ y } R(w, y)) \quad (5)$$

Donde $|w|$ y $|y|$ denote las longitudes de w y y, respectivamente.

2.4. NP-Completos

Estos problemas se llaman NP-Completos (NPC), porque se dice que contienen información completa sobre todos los problemas en NP. El resultado inmediato de esto es que si uno puede resolver un problema de NPC de manera eficiente, entonces todos los problemas de NP pueden resolverse de manera eficiente. En general, hay dos tipos

de problemas NPC llamados problemas de decisión y problemas de optimización. Un problema de decisión es un problema NPC donde la respuesta es sí o no, mientras que el problema de optimización es la mejor solución para un problema dado.

Ahora, dado un problema de NPC, P_1, uno puede probar que un problema, P_2, es NP-Completo si es más difícil o tan difícil como P_1. El P_2 está probado como NP-Complete por [**3**]:

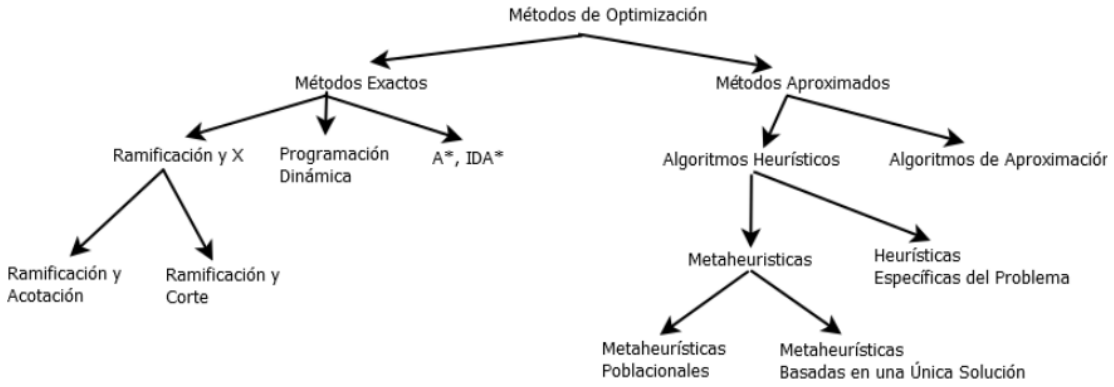
1. Mostrar que está en NP.
2. Construir una transformación, que transforma las instancias, I_1, de P_1 en instancias, I_2, de P_2.
   - Esta transformación debe tener un tiempo de ejecución polinomial
   - Si hay una solución para I_2 entonces también hay una solución para I_1.
   - Si hay una solución para I_1 entonces también hay una solución para I_2.
   - La transformación debe mostrar que P_2 es más difícil o tan difícil como P_1, porque P_1 puede resolverse resolviendo P_2.

2.5. Métodos de Optimización

Dependiendo de la complejidad del problema, puede ser resuelto por un método exacto o un método aproximado. Los métodos exactos obtienen soluciones óptimas y garantizan su optimalidad. Para problemas NP-completos, los algoritmos exactos son algoritmos de tiempo no polinomiales (a menos que P = NP). Los métodos aproximados o heurísticos generan soluciones de alta calidad en un tiempo razonable para uso práctico, pero no hay garantía de encontrar una solución óptima global. [**4**]

2.5.1. Métodos Exactos

En la clase de Métodos Exactos se pueden encontrar los siguientes algoritmos clásicos: programación dinámica, la familia de algoritmos ramificación y X (ramificación y límite,



ramificación y corte, ramificación y precio) desarrollado en la comunidad de investigación de operaciones, restricción programación, y la familia de algoritmos de búsqueda A (A, IDA—profundización iterativa algoritmos) desarrollados en la comunidad de inteligencia artificial. Aquellos métodos enumerativos pueden verse como algoritmos de búsqueda en árbol. La búsqueda se lleva en todo el espacio de búsqueda interesante, y el problema se resuelve subdividiendo en problemas más simples.[**4**]

2.5.2. Métodos Aproximados

En la clase de métodos aproximados, se pueden distinguir dos subclases de algoritmos: algoritmos de aproximación y algoritmos heurísticos. A diferencia de la heurística, que por lo general, encuentran soluciones razonablemente "buenas.[en] un tiempo razonable, los algoritmos de aproximación brindan una calidad de solución comprobable y límites de tiempo de ejecución comprobables.

Las heurísticas encuentran soluciones buenas en instancias de problemas de gran tamaño. Ellos permiten obtener un rendimiento aceptable a costos aceptables en una amplia gama de problemas. En general, las heurísticas no tienen garantía de aproximación sobre las soluciones obtenidas. Se pueden clasificar en dos familias: heurísticas específicas y metaheurísticas. Las heurísticas específicas se adaptan y diseñan para resolver un problema y/o instancia específica. Las metaheurísticas son algoritmos de propósito general que se pueden aplicar para resolver casi cualquier problema de optimizacion. Pueden verse como metodologías generales de nivel superior que puede usarse como una estrategia de guía en el diseño de heurísticas subyacentes para resolver problemas

específicos, problemas de optimización.[**4**]

Algoritmos de Aproximación En los algoritmos de aproximación, hay un garantía sobre el límite de la solución obtenida a partir del óptimo global. Un ϵ-algoritmo de aproximación a genera una solución aproximada no menor que un factor ϵ veces la solución óptima s

DEFINICIÓN **ϵ -Algoritmo de Aproximación** Un algoritmo tiene una aproximación factor ϵ si su complejidad temporal es polinomial y para cualquier instancia de entrada produce un solución a tal que,

$$a \leq \epsilon \cdot s \text{ if } \epsilon > 1$$

$$\epsilon \cdot s \leq a \text{ if } \epsilon < 1$$

Donde s es la solución óptima global, y el factor ϵ define la garantía de ejecución relativa. El factor ϵ puede ser una constante o una función del tamaño de la instancia de entrada. Un ϵ-algoritmo de aproximación genera una garantía de rendimiento absoluta ϵ, si se prueba la siguiente propiedad:

$$(s - \epsilon) \leq a \leq (s + \epsilon)$$

3. Problema de SubGrafos Densos

3.1. Conceptualización

3.1.1. Definición General

En teoría de grafos, la densidad de un grafo es una propiedad que determina la proporción de aristas que posee. Un grafo denso es un grafo en el que el número de aristas es cercano al número máximo de aristas posibles, es decir, a las que tendría si el grafo fuera completo. Al contrario, un grafo disperso es un grafo con un número de aristas muy bajo, es decir, cercano al que tendría si fuera un grafo vacío. [**5**]

3.1.2. Definición Formal

Sea $G = (V, E)$ un grafo simple (sin bucles) con $n = |V|$ vértices y $m = |E|$ aristas.
Si el grafo es no dirigido, su densidad Δ se define formalmente como [**5**]:

$$\Delta = \frac{m}{n(n-1)/2} = \frac{2m}{n(n-1)}$$

La densidad varía entre 0, para grafos vacíos con $m = 0$, y 1, para grafos completos con el máximo número de aristas posibles, a saber, $m = n(n-1)/2$.
Si el grafo es dirigido, su densidad se define como:

$$\Delta = \frac{m}{n(n-1)}$$

En este caso, la densidad alcanza valor 1 para el máximo número de aristas posibles en un grafo dirigido completo, a saber, $m = n(n-1)$.

Si el grafo es además ponderado, sean $W = \{w_1, \ldots, w_m\}$ los pesos de las aristas, entonces la densidad del grafo se define como el promedio de los valores asignados a las aristas:

$$\Delta = \frac{\sum_{i=1}^{m} w_i}{n(n-1)}$$

3.1.3. Problema en Ciencias de la Computación

En ciencias de la computación la noción de subgrafos altamente conectados aparece con frecuencia. Esta noción se puede formalizar de la siguiente manera. Sea $G = (E, V)$ un grafo no dirigido y sea $S = (E_S, V_S)$ser un subgrafo de G. Entonces la densidad de S se define como

$$d(S) = \frac{|E_S|}{|V_S|}$$

El problema del subgrafo más denso es el de encontrar un subgrafo de máxima densidad. En 1984, Andrew V. Goldberg desarrolló un algoritmo de tiempo polinomial para encontrar el subgrafo de densidad máxima utilizando una técnica de flujo máximo . Esto ha sido mejorado por Gallo, Grigoriadis y Tarjan en 1989 para funcionar en tiempo $\mathcal{O}(nm \log(\frac{n^2}{m}))$. [**6**]

3.1.4. SubGrafo Más Denso

Hay muchas variaciones en el problema del subgrafo más denso. Uno de ellos es el problema del subgrafo k más denso, donde el objetivo es encontrar el subgrafo de máxima densidad en exactamente k vértices. Este problema generaliza el problema del Clique y, por lo tanto, es NP-hard en grafos generales. Existe un algoritmo polinomial que aproxima el subgrafo k más denso dentro de una proporción de $n^{\frac{1}{4}+c}$ para cada $c > 0$ si bien no admite una $n^{\frac{1}{poly \log \log n}}$-aproximación en tiempo polinomial a menos que la hipótesis del tiempo exponencial sea falsa. Bajo una suposición más débil de que, no existe PTAS (Polynomial-time approximation scheme) para el problema.

El problema sigue siendo NP-hard en grafos bipartitos y grafos cordales, pero es polinomial para árboles y grafos divididos. Está abierto si el problema es NP-hard o polinomial en grafos de intervalos (propios) y grafos planos; sin embargo, una variación del problema en el que se requiere conectar el subgrafo es NP-hard en grafos planos. [**6**]

3.2. Aplicación

Para grafos dirigidos, Zeleny (1941) definió el índice de asociación de un nodo como la diferencia entre la densidad o media de la ≪intensidad≫ total de la red, y el grado de

salida o ≪elecciones≫ realizadas por el nodo. Denotemos $Is(v)$ el índice de asociación del nodo v. Entonces lo anterior se define formalmnte como:

La densidad se utiliza en análisis de redes sociales desde sus inicios en los años 1950 al menos a partir de Kephart (1950) y Proctor y Loomis (1951) (estos últimos responsables de la centralidad de grado). La densidad es una propiedad relevante para las redes sociales representadas como grafos, que se puede considerar como una medida de centralización de la red. Bott (1990) la propuso como una forma de cuantificar los niveles de ≪entrelazado≫ de una red, y Barnes (1969) para determinar la ≪estrechez≫ de las uniones de redes empíricas, importante en modelos de bloque y otras técnicas algebraicas de análisis. Además, la densidad de un subgrafo sirve para evaluar la cohesión de subgrupos dentro de una red social.[**7**]

3.3. Demostración Generalizada del Problema NP-Completo

3.3.1. Problema

Dada el grafo $G = (V, E)$ y dos enteros a y b. Sea a un conjunto de varios vértices de G tales que hay al menos b aristas entre ellos se conoce como subgrafo denso del grafo G.

3.3.2. Explicación

Para probar el problema del subgrafo denso como NP-completo por generalización, vamos a probar que es una generalización de un conocido problema NP-completo. En este caso, vamos a tomar a Clique como el problema conocido que ya se sabe que es NP-completo, y se explica en Prueba de que Clique es un NP-Completo y necesitamos mostrar la reducción de Clique → Subgrafo Denso.
Nótese que Clique es un subconjunto de vértices de un grafo no dirigido tal que cada dos vértices distintos en el clique son adyacentes.

3.3.3. Prueba

1. **Conversión de entrada:** necesitamos convertir la entrada de Clique a la entrada del Subgrafo denso.

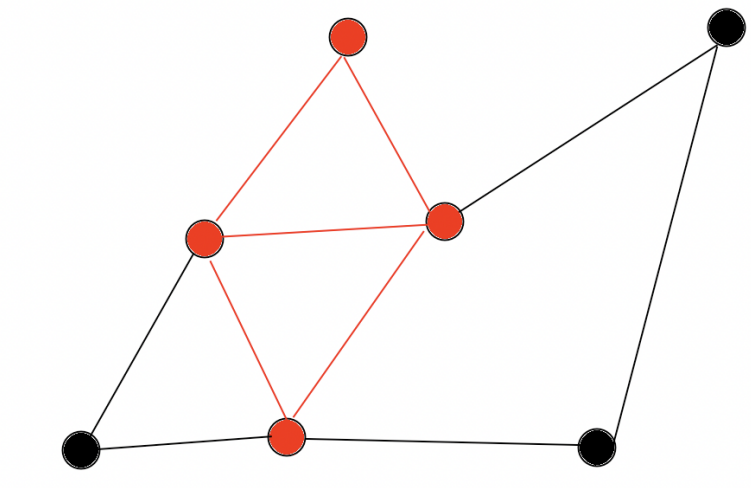
Clique Input: un grafo no dirigido $G(V, E)$ y un entero k.
Dense-Subgraph Input: un grafo no dirigido $G'(V, E)$ y dos enteros a y b.
Vamos a transformar la entrada de Clique para Dense Subgraph tal que

- $G' = G(V, E)$
- $a = k$
- $b = (k * (k - 1))/2$

Esta conversión tomará un tiempo $\mathcal{O}(1)$, por lo que es de naturaleza polinomial.

2. **Conversión de salida:** necesitamos convertir la solución de Dense Graph a la solución del problema Clique.
   La solución del grafo denso dará como resultado un conjunto a que sería un Clique de tamaño k como $k = a$. Por lo tanto, Clique puede utilizar la salida directa de Dense Graph. Dado que no se requiere conversión, nuevamente es de naturaleza polinomial.

3. **Corrección:** Hemos restringido el rango del valor de entrada b tal que $(k \mid 2)$ con valor como $(k * (k–1))/2$.
   Ahora estamos buscando un subgrafo que tenga k vértices y que estén conectados por al menos $(k * (k–1))/2$ aristas.



- Dado que en un grafo completo, n vértices pueden tener como máximo $(n * (n–1))/2$ aristas entre ellos, entonces podemos decir que necesitamos encontrar un subgrafo de k vértices que tengan exactamente $(k * (k–1)))/2$ aristas, lo que significa que el grafo de salida debe tener una arista entre cada par de vértices que no es más que una camarilla de k vértices.
- De manera similar, un Clique de k vértices en un grafo $G(V, E)$ debe tener $(k * (k–1))/2$ aristas que no es más que el subgrafo denso del grafo $G(V, E)$.

Entonces, esto significa que Dense-Subgraph tiene una solución $\iff$ Clique tiene una solución. La reducción completa toma un tiempo polinomial y Clique es NP completo, por lo que Dense Subgraph también es NP completo.

4. **Conclusión:** Por lo tanto, podemos concluir que Dense-Subgraph es NP-Completo. [**8**]

3.4. Demostración Formal del Problema NP-Completo

Se tiene lo siguiente:

Teorema 1. $(k; f(k)) - DSP$ es NP-completo para $f(k) = \theta(k^{1+e})$ donde e puede ser alguna constante positiva menor que uno.

Teorema 2. $(k; f(k)) - DSP$ es NP-completo para $f(k) = ek^2/v^2(1 + O(v^{e1}))$ donde e puede ser cualquier constante positiva menor que uno.

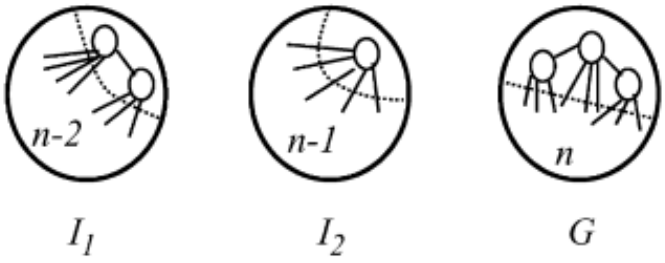
Es obvio que $(k; f(k)) - DSP$ está en NP. Para probar que está en NP-hard, reducimos un problema Clique a $(k; f(k)) - DSP$. Aquí consideramos el problema Clique restringido que pregunta si existe un grafo completo de n-vértices (n-clique) en un grafo de 2n-vértice dado $G = (V, E)$. Se puede demostrar fácilmente que este problema del Clique todavía es NP-completo por reducción del problema general k-clique de la siguiente manera:

Para un grafo de entrada I de n vértices, agregamos un grafo completo de $nk - vértices$ k_{n-k}, conexión bipartita completa entre I y k_{n-k} y k vértices aislados. El grafo consecuente tiene 2n vértices, y tiene un Clique n si y solo si existe un Clique k en I.

Sea $f(k) = nm(2n1) + n(n1)/2$, donde m es un polinomio en n y determinado luego. Construya un grafo $H = (V', E')$ compuesto por una copia G de $G = (V; E)$ y m grafos completos, cada uno de los cuales tiene 2n vértices. H tiene $|V'| = 2n(m+1)$ vértices y $|E'| = |E| + nm(2n1)$ aristas en total. Luego establezca $k = 2nm+n$. Esta construcción de H se puede hacer en tiempo polinomial obviamente.

Lema. *Supongamos que hay m grafos completos $I_1, \ldots, I_m$ de 2n vértices y un (no necesariamente completo) grafo G de 2n vértices. Tome 2nm+n vértices entre esos $2nm + 2n$ unos. Entonces, el número de aristas inducidas se vuelve máximo cuando tomamos todos los vértices de 2nm de $I_1, \ldots, I_m$ (y otros n de G).*

Prueba. Supongamos que tomamos 2nmd vértices de $I_1, \ldots, I_m$ y n+d vértices de G. (Vea la Fig. 1 para $m = 2$ y $d = 3$.) Entonces uno puede ver fácilmente que el número de aristas incluidas no disminuye si abandonamos (cualquiera) d (tres en la Fig. 1) vértices de G y tome d vértices de I_1 e I_2, ya que ambos completos. Esta declaración es obviamente cierto para m y d generales. Así sostiene el lema.



Este lema muestra que un subgrafo de k-vértice más denso de H consta de todos los Clique de 2n vértices y un subgrafo de n vértices de G. Si el número de aristas en este subgrafo es $f(k)$, entonces el número de aristas en el subgrafo tomado de G debe ser $\frac{n(n1)}{2}$; es decir, debe ser una camarilla. Por el contrario, si G tiene una camarilla de tamaño n, entonces obviamente es posible tomar un subgrafo de k-vértices de f(k) aristas.

Para la demostración del Teorema 1, queda demostrar que dado $0 < e < 1$, se puede elegir $f(k)$ de modo que cumpla la condición $f(k) = \theta(k^{1+e})$. Para cualquier fijo $0 < e < 1$

dado, elegimos $m = n^{\frac{1}{e-1}}$, de modo que $k = 2n^{\frac{1}{e}} + n$ y $f(k) = 2n^{1+\frac{1}{e}}n^{\frac{1}{e}} + \frac{n(n1)}{2}$. Así, en términos generales, $\frac{k^{1+e}}{2^{e+1}} < f(k) < \frac{k^{1+e}}{2^{e1}}$, es decir, $f(k) = \theta(k^{1+e})$. Entonces, como $f(k) = |E'|k^2/|V'|^2(1 + O(m^1))$ y $m = O(|V'|^{1e})$, también obtenemos el Teorema 2 del Teorema 1. [**9**]

3.5. Investigación

Kan nan y Vinay fueron los primeros en introducir la noción de densidad y el problema del subgrafo más denso en grafos dirigidos (es decir el problema DDS). Charikar desarrolló un algoritmo exacto para este problema, que completa el costo de tiempo polinomial al resolver $\mathcal{O}(n)$ linealmente.
Khuller y Saha propusieron un algoritmo basado en flujo, que también toma el costo de tiempo polinomial. algoritmos exactos. Sin embargo, todos estos algoritmos exactos son computacionalmente costosos para grafos grandes, por lo que los investigadores han recurrido al desarrollo de algoritmos de aproximación eficientes.
[**10**]

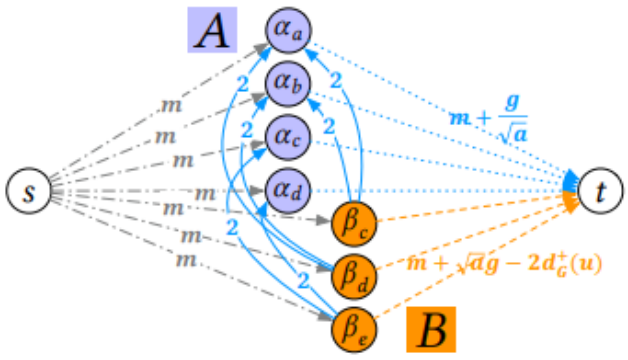
Kannan y Vinay propusieron un algoritmo de aproximación $\mathcal{O}(\log n)$ para calcular el subgrafo más denso. Charikar diseñó un algoritmo de 2 aproximaciones con una complejidad temporal de $\mathcal{O}(n \cdot (n + m))$. Khuller y Saha presentaron un algoritmo de aproximación lineal y afirmaron que logra una aproximación de 2.

Si revisamos el algoritmo exacto de última generación(es decir el algoritmo propuesto por Kan nan y Vinay) y los algoritmos de aproximación (Charikar y Khuller y Saha) para el problema DDS.

Observamos que para los algoritmos de aproximación, tanto los algoritmos (Khuller y Saha y Charikar) lograron un raito de aproximación de 2, pero el primero se ejecuta mucho más rápido que el último. Tenga en cuenta que en este documento, la relación de aproximación se define como la relación de la densidad máxima sobre la densidad del subgrafo devuelto. [**11**]

3.6. Algoritmo Exacto

El algoritmo exacto de última generación calcula el DDS resolviendo un problema de flujo máximo, que generalmente tiene el paradigma de encontrar los subgrafos más densos en grafos no dirigidos. Denotamos este como algoritmo Exacto. Una red de flujo es un grafo dirigido donde hay un nodo fuente s, un nodo sumidero t y algunos nodos intermedios; cada borde tiene una capacidad y la cantidad de flujo en un borde no puede exceder la capacidad del borde. El flujo máximo de una red de flujo es igual a la capacidad de su corte mínimo, $< S, T >$, que divide el conjunto de nodos en dos conjuntos disjuntos, S y T, de modo que sS y tT. [**12**]



Entonces en nuestro Algoritmo 1:

1. Enumeramos los posibles valores de $a = \frac{|S|}{|T|}$ (línea 2).

2. A Través de la búsqueda binaria adivinamos el valor de g de la densidad máxima por cada a (líneas 3-5).

3. Para cada par de a y g, se construye una red de flujo y ejecuta el algoritmo de flujo máximo para calcular el corte de st mínimo $< S, T >$ (líneas 6-11).

4. Si existe un subgrafo inducido $< S, T >$ (es decir $S \setminus \{s\} = \emptyset$ tal que su densidad sea al menos g) y que sea mayor a su densidad ρ^*; entonces actualizamos el DDS D y su densidad correspondiente ρ^* (línea 11).

5. Crea un conjunto V_F de nodos (líneas 14-15), y luego agrega bordes dirigidos con diferentes capacidades entre estos nodos (líneas 16-20).

3.6.1. Algoritmo

Algorithm 1 Agoritmo Exacto o de Fuerza Bruta

Require: G=(V,E)
Ensure: El DDS $D = G[S^*, T^*]$ exacto

$\rho \leftarrow 0$
for $a \in \{\frac{n_1}{n_2} \mid 0, n_2 \leq n\}$ **do**
  $l \leftarrow 0, r \leftarrow max_{u \in V}\{d^+_G(u), d^-_G(u)\}$
  while $r - l \geq \frac{\sqrt{n}-\sqrt{n-1}}{n\sqrt{n-1}}$ **do**
    $g \leftarrow \frac{l+r}{2}$
    $F = (V_F, E_F) \leftarrow BuildFlowNetwok(G, a, g)$
    $< S, T > \leftarrow Min - ST - Cut(F)$
    if $S = \{s\}$ **then**
      $r \leftarrow g$
    else
      $r \leftarrow g$
      if $g > \rho^*$ **then**
        $D \leftarrow G[S \cap A, S \cap B], \rho^* = g$
      end if
    end if
  end while
end for
return D

Algorithm 2 Function BuildFlowNetwok($G = (V, E), a, g$)

$A \leftarrow \{\alpha_u \mid u \in V\}$
$B \leftarrow \{\beta_u \mid u \in V\}, E_F \leftarrow \emptyset$
for $\alpha_u \in A$ **do**
  $add(s, \alpha_u)$ to E_F with capacity m
end for
for $\beta_u \in B$ **do**
  $add(s, \beta_u)$ to E_F with capacity m
end for
for $\alpha_u \in A$ **do**
  $add(\alpha_u, t)$ to E_F with capacity $m + \frac{g}{\sqrt{a}}$
end for
for $\beta_u \in B$ **do**
  $add(\beta_u, t)$ to $\mathrm{E}_F with capacity \mathrm{m} + \sqrt{ag} - 2d_G^+(u)$
end for
for $(u, v) \in E$ **do**
  $add(\beta_u, \alpha_u)$ to $\mathrm{E}_F with capacity 2$
end for
return $F = (V_F, E_F)$

3.6.2. Limitaciones

1. El número de valores posibles de a es n, entonces el bucle *while* de búsqueda binaria tendrá $\mathcal{O}(\log n)$ iteraciones. Calcular el corte de *st* mínimo de una red de flujo requiere un tiempo de $\mathcal{O}(nm)$. En consecuencia, la complejidad de tiempo total es $\mathcal{O}(n^3 m \log n)$, que por lo tanto es muy ineficiente incluso en grafos pequeños. Se demostró que tarda más de 2 días en encontrar el DDS en un grafo con 1200 vértices y 2600 aristas aproximadamente.

2. Esta ineficiencia se puede comprobar con todos los n^2 valores de a, lo cual es muy costoso.

3. La red de flujo F siempre se construye sobre el grafo completo en cada iteración, mientras que el DDS es a menudo un pequeño subgrafo de G.

4. Los límites inicial inferior y superior de ρ^* no son muy ajustados. Por lo tanto, hay espacio para mejorar su eficiencia.

3.7. Algoritmo Aproximado

El algoritmo de aproximación publicado más preciso es $BS-Approx$, que es capaz de encontrar correctamente un resultado de 2 aproximaciones. Similar al Exacto, $BS-Approx$ enumera todos los valores posibles de $a = \frac{|S|}{|T|}$ (línea 2), y para cada a específico, iterativamente elimina el vértice con el grado mínimo de S o T en función de una condición predefinida (línea 8), y luego actualiza S y T, así como el DDS D aproximado (líneas 4-9). [**12**]

3.7.1. Algoritmo

Algorithm 3 Algoritmo Aproximado

Input: G=(V,E)
Output: El DDS D aproximado

```
1 ρ* ← 0, D ← ∅
  for a ∈ {n1/n2 | 0, n2 ≤ n} do
2   S ← V, T ← V  while S ≠ ∅ ∧ T ≠ ∅ do
3     if ρ(S,T) > ρ* then
4       D ← G[S,T], ρ* ← ρ(S,T)  u ← argmin_{u∈S} d⁻_G(u)  v ← argmin_{v∈T} d⁺_G(v)
5       if √a · d⁻_G(u) ≤ 1/√a · d⁺_G(v) then
6         S ← S \ {u}
7       else
8         T ← T \ {v}
9 return D
```

3.7.2. Limitaciones

La complejidad temporal de $BS-Approx$ es $\mathcal{O}(n^2 \cdot (n+m)$, donde la sobrecarga principal proviene del ciclo de enumeración de todos los n^2 valores de a. Aunque es mucho más rápido que el Exacto, sigue siendo ineficiente para grafos grandes. En un grafo con unos 3000 vértices y 30 000 aristas, se tarda unos 3 días en calcular el DDS. Por lo tanto, es imperativo desarrollar algoritmos de aproximación más eficientes.