Instituto Politécnico

Escuela Superior de Cómputo

Calcular el Mayor Cliqué

Proyecto presentado por Flores Aguilar Aldo Ignacio, Mata Franco Carlos Jesus, Ruiz González Ian Alexander

Para la materia de Análisis de Algoritmos

2020

Introducción

El problema del clique máximo es un problema de optimización combinatoria que se clasifica dentro de los problemas NP- duros los cuales son difíciles de resolver. Debido a su complejidad las técnicas convencionales exactas (exhaustivas) tardan mucho tiempo para dar una solución, por lo tanto es necesario desarrollar algoritmos heurísticos que lo resuelvan alcanzando una solución cercana al óptimo en un tiempo razonable. Este problema tiene aplicaciones reales como son: teoría de códigos, diagnóstico de errores, visión computacional, análisis de agrupamiento, recuperación de información, aprendizaje automático, minería de datos, entre otras. Por esta razón es importante usar nuevas técnicas heurísticas y/o metaheurísticas para tratar de resolver este problema, las cuales obtengan mejores resultados en un tiempo polinomial.

Índice general

Introducción **2**

1. Marco Teórico **5**
1.1. Descripción del Problema 5
1.2. Propuesta de Solución 5

2. Síntesis de lectura **6**
2.1. Explicación 6
2.2. Definición del problema 7
2.3. Trabajos relacionados 8
2.3.1. Pasos a seguir 12
2.3.2. Reducción por k-núcleo 12
2.3.3. Reducción por k-truss, coloración y conjunto independiente . . 13
2.3.4. Búsqueda de arboles 15
2.3.5. El nuevo enfoque RMC 17
2.3.6. Inicialización - Algorithm Init (Algorithm 3) 18
2.3.7. El muestreo y la reducción del núcleo - algoritmo scSeed (Algoritmo 4) 19
2.3.8. La reducción por TCI 22
2.4. Estudios experimentales 25
2.5. Conclusión 28

3. Lenguaje y Paquetes **30**
3.1. Lenguaje 30
3.2. Paquetes utilizados 30

4. Uso de la aplicación **32**
4.1. Pantalla principal 32
4.2. Dibujar un Grafo 33
4.2.1. Pintar Nodo 33
4.2.2. Unir Nodos 34
4.2.3. Utilizar muestra 35
4.3. Características 35
4.3.1. Eliminar Nodo 35
4.3.2. Eliminar vertice o arista 36
4.3.3. Cambiar color de fondo 37
4.3.4. Calcular Camino más corto 38
4.3.5. Cargar Mapas 39
4.4. Calcular Máximo Cliqué 40

5. Conclusiones **41**
5.1. Flores Aguilar Aldo Ignacio 41
5.2. Mata Franco Carlos Jesus 41
5.3. Ruiz González Ian Alexander 42

Capítulo 1

Marco Teórico

1.1. Descripción del Problema

El término cliqué proviene de la palabra inglesa y francesa clique, que define a un grupo de personas que comparten intereses en común. En esta analogía, las personas serían los vértices; las relaciones de interés, las aristas; y el hecho de que todas compartan un mismo interés, el grafo completo, es decir, el clique en si.

Dado un grafo no dirigido cualquiera G= (V,E), en el cual V=1,2,...,n es el conjunto de los vértices del grafo y E es el conjunto de aristas. Un clique es un conjunto C de vértices donde todo par de vértices de C esta conectado con una arista en G, es decir C es un subgrafo completo.

1.2. Propuesta de Solución

Para poder dar solucion a este problema NP se propone utilizar un Algoritmo Recursivo, el cual no es el algoritmo más optimo para un grafo muy grande, sin embargo es desarrollado y propuesto por nosostros.

Capítulo 2

Síntesis de lectura

2.1. Explicación

El problema del máximo clique, donde debemos de encontrar el clique con la mayor cantidad de vértices dentro de un grafo dado Tiene varias aplicaciones en diversos campos como la búsqueda comunitaria en:
Redes y redes sociales: Se busca conectar a las personas que están familiarizadas entre si como bordes y candidatos (cliques) para servir a otras comunidades, esto para crear una herramienta eficiente de análisis entre organizaciones y distintas redes en el estudio de estructuras mesoscópicas.
Formación de equipos en redes expertas: Se captura la compatibilidad entre expertos así como sus capacidades individuales para formar equipos donde k-cliques cubre perfectamente un conjunto de capacidades donde k son el número de expertos.
Genética y bioinformática: Donde los grupos de expresión genética (CEG) se modelan como cliques para encontrar grandes equipos en la genética de redes.
Detección de anomalías en redes complejas: son usados para encontrar señales de eventos raros, como el reclutamiento de terrorista o el correo no deseado en la web.

En este articulo se busca un algoritmo aleatorio para el problema del máximo clique mediante diferentes enfoques como lo son en búsqueda de un vértice tras otro, enfoque RMC, clique máximo aleatorio, búsqueda binaria, etc. Donde el objetivo es encontrar una probabilidad de solución del problema $1 - n^{-c}$ donde se determina

el clique máximo mediante la búsqueda de iterativa de un clique-k en S a partir de $k = wc + 1$ hasta que S se convierte en vacío mientras pasan las iteraciones. Sin embargo, debido a la dureza del problema todos estos algoritmos fallan cuando se enfrentan a gráficos masivos (debido al aumento de tecnologías web e internet), se han propuestos diversas soluciones con una buena tasa de probabilidad sobre el clique máximo.

Actualmente todos estos algoritmos tienen una deficiencia importante, ya que dependen del orden inicial del vértice. Por eso trabajaremos con un nuevo enfoque donde:

Primero: Debido a la dureza del problema, se proponen soluciones al problema.

Segundo: En lugar de usar un esquema de ramificación y unión, es decir, ramificar desde cada vértice para enumerar todos los cliques y poder ramificar, se propondrá un esquema de búsqueda binaria con topes en cada iteración hasta que ya no queden más iteraciones.

Tercero: Que cada iteración de la búsqueda binaria se trabaje como un problema de k-cliques en S con una probabilidad de garantía $1 - n^(- c)$ donde para cada semilla en S introducimos una semilla sc y tci para reducir de forma iterativa los subgrafos generados.

Cuarto: Proponemos un nuevo algoritmo iterativo de fuerza bruta para determinar el clique máximo después de una búsqueda binaria desde $k = wc + 1$.

Quinto: Hacemos los estudios experimentales para mostrar la solidez y eficiencia de nuestro algoritmo y enfoque.

2.2. Definición del problema

Modelaremos una red social como un gráfico no dirigido $G = (V, E)$ sin auto bucles o bordes múltiples, donde V y E denotan los conjuntos de vértices y bordes de G, donde n y m denotaran el número de vértices y aristas de G, es decir, $n = |V|ym = |E|$,donde se da por hecho que G esta conectado, de lo contrario, el algoritmo se puede aplicar a cada componente conectado en el grafo.

Un grafo G es un clique si hay bordes entre 2 vértices de G. También llamamos a un

conjunto de vértices CCV una camarilla si el subgrafo inducida por C es una camarilla. C es una camarilla máxima si no existe un superconjunto adecuado de C que también sea una camarilla y C es una camarilla máxima si no existe una camarilla C como tal $|C'| > |C|$. El número de vértices en una camarilla máxima en el gráfico G = (V, E) se denota como w (G) o w (V). Por simplicidad, en la siguiente discusión, usamos $w(G) o\ w(V)$ para denotar el límite superior de la camarilla máxima de G = (V, E) y usamos w (G) o w (V) para representar el límite inferior de la camarilla máxima de G, respectivamente.

2.3. Trabajos relacionados

El clique se estudio por primera vez para modelar grupos de individuos que se conocen entre si y fue modelado por Cook y Karp utilizando la teoría de la completitud NP y los resultados de intratabilidad relacionados para proporcionar una explicación matemática de la dificultad del problema. Tarjan y Trojanowski dieron con el peor de los casos, en 1990 Feige demuestra que es imposible de aproximar el problema con precisión y eficacia hasta que se propone un algoritmo de tiempo polinómico O ((log logn)2) siempre que el gráfico tenga una camarilla de tamaño O $n * log(nb)$ para cualquier constante b. Pero todos estos trabajos son trabajados como grafos pequeños, pocos han aplicado para grafos realmente grandes utilizando métodos como listas de adyacencias, búsquedas binarias. La diferencia con este algoritmo es que ahora buscan limites superiores e inferiores para mejorar las soluciones por aproximación usando una búsqueda local dinámica y el uso de la diversificación, aplicado también en búsqueda tabú y redes neuronales.

Algoritmos anteriores Dejan que el clique máximo encontrado, denotada como Cm, como un límite inferior de w(G). Dejamos que C denote el clique actual, y P =(C) denote el conjunto candidato del cual se seleccionará un vértice para que C crezca a continuación. En otras palabras, poda ramas que no pueden generar cliques más grandes que Cm.

Algorithm 1 *MaxClique* (G)

1: $C_m \leftarrow \emptyset$, sort V in some specific ordering o;
2: **for** $i = 1$ to n according to o **do**
3: $C \leftarrow \{v_i\}$, $P \leftarrow \{v_{i+1}, \ldots, v_n\} \cap \Gamma(v_i)$;
4: sort P in some ordering o';
5: **for** $v_j \in P$ according to the sorting order o' **do** Clique(G, C, P, v_j);
6: **end for**
7: **return** C_m;

8: **Procedure** Clique(G, C, P, v_j)
9: $C \leftarrow C \cup \{v_j\}$, $P \leftarrow P \cap \Gamma(v_j)$;
10: **if** $P = \emptyset$ **then**
11: **if** $|C| > |C_m|$ **then** $C_m \leftarrow C$;
12: **else if** $U(C, P) > |C_m|$ **then**
13: sort P in some ordering o'';
14: **for** $v_k \in P$ according to the sorting order o'' **do** Clique(G, C, P, v_k);
15: **end if**

Figura 2.1: Algoritmo 1

- Línea 1: Lista en un orden.
- Línea 2: recorre hasta.
- Línea 3: Inicializa.
- Línea 4: Ordena en algún orden.
- Línea 5: Quita las ramas infructuosas.
- Línea 6: termina el proceso.
- Línea 7: regresa Cm.
- Línea 8: Procede con el clique.
- Línea 9: Actualiza y regresa un clique.
- Línea 10 y 11: Si encuentra un clique, entonces se actualiza.
- Línea 12 a 14: De lo contrario no encuentra un clique más grande.

Los algoritmos anteriores funcionan bien en muchas redes. Sin embargo, se enfrentan a desafíos cada vez mayores cuando se trata de redes de rápido crecimiento en la vida real. En el fondo de las soluciones existentes y los gráficos masivos del mundo real, observamos tres razones principales que perjudican enormemente la eficiencia y la solidez de los algoritmos existentes.

Primero: El conjunto candidato es muy grande y no tiene un límite fijo, va cambiando.

Segundo: En la practica los vértices de alto grado siempre tienen una gran cantidad de bordes que se conectan con vértices de bajo grado

Tercero: No se utiliza el límite superior de w(G) en los algoritmos existentes. Pierde tiempo para buscar ramas restantes

Visión general del enfoque Se diseña un nuevo algoritmo aleatorio basado en la búsqueda binaria, para mejorar significativamente la robustez y la eficiencia del algoritmo. El algoritmo mantiene un limite inferior y superior de w(G) e intenta encontrar el clique máximo con algunas diferencias. En nuestra búsqueda binaria, en lugar de encontrar una camarilla wt en una iteración directamente, encontramos un conjunto de subgrafos donde pueden existir camarillas wt. El conjunto se denota de la siguiente manera.

$$S = \{(s_1, \Lambda(s_1)), (s_2, \Lambda(s_2)), \ldots, (s_i, \Lambda(s_i)), \ldots\}$$

Donde (si, Δ (si)) es un subgrafo que consiste en una camarilla si y un conjunto candidato Δ (si) desde el cual el clique puede crecer. Para simplificar, utilizamos el término "semilla"para representar tanto la camarilla si como el subgrafo (si, Δ (si)). Además, cada (si, Δ (si)) está asociado con una menor obligado wi y un límite superior wi de w (si U Δ (si)), y todos estos límites se pueden combinar para actualizer wc y wc, st, wc = maxiwi y wc = maxi wi, que a cambio poda las semillas sin fruto. Determinamos los límites inferior y superior para la próxima iteración. Sea wc y wc los límites actuales, y sea wp y wp los límites anteriores. Hay 4 casos como se ilustra en la Fig. 2 basados en wc y wc. 1) El caso óptimo: si S 0 y wc wt, que es equivalente a wc = wc, se ha encontrado el clique máximo y el algoritmo termina.

2) Si S 0 y wc ¡ wt, entonces G no contiene cliques, disminuya wc como wt - 1). 3) Si S 0 wc wp o wc wp, más iteraciones pueden mejorar los límites. 4) Si S 0, wc = wp y wc = wp, otras iteraciones introducen mejoras marginales, finaliza la búsqueda binaria y aplica un algoritmo de búsqueda de fuerza bruta.

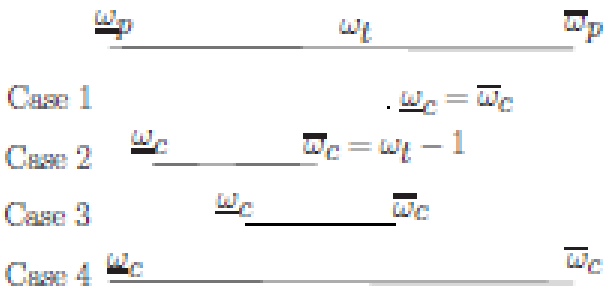


Figure 2: The interval of $\omega(G)$ in next iteration

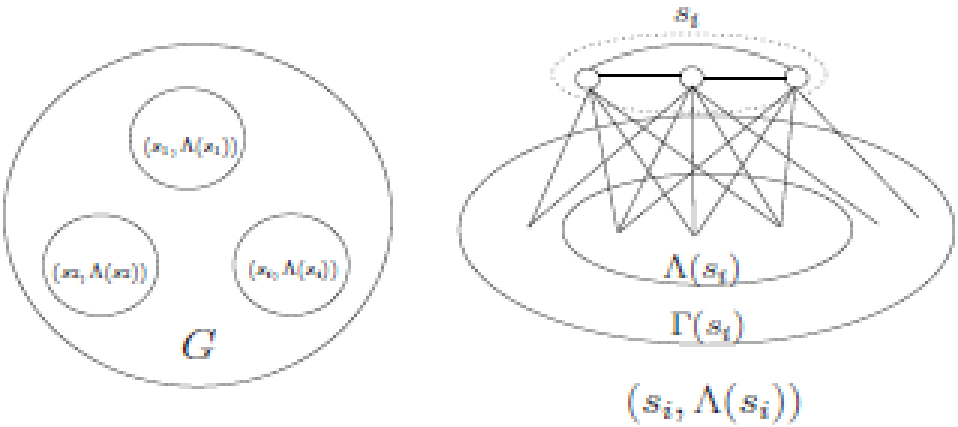


Figure 3: Find a ω_l-clique from a seed set S instead of G

Figura 2.2: Intérvalo w(G)

2.3.1. Pasos a seguir

1. Muestreo de los bordes de G con probabilidad.
2. Generar semillas en triángulos abiertos para que tenga probabilidad de clique.
3. Encuentra el clique máximo en G.

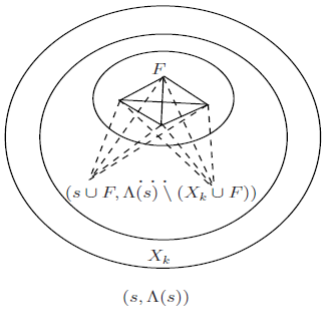
2.3.2. Reducción por k-núcleo

Agrandar s y reducir Δ (s) de la siguiente manera.

$$(s, \Lambda(s_i)) \rightarrow (s \cup F, \Lambda(s) \setminus (X_k \cup F)) \qquad (1)$$

Aquí Xk es un subconjunto de Δ (s) que no puede existir en a (k - —s—- 1) -núcleo en Δ (s) para una k dada, y F es un conjunto de vértices que tienen el potencial de existir en una camarilla máxima de Δ (s) donde F Δ (s) n Xk. Hay dos condiciones para que se seleccione F.

1. Cada vértice en F está conectado a cada vértice en si, lo cual está garantizado por la definición de Δ (si).
2. Cada vértice en F está conectado a cualquier otro vértice en Δ (s) n Xk. En otras palabras, F es una clique a tener en cuenta.



(a) The 1st by k-core

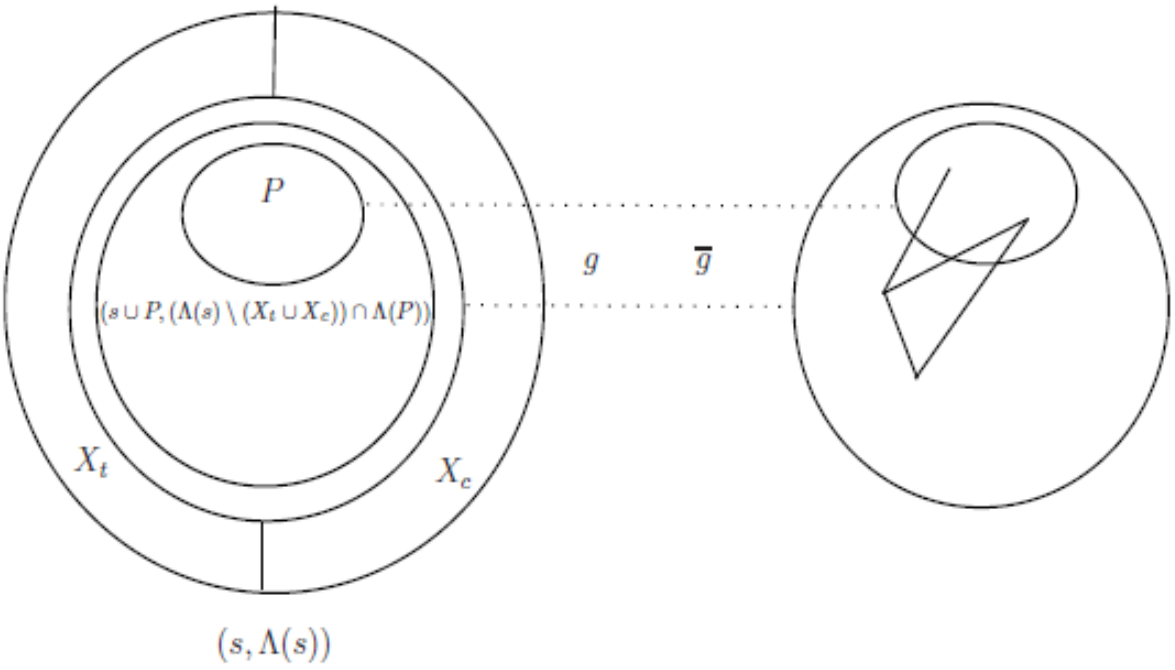
Figura 2.3: Reducción por k-núcleo

2.3.3. Reducción por k-truss, coloración y conjunto independiente

Basado en el anterior, pero con un nuevo enfoque donde:

$$(s, \Lambda(s)) \rightarrow (s \cup P, (\Lambda(s) \setminus (X_t \cup X_c)) \cap \Lambda(P)) \qquad (2)$$

Mediante el truss k, identificamos Xt, que es un subconjunto de Δ (s) que no puede existir, coloreamos, identificamos Xc Δ (s) n Xt que contiene vértices cuyos vecinos se pueden colorear con menos de wt - —s— -1 colores, por lo tanto, el clique no contiene vertices en Xt.



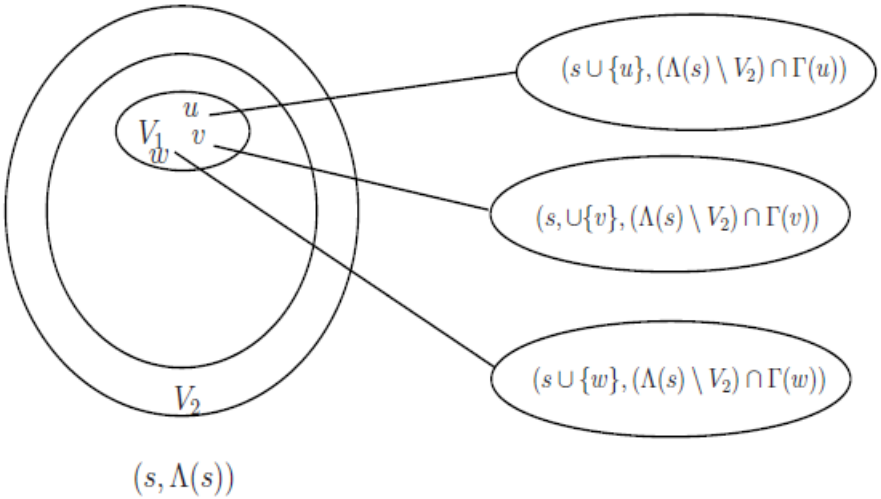
(b) The 2nd by k-truss, coloring, and independent set

Figura 2.4: Reducción por k-truss

El segundo círculo más grande a la izquierda representa un subgrafo g de (s, Δ (s)) al excluir los vértices en (Xt U Xc). Reducimos aún más g usando un conjunto independiente. Sea g la gráfica complementaria de g (el círculo más grande a la derecha). Con la ayuda de un conjunto independiente encontrado entre Δ (s) n (Xt U Xc) en g, podemos extraer un subconjunto P C Δ (s) (Xt U Xc).

La reducción dividiendo

Sea (s, Δ (s)) una subgrafía donde s no puede ampliarse mediante el primer y el segundo enfoque. Proponemos un nuevo algoritmo de búsqueda de fuerza bruta optimizado. Encontramos iterativamente k-camarillas en cada semilla (s, $\Delta(s))S$ a partir de $k = wc + 1$, y encontramos la camarilla máxima cuando S se convierte. Para procesar una semilla (s, Δ (s)), con la ayuda de k, extraemos dos subconjuntos V1, V2 C Δ (s) por coloración gráfica donde V2 representa los vértices cuyos vecinos se pueden colorear con $< k - |s| - 1$ colores y V1 es un subconjunto de Δ (s) nV2 que representa los vértices de color $k - |s|$. Como resultado, los vértices en V1 posiblemente pueden estar en el clique máximo, pero los vértices en V2 no pueden.

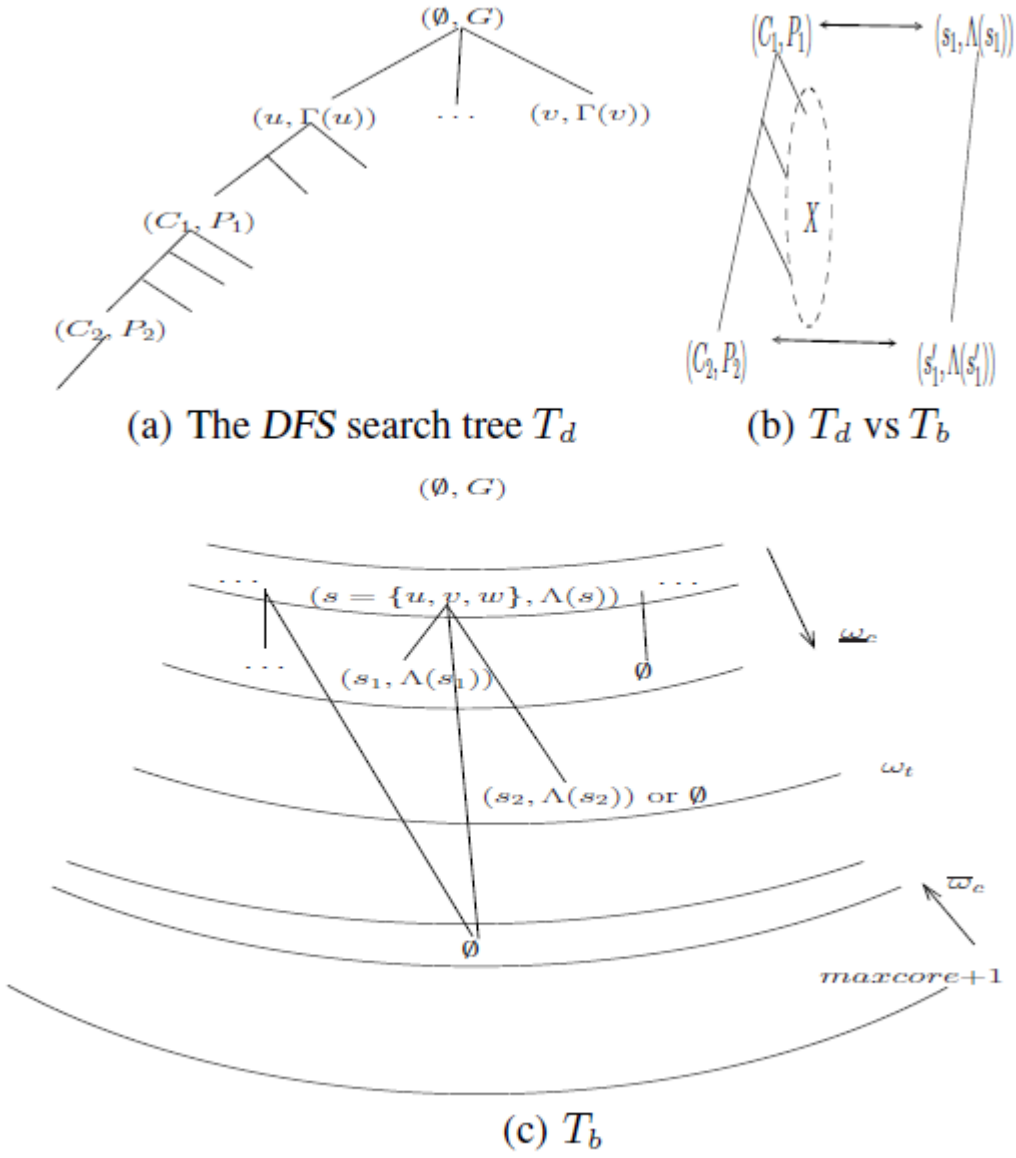


(c) The 3rd by dividing seeds

Figura 2.5: Reducción dividiendo

2.3.4. Búsqueda de arboles

Discutimos las principales diferencias entre los enfoques existentes de ramificación y unión y nuestro nuevo enfoque en términos de árboles de búsqueda, para encontrar las camarillas máximas en gráficos masivos. Deje que Td y Tb denoten el árbol de búsqueda por la marca existente y enfoques vinculados y los nuestros, respectivamente. Usamos nodo en lugar de vértice cuando hablamos de árboles. En Td, un nodo representa un par (C; P), donde C es una camarilla en crecimiento y P es su conjunto candidato. En Tb, un nodo representa una semilla (s; Δ (s)), donde la raíz de ambos árboles es (0; G). Un nodo en ambos árboles representa la misma información, porque s es un clique y Δ (s) es su conjunto candidato. Deje que un nodo hijo de (C; P) sea (C0; P0) en Td, y que un nodo hijo de (s; Δ (s)) sea (s0; Δ (s0)) en Tb. En Td, C0 = C U u donde u se selecciona de P, y P0 = Δ (C) n T(u) Los enfoques existentes de ramificación y unión conducen DFS sobre Td. En Tb, s0 = s [s0 y Δ (s0) = Δ (s) Δ (s0), donde s0 Δ (s). Realizamos BFS sobre Tb. Mostramos las diferencias entre Td y Tb. Vale la pena señalar que en Tb intenta seleccionar más vértices de Δ (s) para agrandar una camarilla en crecimiento en un nodo cuando se ramifica a su nodo hijo, mientras que en Td lo hace seleccionando un solo vértice v de P. En otras palabras, un borde en Tb representa una ruta en Td. Además, los vértices en X en Td pueden podarse mediante nuestro enfoque en Tb en una etapa temprana. Todo se debe a que los candidatos poco prometedores en Δ (s) serán podados lo antes posible. Inicialmente, la s de cualquiera (s, Δ (s)) en el conjunto inicial de semillas S es un triángulo abierto basado en el cual se da el límite inferior wc, y el límite superior ¡wc es maxcore + 1, y wt = —(wc + wc) /2—. En cada iteración de la búsqueda binaria, actualiza wc y wc al explorar el conjunto de semillas actual S. Primero: wc se mueve hacia abajo donde existe al menos una camarilla wc, lo que implica w(G) wc. Segundo: wc se mueve hacia arriba donde no hay cliques (wc + 1), lo que implica w(G) wc. Tercero: wT indica que es posible encontrar wT-camarillas entre wc y wc. Al encontrar a wT-clique, podemos movernos wc a wT, y al encontrar todas las semillas vacías en wt, podemos mover wc a wt - 1.



(a) The *DFS* search tree T_d (b) T_d vs T_b

(c) T_b

Figure 5: The *BFS* search tree T_b

Figura 2.6: Dividing Seeds

2.3.5. El nuevo enfoque RMC

Algorithm 2 *RMC* (G)

1: $(r, \underline{\omega_c}, \overline{\omega_c}, C_m) \leftarrow Init(G)$;
2: **if** $r = 1$ **then return** C_m;
3: **while** $\overline{\omega_c} - \underline{\omega_c} \geq \theta$ **do**
4: $\quad \omega_t \leftarrow \lfloor (\overline{\omega_c} + \underline{\omega_c})/2 \rfloor, \overline{\omega_p} \leftarrow \overline{\omega_c}, \underline{\omega_p} \leftarrow \underline{\omega_c}$;
5: $\quad (\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow scSeed(G, \omega_t)$;
6: $\quad$ **switch** (states of $S, \underline{\omega_c}, \omega_t$)
7: $\quad$ **case** $S = \emptyset$ and $\underline{\omega_c} \geq \omega_t$: **return** C_m;
8: $\quad$ **case** $S = \emptyset$ and $\underline{\omega_c} < \omega_t$: $\overline{\omega_c} \leftarrow \omega_t - 1$; **break**;
9: $\quad$ **case** $S \neq \emptyset$:
10: $\quad\quad (\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow tciSeed(G, S, \omega_t)$;
11: $\quad\quad$ **switch** (states of $S, \underline{\omega_c}, \omega_t, \underline{\omega_p}, \overline{\omega_p}$)
12: $\quad\quad$ **case** $S = \emptyset$ and $\underline{\omega_c} \geq \omega_t$: **return** C_m;
13: $\quad\quad$ **case** $S = \emptyset$ and $\underline{\omega_c} < \omega_t$: $\overline{\omega_c} \leftarrow \omega_t - 1$; **break**;
14: $\quad\quad$ **case** $\underline{\omega_c} = \underline{\omega_p}$ and $\overline{\omega_c} = \overline{\omega_p}$: terminate the while loop;
15: $\quad\quad$ **end switch**
16: $\quad$ **end switch**
17: **end while**
18: $\omega_t \leftarrow \underline{\omega_c} + 1$;
19: $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow scSeed(G, \omega_t)$;
20: **if** $S = \emptyset$ **then return** C_m;
21: $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow tciSeed(G, S, \underline{\omega_c} + 1)$;
22: **while** $S \neq \emptyset$ **do** $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow divSeed(G, S, \underline{\omega_c} + 1)$;
23: **return** C_m;

Figura 2.7: RMC

Le damos nuestro algoritmo RMC ((Randomized Maximum Clique) en Algorithm 2. En Algorithm 2, el algoritmo Init (Algorithm 3) inicializa wc, wc, y Cm, y devuelve una variable r indica si la camarilla máxima ya se encuentra en el gráfico G (Línea 1). Si r = 1, entonces Cm es la camarilla máxima, el algoritmo termina devolviendo Cm (Línea 2). De lo contrario, aplica una búsqueda binaria para reducir la brecha entre el límite inferior wc y el superior enlazado wc en un ciclo while (Línea 3-17). El ciclo continuará si la diferencia entre wc y wc es mayor o igual que un umbral. En otras palabras, la búsqueda binaria se detendrá si otras iteraciones no reducen significativamente el espacio de búsqueda. En cada iteración, intentamos encontrar a wT-clique, donde b (wc + wc) = 2c (Línea 4). Primero, usamos un algoritmo aleatorio scSeed (Algoritmo 4) para actualizar wc y wc, y obtenemos un conjunto

Algorithm 3 *Init* (G)

1: compute core(u) for every u in G;
2: let the max core number be cm, $\overline{\omega_c} \leftarrow cm + 1$;
3: **if** the max core of G is a clique **then**
4: $C_m \leftarrow$ max core of G, $\underline{\omega_c} \leftarrow |C_m|$;
5: **return** $(1, \underline{\omega_c}, \overline{\omega_c}, C_m)$;
6: **end if**;
7: **for** each vertex v with core$(v) \geq \underline{\omega_c}$ in non-ascending order of core numbers **do**
8: greedily generate a maximal clique C starting from v;
9: $C_m \leftarrow C, \underline{\omega_c} \leftarrow |C_m|$ if $|C| > \underline{\omega_c}$;
10: **end for**
11: **if** $\underline{\omega_c} = \overline{\omega_c}$ **then return** $(1, \underline{\omega_c}, \overline{\omega_c}, C_m)$;
12: compute graph coloring, and the color number as cn;
13: **if** $\overline{\omega_c} > cn$ **then** $\overline{\omega_c} \leftarrow cn$;
14: **if** $\underline{\omega_c} = \overline{\omega_c}$ **then return** $(1, \underline{\omega_c}, \overline{\omega_c}, C_m)$;
15: **else return** $(0, \underline{\omega_c}, \overline{\omega_c}, C_m)$;

Figura 2.8: Init

de semillas S donde posiblemente existen t-cliques (Línea 5). Aquí, scSeed aplica un muestreo uniforme en los bordes de G para extraer semillas si = (u; v; w) st (u; v) y (u; w) se muestrean y (v; w) es un borde en G. Vale la pena señalar que una semilla es un triángulo abierto. Como lo demuestra el Teorema 7.1, con alguna probabilidad de muestreo específica p, cada camarilla contiene al menos una semilla con alta probabilidad. Para cada semilla si, scSeed aplica la descomposición del núcleo [5] en Δ (si), poda los vértices que existen en no wT-cliques, y mueve vértices que se conectan a todos los demás a si. Esto da como resultado un límite superior wI y un límite inferior wI de w(Si [Δ (si)). Utilizamos los límites obtenidos para las semillas juntos para actualizar wc y wc, que a cambio elimina las semillas sin fruto y hace que S sea mínimo.

2.3.6. Inicialización - Algorithm Init (Algorithm 3)

Toma un gráfico G como entrada y devuelve una tupla de 4. Aquí, Cm es una clique, wc y wc son los límites inferior y superior, y r es un indicador de si Cm es la

camarilla máxima. Primero calculamos el número de núcleo para cada vértice u en G, denotado como núcleo (u), utilizando el algoritmo de descomposición del núcleo (línea 1). El número de núcleo máximo sea cm e inicialice wc como cm + 1. Si el núcleo máximo se encuentra como una camarilla, entonces se encuentra la camarilla máxima, devolviendo el resultado (línea 3-6). Discutimos cuándo el núcleo máximo de G no es una camarilla (Línea 7-15). Encontramos la camarilla Cm entre todas las camarillas máximas encontradas con avidez para cada vértice v si su número central (núcleo (v)) es el límite superior actual wc (Línea 7-10). Si los límites superior e inferior actuales son iguales, devolvemos el resultado ya que Cm encontrado es la camarilla máxima (Línea 11). A continuación, actualizamos aún más el límite superior actual usando el color del gráfico. Deje que el número central de G sea cn (Línea 12). ! c se reduce a cn si! c¿cn. Finalmente devolvemos el resultado. Tenga en cuenta que en esta etapa, Cm encontrado es la camarilla máxima si wc = wc.

2.3.7. El muestreo y la reducción del núcleo - algoritmo scSeed (Algoritmo 4)

La segunda fase es actualizar S usando la reducción por k-core (Ec. (1)). Ampliamos sy reducimos Δ (s) para cada semilla (s; Δ (s)) en S, y eliminaremos todo (s; Δ (s)) de S si no puede ayudar a encontrar una k-camarilla donde k se proporciona como una entrada del algoritmo (línea 7-33). En la primera fase, muestreamos un conjunto de bordes del conjunto de bordes E de G, denotado como Es, usando un muestreo uniforme. Luego construimos el conjunto inicial de semillas S. Para cada semilla, (s; Δ (s)), en S, s es un triple abierto s = (u; v; w) si ambos (u; v) y (u; w) están en el conjunto de bordes muestreados Es y (v; w) están en E. Para cada semilla (s; Δ (s)), determinamos su límite superior! s como minfcore (u); núcleo (v); núcleo (w) g +1. Discutimos la probabilidad de muestreo p.

Teorema 7.1: Sea Es un subconjunto de E de G mediante un muestreo uniforme de bordes de G con una probabilidad de muestreo de bordes p. Sea ST un conjunto de triángulos abiertos, s = (u, v, w), de modo que tanto (u, v) como (u, w) estén en Es y (v; w) estén en E. Cada k-camarilla en G contiene al menos un triángulo en ST con probabilidad 1–$n^(c)$, donde n es el número de vértices en G.

$$p = \sqrt{2 \cdot c \cdot \ln n/(k \cdot (k-1) \cdot (k-2))}$$

Bosquejo de prueba

Supongamos que E1 denota el evento de que uno de esos triángulos abiertos específicos se muestrea a partir de G. Luego, deje que E denote el evento de que se muestree al menos un triángulo abierto para una camarilla k, y deje que E denote que el evento E no ocurre.

$$\begin{aligned} Pr(\mathcal{E}_1) &= p^2 = 2 \cdot c \cdot \ln n/(k \cdot (k-1) \cdot (k-2)) \\ Pr(\overline{\mathcal{E}}) &= (1 - Pr(E_1))^{k \cdot \binom{k-1}{2}} \leq n^{-c} \end{aligned}$$

Therefore, we can conclude $Pr(\mathcal{E}) \geq 1 - n^{-c}$.

En la segunda fase, actualizamos cada semilla (s; Δ (s)) en S y actualizamos S siguiendo el orden no ascendente en los límites inferiores (wS). Es importante tener en cuenta que la k inicial en el bucle (línea 7-33) es la entrada k del algoritmo para una k-clique y k puede aumentar para podar más semillas de S. Primero, si ws de una semilla (s, Δ (s)) es menor que k, la semilla se eliminará de S, ya que no puede conducir a una k-camarilla (Línea 8) En segundo lugar, verificamos si se puede encontrar una k-camarilla en (s, Δ (s)) por la condición de —Δ (s) — + —s— ¡k. Si esta condición no se cumple, la semilla se eliminará de S (Línea 9-11).

En tercer lugar, actualizamos más una semilla si la condición no se cumple (Línea 12-31). Discutimos el tercer caso a continuación. Supongamos que encontramos que Δ (s) forma una camarilla, actualizamos la camarilla máxima actual Cm por (s; Δ (s)) y actualizamos el límite inferior actual para que sea wc = —Cm—. Vale la pena señalar que en cada iteración k permanece sin cambios o aumenta. Por lo tanto, el Cm actualizado no puede ser más pequeño que el encontrado anteriormente. Eliminamos (s, Δ (s)) de S, ya que es la camarilla máxima actual. Seguimos la ecuación. (1), calcular Xk, elimine Xk de Δ (s) y calcule F (línea 16-18). Aquí, Xk es un subconjunto de vértices en Δ (s) donde cada vértice u tiene un número central que es menor que k - —s— -1. F es un subconjunto de vértices en Δ (s) n Xk donde cada

vértice se conecta a todos los demás vértices en Δ (s) n Xk. En otras palabras, s U F posiblemente puede formar una camarilla máxima. Hay varios casos:
Caso-i) Cuando Δ (s) n F =0: Si (s, Δ (s)) no puede conducir a una k-clique porque F está vacío, la semilla (s, Δ (s)) se eliminará de S (Línea 19-21). De lo contrario, si F no está vacío, la camarilla máxima actual Cm puede actualizarse con s U F, y la semilla puede eliminarse de S, ya que la semilla se trata como la camarilla máxima actual (Línea 21-23).
Caso-ii) cuando Δ (s) n F 6 =0: aumentará s por s [F y reducirá Δ (s) por Δ (s) n F. Además, si js [Fj + 1 k, implica se ha encontrado una k-clique (Línea 24), y actualizaremos el actual camarilla máxima por un subgrafo con s [F [fvg donde v se toma de Δ (s) n F (Línea 25). Tenga en cuenta que el límite inferior wc se actualiza cuando se actualiza la camarilla máxima actual. Después de considerar los casos, actualizamos k para que sea wc +1 if wc k (Línea 31). Finalmente, devolvemos (wc, wc; Cm; S) donde wc es el número de núcleo del núcleo máximo encontrado para semillas no vacías.

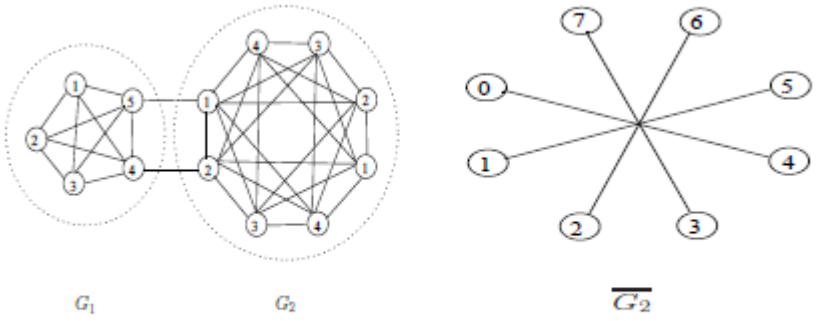


Figure 7: Reduction by TCI

Figura 2.9: Dividing Seeds

2.3.8. La reducción por TCI

El algoritmo tciSeed se muestra en el Algoritmo 5, que toma 3 entradas, un gráfico G, un conjunto de semillas S y un valor específico de k-clique k, y devuelve una tupla de 4 (wc, wc, Cm, S), donde wc y wc son un límite inferior y un límite superior, y Cm es la camarilla máxima actual, y S es un conjunto de semillas reducido. Como se da en la ecuación. (2), en tciSeed, para cada semilla (s, Δ (s)) / S, reducimos Δ (s) para que sea Δ (s) n (Xt U Xc), donde Xt y Xc son dos subconjuntos en Δ (s) de manera que cualquier vértice en Xt U Xc no pueda aparecer en una k-camarilla. Aquí Xt contiene los vértices que no pueden estar en un (wT-—s—) -truss en Δ (s), y Xc contiene vértices cuyos vecinos están coloreados con menos de wt - —s— - 1 colores La corrección de Xt es obvia.

Teorema 7.2: Dado un gráfico G con un color gráfico, si w(G) k, entonces cualquier vértice v con vecinos coloreados con menos de k - 1 colores no se pueden incluir en ninguna k-camarilla de G. Bosquejo de prueba: Suponga lo contrario, es decir, hay un vértice v cuyos vecinos se pueden colorear con menos de k - 1 colores, mientras que v se incluye en una k-camarilla. Entonces, el subgrafo inducido por v U T(v) contiene una k-camarilla y se puede colorear con menos de k colores, lo que conduce a una contradicción. Por lo que agrandamos s en la semilla de (s, Δ (s)) por un subconjunto P Δ (s), donde todos los vértices en P pertenecen a una camarilla máxima en Δ (s). Encontramos P como los vértices en un conjunto independiente máximo en el gráfico complementario Δ(s) en un algoritmo compIS, basado en el teorema 7.3.

Teorema 7.3: En un gráfico G, si hay un vértice v con grado (v) = 1, hay al menos un conjunto independiente máximo de G que contiene v en G.

Bosquejo de prueba: Suponga lo contrario, es decir, ninguno de los conjuntos independientes máximos contiene v en G. Suponga que I es un conjunto independiente máximo en G. Deje que el vecino único de v sea u. Solo hay dos casos relacionados con la existencia de u en I. Primero, si u I, podemos tener un conjunto independiente máximo I0 reemplazando u con v de modo que I‘ -¿I U v u. En segundo lugar, si u = (no existe en) I, podemos tener un conjunto independiente más grande I0 ampliando I a I [fvg de modo que I‘ -¿I Uv. Esto lleva a la conclusión de que al menos un conjunto independiente máximo contiene v.

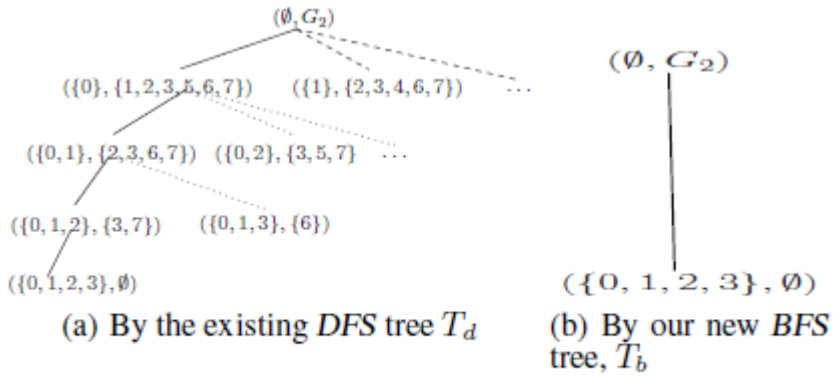


Figure 8: *DFS* T_d **vs** *BFS* T_b

Figura 2.10: DFS vs BFS

La reducción dividiendo Hay casos en los que la búsqueda por fuerza bruta es inevitable, en tales casos son cuando S no está vacío. Mediante la búsqueda de fuerza bruta existente, para cada semilla (s; Δ (s)) en S, se ramifica desde cada vértice v Δ (s) para enumerar camarillas máximas y podar ramas sin fruto.

Como se indica en RMC (Algoritmo 2) en la Línea 22, encontramos la camarilla máxima del conjunto de semillas S llamando al algoritmo divSeed hasta que S se vacíe. El Algoritmo divSeed se da en el Algoritmo 7, que toma 3 entradas: el gráfico G, el conjunto de semillas S y el valor ak para una k-camarilla específica. Tal valor k ofrece más oportunidades para que divSeed pueda podar. Al igual que scSeed y tciSeed, divSeed devuelve 4 tuplas (wc, wc, Cm, S). A diferencia de scSeed y tciSeed, divSeed devuelve un conjunto de semillas, denotado como S0, para reemplazar la entrada S. Aquí, una semilla (s, Δ (s)) S se reemplaza por varias semillas más pequeñas y densas (s0, Δ (s0)) en S0. Para cada semilla (s, Δ (s)) en S, calculamos V1 y V2. Como se discutió, V1 es un subconjunto de Δ (s) donde cada vértice en V1 tiene un color mayor o igual a k - —s—, y V2 es un subconjunto de Δ (s) donde por cada vértice en V2 sus vecinos en Δ (s) pueden colorearse con menos de k - —s— -1 colores (Línea 3-4). Según el teorema 7.2, los vértices en V2 no pueden estar en una k-camarilla. Eliminamos los vértices en V2 de Δ (s), y refinamos V1 para que sea V1 n V2 (Línea 5). Mostramos que cualquier k-clique de s [Δ (s) debe contener

Algorithm 5 *tciSeed* (G, S, k)

1: **for** each $(s, \Lambda(s)) \in S$ **do** $\overline{\omega}_s \leftarrow |s| + |\Lambda(s)| - 1, \underline{\omega}_s \leftarrow |s| + 1; \underline{\omega}_s$;
2: **for** each $(s, \Lambda(s)) \in S$ in non-ascending order of $\overline{\omega}_s$ and $\underline{\omega}_s$ **do**
3: **if** $\overline{\omega}_s < k$ **then** $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
4: compute the $(k\text{-}|s|)$-truss from $\Lambda(s)$ and X_t;
5: $\Lambda(s) \leftarrow \Lambda(s) \setminus X_t$;
6: **if** the $(k\text{-}|s|)$-truss is a clique **then**
7: $C_m \leftarrow s \cup \Lambda(s), \underline{\omega_c} \leftarrow |C_m|$;
8: $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
9: **else if** $\Lambda(s) \neq \emptyset$ **then**
10: compute graph coloring for $\Lambda(s)$ and X_c;
11: $\Lambda(s) \leftarrow \Lambda(s) \setminus X_c$;
12: **if** $\Lambda(s) = \emptyset$ **then** $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
13: $(P, I) \leftarrow$ *compIS* $(\overline{\Lambda(s)})$;
14: $(s, \Lambda(s)) \leftarrow (s \cup P, \Lambda(s) \cap \Lambda(P))$;
15: **if** $|I| + |s| \geq k$ **then** $C_m \leftarrow s \cup I, \underline{\omega_c} \leftarrow |C_m|$;
16: **end if**
17: **if** $\underline{\omega_c} \geq k$ **then** $k \leftarrow \underline{\omega_c} + 1$;
18: **end for**
19: **return** $(\underline{\omega_c}, \overline{\omega_c}, C_m, S)$;

Algorithm 6 *compIS* $(G = (V, E))$

1: $P \leftarrow \emptyset, I \leftarrow \emptyset$;
2: add every $(v, \delta(v))$ into a min-heap H for $v \in V$;
3: initialize t to be $true$, and $state[1...|V|]$ be all zeros;
4: **while** $H \neq \emptyset$ **do**
5: $v \leftarrow$ getMin(H);
6: **if** $state[v] = 0$ **then**
7: $state[v] \leftarrow 1, I \leftarrow I \cup \{v\}$;
8: **if** $\delta(v) > 1$ **then** $t \leftarrow false$;
9: **if** $t = true$ **then** $P \leftarrow P \cup \{v\}$;
10: **for** every u with $(u, v) \in G$ and $state[u] = 0$ **do**
11: $state[u] \leftarrow 1$;
12: **for** each w with $(u, w) \in G$ and $state[w] = 0$ **do**
13: $\delta(w) \leftarrow \delta(w) - 1$;
14: **end for**
15: **end for**
16: **end if**
17: **end while**
18: **return** (P, I);

Figura 2.11: Algoritmos

al menos un vértice de un subconjunto V1 Δ (s) Teorema 7.4.

Teorema 7.4: Para un gráfico G, si w(G) k, entonces cualquier k-clique de G debe

contener al menos un vértice con el color k.
Bosquejo de prueba: Mostramos la afirmación por contradicción, es decir, suponemos que hay una camarilla k con todos los vértices coloreados ¡k en G. Luego, la coloración sigue siendo válida si eliminamos todos los otros vértices que no están incluidos en la camarilla k. Esto implica que obtenemos un color para una camarilla k que usa ¡k colores, lo que lleva a una contradicción. Por lo tanto, cualquier k-clique en G contiene al menos un vértice con color k. 2
En divSeed, en el bucle (Línea 6-28), obtenemos una nueva semilla más pequeña / más densa, (s0, Δ (s0)) de la semilla actual (s, Δ (s)) para cada vértice v en V1. Aquí, s0 se convierte en s U v, y Δ (s0) se convierte en Δ (s) (v) (Línea 7). Luego, reducimos la nueva semilla (s0; Δ (s0) usando técnicas similares utilizadas en el algoritmo tciSeed (línea 6-28).

2.4. Estudios experimentales

Hemos realizado estudios experimentales utilizando 22 gráficos grandes reales para comparar RMC con varios enfoques de vanguardia, incluyendo FMC, PMC, MCQD, cliquer, que son algoritmos exactos y GRASP [1], que es un algoritmo de aproximación en una maquina con CPU Intel Core i7-4770 de 3.40 GHz, 32 GB de RAM y Linux. La unidad de tiempo utilizada es la segunda y establecemos el límite de tiempo en 24 horas. Delicious, Digg, Flixster, Foursquare, Friendster y Lastfm se descargan de http://socialcomputing.asu.edu/pages/conjuntos de datos La información detallada de los conjuntos de datos del mundo real se resume en la Tabla 1.

La eficiencia en conjuntos de datos reales: La Tabla 2 muestra la eficiencia de RMC y sus comparaciones. Las columnas 12 y 15 muestran los tamaños de las camarillas máximas encontradas por GRASP y RMC, respectivamente. Descuidamos los resultados de otros algoritmos ya que son algoritmos exactos. Como se demostró, RMC supera a todos los demás en todos los conjuntos de datos probados significativamente. De la Tabla 2, RMC, PMC y FMC superan significativamente a MCQD,

Algorithm 7 *divSeed* (G, S, k)

$S' \leftarrow \emptyset$;
for each $(s, \Lambda(s)) \in S$ **do**
  $V_1 \leftarrow \{u \mid u \in \Lambda(s) \wedge \mathsf{color}(u) \geq k - |s|\}$;
  $V_2 \leftarrow$ the vertices in $\Lambda(s)$ whose neighbors in $\Lambda(s)$ can be colored with $< k - |s| - 1$ colors;
  $\Lambda(s) \leftarrow \Lambda(s) \setminus V_2, V_1 \leftarrow V_1 \setminus V_2$;
  for each $v \in V_1$ **do**
    $s' \leftarrow s \cup \{v\}, \Lambda(s') \leftarrow \Lambda(s) \cap \Gamma(v)$;
    if $|s'| + |\Lambda(s')| < k$ **then continue**;
    if $\Lambda(s')$ is a clique **then**
      $C_m \leftarrow s' \cup \Lambda(s'), \underline{\omega_c} \leftarrow |C_m|$;
    else
      $X_k \leftarrow \{u \mid u \in \Lambda(s') \wedge \mathsf{core}(u) < k - |s'| - 1\}$;
      compute F from $\Lambda(s') \setminus X_k$;
      $\Lambda(s') \leftarrow \Lambda(s') \setminus X_k$;
      if $\Lambda(v') \setminus F = \emptyset$ **then**
        if $|F| \geq k - |s'|$ **then** $C_m \leftarrow s' \cup F, \underline{\omega_c} \leftarrow |C_m|$;
      else
        if $|F| \geq k - |s'| - 1$ **then**
          $C_m \leftarrow s' \cup F \cup u$ where $u \in \Lambda(v) \setminus F$;
          $\underline{\omega_c} \leftarrow |C_m|$;
          $s' \leftarrow s' \cup F, S' \leftarrow S' \cup (s', \Lambda(s'))$;
        else
          $s' \leftarrow s' \cup F, S' \leftarrow S' \cup (s', \Lambda(s'))$;
        end if
      end if
    end if
    if $\underline{\omega_c} \geq k$ **then** $k \leftarrow \underline{\omega_c} + 1$;
  end for
end for
$(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow tciSeed(G, S', k)$;
return ;

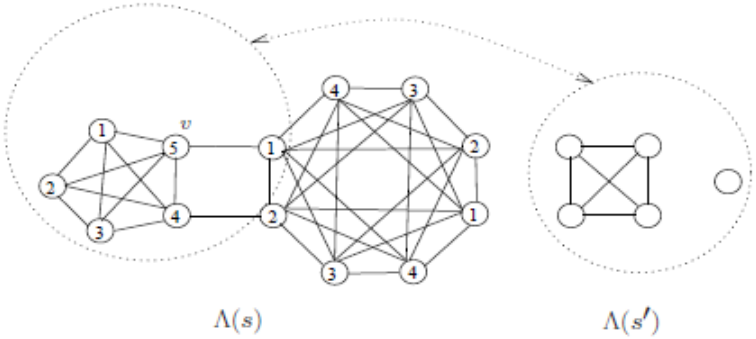


Figure 9: The reduction by dividing

Figura 2.12: Algoritmo divSeed

Table 1: Summarization of datasets

Graph	$\|V\|$	$\|E\|$	d_{max}	c_{max}	t_{max}	$\|\omega(G)\|$
Amazon	403,394	2,443,408	2,752	10	11	11
BerkStan	685,230	6,649,470	84,230	201	201	201
Epinions	75,879	405,740	3,044	67	33	23
Gnutella	62,586	147,892	95	6	4	4
Google	875,713	4,322,051	6,332	44	44	44
LiveJournal	4,036,538	34,681,189	14,815	360	352	327
NotreDame	325,729	1,090,108	10,721	155	155	155
Pokec	1,632,803	22,301,964	14,854	47	29	29
Slashdot0811	77,360	469,180	2,539	54	35	26
Slashdot0902	82,168	504,229	2,552	55	36	27
Stanford	281,903	1,992,636	38,625	71	62	61
WikiTalk	2,394,385	4,659,563	100,029	131	53	26
wikivote	7,115	100,762	1,065	53	23	17
Youtube	1,138,499	2,990,443	28,754	51	19	17
Orkut	3,072,627	117,185,083	33,313	253	78	51
BuzzNet	101,163	2,763,066	64,289	153	59	31
Delicious	536,408	1,366,136	3,216	33	23	21
Digg	771,229	5,907,413	17,643	236	73	50
Flixster	2,523,386	7,918,801	1,474	68	47	31
Foursquare	639,014	3,214,986	106,218	63	38	30
Friendster	5,689,498	14,067,887	4,423	38	31	25
Lastfm	1,191,812	4,519,340	5,150	70	23	14

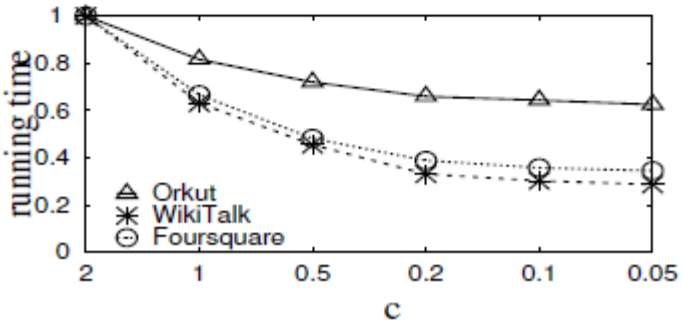


Figure 10: The Efficiency of RMC: Varying c

Figura 2.13: Dataset

cliquer y GRASP en la mayoría de los gráficos.

La influencia del parámetro c: Mostramos la influencia de c en el Teorema 7.1 en la precisión y eficiencia de RMC. Aquí mostramos los resultados utilizando 3 conjuntos de datos: Orkut, WikiTalk y Foursquare. Los conjuntos de datos restantes son similares a uno de los tres. La Fig. 10 demuestra el tiempo de ejecución empleando

Table 2: Performance of *FMC*, *PMC*, *MCQD*, *cliquer*, *GRASP* and *RMC* on real-world networks

Graph	FMC		PMC		MCQD		cliquer		GRASP			RMC					
	time	mem	time	mem	time	mem	time	mem	time	mem	$\omega(G)$	time	mem	$\omega(G)$	t_{div}	ω_c	$\overline{\omega_c}$
Amazon	0.1	112	**0.09**	114	-	-	868.1	19,437	63.17	85	9	0.1	67	11	0	11	11
BerkStan	-	-	1.04	272	-	-	-	-	708.5	224	53	**0.23**	144	201	0	201	201
Epinions	5.65	15	0.26	46	7.33	647	31.81	696	15.68	18	22	**0.11**	12	23	0.02	23	26
Gnutella	**0.01**	8	0.05	8	4.37	10	22.52	470	2.4	8	4	0.03	8	4	0.01	4	6
Google	0.35	216	0.77	250	-	-	-	-	80.61	163	20	**0.24**	135	44	0	44	44
LiveJournal	-	-	6.15	1,412	-	-	-	-	508.49	995	102	**1.82**	788	327	0.01	326	327
NotreDame	**0.01**	215	0.19	474	-	-	449.73	12,857	30.01	61	154	0.2	46	155	0	155	155
Pokec	-	-	9.51	1,359	-	-	-	-	369.18	575	13	**2.79**	407	29	0	29	29
Slashdot0811	6.66	21	0.18	38	7.47	1,943	33.24	724	23.26	18	24	**0.06**	13	26	0.01	25	32
Slashdot0902	10.18	18	0.19	36	8.53	1,936	37.5	817	14.45	19	25	**0.07**	14	27	0.01	27	29
Stanford	-	-	0.48	68	-	-	428.31	9,550	201.31	85	49	**0.19**	51	61	0	61	61
WikiTalk	2,767.95	325	4.87	336	-	-	-	-	298.42	415	24	**2.45**	314	26	0.41	22	37
wikivote	1.53	3	0.08	49	0.06	3	0.29	435	2.59	3	17	**0.03**	2	17	0.01	17	21
Youtube	6.13	187	2.04	191	-	-	-	-	1,145.82	216	15	**1.26**	156	17	0.08	17	20
Orkut	-	-	208.97	1,692	-	-	-	-	9,132.23	2,520	25	**45.01**	1,623	51	3.76	50	65
BuzzNet	28,354.4	88	14.37	52	13.15	1479	51.35	1236	758.99	81	30	**4.78**	48	31	1.89	23	59
Delicious	0.27	83	0.26	86	-	-	1,476.15	31,987	33.14	103	20	**0.1**	73	21	0.01	20	21
Digg	-	-	10.09	1,388	-	-	-	-	593.21	200	44	**2.13**	146	50	0.38	45	58
Flixster	71.10	434	2.25	423	-	-	-	-	120.52	473	26	**1.04**	358	31	0.06	31	33
Foursquare	-	-	39.36	141	-	-	13,698.37	-	470.83	153	27	**23.42**	113	30	1.82	29	33
Friendster	10.12	823	2.88	852	-	-	-	-	254.74	939	14	**1.09**	765	25	0	25	25
Lastfm	25.61	205	2.55	265	-	-	-	-	80.84	243	14	**1.41**	177	14	0.2	14	17

Figura 2.14: Performance

diferentes valores de c. Aquí, establecemos el tiempo de ejecución como 1 cuando c = 2, y usamos la relación para indicar el tiempo de ejecución con c menor. Como puede verse

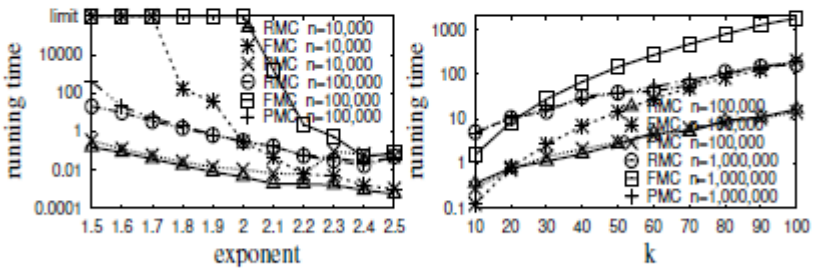
1. Emplear c más pequeño mejora significativamente el rendimiento de RMC,
2. La mejora es marginal cuando c es lo suficientemente pequeña, y
3. La mejora difiere significativamente en diferentes gráficos.
4. La influencia de c en la precisión de RMC se muestra en la Tabla 3.

2.5. Conclusión

Nuestro enfoque RMC emplea un esquema de búsqueda binaria. Específicamente, RMC mantiene un límite inferior wc y un límite superior wc de w(G) e intenta encontrar una wT-clique en cada iteración donde wT = b (wc + wc) = 2c. Debido a lo incompleto de NP de encontrar un wT-clique en cada iteración, RMC muestrea un conjunto de semillas S st encontrar a! T-clique en el gráfico G es equivalente a encontrar una wT-clique en S con garantías de probabilidad (1–$n^{(}$ − C)). Para

Table 3: The accuracy of RMC: Varying c

c	Orkut			WikiTalk			Foursquare		
	$\underline{\omega_c}$	$\overline{\omega_c}$	$\omega(G)$	$\underline{\omega_c}$	$\overline{\omega_c}$	$\omega(G)$	$\underline{\omega_c}$	$\overline{\omega_c}$	$\omega(G)$
2	50	65	51	22	55	26	29	33	30
1	50	65	51	22	37	26	29	33	30
0.5	50	65	51	22	37	26	29	33	30
0.2	50	65	51	22	37	26	29	33	30
0.1	50	65	51	22	37	26	29	33	29
0.05	50	65	51	22	37	26	29	33	29



(a) PowerLaw graph (b) Watts-Strogatz model

Figure 11: *FMC*, *PMC* and *RMC* on synthetic graphs

Figura 2.15: Table 3

las semillas restantes cuyas camarillas máximas probablemente puedan actualizar la camarilla máxima Cm encontrada hasta ahora, proponemos un nuevo algoritmo iterativo de búsqueda de fuerza bruta divSeed para encontrar la camarilla máxima cuando el conjunto de semillas S se vacía.

Capítulo 3

Lenguaje y Paquetes

3.1. Lenguaje

El lenguaje que se utilizó fue Java, ya qué es un lenguaje con el cual todos los integrantes están familiarizados y tienen experiencia. De igual manera es un lenguaje amigable para aplicaciones GUI e interfaces, lo cual nos ayudó mucho para el dibujo de grafos.

3.2. Paquetes utilizados

java.awt.event.ActionEvent: Un evento semántico que indica que ocurrió una acción definida por componentes. Este evento de alto nivel es generado por un componente (como lo son los botones en nuestro programa) cuando ocurre la acción específica del componente (como ser presionado). El evento se pasa a todos los objetos ActionListener que se registraron para recibir dichos eventos utilizando el método addActionListener del componente

import java.util.*;

import java.util.ArrayList: La usamos para los objetos ArrayList y usarlos en una lista redimensionable en la que tendremos disponibles los métodos más habituales para operar con listas. Los usamos para extraer, construir e imprimir los elementos

grafos y aristas.

import javax.swing.JFileChooser: Es una forma rápida y fácil de solicitar al usuario que elija un archivo o una ubicación para guardar archivos. Lo usamos para subir imagenes de fondo para interpretar un mapa con los grafos.

import javax.swing.JOptionPane.

import java.awt.BasicStroke: Analiza las subrutas no cerradas y segmentos de guión que se extienden más allá del final. Con sus métodos encontramos los nodos que encierran las aristas. import java.awt.Color: Lo usamos para cambiar los colores de los objetos en la interfaz.

import java.awt.Graphics: Lo usamos para las operaciones graficas.

import java.awt.Graphics2D: Amplía la clase Graphics, haciendo uso de sus métodos podemos darle valores enteros a los nodos, son los números que se ven junto a los nodos.

import java.awt.RenderingHints.

Capítulo 4

Uso de la aplicación

4.1. Pantalla principal

Figura 4.1: Pantalla principal

Al comenzar la aplicación se puede observar una pantalla en la cual se tiene en un principio un fondo gris, el cual usaremos para dibujar nuestros grafos.
Del lado izquierdo tenemos un botón para calcular el máximo cliqué y un botón de Desmarcar que de momento no tiene función. En la parte superior se tiene una serie de opciones que nos ayudarán en diversas situaciones; las cuales veremos más adelante con detalle.

4.2. Dibujar un Grafo

4.2.1. Pintar Nodo

Para dibujar un grafo solo se requiere dar click sobre la superficie gris que se aprecia al iniciar la aplicación. En donde se de click es donde se pintará el nodo de nuestro Grafo, entonces podemos comenzar a dibujar los nodos que compongan nuestro grafo.

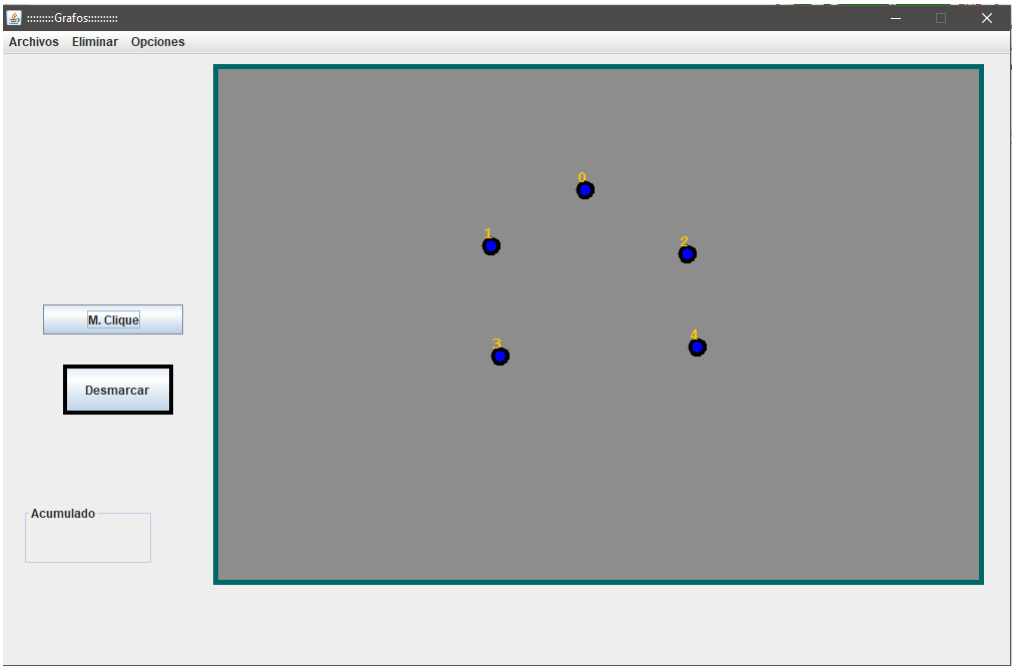


Figura 4.2: Dibujar nodos

4.2.2. Unir Nodos

Para unir los nodos de nuestro Grafo lo primero que haremos sera dar click en un nodo, el cuál será nuestro origen y luego dar click sobre el nodo destino para que exista una unión entre ellos. Al realizar esto nos abrirá una pequeña ventana donde tendreos que teclear el peso que habrá entre estos dos nodos, ya qué de igual forma el programa funciona para calcular el camino más corto.
Realizar esto para unir todos los nodos que tengan relación en nuestro grafo.
Al final tendremos pintado nustro grafo para poder caluclar el máximo cliqué o calcular el camino más corto.

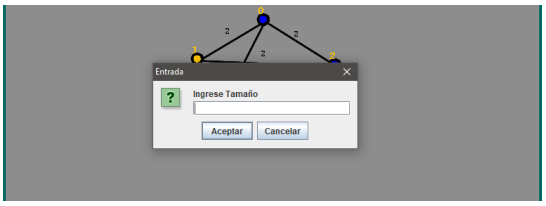


Figura 4.3: Definir peso de unión

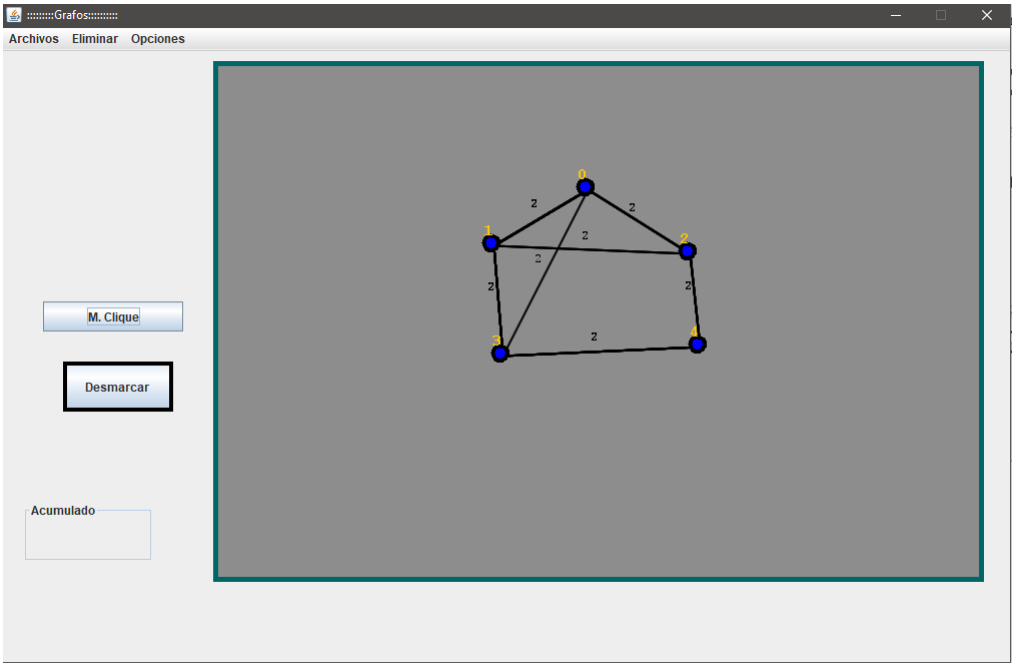


Figura 4.4: Dibujar vertices

4.2.3. Utilizar muestra

Si no se tiene un grafo para realizar alguna prueba se puede cargar un grafo que se encuentra a modo de muestra.
Nos dirigimos a la parte superior en la opcion de archivo damos click y nos desplazamos a la opción de Muestra. Al dar click en esta opción nos cargará un Grafo muestra con el cual podemos trabajar.

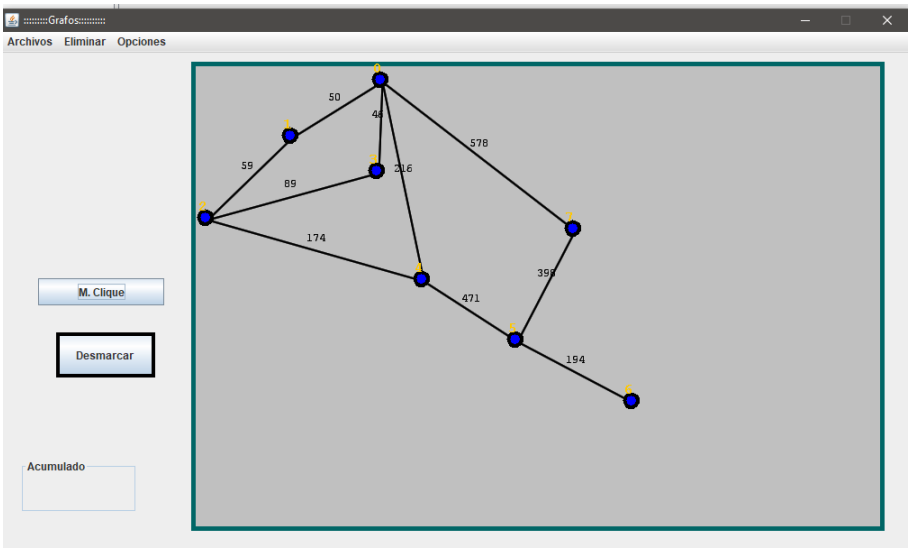


Figura 4.5: Dibujar vertices

4.3. Características

4.3.1. Eliminar Nodo

Dentro de la aplicación es posible eliminar nodos si es que hubo algún error. Para ello lo que se tiene que realizar es lo siguiente.

1. Dirigirirnos a la parte superior donde dice Elimiar y dar click.
2. Se desplegará una lista de 3 opciones (Eliminar nodo, Eliminar Arista, Eliminar todas las aristas).

3. Seleccionamos elimar Nodo, abrirá una pequeña ventana donde nos pedirá el nodo a eliminar.

4. Digitamos el nodo y damos click en aceptar.

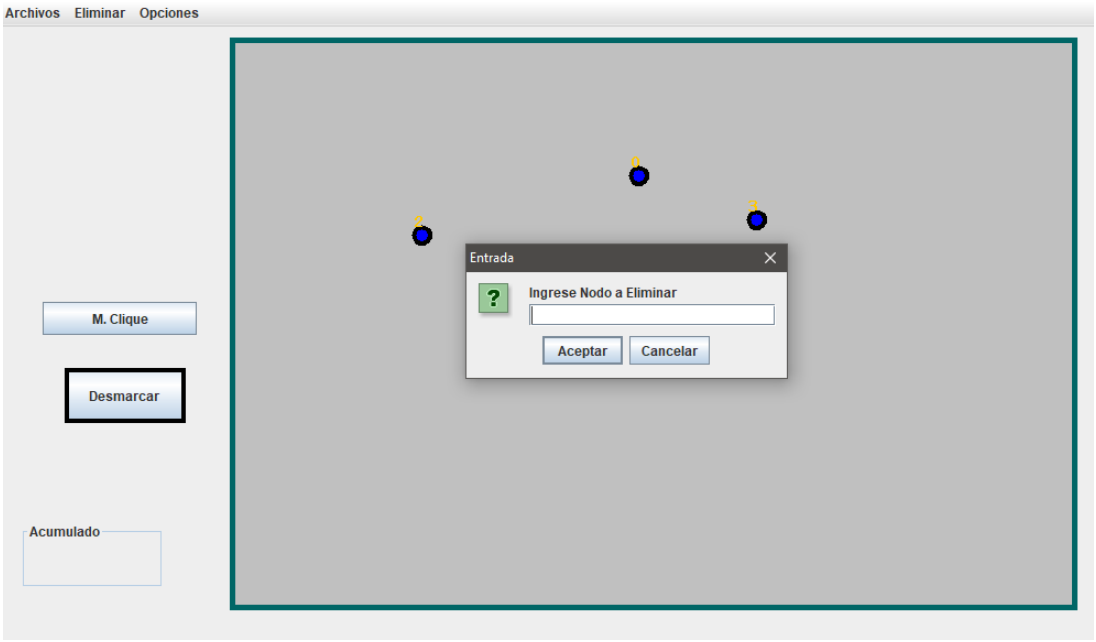


Figura 4.6: Eliminar Nodo

4.3.2. Eliminar vertice o arista

Para ello realizamos los mismos pasos que en eliminar nodo solo que seleccionamos la opción de eliminar arista, en esta ocación nos saldrá una ventana pero debemos digitar dos nodos para saber que arista eliminaremos.

De igual manera podemos elimnar todos los vertices. Para esto solo se tiene que seguir los pasos antes mencionados para eliminar nodo y arista (vertice) pero ahora daremos click en Eliminar todas las aristas.
Al dar click se eliminarán todas las aristas existentes en nuestro grafo, esto nos ayuda a que si no tenemos muy claro en que estamos mal en el grafo poder empezar desde cero a realizar las relaciones.

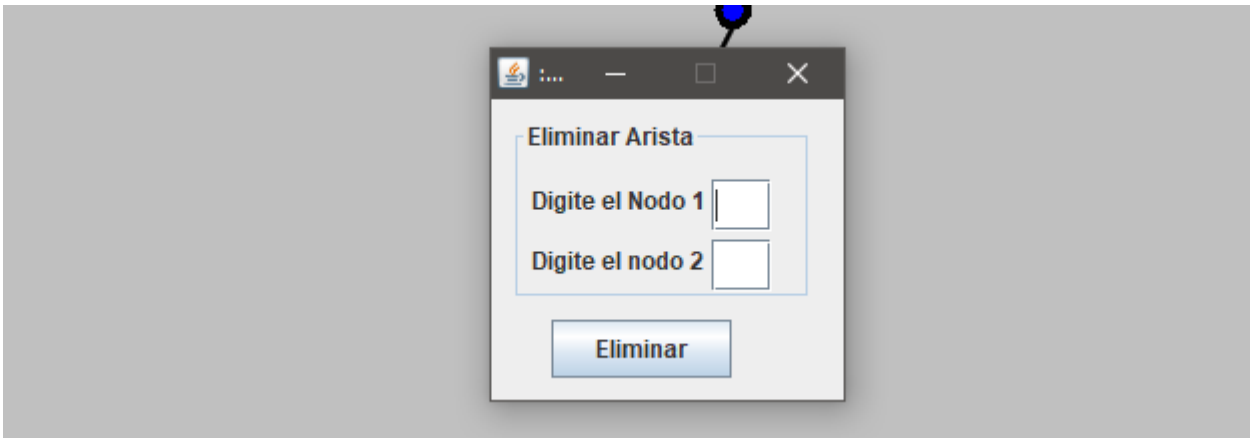


Figura 4.7: Eliminar vertice

4.3.3. Cambiar color de fondo

Es posible cambiar el color de fondo de nuestra área de trabajo.
Para ello solo tenemos que ir a la parte superior y dar click en la opción de opciones luego donde dice color, lo que nos abrirá del lado izquierdo una pequeña paleta de colores en donde el usuario podrá elegir entre 12 colores distintos, al dar click sobre cualquier color en automático se realizará el cambio de color de fondo para que se utilice al agrado de cada persona o dependiendo si es un lugar con mucha iluminacion, etc.

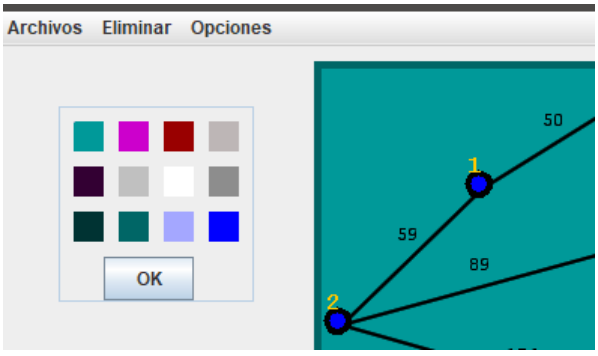


Figura 4.8: Cambiar color

4.3.4. Calcular Camino más corto

Debido a que se tomó de un proyecto ya implementado el prgrama cuenta con la capacidad de calcular el camino más corto de un nodo a otro.
Si queremos utilizar esta caracteristica lo que necesitamos es seguir los pasos siguientes:

1. Dirigirnos a la parte superior y dar click en Archivos.

2. Seleccionar la opción de Camino más Corto.

3. abrirá una ventana para que indiquemos nodo origen y luego otra para el nodo destino.

4. Despues de ingresar los datos damos click y nos pintará de color la ruta del camino más corto.

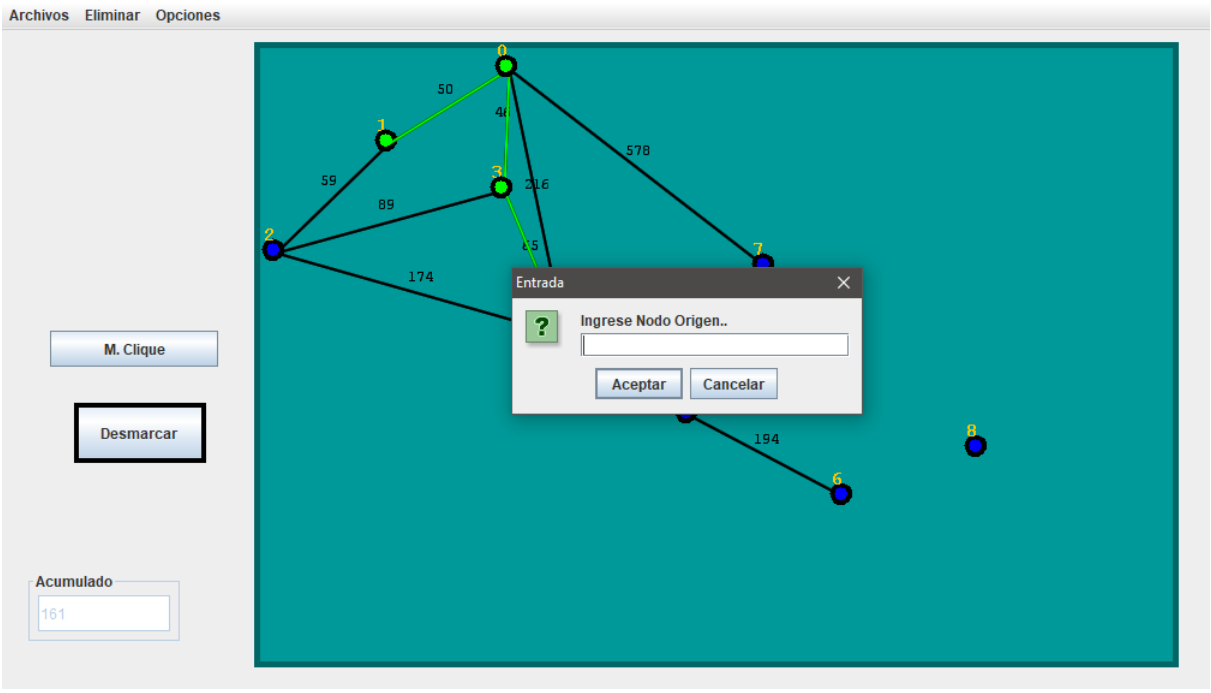


Figura 4.9: Cambiar color

4.3.5. Cargar Mapas

Una de las caracteristicaas y ventajas que pudimos econtrar e implementar en esta aplicación es el cargar cualquier mapa (imagen) de fondo para poder pintar el grafo sobre éste, de forma que se visualice un problema más real. Así, de igual manera sea amigable y entendible para el usuario.

Por ejemplo se puede cargar el mapa de algún Pais, mapa de alguna ruta de camiones, metro, suburbano, etc, para poder definir cual es la ruta más cerca para desplazarse de un punto a otro.
Podriamos cargar un mapa con imagenes de personas y definir gustos en común para determinar cual sería el mayor clique entre este grupo de personas.

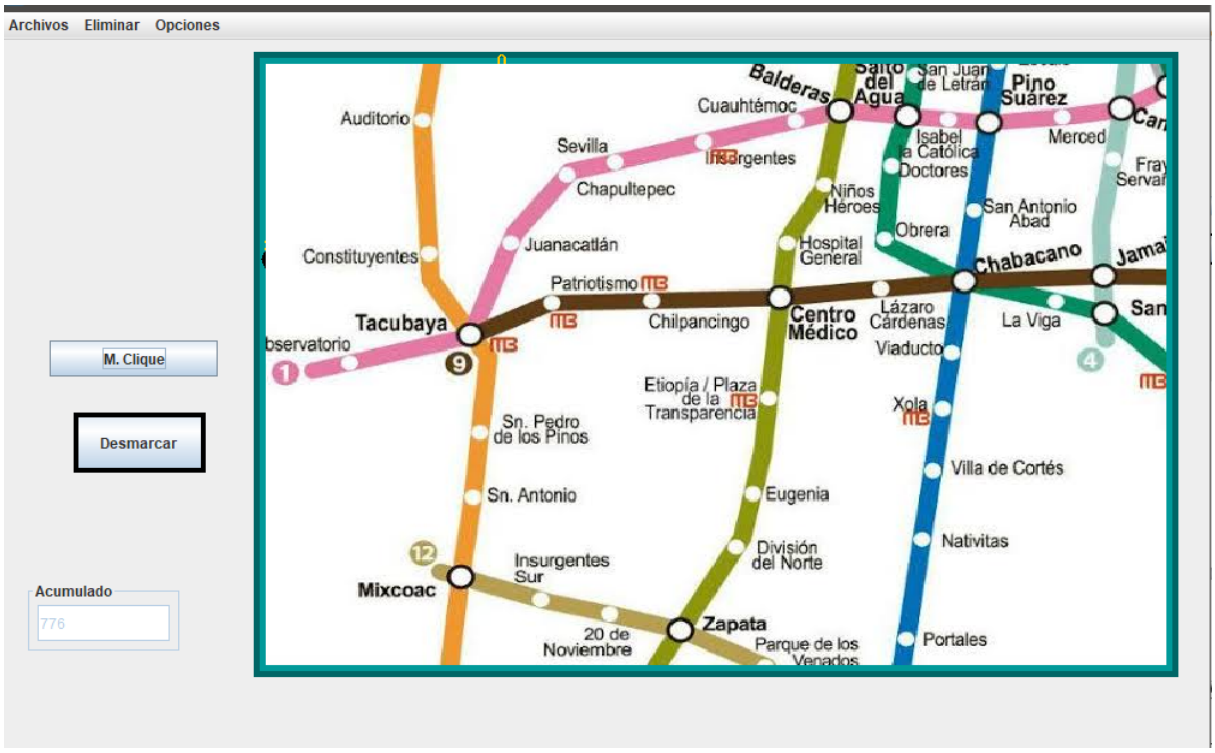


Figura 4.10: Cambiar color

4.4. Calcular Máximo Cliqué

Para poder calcular el Máximo Cliqué, tenemos que primero haber pintado un grafo con todo y sus relaciones entre nodos.
Ahora nos dirigimos al lado izquierdo de la pantalla donde se encuentra un botón que dice M.Clique y damos click. Al terminar el proceso del algoritmo se mostrará una pequeña ventana donde dice que se ha obtenido el máximo cliqué y el tamaño del mismo. Al cerrar la ventana podemos ver en la consola que nos imprime los nodos que conforman este máximo cliqué puede haber más de un máximo cliqué ya qué, puede que existan más de dos soluciones para el problema planteado.

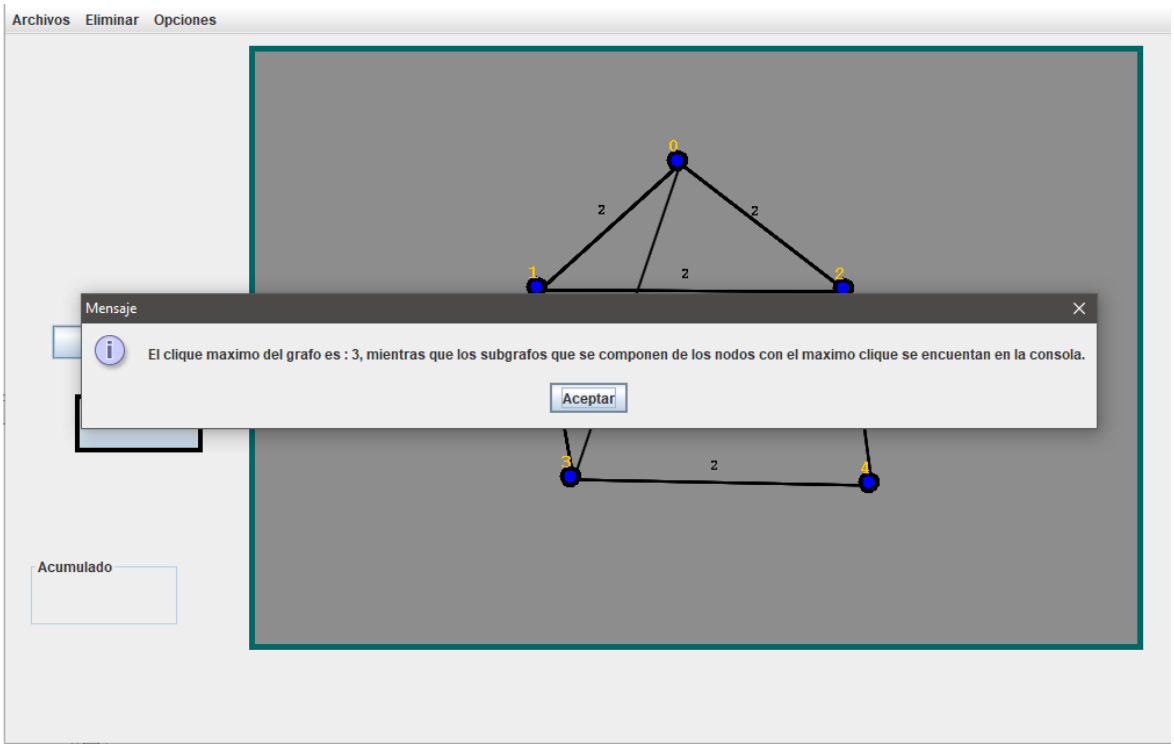


Figura 4.11: Calcular Máximo Cliqué

Capítulo 5

Conclusiones

5.1. Flores Aguilar Aldo Ignacio

Como se mencionó antes, el equipo partió de un proyecto con el que ya contaba, el cual ya era capaz de visualizar un panel donde se podía dibujar grafos y mostrar el camino mínimo entre nodos. El equipo tuvo que emplear nuevas técnicas en java donde pudiera dar uso a métodos gráficos como eventos que pudieran ser usados como listas. Si se hubiera usado un método donde el usuario solo introdujera las matrices de adyacencia y de coeficientes; el programa hubiera sido mucho más simple de programar, pero el encanto de los grafos se encuentra en que estos tienen que ser gráficos, este tipo de esfuerzos se llevan a cabo para que el usuario pueda manejar el programa de una manera simple, y el equipo está de acuerdo en que el uso del proyecto es bastante intuitivo, me encuentro satisfecho con el funcionamiento del programa, pero espero poder desarrollarlo un poco más y resolver ciertas limitaciones que podrían presentarse.

5.2. Mata Franco Carlos Jesus

Finalmente con el termino del proyecto podemos encontrar que el problema del **Máximo Cliqué** tiene distintas aplicaciones en la vida real enfocadas principalmente en el análisis del comportamiento de ciertos individuos o elementos que tienen caracterisiticas o relaciones en común ya sea en el área social o en algún ámbito

biologíco, etc.
En cuanto a la aplicación se pudo observar que cumple con el cálculo del máximo cliqué y que, se puede usar en un caso más real, ya qué como se mencionó antes se puede cargar un mapa o imagen para pintar el grafo sobre la misma. Además que es una interfáz amigable para el usuario tanto para el uso como para el entendimiento del resultado.

5.3. Ruiz González Ian Alexander

Después de todo el trabajo realizado, encuentro la aplicación de algoritmos como algo más bastó de lo que pensaba. Tenía la idea de usar un algoritmo para cada cosa pero no me podía imaginar que un solo algoritmo tendría tantas aplicaciones diversas. Partir de un problema similar al agente viajero y darle una perspectiva a grupos de redes sociales, adn, geología, solo para encontrar la mayor compatibilidad entre grupos de individuos, genes, etc. Es algo increíble! El desarrollo del proyecto partiendo de que tenga una interfaz gráfica fue algo nuevo, no había tenido la necesidad de crearlas antes y descubrir algunos trucos en netbeans fue clave en todo esto, la simplicidad de nuestro programa lo hace útil para practicar en él y meterse de lleno al problema del clique máximo, el apartado de cargas mapas (aunque sólo sea una imagen) lo hace más útil a un aplicación real (Google maps en mini) y es mejor trabajar un ejercicio de este tipo con algo que puedas ver de forma práctica a solo con matrices o nodos.