Evaluación 2 - Taller de Programación
Clique en grafos

Prof: Pablo Román

Abril 17 2024

1 Objetivo

Desarrollar una búsqueda del *clique* máximo de mayor tamaño que pueda existir en un grafo. Se explorará el algoritmo de Bron-Kerbosh para ser adaptado en la búsqueda del mayor clique. Se implementará en el lenguaje C++, con la normativa del curso de compilación separada, la estructuración y buenas prácticas de un código orientado al objeto. Adicionalmente se podrán utilizar estructuras de datos de STL que permita obtener eficiencia en la resolución del problema particular.

2 Problema del máximo clique

En un grafo no dirigido con nodos $\{i = 0 \dots n-1\}$ se puede representar mediante la matriz de $n \times n$ de adjacencia A. Donde $A[i][j] = 1$ si el nodo i esta conectado con j, en caso contrario es 0. Existen varias otras representaciones dependiendo cual es la información a la cual acceder. Un ejemplo de matriz de adjacencia es:

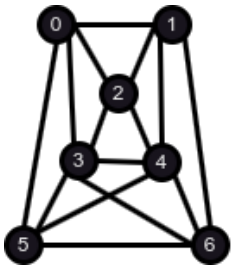

Figure 1: Ejemplo de grafo y su matriz de adyacencia.

$$\begin{array}{ccccccc} 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Un clique es un subconjunto de nodos donde todos se encuentran conectados con todos (`https://en.wikipedia.org/wiki/Clique_(graph_theory)`). Por ejemplo el clique mas pequeño de un grafo no vacío contiene solo un nodo ya que existe solo 1, si es que se puede conectar el nodo con sigo mismo. En la figura 2 se muestran varios subgrafos de un grafo de tamaño 7. El subgrafo marcado en rojo es un clique. **Un clique máximo es aquel que no se le puede agregar un vértice adicional de forma que siga siendo clique.** Dentro de un grafo pueden existir varios **cliques máximos**. Encontrar el clique máximo de mayor tamaño tiene importantes aplicaciones. Este problema es de gran interés en variados problemas de la vida real, en análisis de textos o detección de plagio se requieren segmentos de textos/musica de documentos/audios que se encuentren a una distancia mínima de modificación. Aquellos documentos/audios que conformen un clique tienen alta probabilidad de tener plagio. La misma técnica se

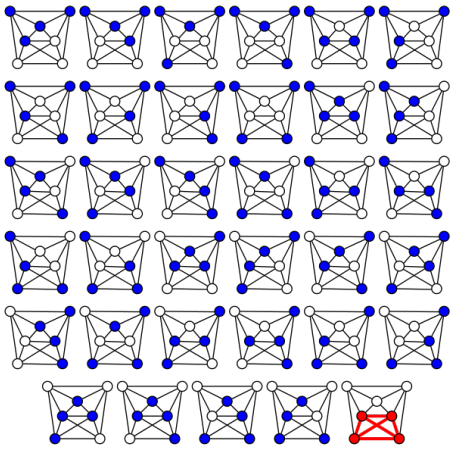

Figure 2: Clique máximo (rojo) de tamaño 4 en grafo de 7 nodos (Fuente: Wikipedia)

utiliza para agrupar cadenas de dna similares en genética. Se utiliza en el análisis de redes sociales para encontrar comunidades que comparten las mismas preferencias. Por lo mismo en marketing, para seleccionar que campaña dirigir a que comunidad. El problema de encontrar el máximo entre todos los clique se ha investigado en ciencia de la computación durante decadas (`https://link.springer.com/chapter/10.1007/978-1-4757-3023-4_1`). Este problema es NP-hard, pero se han generado múltiples algoritmos que logran encontrar en tiempo exponencial la solución. Muchas de las optimizaciones posibles tienen relación con heurísticas y estructuras de datos que hacen que las búsquedas sean mas eficientes.

3 Algoritmos asociados al clique máximo: Bron-Kerbosh

El algoritmo de Bron-Kerbosh (`https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch_algorithm`) es un algoritmo clásico inicialmente formulado de manera recursiva y que permite muchas variantes para ser optimizado tanto para la búsqueda (utilizando heurísticas) o de implementación (mejorando estructuras de datos).

3.1 Bron-Kerbosh Recursivo

El algoritmo en su versión general se presenta:

Algorithm 1 BronKerbrosh(R,P,X,CLIQUE)

if $P \rightarrow$ isEmpty() & $X \rightarrow$ isEmpty() **then** ▷ Si no quedan mas vértices a revisar (pasaron todos a R) y de ellos no hay ningún eXcluído entonces tenemos un clique máximo en R
  return $CLIQUE \rightarrow$ push(R)
end if
  ▷ R Si no se cumple la condición aún no es un clique máximo porque quedan vértices por visitar en P
for $x \in P$ **do** ▷ Notar que si P es vacío y X no lo es, entonces se descarta este R porque retorna inmediatamente.
  $R1 = R \cup \{v\}$
  $P1 = P \cap$ Vecinos(v) ▷ Consideramos explorar solo los vecinos de v
  $X1 = X \cap$ Vecinos(v) ▷ Considerar los excluidos que sean vecinos de v
  BronKerbrosh($R1, P1, X1$) ▷ Aquí buscamos aumentar el clique en R revisando todos los vecinos de v
  ▷ No se agrega v a R si la recursión anterior falla en encontrar clique
  $P = P \rightarrow$ delete(v) ▷ Sacamos v de los vértices Potenciales
  $X = X \cup \{v\}$ ▷ Se marca v eXcluído
end for
return CLIQUE ▷ Se retorna todo lo que haya sido encontrado en cada recursión

El algoritmo 1 es recursivo , se inicia con R y X ambos vacíos, y P con todos los vértices del grafo. R es el conjunto Resultado que va acumulando vertices de P para conformar un clique. X es el conjunto de los vértices que

se eXcluyen en el proceso. Esta exclusión se hace para no repetir el posible clique que se buscó en la recurrencia anterior. P el conjunto de los Potenciales vértices de un clique y $CLIQUE$ el conjunto que acumula conjuntos cliques máximos encontrados. El algoritmo agrega uno a uno los vértices que están en P a R y revisa en cada paso que se encuentre conectado con todos los vértices utilizando el conjunto Vecinos(v). Porque si v esta en un clique todos sus vecinos deberían estar en el clique. Esto lo hace en la recursión BronKerbrosh($R \cup v, P \cap \text{Vecinos}(v), X \cap \text{Vecinos}$). Cuando se vacía P pero sobra algún un elemento en X entonces R ya fue explorado. Porque significa que estamos revisando nuevamente R. Por ejemplo si tuviésemos solamente los nodos

$$(0, 1, 2)$$

, y que estamos en el segundo ciclo de la iteración con $v = 1$, $R = \{0\}$, $P = \{1, 2\}$ $X = \{0\}$, para la primera recursión se conforma $v = 2$, $R = \{0, 1\}$, $P = \{2\}$ $X = \{0\}$, para la segunda $R = \{0, 1, 2\}$, $P = \{\}$ y $X = \{0\}$; lo que termina sin nuevo clique. En cada recursión tanto X como P se restringen al nodo por revisar v y todos sus vecinos. Esto depende de la función Vecinos(v) que entrega el conjunto de vecinos de un cierto vértice v. Notar si en el grafo original colocado en P inicialmente, tiene todos sus nodos conectados con todos (es decir es un clique en si mismo), entonces los vecinos de cualquier v son todo el resto del conjunto. En ese caso la recursión va a ir disminuyendo el conjunto P en el nodo v y su intersección con sus vecinos sigue siendo todo el conjunto P restante. En cada recursión se agrega el nodo v a R hasta que R sea igual a P original. Esto retorna el grafo original que era de por si un clique.

¿Que pasa con el resto de las ramas de la recursión relativas al for? Cuando retorna finalmente la llamada a la función se agrega al conjunto X el vértice v. Si el vértice nuevo v' que se agrega es parte del clique anterior anterior entonces después en algunas de las intersecciones recursivas X no va a ser vacío incluso cuando se acaben los valores de P. Esta es la utilidad de X porque el caso anterior evita que se repita el clique anterior. Si el vértice nuevo v' que se agrega es parte de otro clique máximo entonces la recursión solo selecciona el otro clique máximo.

El problema de esto es que se generan muchas ramificaciones en el peor caso igual a la cantidad de vértices N, es decir del orden de N^k ramificaciones donde k es la profundidad!

3.2 Variaciones más eficientes del algoritmo de Bron-Kerbosh

El problema principal es cuando se realiza la búsqueda en todas las ramas revisando todas las combinaciones posibles y lo relativo a la recursión que repite casos posibles. Es posible acelerar en forma Heurística este algoritmo lo que produce que se pueda lograr algún resultado en un tiempo razonable.

1. **Algoritmo de Bron-Kerbosh con pivote**: En este caso se evita seleccionar el vértice v en la vecindad de cierto vértice u denominado pivote. Esto disminuye la ramificación de las llamadas recursivas. Si justamente se escoja el vértice u que se encuentre en el clique máximo entorno a v en todas las ramas entonces nunca va a encontrar nada. Si se efectúa de manera aleatoria entonces hay gran probabilidad de seguir encontrando el clique máximo, pero puede que no disminuya mucho la ramificación. Otra forma es escoger el primer nodo posible o el de mayor grado, pero en cierto casos se muestra que no agrega mayor eficiencia. Una estrategia que muestra mayor estabilidad (`https://www.sciencedirect.com/science/article/pii/S0304397508003903`) es escoger u en el conjunto $P \cup X$ de tal forma que u sea el nodo con mayor cantidad de conexiones con $P \cap \text{Vecinos}(u)$. El algoritmo 2 detalla el procedimiento descrito, pero la modificación es menor.

2. **Selección heurística del vértice en el ciclo:** Durante la ramificación de la recursión (en el for) se selecciona un nodo de cierto conjunto. En la recursión original es P, en la recursión con pivote es $P \setminus \text{Vecinos}(u)$. El asunto es cual es el criterio para seleccionar el mejor vértice. Una heurística comúnmente utilizada es escoger en dicha selección el nodo con mayor grado para aumentar la probabilidad de encontrar un clique de mayor tamaño.

3. **Poda de ramificaciones (evitar algunas llamadas recursivas):** La idea es evitar llamadas recursivas que lleven a resultados inconducentes (por ejemplo cliques de menor tamaño). El problema es la detección de dichas ramificaciones. Para el caso en que sólo se requiera encontrar el mayor clique máximo se puede estimar una cota para el clique a encontrar en dicha rama. $P \rightarrow size + R \rightarrow size$ es una cota superior del tamaño que pueda llegar a tener el clique en dicha rama. Si el tamaño del clique máximo obtenido hasta el momento o el tamaño estimado es T y la rama actual tiene el potencial de llegar a un clique de tamaño $P \rightarrow size + R \rightarrow size < T$ entonces conviene no proseguir la exploración de dicha rama. Esto podaría pasos innecesarios en el árbol de ramificaciones recursivas. Para estimar el tamaño del clique de mayor tamaño se puede considerar el máximo que se dispone o que en un clique de tamaño T todos sus nodos tienen

Algorithm 2 BronKerbroshPivote(R,P,X,CLIQUE)

```
if P → isEmpty() & X → isEmpty() then
    return CLIQUE → push(R)
end if
u = getOptimalPivot(P, X)                    ▷ Obtenemos el pivote
for x ∈ P \ Vecinos(u) do                    ▷ No consideramos los vecinos del pivote al ramificar
    R1 = R ∪ {v}
    P1 = P ∩ Vecinos(v)
    X1 = X ∩ Vecinos(v)
    BronKerbrosh(R1, P1, X1)
    P = P → delete(v)
    X = X ∪ {v}
end for
return CLIQUE
```

exactamente $T-1$ conexiones (grado del vértice), por lo tanto si en alguna rama existen suficientes nodos de grado $T-1$ a revisar y es el máximo grado con esas características entonces T es un candidato a un clique se se pueda encontrar. Otra forma más sofisticada de podar una ramificación es colorear el grafo actual. Un grafo coloreado con C colores tiene cada vértice asociado con un color distinto a todos los colores de sus vecinos. Ahora en un clique la cantidad de colores que pueden haber es igual al tamaño del clique, debido a que están todos conectados con todos. Entonces si consideramos que una rama tiene un potencial clique en $P \cup R$ y se disponen de $C < T$ colores entonces se puede descartar dicha rama. Porque el clique tendrá menos vértices que dichos colores. El coloreo óptimo de un grafo es también un problema NP-completo. Pero existen algoritmos mas rápidos para realizar esto `https://en.wikipedia.org/wiki/Greedy_coloring`.

4 Implementación

Se requiere implementar diferentes objetos que permitan encontrar el mayor clique contenido en un grafo. La implementación debe ser lo mas eficiente posible. Intersecciones/Uniones/Restas/otras entre conjuntos, selección de vértices y otras operaciones (delete, insert,find ....) deben ser lo más eficientes posibles. La implementación debe ser en base al algoritmo de Bron-Kerbosh con modificaciones heurísticas, de eficiencia y adaptación a encontrar el clique de tamaño máximo. Debe incluir una interfaz de usuario que permita indicar el nombre del archivo con la matriz de números en desorden. Estos números vienen separados por un espacio. Los archivos de entrada:

```
4
0 1
0 2
1 2
```

Donde la primera fila indica la cantidad de vertices y las siguientes columnas indican conexión entre vértice inicial a vértice final. Para este caso la solución es $\{0, 1, 2\}$

Otro ejemplo:

7
0 1
0 2
0 3
0 5
1 2
1 4
1 6
2 3
2 4
3 4
3 5
3 6
4 5
4 6
5 6

En este caso la solución es $\{3, 4, 5, 6\}$.

La salida a entregar en el programa main es imprimir los nodos que conforman el clique mayor, su tamaño y el tiempo que demoró la ejecución.

Además el entregable debe contener:

- Un makefile que compile programa principal y programas de test con compilación separada.
- Un test por cada clase generada. Debe comprobar que efectivamente la clase funcione.
- Varias clases implementando razonablemente la funcionalidad pedida.
- Un main.cpp con un menu para seleccionar archivo de entrada. Se resuelve el problema entregando, los nodos del clique, si es que tiene solución y el tiempo que demoró en resolverse. Si no tiene solución debe indicarlo. Se repite en un loop que incluye un salir.
- pdf de informe según plantilla a entregar vía classroom
- varios ejemplos de entrada. Dichos ejemplos los puede obtener de la página de Google.
- Formato de nombre y carpetas que se especificarán por classroom.

El programa a construir debe estar codificado en C++. Además de las estructuras de datos auxiliares más **una heurísticas** que permita que la simplificación sea los más óptima posible. Debe entregar una carpeta comprimida cuyo nombre corresponda a ApellidoNombre.zip (u otra compresión). Toda clase debe ser implementada por separado en dos archivos (.h y .cpp). El nombre de una clase siempre comienza en Mayúscula y el archivo que la implementa tiene su nombre. No se permite definir funciones fuera de las clases y cada clase debe servir a un propósito o abstracción.

Se verificará eficiencia en casos de grafos de 50 nodos entregados vía classroom.

1. (10%) Informe : contiene una descripción de la estrategia de resolución que se utilizó y el porqué su implementación es eficiente.
2. (30%) Código : se revisa si compila (0 pts si no compila), si se implementa el algoritmo propuesto, si tiene la estructura indicada (clases+test+main+makefile), si se efectúa compilación separada, si esta todo descrito por clases y no hay funciones sueltas salvo la función main, si el makefile opera bien, si el código se entiende (comentarios, variables con nombres descriptivos, identación, sin repeticiones forzadas), si se encuentra 1 test adecuado por clase, si el programa se ejecuta sin problemas y efectúa las operaciones que se espera de la tarea (si no ejecuta en ningún caso 0 pts). **Si tiene funciones fuera de las clases tiene 0 puntos.**
3. (50%) Programa correcto y Eficiencia (0 pts si no funciona.): Se revisa que no existan errores de lógica y/o ejecución, que entregue los resultados correctos en todos los casos (10%). Se revisa que se haya implementado el problema de manera eficiente (40%, 0 si no hay heurística o estructuras de datos eficientes).

4. (10%) Revisión por pares. Cada compañero revisa el código a otro con una nota. Se revisará dicha revisión con otra nota. Si la revisión tiene defectos notables, la nota anterior se reemplaza por otra nota indicada por el profesor. La nota final es la nota propuesta por un par (si es que está correcta) promediada con la evaluación de la revisión que realizó el alumno.

5. (+ 1pto) Bonus al programa que genera la solución en el menor tiempo posible. Si hay varios con la menor cantidad de pasos, entonces se selecciona al programa mas rápido.

5 Entrega

27 de mayo a las 11:55PM entrega que incluye lo anterior. 1 punto por día de atraso considerando entrega hora/fecha posterior a la indicada. 3 días después: entrega de la revisión de a pares. El archivo entregable es una carpeta con nombre-rut del alumno y comprimida en un archivo con nombre-rut con los archivos requeridos.

vía Classroom