Proyecto 3 Estructuras de Datos: Mini-Twitter

Pablo Plaza Soto Vicente Varas Pavez

23 de Junio de 2017

Introducción

El problema dado consiste en la creación de una red social tipo Twitter. Para esto se utiliza un digrafo que representa nodos que siguen a otros nodos. La entrada se hace mediante un archivo de entrada con comandos que permiten: agregar una relación entre nodos, encontrar si un nodo está presente, encontrar cliques en el grafo, desplegar una versión compacta del grafo, y desplegar los nodos más seguidos. Como los nodos se añaden junto a las relaciones, el grafo resutante será uno débilmente conexo, no se puede añadir nodos aislados.

Para la implementación se contempla la creación de dos clases: Graph y Node. Graph representa el grafo del mini-twitter y se implementa con una lista de adyacencia que corresponde a un vector(std::vector) de "Node".

Además de este vector y las funciones requeridas Graph tiene dos vectores de vectores de nodos, usados para almacenar los cliques y el grafo compactado, y dos funciones privadas: "BK" y "BK2". Estas dos funciones son implementaciones de la forma más básica del algoritmo de Bron-Kerbosch para encontrar cliques. La diferencia entre ellas es que una encuentra sólo los cliques maximales compuestos por mas de 3 nodos, para el método Clique(), mientras que la otra encuentra todos los cliques maximales y se usa para el método Compact(). Al crear estas funciones hubo que considerar que se está trabajando con un grafo dirigido y en ese sentido se consideró que nodos vecinos son aquellos que se "siguen" mutuamente.

Implementación de los métodos

Find(u)

Para este método se hicieron dos implementaciones distintas, las que se encuentran en la clase Graph como Find_g(u) y Find_map(u). La primera usa BFS para recorrer el grafo y encontrar el nodo, mientras que la segunda usa la clase Map de la STL. A continuación se encuentra la comparación teórica y experimental de estas implementaciones.

Comparación teórica

- Find_g:
  Basado en BFS. Se recorre el grafo a partir del primer nodo en el vector (de un total de n nodos), se revisa si ese nodo tiene por nombre el string buscado, y si no lo tiene se visita a sus vecinos, lo que en total, para todo el grafo, se hará a lo más m veces, siendo m el número de aristas del grafo. Por tratarse de un dígrafo, es posible que una sola BFS no visite todos los nodos, y ya que el objetivo es encontrar un nodo en particular, se incorporó una revisión de qué nodos falta por revisar cuando la queue queda vacía, haciendo BFS desde el nodo que falte, hasta así visitarlos todos. En cuanto a espacio adicional, requiere n booleanos para marcar los nodos visitados y la queue ya mencionada.

  En el peor caso, en que todos los nodos estén aislados, tendría complejidad teórica $O(n^2)$, sin embargo este caso no se puede dar en este problema, ya que los nodos siempre se crean como seguidores de otro nodo, siendo así esperable una complejidad más cercana a $O(n + m)$.

- Find_map:
  Consiste en ingresar los nodos a un std::map usando su nombre como clave, para luego usar el método count para ver si hay algún nodo en el mapa con el string indicado como clave. Para esto necesita que se guarde el map con los punteros a los nodos, lo que implica usar espacio adicional.

  Ya que la inserción de los nodos al map se hace cuando son creados (en el método add), la complejidad es la del método count, es decir $O(log(n))$. Si la inserción se hiciera dentro del método Find, la complejidad sería n veces la del método insert, o sea $O(n * log(n))$. Cabe señalar que colocar la inserción en el método add no altera su complejidad, ya que add tiene que llamar a un método find, que de ser éste tendría la misma $(log(n))$.

Comparación experimental

En la figura 1 se muestra los tiempos en segundos que tardaron las dos versiones del método Find en buscar los nodos indicados para el grafo de la figura 2. El primer nodo creado es A, por lo que Find_g siempre empieza por él. Se ve que en todos los casos Find_map demora menos, y con tiempos similares se encuentre el nodo o no. Por esto, es preferible usar Find_map siempre que no haya problemas de espacio con almacenar el map.

Método	Nodos encontrados				
	A	B	C	E	J
Find_g	1,10E-05	7,00E-06	5,00E-06	7,00E-06	1,30E-05
Find_map	1,00E-06	1,00E-06	1,00E-06	1,00E-06	1,00E-06
Método	Nodos encontrados				
	K	L	P	Q	W
Find_g	1,30E-05	1,40E-05	1,50E-05	1,60E-05	1,70E-05
Find_map	2,00E-06	2,00E-06	2,00E-06	2,00E-06	1,00E-06
Método	Nodos no encontrados				
	X	Y	Z		
Find_g	1,80E-05	1,70E-05	1,70E-05		
Find_map	2,00E-06	2,00E-06	1,00E-06		

Figura 1: Tiempos de búsqueda de las variantes de Find.

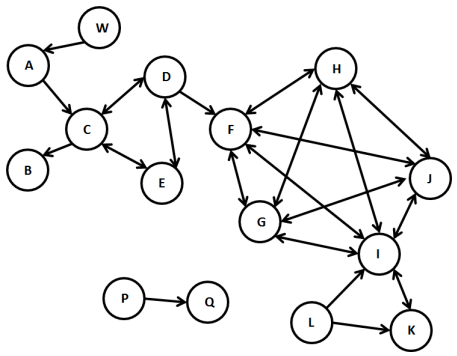

Figura 2: Grafo usado en la comparación experimental

Clique()

Para este método se implementó el algoritmo Bron-Kerbosch que encuentra los cliques maximales de un grafo. Este es un algoritmo recursivo que se basa en la utilización de tres conjuntos: R, P y X. En R se almacena un posible clique maximal, P contiene los nodos vecinos de R y X contiene a los nodos que ya se han considerado. Para representar a estos conjuntos se usaron vectores auxiliares denominados de la misma forma. La función "BK" recibe como argumento estos tres vectores y se llama de forma recursiva. Cuando los vectores P y X están vacios, se reporta R como clique maximal, agregandolo al vector de vectores de nodos cliques, que es una variable privada de la clase Graph. A continuación se muestra el pseudocódigo del algoritmo, en el que se basó la implementación.

```
BronKerbosch(R, P, X):
   si P y X estan vacíos:
       reportar R como clique maximal
   para cada vertice v en P:
       BronKerbosch(R unión {v}, P intersección N(v), X intersección N(v))
       P := P \ {v}
       X := X unión {v}
```

Compact()

Este método se implementó usando el mismo algoritmo que el usado para encontrar los cliques maximales, sólo que en vez de tomar en cuenta solo los de tamaño mayor o igual a 3, se consideraron todos. Con esto se consigue identificar todos los subgrafos completos maximales del grafo. La implementación de este algoritmo se encuentra en la función "BK2". Estos subgrafos se almacenaron en un vector y luego se revisaron sus relaciones de forma iterativa a la vez que se despliegan de la forma requerida.

Follow(n)

Para la implementación de este método se crea una priority_queue que almacena punteros a Nodes en un vector y usa la clase comparaSeg para hacer las comparaciones entre nodos según su número de seguidores (obtenido con getSeg()). Se insertan todos los nodos en esta priority_queue y luego se hace top y pop n veces, entregando así los n nodos con más seguidores.

Resumen de ficheros

- Graph.h y Graph.cpp:
  Implementan la clase Graph que representa las relaciones del mini-twitter.
- Node.h y Node.cpp:
  Implementan los nodos usados en la clase Graph.
- main.cpp:
  Programa principal que lee el archivo con las entradas. Instancia un Graph y usa sus métodos. Al ejecutarlo se debe entregar el nombre del archivo con las entradas como argumento.(vg: $./main input.txt)
- input.txt:
  Archivo de texto con las entradas de ejemplo dadas.