Universidad Simón
Bolívar

Reporte de Laboratorio de Semana 5

Análisis de algoritmos sobre grafos

Autor:
Christopher Gómez

Profesor:
Guillermo Palma

Laboratorio de Algoritmos y Estructuras de Datos III (CI2693)

15 de noviembre de 2021

1. Metodología

El siguiente estudio experimental consiste en correr un conjunto de algoritmos sobre grafos no dirigidos para la búsqueda de componentes conexas y árboles mínimos cobertores sobre tres grafos de prueba de distinto número de vértices y lados, midiendo sus tiempos de ejecución y tomando nota de los resultados.

Solamente se toma el tiempo de ejecución del algoritmo.

Los algoritmos a ejecutar, implementados en el lenguaje de programación Kotlin, serán los siguientes:

- **Algoritmos para hallar componentes conexas**:
  - Basado en la estructura de conjuntos disjuntos.
  - Basado en el algoritmo de Búsqueda en Profundidad.
- **Algoritmos para hallar árboles mínimos cobertores**:
  - Algoritmo de Kruskal (usando en la estructura de conjuntos disjuntos).
  - Algoritmo de Prim (usando una cola de prioridad basada en un heap binario).

Los siguientes son los detalles de la máquina y el entorno donde se ejecutaron los algoritmos:

- **Sistema operativo**: Debian 11 Bullseye 64-bit.
- **Procesador**: Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz.
- **Memoria RAM**: 4,00 GB (3, 88 GB usables).
- **Compilador**: kotlinc-jvm 1.5.31 (JRE 11.0.12+7-post-Debian-2).
- **Entorno de ejecución**: OpenJDK Runtime Environment (build 11.0.12+7-post-Debian-2).
- **Flags (máquina virtual)**: -Xmx2g.

2. Resultados experimentales

2.1. Algoritmos para hallar componentes conexas

Los Cuadros 1 y 2 muestran los resultados obtenidos de aplicar los algoritmos para hallar componentes conexas sobre los grafos de prueba.

Como se puede observar em el Cuadro 1, ambos algoritmos muestran los mismos resultados al momento de hallar componentes conexas de un grafo no dirigido, lo cual nos indica que, a pesar de diferir en su funcionamiento, son equivalentes y dan el resultado correcto.

Archivo	Conjuntos disjuntos	Búsqueda en Profundidad
`pequenio.txt`	3	3
`mediano.txt`	1	1
`grande.txt`	1	-

Cuadro 1: Número de componentes conexas detectadas según cada algoritmo.

	Tiempo (ms)	
Archivo	**Conjuntos disjuntos**	**Búsqueda en Profundidad**
`pequenio.txt`	8	1
`mediano.txt`	15	2
`grande.txt`	9423	-

Cuadro 2: Tiempo de ejecución de los algoritmos para hallar componentes conexas.

Por otro lado, es notorio observando el Cuadro 2 que el algoritmo basado en Búsqueda en Profundidad se ejecuta considerablemente más rápido que el basado en la estructura de Conjuntos Disjuntos para los archivos `pequenio.txt` y `mediano.txt`, sin embargo, para el archivo `grande.txt` se obtuvo un resultado singular, y es que mientras el algoritmo basado en Conjuntos Disjuntos se ejecutó en un tiempo de 12,8s aproximadamente, el algoritmo basado en Búsqueda en Profundidad arroja el siguiente error de memoria antes de terminar de ejecutarse, por lo que no se obtienen resultados:

```
Exception in thread "main" java.lang.StackOverflowError
    at ve.usb.grafoLib.GrafoNoDirigido.adyacentes(GrafoNoDirigido.kt:142)
    at ve.usb.grafoLib.ComponentesConexasDFS.dfsVisit(ComponentesConexasDFS.kt:52)
    at ve.usb.grafoLib.ComponentesConexasDFS.dfsVisit(ComponentesConexasDFS.kt:57)
    at ve.usb.grafoLib.ComponentesConexasDFS.dfsVisit(ComponentesConexasDFS.kt:57)
                                        .
                                        .
                                        .
```

La razón de que esto ocurra en el archivo `grande.txt` es que este contiene un grafo de un millón de vértices y más de siete millones de aristas, y además, como se observa en el resultado dado por el otro algoritmo, que el grafo tiene una sola componente conexa, lo cual implica que la búsqueda profunda puede necesitar mantener más de 3 mil llamadas recursivas[1] para transversar determinados caminos en el grafo, causando rápidamente[2] un desbordamiento de la pila (con la configuración referida en la sección 1).

Salvo por esta excepción, los resultados obtenidos para estos algoritmos concuerdan con la teoría, de la cual podemos deducir que el algoritmo basado en Conjuntos Disjuntos, usando las heurísticas de compresión de camino y de unión por rango, tienen un tiempo de ejecución de $O((|V| + ||E)\alpha(|V|))$ (donde $\alpha(|V|)$ es una constante menor a 4), ya que involucra $|V| + |E|$ operaciones de `Make-Set`, `Find-Set` y `Union`, de las cuales $|V|$ son `Make-Set`. Por otra parte, el tiempo de ejecución del algoritmo basado en DFS es de solo $O(|V| + |E|)$, lo cual indica que las constantes involucradas en el otro algoritmo hacen una diferencia notoria en la práctica.

[1] Cifra tomada de una prueba que no se halla en el código fuente adjunto, en la que se cuentan la cantidad de llamadas recursivas que se mantienen en la ejecución del algoritmo. El resultado mostró 3336 llamadas empiladas antes de lanzar error por desbordamiento de pila.

[2] En menos de 2s. según mediciones con herramientas externas

Archivo	Tiempo (ms)
pequenio.txt	0
mediano.txt	3
grande.txt	1164

Cuadro 3: Tiempo de ejecución del DFS Iterativo para hallar componentes conexas.

A raíz de estas últimas pruebas, se decidió adicionalmente implementar una versión iterativa de la Búsqueda en Profundidad, evitando las llamadas recursivas de `DFS-Visit` mediante una pila de vértices. Los resultados se recogen en el Cuadro 3, y del mismo podemos ver que, a pesar de obtener unos resultados similares a los de la versión recursiva para los mismos archivos de prueba, esta vez sí se logra terminar la ejecución y el tiempo para `grande.txt` es más de 8 veces menor al resultado para el algoritmo de Conjuntos Disjuntos, lo cual deja en evidencia la eficiencia del algoritmo.

2.2. Algoritmos para hallar árboles mínimos cobertores

En los Cuadros 4 y 5 se muestran los resultados de ejecutar los algoritmos de Kruskal y Prim sobre los archivos dados.

Archivo	Kruskal	Prim
gnd-pequenio.txt	1.81	1.8099999999999998
gnd-mediano.txt	10.463510000000001	10.463509999999994
gnd-grande.txt	647.6630695500033	647.6630695499884

Cuadro 4: Peso calculado del árbol mínimo cobertor hallado según cada algoritmo.

	Tiempo (ms)	
Archivo	Kruskal	Prim
gnd-pequenio.txt	22	2
gnd-mediano.txt	34	9
gnd-grande.txt	21016	5762

Cuadro 5: Tiempo de ejecución de los algoritmos para hallar árboles mínimos cobertores.

Los resultados obtenidos muestran que en todos los casos el algoritmo de Prim es más eficiente que la implementación del algoritmo de Kruskal usando Conjuntos Disjuntos. Como se puede observar en el Cuadro 4, ambos algoritmos hallan árboles mínimos cobertores con pesos muy parecidos, lo cual verifica que en ciertos casos tanto el algoritmo de Prim como el de Kruskal, a pesar de funcionar de forma distinta, hallan el mismo árbol mínimo cobertor. Aunque no se verificó para los archivos `gnd-mediano.txt` y `gnd-grande.txt`, para `gnd-pequenio.txt`, efectivamente, los árboles mínimos cobertores son idénticos, sin embargo, el cálculo del peso da una diferencia de $2 * 10^{-17}$ entre Kruskal y Prim, lo cual puede deberse a que en cada arbol las aristas son agregadas en la lista de lados en orden distinto, generando errores de redondeo en la aritmética de precisión finita que se efectúa en el computador para operaciones de punto flotante al momento de sumar los pesos en distinto orden.

Por otro lado, para los árboles en `gnd-mediano.txt` y `gnd-grande.txt` no se puede asegurar si ambos árboles son exactamente los mismos y se trata nuevamente de errores en la aritmética, o si son distintos árboles, pero sí se puede notar que tanto Kruskal como Prim son capaces de dar resultados

razonables, aunque para el caso de Prim dependa del vértice que se escoja como fuente.

Tal como se enuncia en la teoría, los algoritmos con Conjuntos Disjuntos no suelen preferirse por encima de otros algoritmos para resolver los mismos problemas. En este caso, aunque tanto el algoritmo de Prim como el de Kruskal tienen teóricamente un tiempo asintótico de $O(|E|log|V|)$, en la práctica podemos ver que que el algoritmo de Prim resulta siempre alrededor de 4 veces más rápido, lo cual puede ser un indicativo de que las operaciones sobre la estructura de Conjuntos Disjuntos pueden resultar más costosa que las operaciones sobre heaps binarios, y que el procedimiento de ordenar las aristas por sus pesos esconde constantes que en aplicaciones prácticas en necesario tener en cuenta.

2.3. Conclusiones

Finalmente, la implementación de estos algoritmos, los resultados obtenidos en las pruebas, y su posterior análisis permiten obtener las siguientes conclusiones:

- Aunque la estructura de Conjuntos Disjuntos es facil de implementar y es útil para usar en algoritmos más sencillos de entender, generalmente suelen tener un peor desempeño en la práctica.
- Los algoritmos recursivos pueden resultar más elegantes y fáciles de entender, sin embargp, en ciertos casos se puede presentar un número excesivo de llamadas recursivas, dando lugar a fallos en su funcionamiento, por lo que es recomendable cuidar los detalles de sus implementaciones o usar alguna variante iterativa si se sabe que el algoritmo será usado para propósitos que manejen instancias grandes de problemas y pocos recursos.
- La Búsqueda en Profundidad puede ser modificada para muchos propósitos distintos, además de el de encontrar caminos o marcar tiempos.
- Algoritmos con la misma complejidad temporal pueden desempeñarse en la práctica de forma muy distinta.
- Es recomendable implementar formas de evitar errores en la aritmética si es de suma importancia la precisión de los cálculos para los pesos de los AMC.
- Es importante asegurar la escalabilidad de las implementaciones que se hacen, pues a menudo las aplicaciones de la vida real involucran instancias de problemas gigantescas, y se necesita no perder eficiencia ni eficacia en los resultados. Asegurar la escalabilidad requiere en algunos casos repensar los algoritmos o el funcionamiento de las estructuras que este involucra, o hasta implementarlas y prescindir de librerías, para que se acoplen de la mejor forma a las necesidades.