Evaluación 3 - Taller de Programación
Máximo Flujo en Grafos

Prof: Pablo Román

Diciembre 03 2024

1 Objetivo

Desarrollar una aplicación eficiente para encontrar soluciones al problema de máximo flujo generalizado. En particular se expone el problema de configurar la distribución de materiales escolares desde bodegas a diferentes liceos públicos. Para estos fines puede basarse en los algoritmos de Ford-Fulkerson, Edmonds Karp, Dinic u otra variante más eficiente. Se implementará en el lenguaje C++, el uso de makefile, la estructuración y buenas prácticas de un código orientado al objeto. Se utilizarán las librerías de C++ STL para incrementar la eficiencia.

2 Problema a resolver: distribución de productos desde diferentes orígenes a distintos destinos

El problema consiste en distribuir mercadería o bienes. Dicha planificación es muy común en proyectos de ingeniería relativos a la distribución de bienes para supermercados, o para materiales escolares en liceos de Chile, o la forma en que se despacha en mercadolibre o Amazon. Se disponen de varios orígenes o_k o fuentes y varios destinos o sumideros s_l. Entre medio hay centros i de redistribución o Hub, entre los centros de distribución $i \rightarrow j$ hay caminos para repartir productos. Dichos caminos tienen una capacidad máxima c_{ij} para transportar productos. El asunto es que se tiene que llevar el máximo posible total a los distintos destinos dada la configuración de capacidades de transporte. Este problema es conocido como el problema de máximo flujo con múltiples fuentes y sumideros. La salida de este problema son el flujo máximo de salida por cada fuente y de entrada por cada sumidero. Este

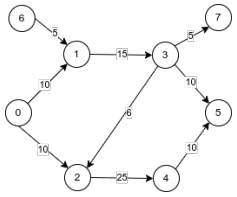

Figure 1: Red con dos fuentes (0,6) y dos sumideros (5,7).

problema se puede transformar a un problema clásico de máximo flujo considerando agregar una fuente artificial única conectada a cada fuente. Dichas conexiones se les asigna una capacidad máxima de asociado a la máxima capacidad que pueda salir de cada fuente. Esto se calcula sumando las capacidades que salen de cada fuente. Lo mismo para los sumideros, se genera un sumidero artificial que recibe una conexión de cada sumidero con capacidad igual a la suma de las capacidades que llegan a cada sumidero. De esta forma se puede resolver con los algoritmos

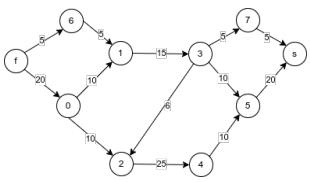

Figure 2: Red de Figura 1 modificada ahora con únicos fuente (f) y sumidero (s)

tradicionales de flujo máximo.

3 Algoritmos conocidos para resolver el problema del Flujo Máximo en Grafos

Los algoritmos de flujo máximo en grafos son uno de los más importantes algoritmos conocidos. La razón es que son uno de los pocos que resuelven de manera exacta un problema sobre grafos dirigidos, lo que es excepcional entre los tipos de problemas basados en grafos. Por lo general, uno se topa con problemas exponenciales en complejidad. Revisamos los algoritmos disponibles para resolver este problema de flujo máximo en grafos dirigidos con capacidades enteras, una fuente y un sumidero.

Algoritmo de Ford-Fulkerson Este algoritmo fue la base de muchas otras estrategias de resolución del problema de flujo máximo. Se basa en el concepto de "camino aumentante". Este corresponde a una secuencia de vértices sin repetición y conectados por aristas, en un orden que se inicia en el vértice fuente y termina en el vértice sumidero. A este camino se le asigna un número que indica el flujo máximo que puede pasar en forma adicional por el grafo. El flujo de un camino no puede sobrepasar la mínima capacidad entre todas las arista que contiene mas los flujos que el grafo ya contiene. Entonces no siempre para cualquier camino se va tener un flujo que no sea 0, puede que las capacidades ya estén saturadas. Un camino aumentante es aquel que agrega mas flujo a la red, es decir a la salida sale más flujo que antes. La idea entonces es buscar caminos aumentantes modificando los flujos hasta no poder encontrar ningún camino que pueda aumentar el flujo total (que llega al sumidero o que sale de la fuente).

El algoritmo se puede resumir de una manera muy simple:

Algorithm 1 Algoritmo de Ford-Fulkerson

Require: G Graph
  while $(G \rightarrow existAugmentingPath())$ **do**
    $c = G \rightarrow getPath()$
    $f = c \rightarrow getMaxFlux()$
    $G \rightarrow updateFlux(c, f)$
  end while
  return $G \rightarrow getFlux()$

La forma en que esta presentado el algoritmo anterior es más bien para expresar de manera mas sencilla el funcionamiento global. No se implementa realmente la función $(G \rightarrow existAugmentingPath())$ que revisa si aún existen caminos aumentantes. En los libros de texto esta parte de la búsqueda se deja libre. Notar que dependiendo de la forma de buscar caminos aumentantes, es si se tiene garantía de término del algoritmo. Está flexibilidad origina una variedad de algoritmos que modifican esta búsqueda de caminos de forma de saturar de manera más eficientemente.

Algoritmo de Edmonds-Karp Este algoritmo específica la búsqueda de un camino aumentante en anchura. Esto lleva a encontrar el camino más corto al sumidero y garantiza término del algoritmo. El camino se forma buscando mientras existan aristas que tengan capacidad disponible para enviar flujo, si eso ocurre se agrega la arista al camino y se sigue conformando otros caminos. La complejidad de este algoritmo es de orden $O(V * A^2)$, donde V es el número de vértices y A es el número de aristas. Para grafos grandes este algoritmo puede llegar a tener una complejidad grande, digamos 1000 nodos 10000 aristas significa 100.000.000.000 de iteraciones. Un video explicativo de este algoritmo lo pueden encontrar en `https://youtu.be/RppuJYwlcI8?si=7UnqopfC_XESSFHk`, incluye explicación de código en `https://youtu.be/OViaWp9Q-Oc?si=sH8ui_2sQkE-5yVW`.

Algoritmo de Dinic Este algoritmo extiende con nuevos concepto el algoritmo de Ford-Fulkerson, y logra eficiencias mayores. Al igual que el algoritmo de Edmonds-Karp se realiza una búsqueda en anchura pero su complejidad estimada es $O(V^2 * A)$. Para el caso hipotético anterior de 1000 vértices resultan 10.000.000.000 iteraciones es decir 10 veces menos que el algoritmo anterior. Este algoritmo se basa en generar un sub-grafo que partiendo desde la fuente genera niveles de profundidad (nuevo atributo por vértice) al navegar el grafo en anchura. Este es el llamado "grafo de niveles" y solo considera vértices que se puedan acceder por aristas con capacidad disponible y sin volver atrás en el nivel. Se busca en profundidad los caminos aumentantes en este grafo y actualizando los flujos hasta saturar las capacidades disponibles, esto se llama "Flujo Bloqueante". Posterior a esto se genera un nuevo sub-grafo de niveles volviendo a repetir el proceso hasta no disponer de sub-grafos de niveles. Un video altamente explicativo de este algoritmo está en `https://youtu.be/M6cm8UeeziI?si=FEhCDoHfdf6q-Ulg`

y en la misma serie tienen videos de implementaciones Java de sus algoritmos además de un repositorio de github https://github.com/williamfiset/Algorithms/blob/master/src/main/java/com/williamfiset/algorithms/graphtheory/networkflow/examples/DinicsExample.java.

Este algoritmo tiene altas capacidades de ser implementado de manera mucho más eficiente. En particular el algoritmo de Sleator-Tarjan https://www.cs.cmu.edu/~sleator/papers/dynamic-trees.pdf utiliza una estructura de datos para poder combinar caminos aumentantes para producir otros https://www.arl.wustl.edu/~jon.turner/cse/542/text/sec19.pdf. Su complejidad llega a ser $O(V * A * \log(V))$.

Algoritmos del estado del arte La importancia de este problema en ingeniería a llevado a investigar nuevos algoritmos (https://en.wikipedia.org/wiki/Maximum_flow_problem) que tienen tiempos de ejecución mucho más eficientes y que pueden resolver problemas con grandes cantidades de aristas (https://www.quantamagazine.org/researchers-achieve-absurdly-fast-algorithm-for-network-flow-20220608/). En particular se buscan algoritmos que sean de orden lineal en el número de aristas https://inf.ethz.ch/news-and-events/spotlights/infk-news-channel/2023/12/almost-linear-time-algorithms-for-maximum-flow-and-minimum-cost-flow.html. Por ejemplo https://dl.acm.org/doi/10.1145/3610940.

4 Implementación

Se requiere implementar un programa que resuelva el problema del máximo flujo con múltiples fuentes y sumideros. Se entregarán grafos de ejemplo de gran tamaño por lo que el programa debe ser implementado de la forma más eficiente posible. Debe cuidar no tener clases con excesivas responsabilidades. Por ejemplo no se puede tener una clase que realice todo el algoritmo. Debe separar la responsabilidad de tener objetos dedicados al grafo y otros dedicados al algoritmo. Aun cuando el problema es exacto, no esta demás utilizar heurísticas para acelerar la búsqueda de caminos aumentantes.

Otro temas relativo a la orientación al objeto: No deben haber funciones sueltas, solo clases y funciones main para test y programa principal. Para ello será importante la forma en que se representará **el estado del sistema** y **las operaciones** a realizar. Es importante que el programa final deberá resolver en forma eficiente el problema. **En esta tarea DEBE utilizar librerías de STL que implementen tipos de datos eficientes**.

Debe incluir una interfaz de usuario (menú) simple que permita indicar el nombre del archivo que contiene el grafo con la configuración inicial.

El programa recibe como entrada la descripción del grafo como un archivo de texto. Este es un archivo que contiene:

1. La primera fila enumera los vértices fuentes
2. La segunda fila enumera los vértices sumideros
3. A partir de la tercera fila, se tienen 3 columnas que indican aristas y su capacidad. En este orden: origen destino capacidad

Por ejemplo para el caso del grafo de la figura 1:

```
0 6
5 7
0 1 10
0 2 10
1 3 15
2 4 25
3 2 6
3 5 10
3 7 5
4 5 10
```

La salida a entregar corresponde a valor de de flujo que sale de cada fuente, valor de flujo que entra a cada sumidero, el flujo total, y tiempo en resolver:

```
Fuentes:
0 20
6 5
```

```
Sumideros:
5 20
7 5
Flujo Total: 25
Tiempo: 0.1 [ms]
```

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.
- Cada clase son 2 archivos (.cpp) y (.h), que debe ser compilados en forma separada.
- Un main.cpp con un menú para seleccionar archivo de entrada. Se resuelve el problema entregando los colores por vértice y el tiempo que demoró en resolverse en milisegundos. Se repite en un loop que incluye un salir.
- pdf de informe
- varios ejemplos de entrada que demuestren que su programa funciona.

El programa a construir debe estar codificado en C++, implementando las funcionalidades de la manera mas eficiente posible. Debe entregar una carpeta comprimida cuyo nombre corresponda a ApellidoNombre.zip (u otra compresión). Esta carpeta contiene archivos Header (.h), clases (.cpp), programas principales (main.cpp y test_XX.cpp), al menos un test por clase, un makefile con compilación separada. 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 sólo propósito o abstracción.

1. (10%) Informe : contiene una descripción de la estrategia de resolución para encontrar la menor cantidad de colores, el porqué su implementación es eficiente y que explique como se utiliza su programa o cuidados que hay que tener.
2. (30%) Código : se revisa si compila (0 pts si no compila), si se implementa el algoritmo propuesto de tipo B&B, si tiene la estructura indicada (clases+test+main+makefile), si se efectúa compilación separada, si está todo descrito por clases y no hay funciones sueltas salvo la función main, si el makefile opera bien, si el código se entiende (comentarios claros y útiles, variables con nombres descriptivos, identación, sin repeticiones forzadas), si se encuentra 1 test adecuado por clase, si no tiene problemas de memoria, Se revisa que no existan errores de lógica y/o ejecución, que entregue los resultados correctos en todos los casos (si no ejecuta en ningún caso 0 pts). **Si tiene funciones fuera de las clases tiene 0 puntos.** Se revisa que haya una clara separación de responsabilidad entre clases (0% si no hay separación, 20% si hay algún grado de separación de responsabilidades en el algoritmo)
3. (50%) Programa Eficiente (0 pts si no funciona o no tiene ningún intento de hacerlo eficiente.): Se revisa que se haya implementado el problema de manera eficiente. Según lo que se exprese en el informe se verificará en el código cual es su propuesta de eficiencia. Por ejemplo si se utiliza una implementación conocida y no hay intento de mejorarlo son 0pts. Pero basta con que indique una buena justificación reflejada en el código y en los tiempos de ejecución.
4. (10%) Revisión por pares. Cada compañero revisa el código a otro con una nota. La asignación es aleatorea, puede que alguien revise su propio código. Se revisará dicha revisión con otra nota.
5. (+ 1pto) Bonus al programa que genera la solución con el menor tiempo posible para ejemplos difíciles.

5 Entrega

Jueves 02 de Enero a las 11:55PM entrega que incluye lo anterior. 1 punto por día de atraso considerando entrega hora/fecha posterior a la indicada. 06 de Enero 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