Dinitz vs Dinic(-Even)

Daniel Penazzi

28 de abril de 2021

Tabla de Contenidos

Complejidad “naive”de Dinitz

- ¿Cual es la complejidad de Dinitz?
- Primero, depende de cuantos networks auxiliares tengamos que construir.
- Veremos que no puede haber mas de n networks auxiliares.
- La construcción de cada uno es $O(m)$.
- ¿Cuál es la complejidad de encontrar un flujo bloqueante en un network auxiliar?, o equivalentemente, en el algoritmo original de Dinitz, ¿cuál es la complejidad de desconectar s de t?

- Como estamos corriendo Greedy, un analisis naive de la complejidad diria que es $O(m^2)$, pues esa es la complejidad de Greedy.
- Es decir, cada camino en el network auxiliar satura al menos un lado, y en el network auxiliar los caminos no se des-saturan, asi que hay a lo sumo m caminos.
- Como usamos DFS en cada camino, la complejidad de hallar un camino es $O(m)$.
- Lo cual nos da complejidad total de hallar un flujo bloqueante (o desconectar s de t) en un network auxiliar igual a $O(m^2)$.

- Pero entonces la complejidad total de Dinica seria $n.(O(m) + O(m^2)) = O(nm^2)$, la misma que la de Edmonds-Karp!
- ¿Para qué hacer todo este lio entonces?
- Es que ese es un analisis de una implementación “naive” de Dinitz.
- Es básicamente, como corrimos el ejemplo.
- Pero el ejemplo era chico, y ademas en un caso chico visualmente podemos hacer cosas que la computadora no puede.

Un poco mas de sutilezas

- Pej, en el primer network auxiliar del ejemplo, luego de encontrar caminos teniamos:
- Era fácil ver que no habia mas caminos aumentantes, viendo que si salimos por D llegamos a C y luego no podemos avanzar mas, y si salimos por G llegamos a H y no podemos avanzar mas.
- En el ejemplo, como lo estamos haciendo a mano, y es un ejemplo relativamente chico, es fácil darse cuenta visualmente cuál camino buscar y descartar “dead ends” .
- Pero ¿que pasaría si fuese incluso un poquito mas complicado?

Un poco mas de sutilezas

- Pej
- En ese ejemplo podemos ver, con cierta dificultad, que efectivamente no hay mas caminos.
- Pero ¿cómo haria un programa para hacerlo?
- Bueno, simplemente correria DFS y una vez que se le acaba la pila, terminaria. No hay problema.
- Ese último DFS recorreria casi todos los vértices y lados haciendo backtrackings, pero es una sola vez.

Un poco mas de sutilezas

- Pero supongamos que fuese:
- En este caso, hay mas caminos entre s y t, y encontrar cada uno puede ser engorroso.
- Veamos cómo seria una implementación de DFS usual.

Un poco mas de sutilezas

- Iria construyendo la pila s, D, C, F, P, U, backtrack a P, seguir por W, backtrack a P, F, C, seguir a M, M ignora P porque ya estuvo en la pila, pero sigue por Q, W, backtrack Q, M, C, D, s sigue por G, H, ignora M porque estuvo en la pila, sigue por K, ignora Q porque estuvo en la pila, sigue por R, ignora W porque estuvo en la pila, sigue por X, backtrack todo el camino hasta s, sale por Y, finalmente encuentra s, Y, E, L, N, V, t.
- Luego, asumiendo que se satura sólo el lado $\overrightarrow{YE}$, tiene que repetir TODO eso, hasta encontrar el camino s, Y, I, L, N, V, t.
- Luego, asumiendo que se satura sólo el lado $\overrightarrow{YI}$, tiene que repetir todo eso, hasta encontrar el camino s, Y, J, L, N, V, t, etc

Un poco mas de sutilezas

- En este ejemplo o uno similar, cada encuentro de camino es efectivamente $O(m)$ y la complejidad total seria efectivamente $O(m^2)$ hasta poder terminar con el network auxiliar.
- El problema es que no estamos teniendo “memoria” entre una corrida de DFS y la siguiente.
- Cuando lo hacemos a mano nos podemos dar cuenta y no repetir toda esa busqueda inicial.
- Y lo mismo debemos hacer en la computadora.

Un poco mas de sutilezas

- Esa en realidad es la idea de la implementación de Ever:
  - cuando corremos DFS, si llegamos a un vértice x que no tiene vecinos, debemos hacer un backtrack, borrando a x de la pila y usando el vértice anterior a x en el camino para seguir buscando.
  - Esa información de que es inútil seguir buscando por x no deberiamos perderla y hay que "guardarla" para futuras corridas de DFS.
  - La forma que tiene Ever de "guardar"esa información es simplemente borrar x, o bien, si hacemos backtrack desde x a z, borrar el lado $\overrightarrow{zx}$.
- La idea original de Dinitz es distinta.

Diferencia entre la version rusa y la occidental de Dinitz

- La diferencia entre Dinitz y Dinic-Even es en cómo y cuando se actualiza el network a medida que encontramos nuevos caminos.
- Ambos borran los lados saturados en un camino, pero luego, como dije recien, Ever borra varios lados (o vértices) extras mientras corre DFS, cada vez que tiene que hacer un backtrack.
- En la versión original de Dinitz, el network auxiliar se construye de forma tal que DFS nunca tenga que hacer backtrack.
- Y cada vez que se encuentra un camino entre s y t y se cambia el flujo, tambien se cambia el network auxiliar para seguir teniendo esta propiedad.

Dinitz vs Dinic-Even

- Es decir, en el original se usa un poco mas de tiempo actualizando el network auxiliar luego de cada camino, para asegurarse que cada busqueda de camino por DFS demore lo mismo, pues ninguna hace backtrack.
- mientras que en la versión de Even, no se pierde tanto tiempo actualizando el network auxiliar entre caminos, pero las busquedas DFS no demoran todas igual pues algunas modificarán el network mas que otras.
- Esto trae algunas diferencias a la hora de, pej, probar la complejidad, aunque las complejidades de ambas versiones son iguales.

Dinitz vs Dinic-Even

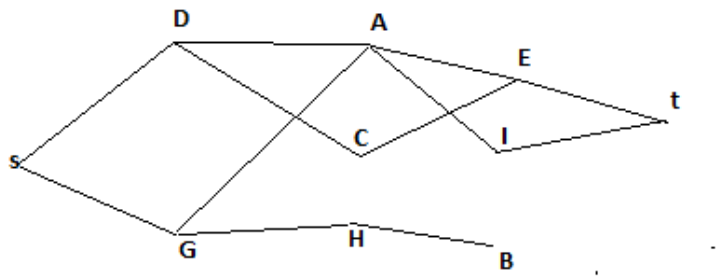
- En la versión de Dinitz, pej, luego del primer camino del ejemplo, cuando llegamos a
- Dinitz notaria que C no tiene caminos de salida, lo borraria, y borraria los lados que entran a C ($\overrightarrow{AC}$, $\overrightarrow{DC}$)
- Luego como D no tendria lados de salida, lo borraria, y al lado $\overrightarrow{sD}$.
- Ever no haria eso en este paso, ni durante el DFS siguiente, pues encontraria antes el camino $sBHt$ como hicimos nosotros.

Dinitz vs Dinic-Even

- Recien luego de ese camino, durante el ultimo DFS, Ever encontraria s, D, C y al hacer backtrack borraria los lados $\overrightarrow{sD}, \overrightarrow{DC}$ y luego de ir por s, G, H, borrraria los lados $\overrightarrow{sG}, \overrightarrow{GH}$.
- Dinitz no tendria que hacer eso, pues luego de haber borrado C, D luego del primer camino, y haber encontrado tambien $sBHt$, borraria H y G y cuando quisiera hacer su último DFS, no podria pues s no tendría lados de salida.
- Entonces podemos decir que Ever es mas "lazy" y sólo borra lados si los encuentra y se da cuenta que no los necesita, mientras que la versión original de Dinitz es mas proactiva y borra todos los lados que sabe que son inútiles aún si luego nunca los encontraria. (como el $\overrightarrow{AC}$ en el ejemplo).

Dinitz vs Dinic-Even

- O, pej, luego de construir el segundo network auxiliar:



- Dinitz borraria los vértices H, B y los lados $\overrightarrow{GH}$, $\overrightarrow{HB}$ antes de empezar el primer camino.
- Queda analizar en detalle las complejidades de ambas implementaciones.