Algorithm Design and Analysis

Network Flow Part II: Advanced Flow Algorithms

Roadmap for today

- Review network flow and the ***Ford-Fulkerson*** algorithm
- Make the Ford-Fulkerson algorithm faster!
  - The ***Edmonds-Karp*** algorithm
  - ***Dinic's*** algorithm: The layered graph, and blocking flows

Network Flow recap

- A ***flow network*** is a directed graph with:
  - ***capacities*** $c(u, v)$
  - A ***source vertex*** s and ***sink vertex*** t
- A ***flow*** is an assignment of values to edges:
  - ***Capacity constraint:*** $0 \le f(u, v) \le c(u, v)$
  - ***Conservation constraint:*** "flow in = flow out" for all vertices except s, t

$$\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$$

- The value of a flow is the net flow out of the source (can prove via conservation that is = net flow into sink)

Network Flow recap

- The ***maximum flow*** problem is to find a flow of maximum value
- We learned the ***Ford-Fulkerson*** algorithm:
  - Define the ***residual network:***

$$c_f(u, v) =$$

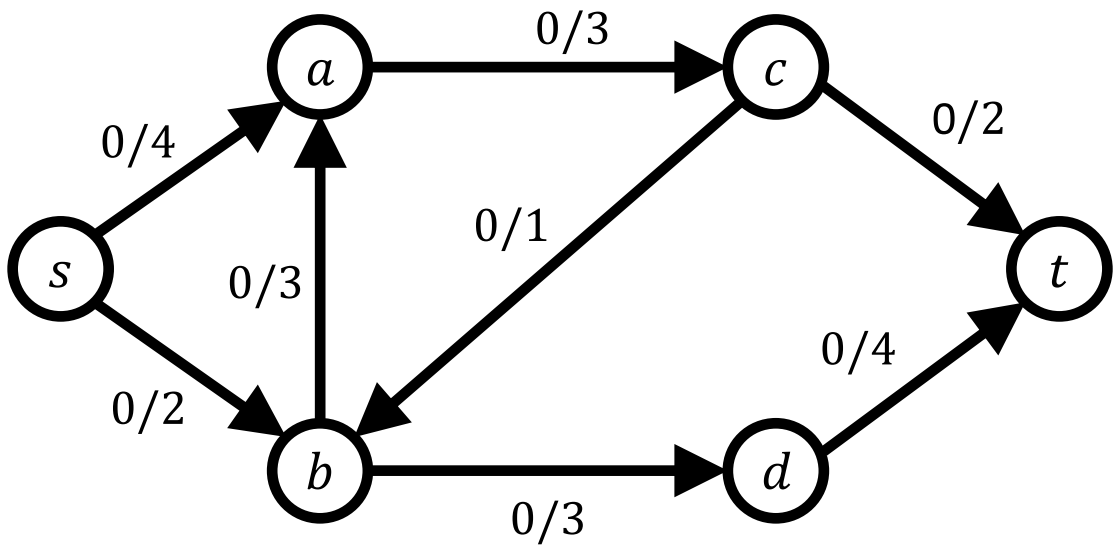
$$c_f(v, u) =$$

  - Then the algorithm is:

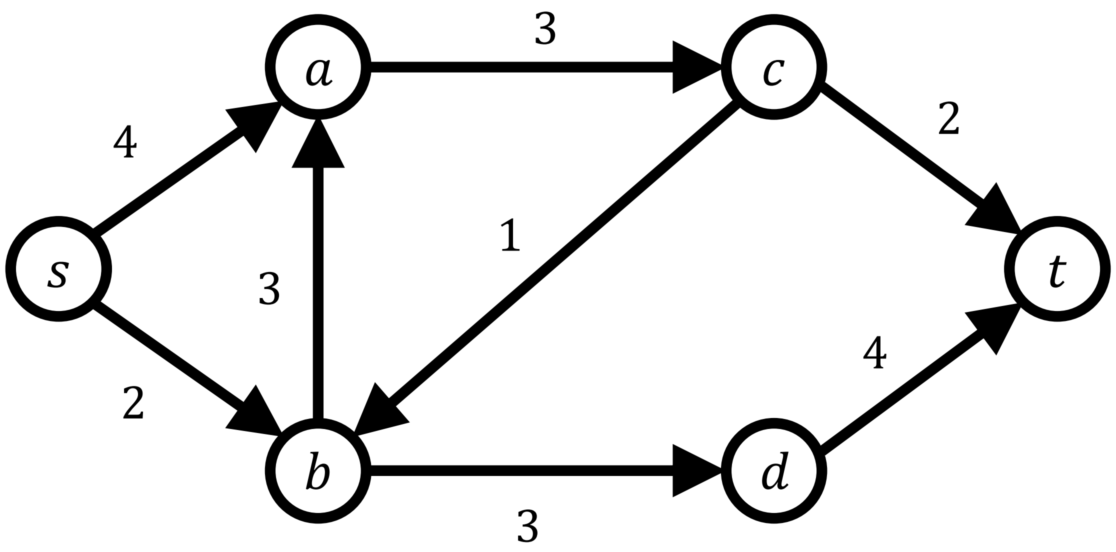
    while there exists a path in the residual network:
      add flow to that path.

Residual capacity:

$$\begin{cases} c_f(u,v) = c(u,v) - f(u,v), \\ c_f(v,u) = c(v,u) + f(u,v) \end{cases}$$



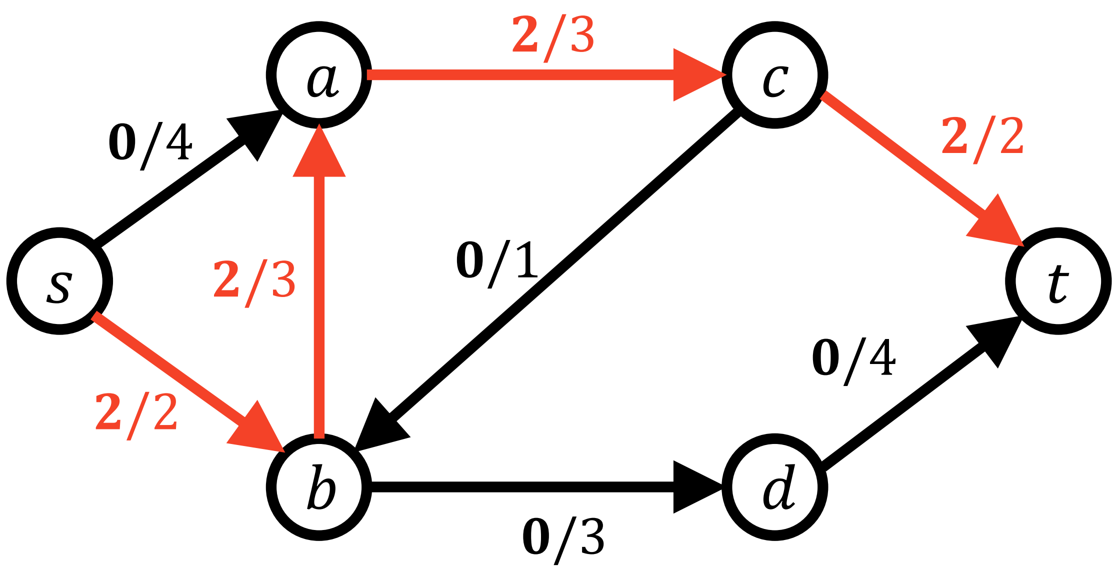
Flow network G



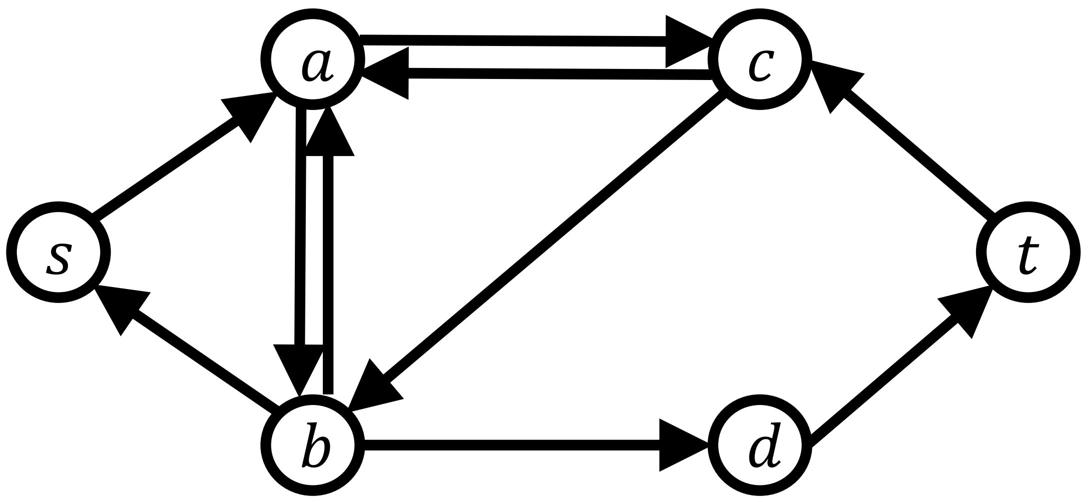
Residual network G_f

Residual capacity:

$$\begin{cases} c_f(u,v) = c(u,v) - f(u,v), \\ c_f(v,u) = c(v,u) + f(u,v) \end{cases}$$



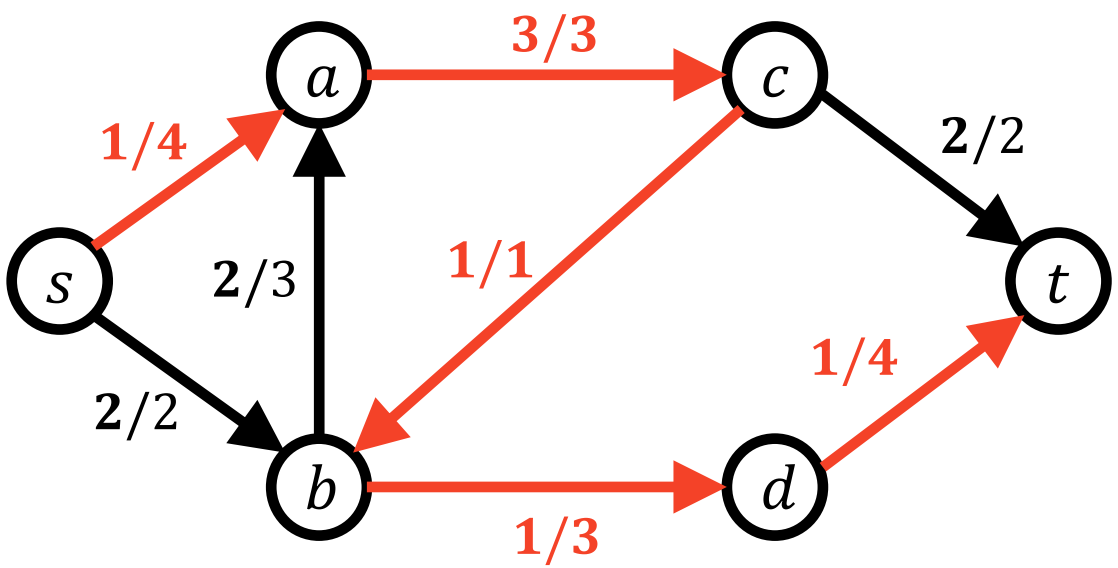
Flow network G



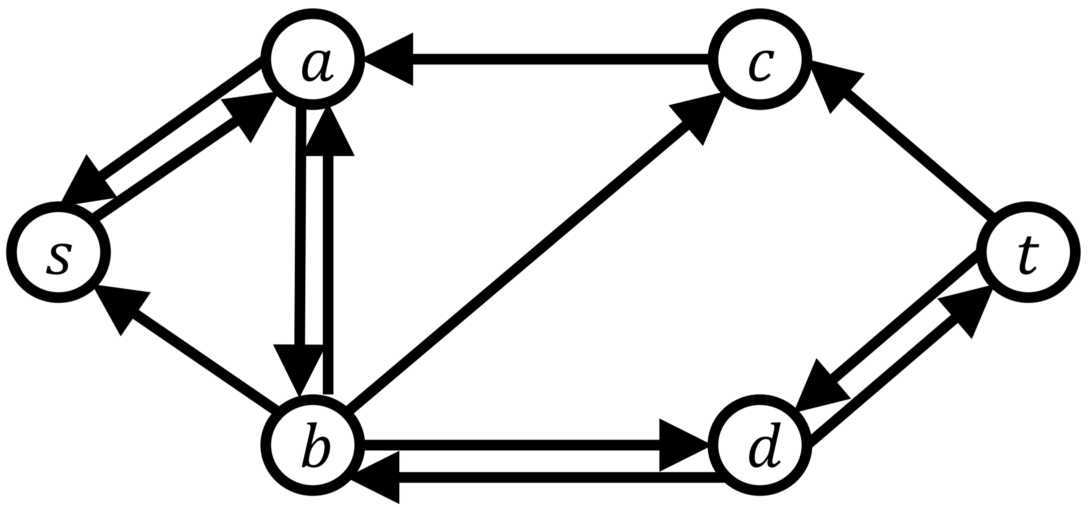
Residual network G_f

Residual capacity:

$$\begin{cases} c_f(u,v) = c(u,v) - f(u,v), \\ c_f(v,u) = c(v,u) + f(u,v) \end{cases}$$



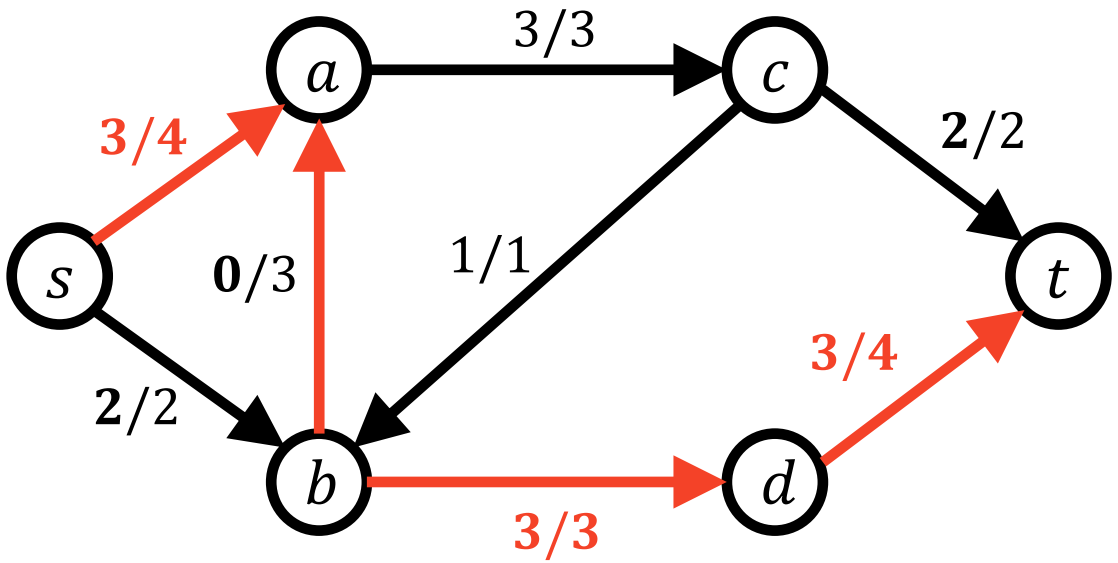
Flow network G



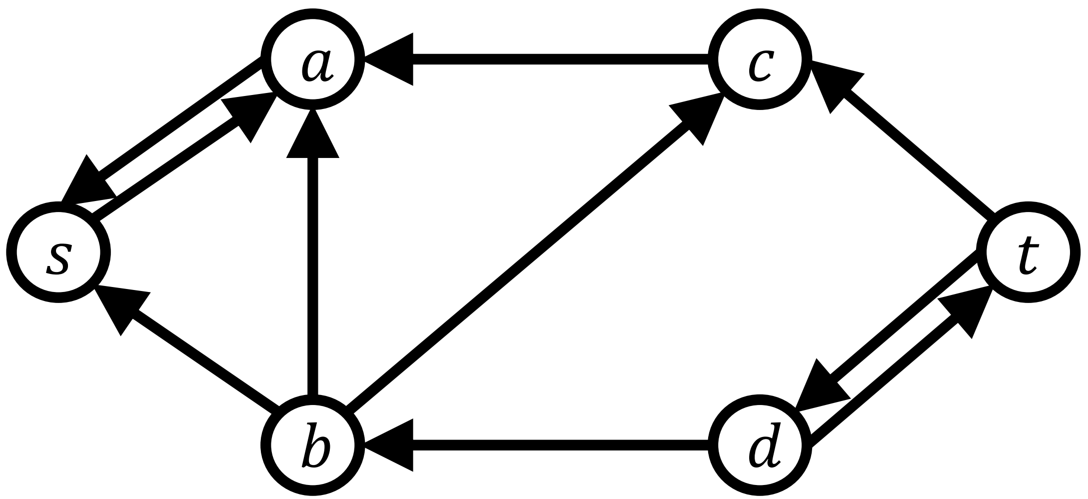
Residual network G_f

Residual capacity:

$$\begin{cases} c_f(u,v) = c(u,v) - f(u,v), \\ c_f(v,u) = c(v,u) + f(u,v) \end{cases}$$



Flow network G



Residual network G_f

Worst-case runtime

Theorem: Ford-Fulkerson runs in $\boldsymbol{O(mF)}$ **time** (with integer capacities)

(Or $O((n+m)F)$ but we might as well assume that G is connected)

How to make it faster?

- Ford-Fulkerson finds *any* augmenting path until there are none left
- ***Idea***: Can we find "good" augmenting paths that guarantee a better running time? Yes!

- **Idea #1**:

- **Idea #2**:

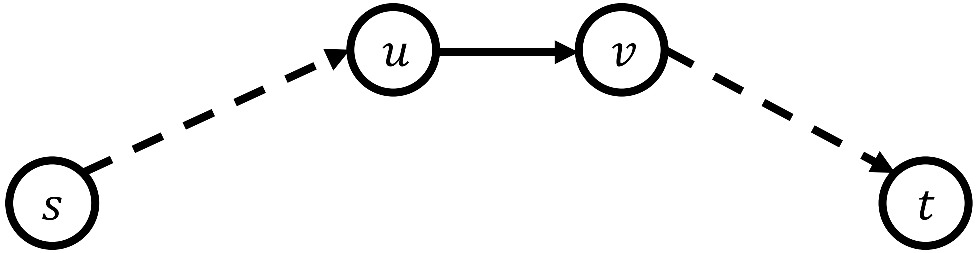
Edmonds-Karp (Shortest Augmenting Paths)

- When we described Ford-Fulkerson, we found *any* augmenting path, usually via DFS as the simplest possible implementation
- If we use a ***BFS*** instead, we get a shortest augmenting path (fewest possible edges)

Theorem: Edmonds-Karp runs in $O(nm^2)$ time (polynomial!)

Analysis

Lemma: Let d be the distance from s to t. In Edmonds-Karp, d never decreases.



Analysis

Lemma: After m iterations, d **must** increase.

Analysis

Conclusion:

- Each iteration takes:
- Iterations per value of d:
- d can increase:

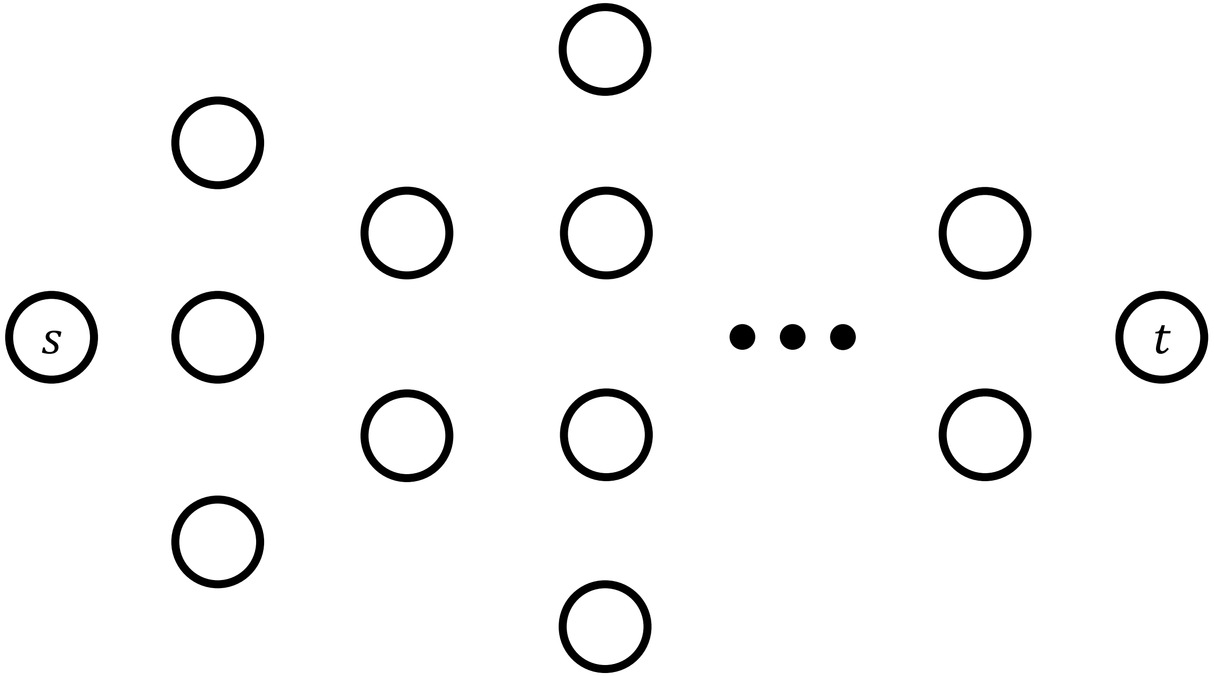
Redundancy in Edmonds-Karp

- Edmonds-Karp does up to m augmenting paths for each value of d
- Hmm... is something redundant here?

- Does BFS only find you one shortest path?
- No! It finds every shortest path (from the source s)!
- ***Dinic's algorithm***: Find many shortest augmenting paths per BFS call to save work!

The "layered graph"

- a.k.a. **level graph**, a.k.a. **admissible graph**.
- Given a network G_f, what do we get when we run BFS?
- We want to find augmenting paths **in the layered graph**.
- Algorithm? Find augmenting paths using DFS until none remain?

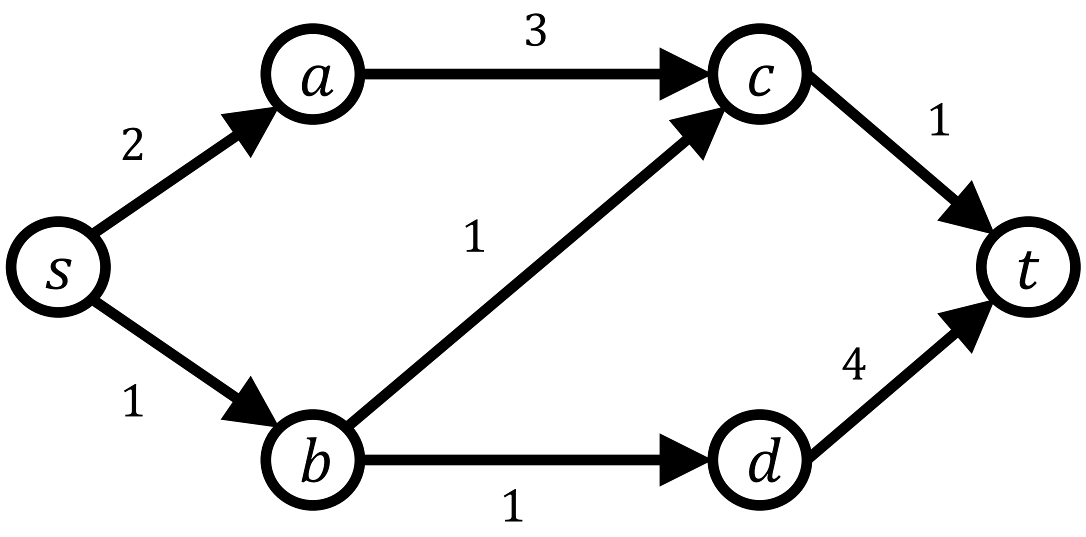


***Time per iteration*:** ***# Iterations*:** ***# layers*:**

Blocking flows

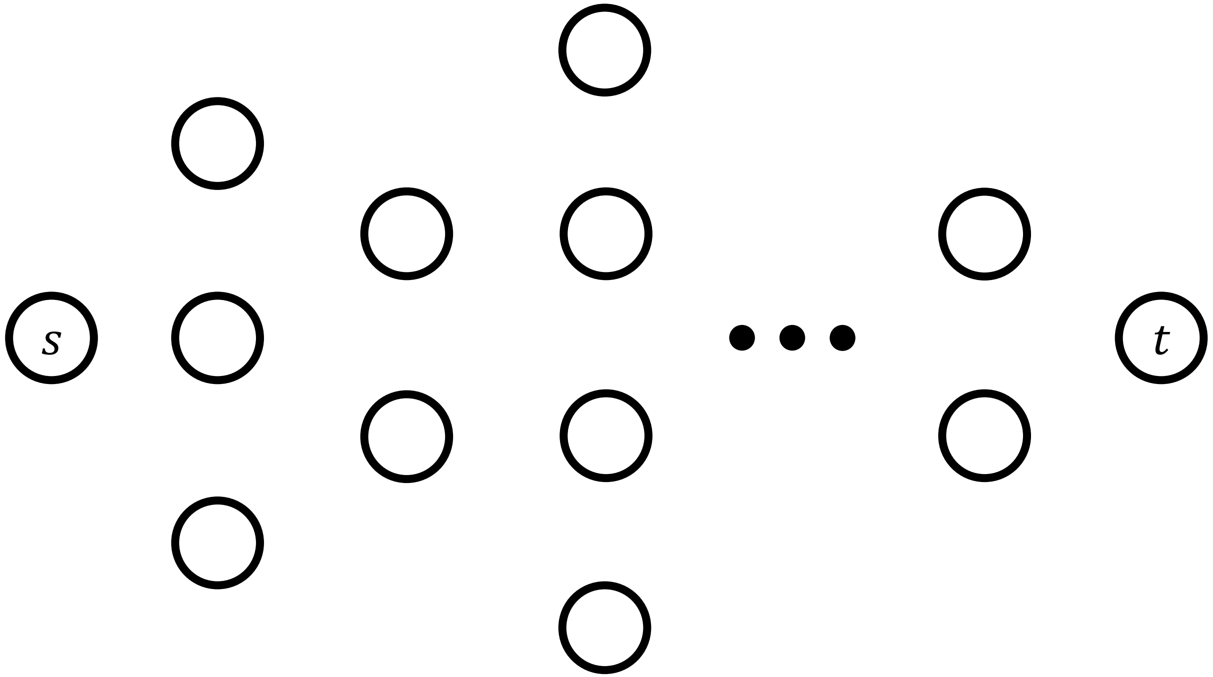
Definition: A ***blocking flow*** in a flow network G' is a flow that saturates at least one edge in every $s - t$ path in G'

Note: Not the same thing as a maximum flow! (Every maximum flow is a blocking flow, but the reverse is not true)



Algorithmic goal

- We want to find a ***blocking flow*** in the ***layered graph***
  - Faster than just finding augmenting paths independently one by one



Blocking flow algorithm

- Perform DFS to find capacitated $s - t$ paths
- When we traverse an edge that does not lead to t, mark it as "dead", i.e., logically delete it from the network
- In future DFS's, dead edges are not considered!

Note: Since we are looking for a blocking flow, not a maximum flow, we don't need to enable back edges after finding each $s - t$ path!

- Why? A back edge always make the distance longer, so it can not possibly be in the layered graph for the current value of d. (Same proof as when we analyzed Edmonds-Karp)

Dinic's algorithm

while the flow is not maximum:
    compute the layered graph of G_f for the current distance d
    find a blocking flow in the layered graph
    augment f with the contents of the blocking flow

Correctness: Finding a blocking flow saturates every shortest path, so the distance d must increase. After increasing the distance n times there are no more augmenting paths, so the flow is maximum.

Analysis

Dinic's on unit-capacity graphs

- Many problems modelled using network flow use only use capacity 1
  - Example: bipartite matching from last lecture
- Unit-capacity networks have low max flow ($F \leq m$) so algorithms ought to be faster

Theorem: Dinic's runs in $O(\sqrt{m}\, m)$ time on unit-capacity networks.

Proof by two lemmas:

- We can find a blocking flow in a unit-capacity network in $O(m)$
- $O(\sqrt{m})$ blocking flows is sufficient to find a maximum flow in a unit-capacity network.

Dinic’s on unit-capacity graphs

Lemma: We can find a blocking flow in a unit-capacity network in $O(m)$

Dinic’s on unit-capacity graphs

Lemma: $2\sqrt{m}$ blocking flows is sufficient to find a maximum flow in a unit-capacity network.

Take-home messages

- Maximum flow can be solved in polynomial time!
- Edmonds-Karp (shortest augmenting paths) runs in $O(nm^2)$
- Dinic's runs in $O(n^2m)$, better for sparse graphs!
  - Try to review and understand ***blocking flows***
- Dinic's runs even faster, $O(\sqrt{m}\, m)$ on unit-capacity graphs