CSE 417
Network Flows (pt 1)
Max Flow

UNIVERSITY *of* WASHINGTON

Reminders

> **HW6 is due in one week**
  - solve a Knapsack-type problem
  - many additional wrinkles
  - will need some optimization

Review of last lecture

- Optimal substructure: (small) set of solutions, constructed from opt solutions to sub-problems, that always contains the optimal one

- Can <u>construct</u> the optimal solution for each sub-problem
  - we usually just recorded the value of the solution to save time & space
  - with careful data structures, you can sometimes record solutions just as quickly
    - BUT that does not allow you to save space by only keeping last col / row
  - we used the same trick for midpoints as for values
    - compute value of the solutions from values on sub-problems
    - compute midpoint of the solutions from midpoints on sub-problems

Foreword

- Back to modeling…
  - shortest path
  - binary / ternary search
  - network flows
- Algorithms for network flows are unlike D&C and DP
  - both of those relied on solutions to sub-problems
  - none of the algorithms we will discuss work that way
  - they are more like coordinate descent than D&C or DP
    - maintain a possible solution and improve it to optimality

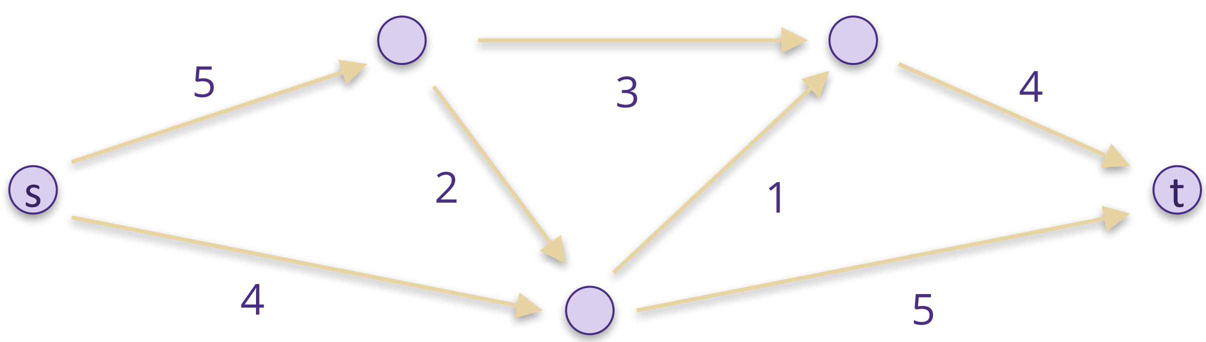
Outline for Today

> Maximum Flow
> Applications
> Ford-Fulkerson
> Other Algorithms

Flows

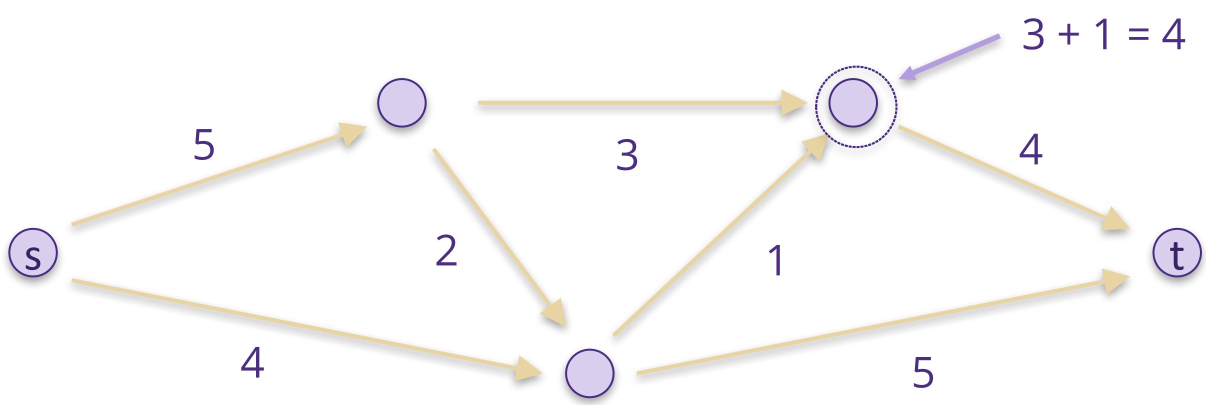
> Let s and t be two nodes of a graph G.
>   – s is called the "source" and t the "sink"

> A *flow* assigns a non-negative number, f_e, to each edge e
>   – represents the amount of water / cars / etc. moving along the edge per unit time



Flows

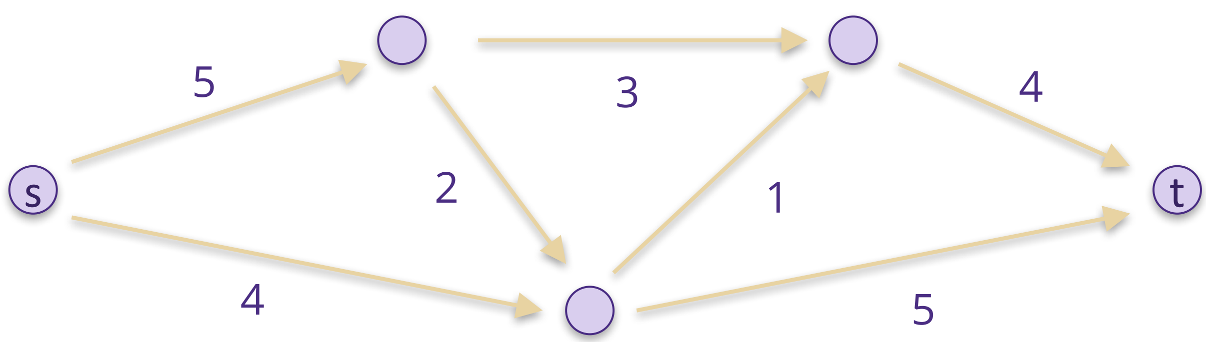
- A flow is ***balanced*** if, at every node other than s or t, the amount of incoming flow equals the amount of outgoing flow
  - incoming flow at n = sum of flows on edges *into* n
  - these are called “flow balance constraints”



Flows

> Call incoming flow – outgoing flow at node n the *excess* flow
  – flow balance constraint says excess flow = 0 everywhere but s & t

> **Fact**: if f is balanced, then excess flow at t = -excess flow at s
  – intuition: -excess flow leaving s cannot pool at any other node, so it ends up at t



Flows

- Call incoming flow – outgoing flow at node n the *excess* flow
  - flow balance constraint says excess flow = 0 everywhere but s & t

- **Fact**: if f is balanced, then excess flow at t = -excess flow at s
  - intuition: -excess flow leaving s cannot pool at any other node, so it ends up at t
  - proof
    - sum of excess flows of all nodes is 0
      - every edge flow appears twice — once incoming (+) and once outgoing (-) — so sum is 0
    - values are zero everywhere but s and t, so excess(s) + excess(t) = 0

- The excess flow at t is called the *value* of the flow

Maximum Flow

> **Problem**: Given a graph G, two nodes s and t, and edge capacities, c_e for each edge e, find the maximum value of any *balanced* flow where the flow on each edge e no more than c_e.

> Call a flow ***feasible*** if it satisfies both

– flow balance constraint: excess(n) = 0 for each $n \neq s, t$

– capacity constraint: $f_e \leq c_e$ for each edge e

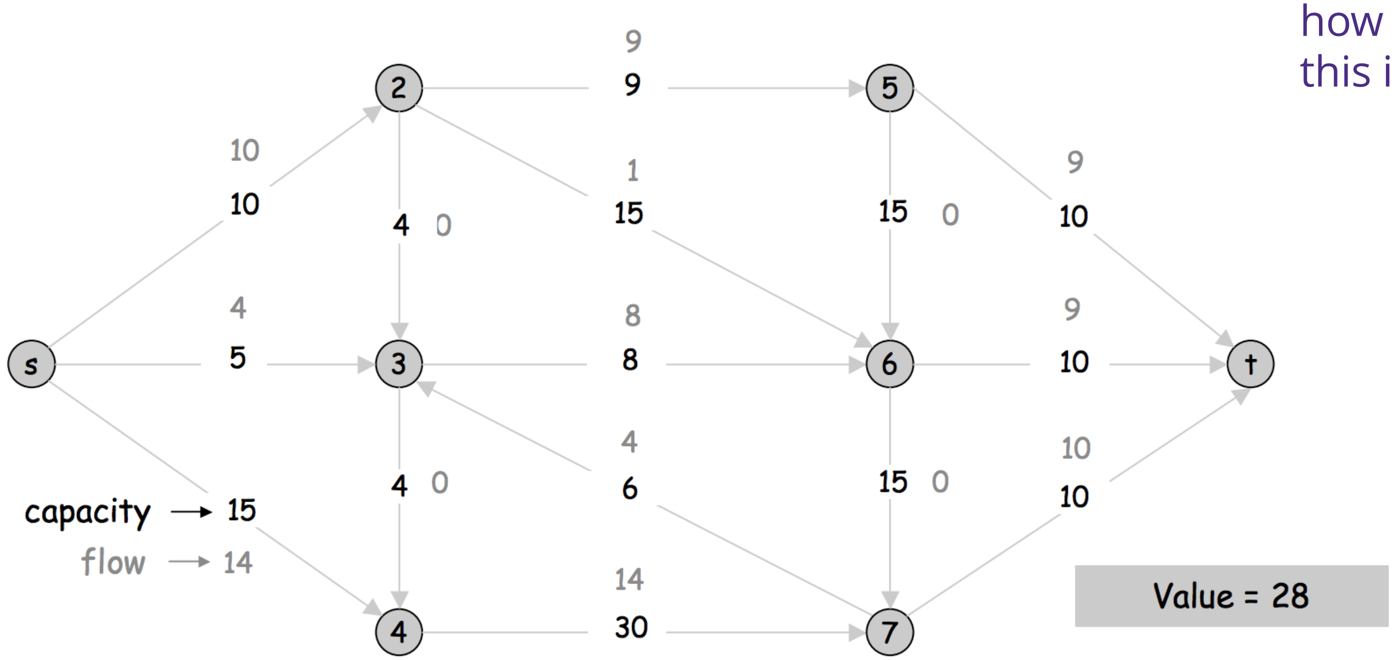
problem is to find feasible flow of maximum value

> Given flow, easy to check that constraints are satisfied

– not easy to see if value is optimal… more on that later

Maximum Flow Example

Will see next time how to check that this is optimal…



Outline for Today

> Maximum Flow
> Applications
> Ford-Fulkerson
> Other Algorithms

Applications

- Project selection
- Airline scheduling
- Baseball elimination
- Image segmentation
- Network connectivity
- Network reliability
- Intrusion detection
- Distributed computing
- Egalitarian stable matching
- Security of statistical data
- Data mining
- Multi-camera scene reconstruction
- ...

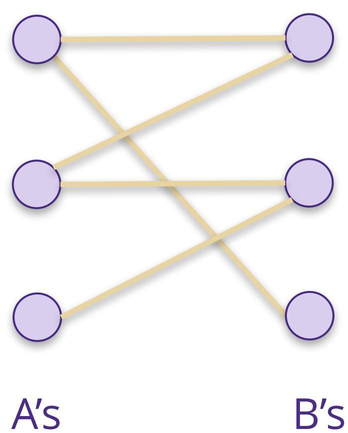
Bipartite Matching

> **Problem**: Given two lists of objects, A and B, and a set of allowed matches {(a, b)}, find the largest possible subset of the matches with the property that element in A or B is **not** matched 2+ times.

> **Q**: What does this have to do with graphs?

Bipartite Matching

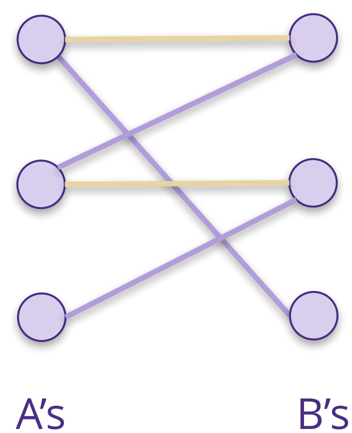
> **Problem**: Given two lists of objects, A and B, and a set of allowed matches {(a, b)}, find the largest possible subset of the matches with the property that element in A or B is **not** matched 2+ times.



find a subset of edges (a,b) with no two edges sharing the same a or b

Bipartite Matching

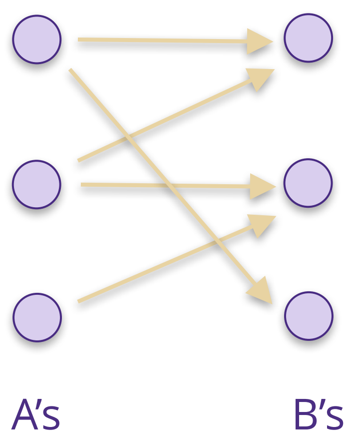
> **Problem**: Given two lists of objects, A and B, and a set of allowed matches {(a, b)}, find the largest possible subset of the matches with the property that element in A or B is **not** matched 2+ times.



optimal matching
has 3 matched pairs

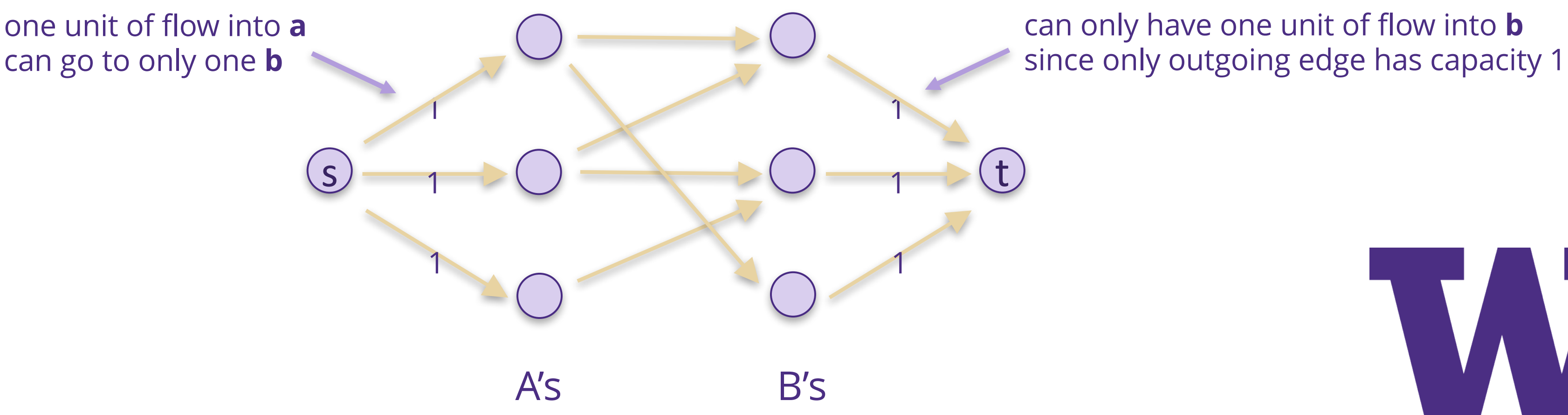
Bipartite Matching

- > Solve bipartite matching by modeling as maximum flow
  - – flow along an edge indicates a match: 1 for matched, 0 for not
    - > also flow along each edge must be 0 or 1 (more on that later…)



Bipartite Matching

- Solve bipartite matching by modeling as maximum flow
  - flow along an edge indicates a match
  - capacity constraints ensure each object is chosen at most once



Bipartite Matching

- Maximum bipartite matching is no harder than maximum flow
  - (assuming we can require a 0/1 flow)
- **Moral**: information about “allowed pairs” is a (bipartite) graph
  - any such information in a problem is a hint to model with network flows
  - in fact, network flow problems on bipartite graphs are not actually easier than general graphs for this problem
    - (may show the reduction later if we have time...)

More on Matching

- Maximum matching be solved on non-bipartite (general) graphs, BUT that problem is harder
  - separate theory of matchings and associated algorithms
  - one exception: easy to find "stable marriage" matchings

- Foreword: can also consider weighted graphs…
  - find the matching that maximizes the total weight of the matching
  - also called the "assignment problem"
  - this is a <u>harder</u> problem than the basic version
    - related: min cost flow is harder than maximum flow

Outline for Today

- Maximum Flow
- Applications
- Ford-Fulkerson
- Other Algorithms

Ford–Fulkerson

- **Idea**: maintain a feasible flow and continue to improve it
  - once we cannot find a way to improve the flow, then we can (hopefully) prove it is actually optimal

- Can start with flow $f_e = 0$ for each edge e
  - satisfies capacity constraints since $c_e \geq 0$
  - satisfies balance constraints since incoming flow = outgoing flow = 0

Ford-Fulkerson

- To improve it, find an s ~> t path along which we can send more flow

- In more detail:
  - create a graph, G(f), with only edges along which we can send more flow
    - *e.g.*, edge e with $f_e < c_e$ can get $c_e - f_e$ more flow and still satisfy capacity constraints
  - a path from s to t in G(f) gives us a way increase the flow
    - increase every edge on path by the minimum allowed increase of all the edges
    - intermediate nodes get more in but also more out flow, so balance is preserved

- If there is no such path, then we cannot improve it
  - we will prove this formally later…

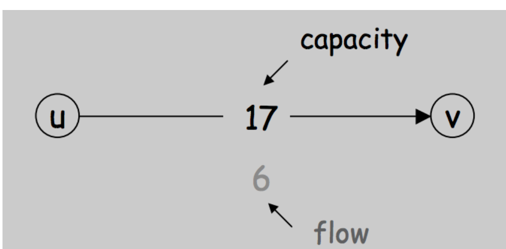
Residual Graph

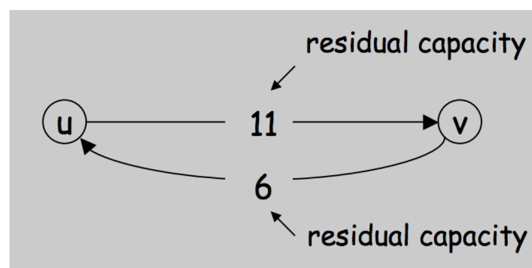
G(f) is called the "residual graph"

- Given flow f, **define** G(f) to have the same nodes as G
- If $e = (u,v)$ is an edge of G with $f_e < c_e$,
  then G(f) has edge e with capacity $c_e - f_e$
  - represents the ability to push more flow along e
- If $e = (u,v)$ is an edge of G with $0 < f_e$,
  then G(f) has an edge $e' = (v,u)$ with capacity f_e
  - represents the ability to push *less* flow along e

Residual Graph

- Given flow f, **define** G(f) to have the same nodes as G
- If $e = (u,v)$ is an edge of G with $f_e < c_e$,
  then G(f) has edge e with capacity $c_e - f_e$
- If $e = (u,v)$ is an edge of G with $0 < f_e$,
  then G(f) has an edge $e' = (v,u)$ with capacity f_e

G


G(f)


Residual Graph

- Given flow f, define G(f) to have the same nodes as G
- If $e = (u,v)$ is an edge of G with $f_e < c_e$,
  then G(f) has edge e with capacity $c_e - f_e$
- If $e = (u,v)$ is an edge of G with $0 < f_e$,
  then G(f) has an edge $e' = (v,u)$ with capacity f_e
- Can get capacity **both** from (u,v) and also from (v,u)
  - if so, add their capacities in the same direction together
  - (only want one edge from u to v)

Ford-Fulkerson

```
let flow f = 0 for every edge
repeat
    find an s ~> t path in G(f)
    if none exists:
        break
    else:
        let δ = min capacity of any edge on the path
        increase flow f by δ along each edge
```

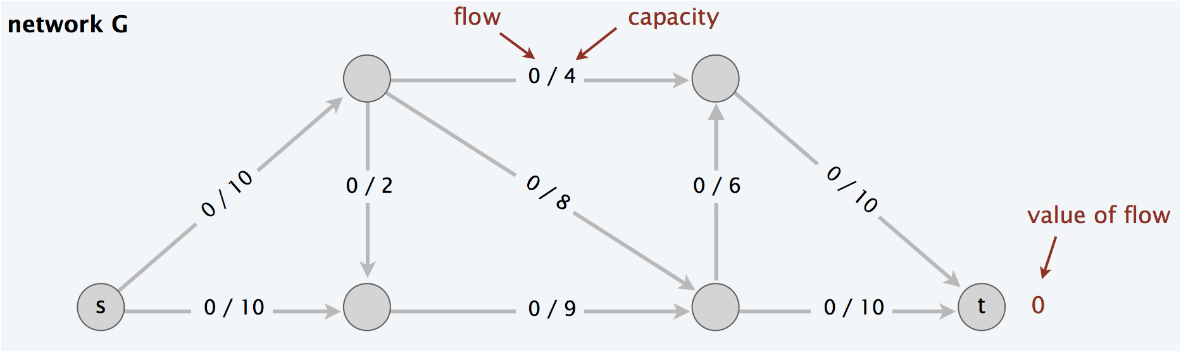
start with a feasible flow

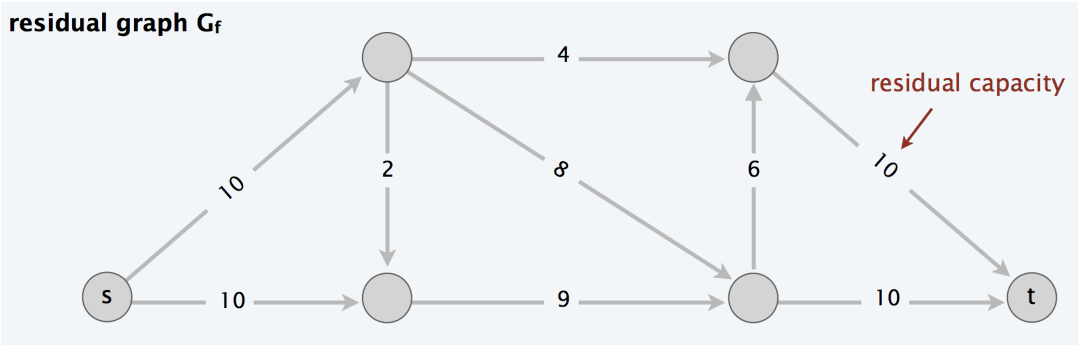
e.g., use DFS or BFS

still satisfies both balance and capacity constraints by previous argument

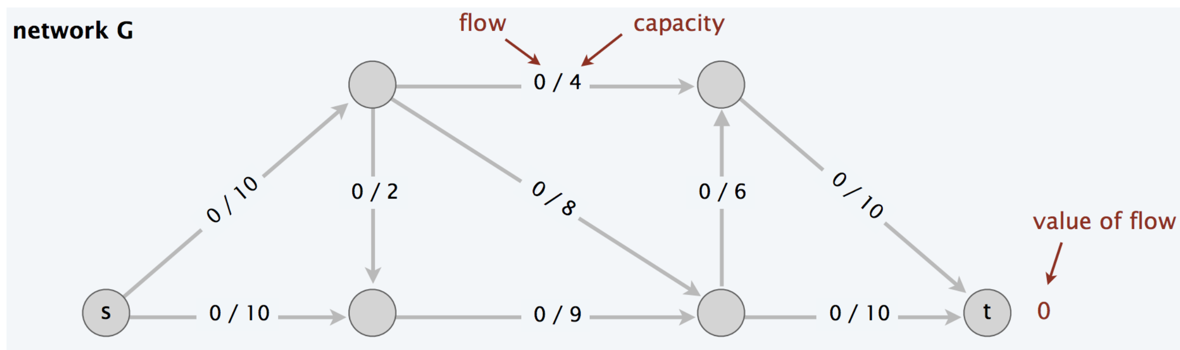
actually may mean decreasing flow if from a reverse edge

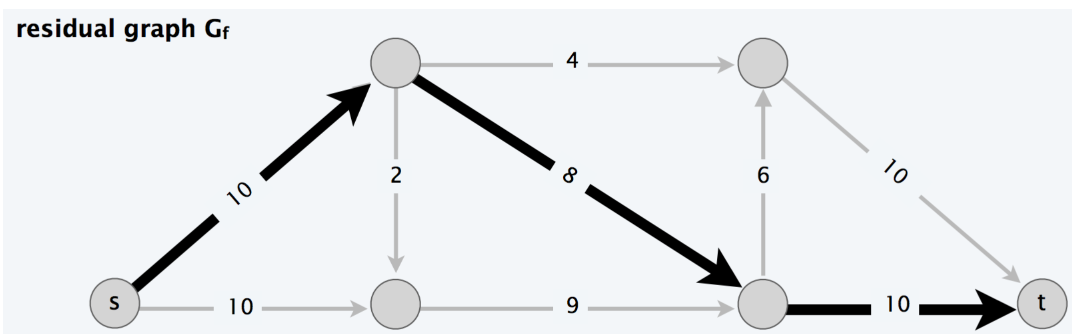
Ford–Fulkerson



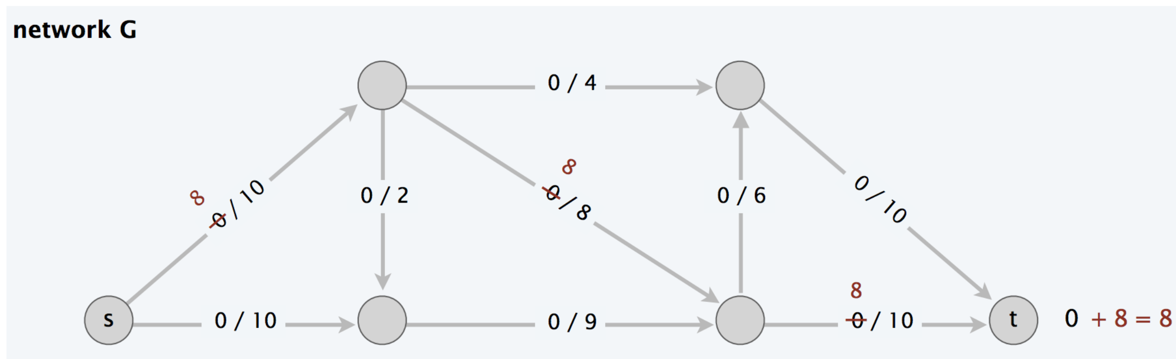


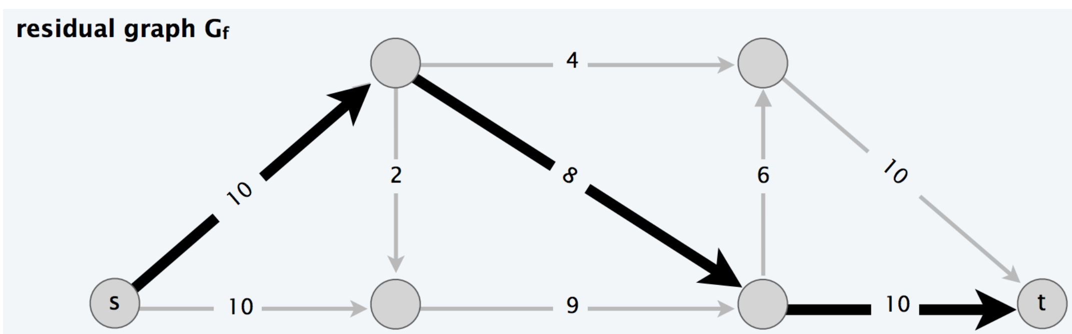
Ford–Fulkerson



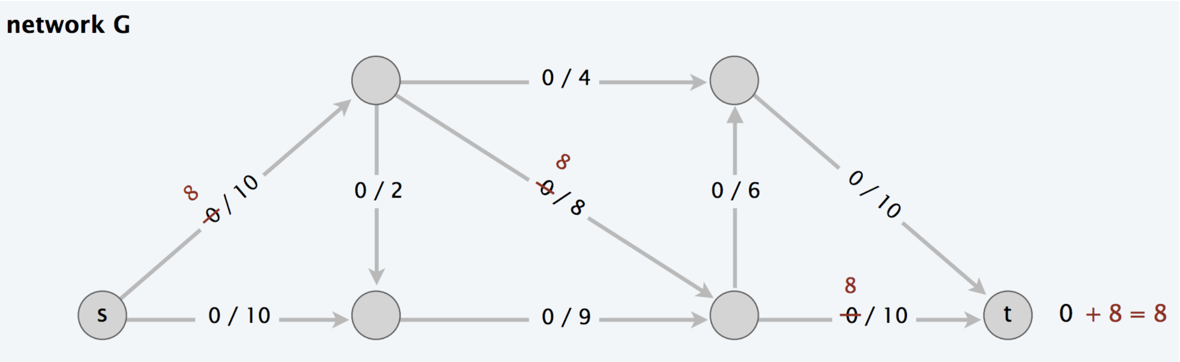


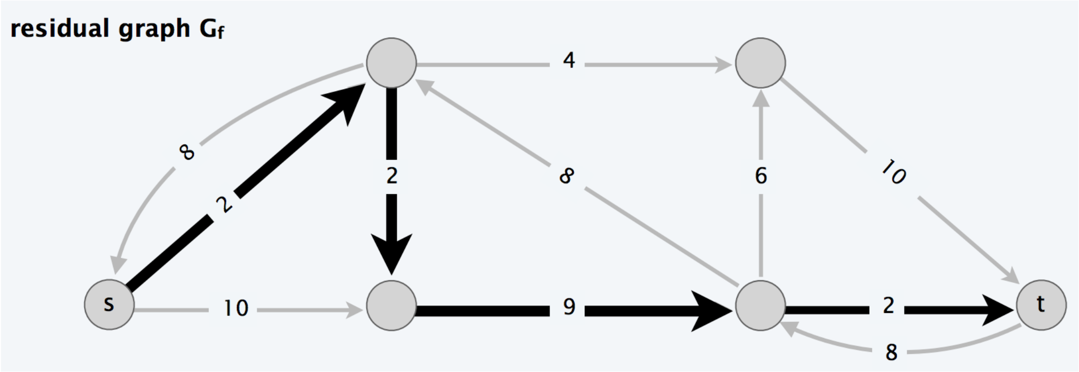
Ford–Fulkerson



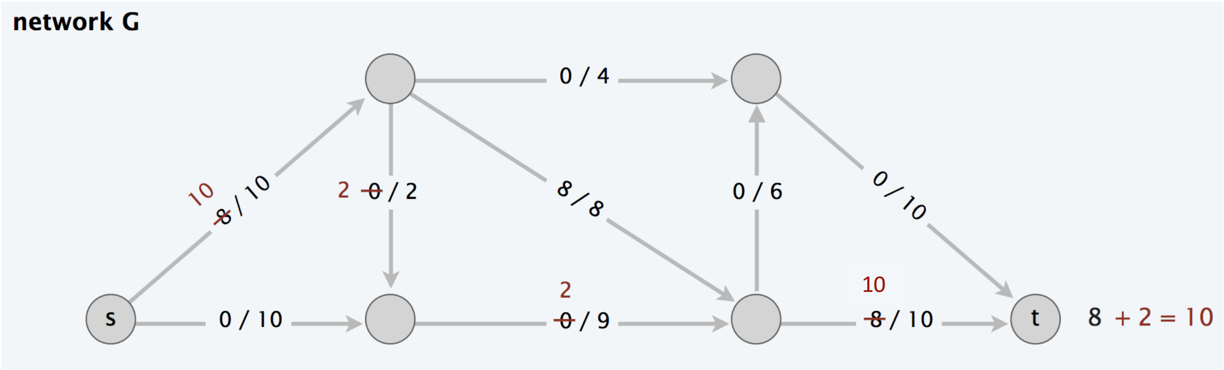


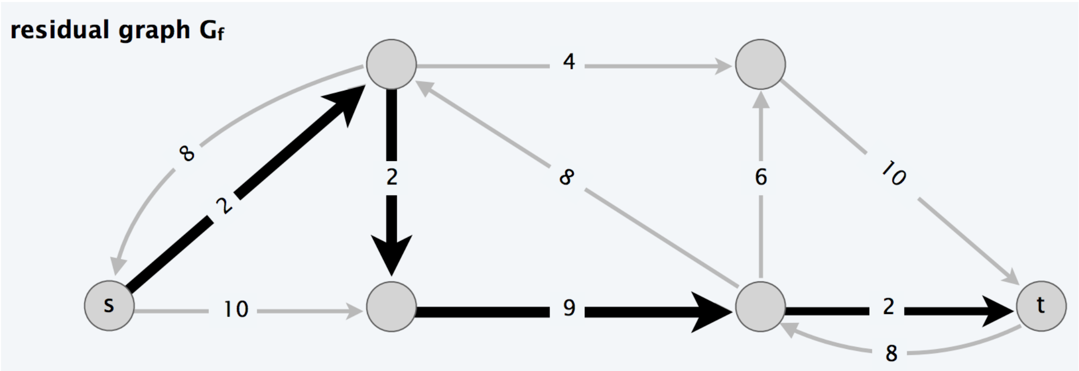
Ford–Fulkerson



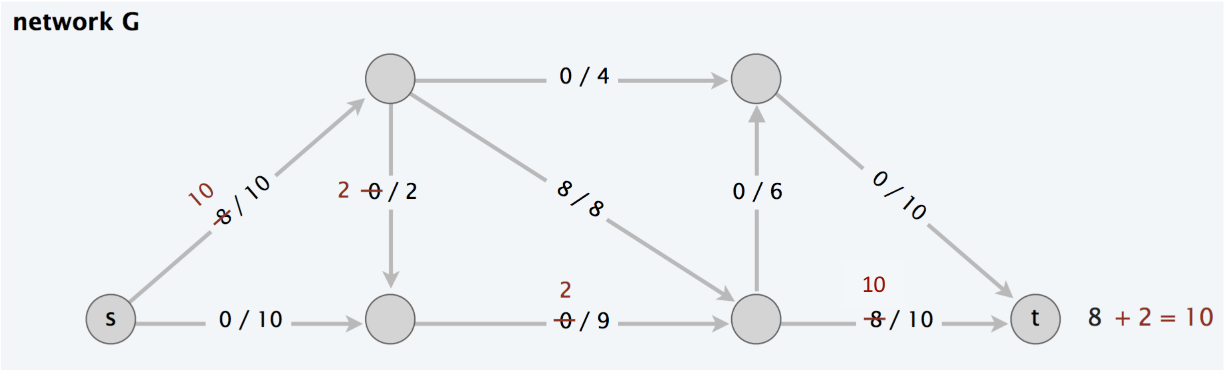


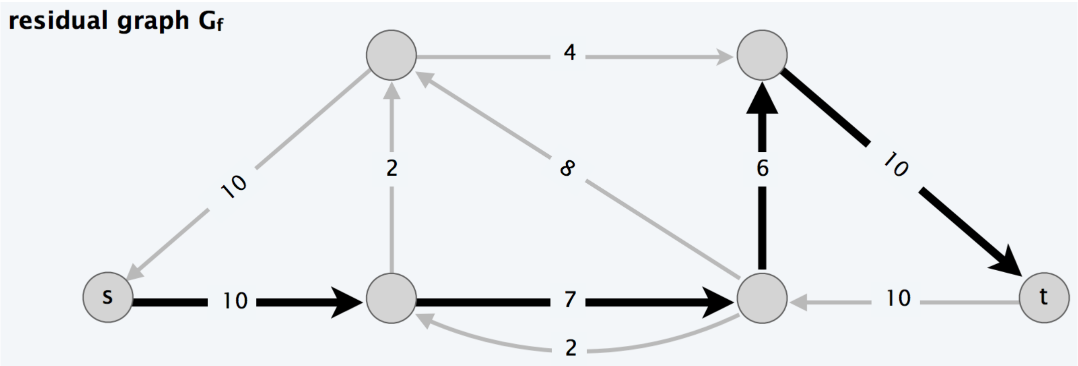
Ford–Fulkerson



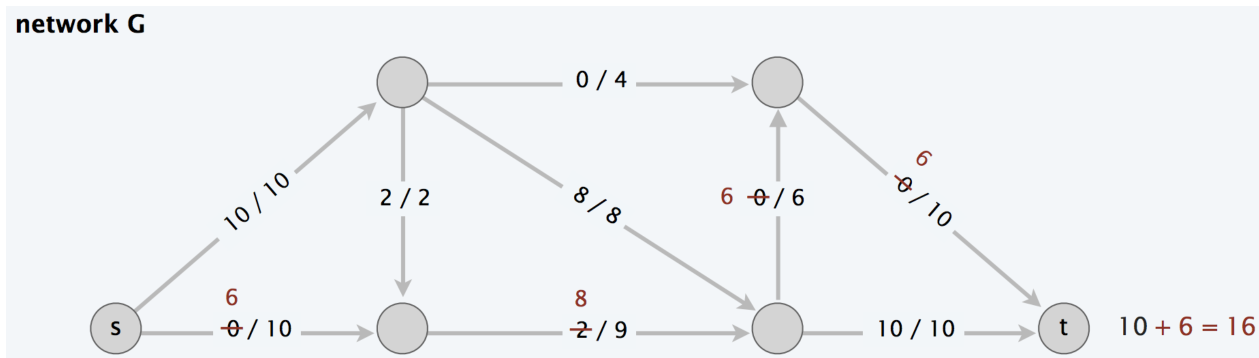


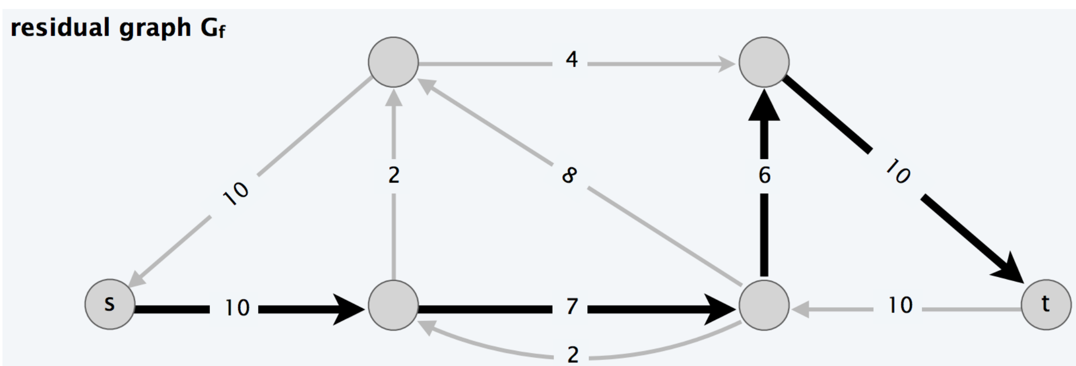
Ford–Fulkerson



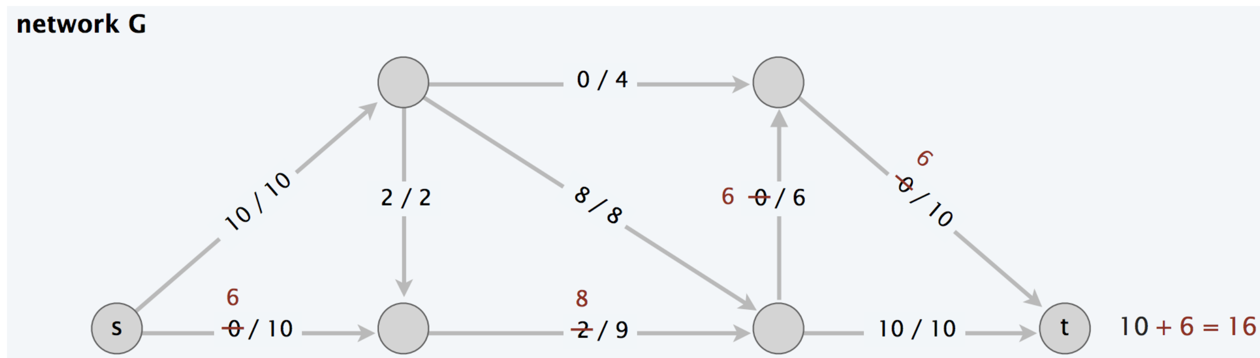


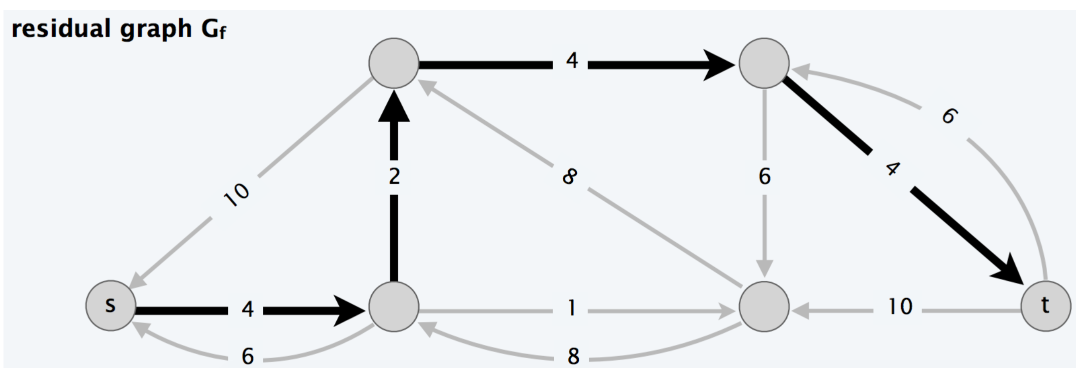
Ford–Fulkerson



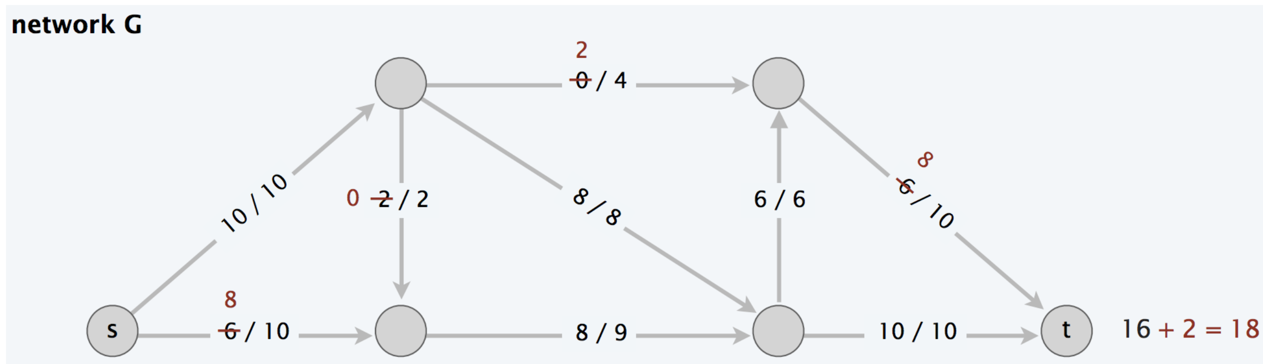


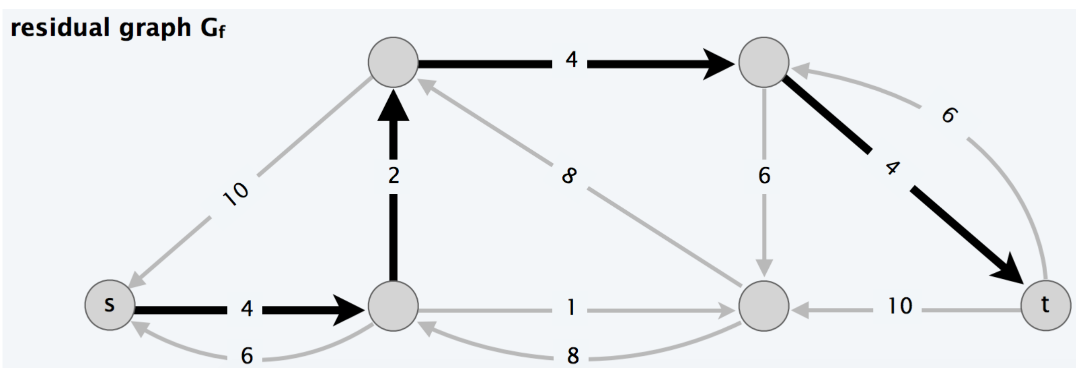
Ford–Fulkerson



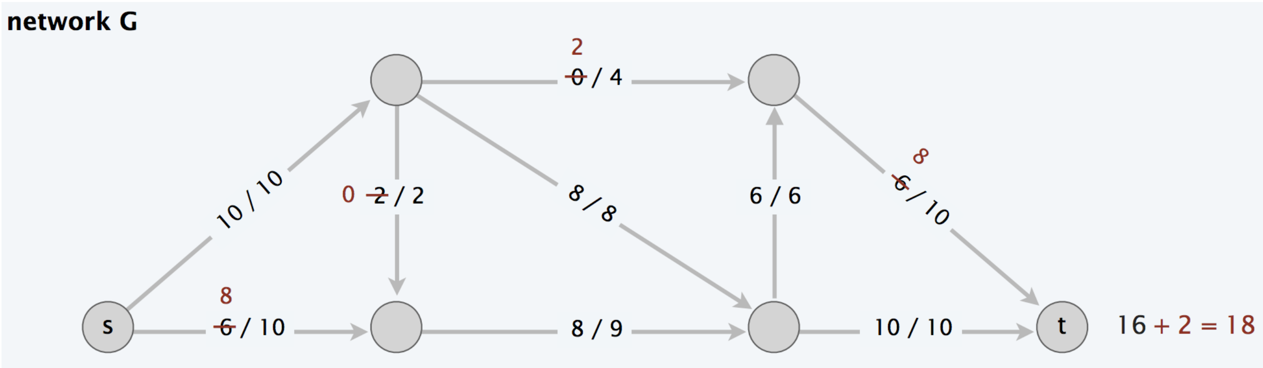


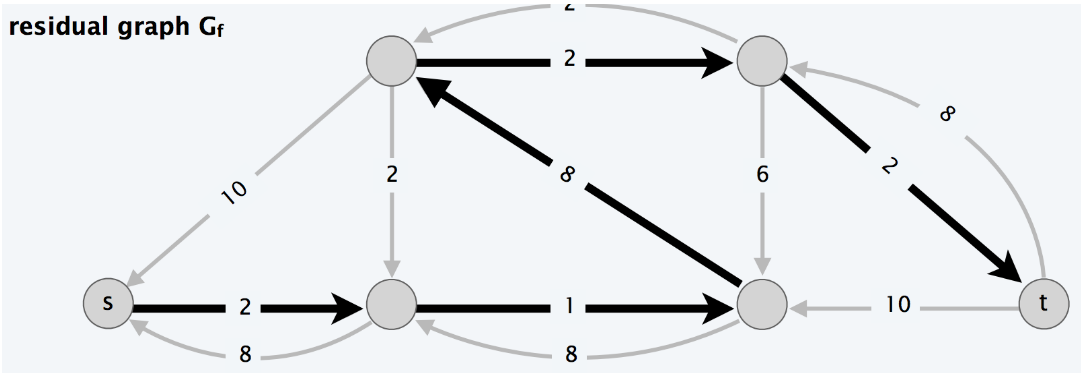
Ford–Fulkerson



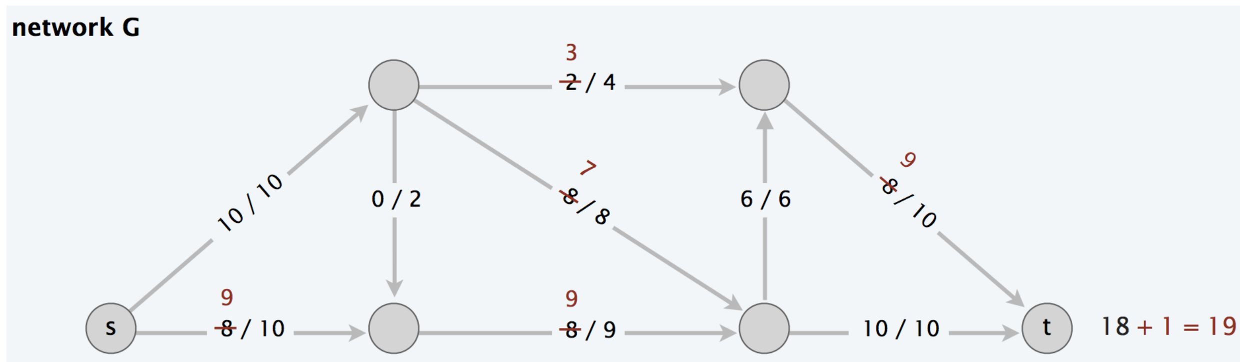


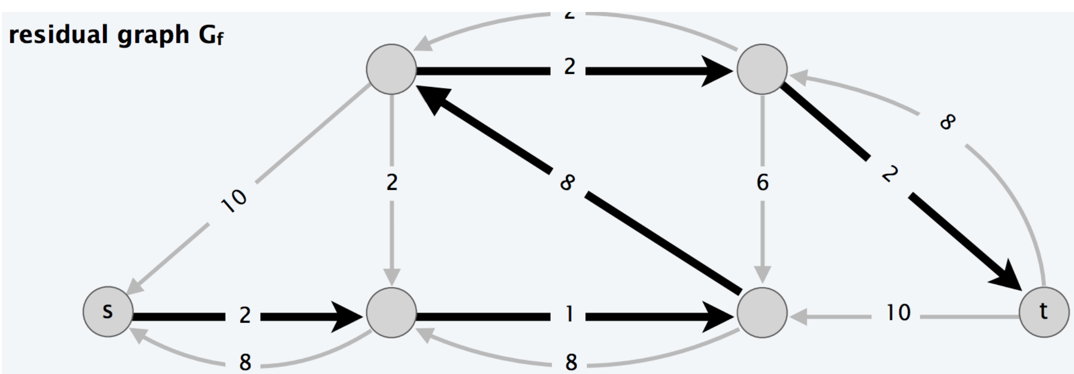
Ford–Fulkerson



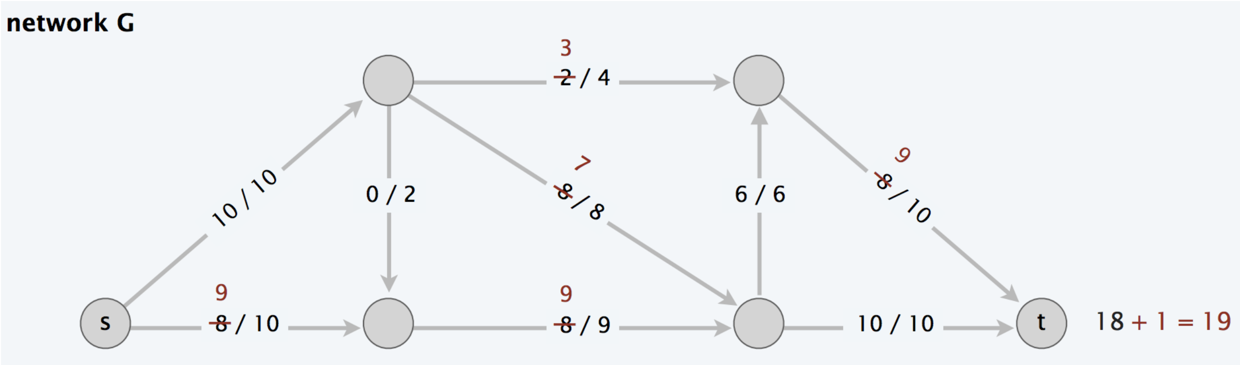


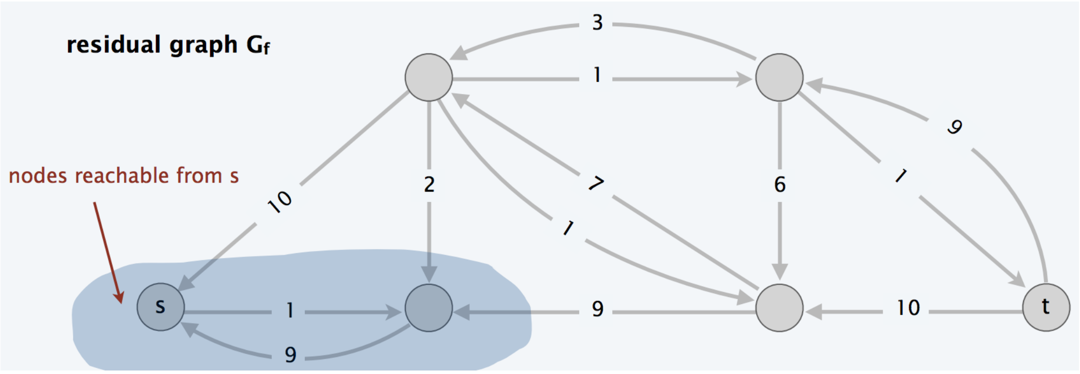
Ford–Fulkerson





Ford–Fulkerson





Ford–Fulkerson

> Sometimes called the "augmenting path" algorithm
  - paths along which we push more flow are called augmenting paths
  - the process of increasing flow along such a path is called an "augmentation"

> Ford-Fulkerson maintains a feasible flow throughout

> It improves the flow through a series of augmentations

> When no augmenting path exists, the flow is optimal
  - (still just a claim… we haven't proven it)

Ford-Fulkerson Correctness

- Proof of correctness depends on a concept will discuss next time…
- For today, we will assume this,
  and consider what consequences that has…

Ford–Fulkerson Application

- **Theorem**: If all the capacities are integers, then there is a maximum flow where each edge flow is *integral*.

- Proof: Ford-Fulkerson will return an integral flow.
  - Feasible flow starts out integral (all zeros).
  - Each augmentation increases integral flows by an integer amount
    - adjustment is min (capacity – flow) over edges
    - if capacities are integers and flows are integers, then capacity – flow is an integer
    - minimum of a set of integers is an integer
  - Hence, the flow is always integral, including at the end.

Ford-Fulkerson Running time

- Each augmentation takes O(m) time
  - O(1) per edge on the path, and at most m edges on the path
- Each iteration increases the value of the flow by at least 1
  - (see previous slide)
- If the maximum capacity on any edge is U, then max value is nU
  - value is sum of flow on (< n) edges coming into the sink
- Ford-Fulkerson running time is O(nmU)

Ford–Fulkerson Running time

- Ford-Fulkerson running time is $O(nmU)$

- **Q**: Is that good?
- **A**: Yes, if U is small
  - in general, $\Theta(nm)$ is the best we should hope for
    - more on this shortly…
    - recall that shortest path also takes $\Theta(nm)$ time if negative weights allowed
  - so this is essentially optimal if $U = O(1)$

Ford–Fulkerson Running time

- Ford-Fulkerson running time is O(nmU)

- **Q**: Is that good?
- **A**: No, if U is large
  - like Knapsack, this is not efficient in the worst case sense
    - this is a pseudo-polynomial time algorithm
  - even in practice, if U is large, this could be very slow

Ford–Fulkerson Running time

- Not hard to make Ford-Fulkerson truly polynomial...
- **Theorem (Edmonds-Karp)**: If we choose *shortest* augmenting paths, then Ford-Fulkerson runs in $O(nm^2)$ time
- Note: that algorithm may not actually be faster in practice
  - other heuristics may work better
  - e.g., choose the path with the largest resulting augmentation

Outline for Today

> Maximum Flow
> Applications
> Ford-Fulkerson
> Other Algorithms

Improvements to Ford-Fulkerson

- Capacity scaling
  - round capacity 1,234 down to 1,000, then F-F, then 1,200, F-F, 1,230, F-F, etc.
    - flow from last F-F is starting point for next… each has less work to do
  - requires only $O(m \log U)$ augmentations, so running time is $O(m^2 \log U)$
    - only faster if $U \gg m/n$
- Shortest augmenting path with lazy path construction
  - reduce work across multiple shortest path calculations by saving information
  - reduces the worst case running time to $O(n^2 m)$
- Both together reduce running time to $O(nm \log U)$
  - quite good

Other Algorithms

- Generic pre-flow push — $O(n^2 m)$
- FIFO pre-flow push — $O(n^3)$
  - essentially optimal if $m = \Theta(n^2)$
- Excess scaling pre-flow push — $O(nm + n^2 \log U)$
  - essentially optimal if $m >> n \log U$

Other Algorithms cont.

- Dinic's algorithm $O(n^2m)$
- Dinic's algorithm + dynamic trees $O(nm \log n)$
  - (data structure of Sleator & Tarjan)
- min(Orlin, KRT) $O(nm)$
  - KRT = King, Rao, & Tarjan
  - KRT runs in time $O(nm \log_b n)$, where $b = m / (n \log n)$
    - if $m >> n \log n$, then $\log_b n = O(1)$
  - KRT is $O(nm)$ except for sparse graphs, Orlin is $O(nm)$ on those

Other Algorithms cont.

> A**nother** algorithm is actually fastest in practice…

Algorithms for Special Cases

- Unit capacities $O(\min(n^{2/3}, m^{1.5}))$
  - breaks the $\Theta(nm)$ barrier

- Bipartite graphs $O(n_1^2 m)$
  - where node set $N = N_1$ union N_2 (two sides)
  - some applications (e.g., network testing) have $n_2 = \Theta(n_1^2) = O(n)$

Max Flow Algorithms Summary

> Ford-Fulkerson is essentially optimal when U = O(1)

> If a library is available, **use it**
  - may give you either an algorithm that is extremely fast in practice OR one of the (complex) algorithms with great worst case performance

> If no library is available, start with Ford-Fulkerson
  - add capacity scaling and/or lazy shortest paths if necessary
  - result is only a log U factor worse than optimal