EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

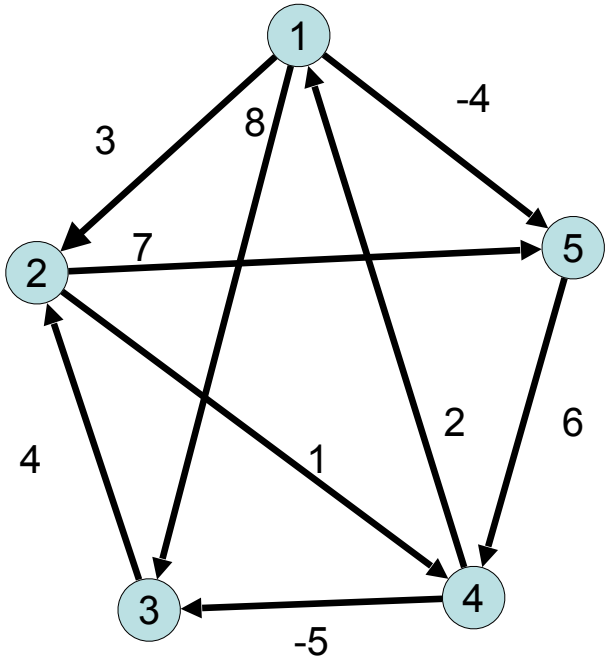
Prof: David Castañón, dac@bu.edu

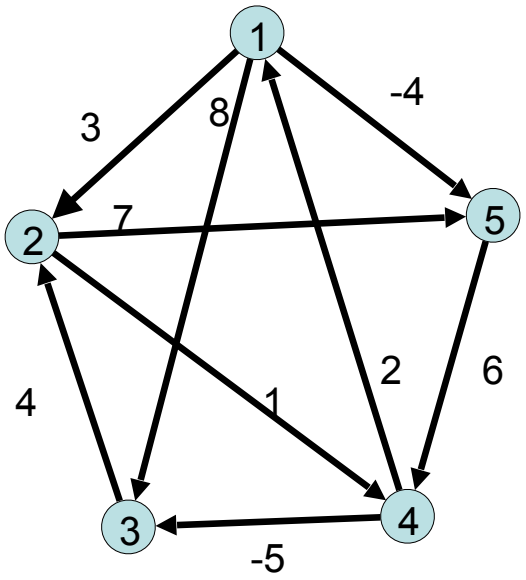
GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Floyd-Warshall: Example 2

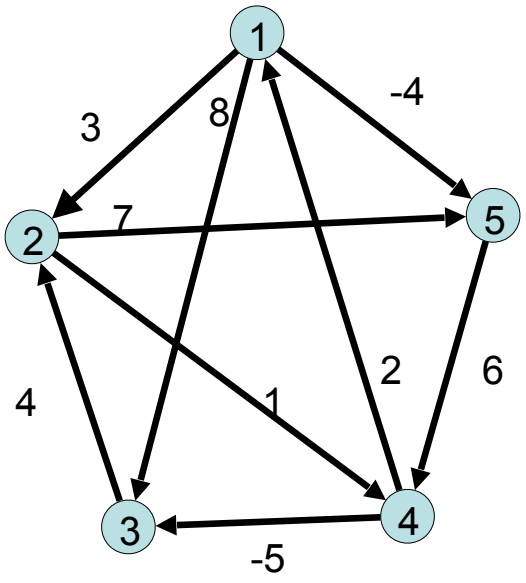




Floyd-Warshall

Initial distances and predecessors

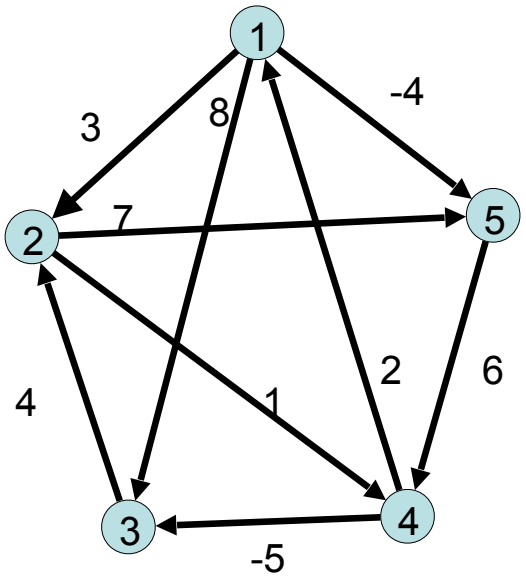
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 1

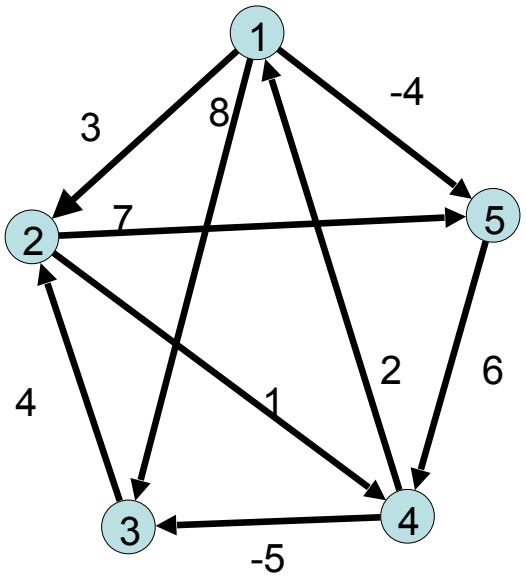
$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 2

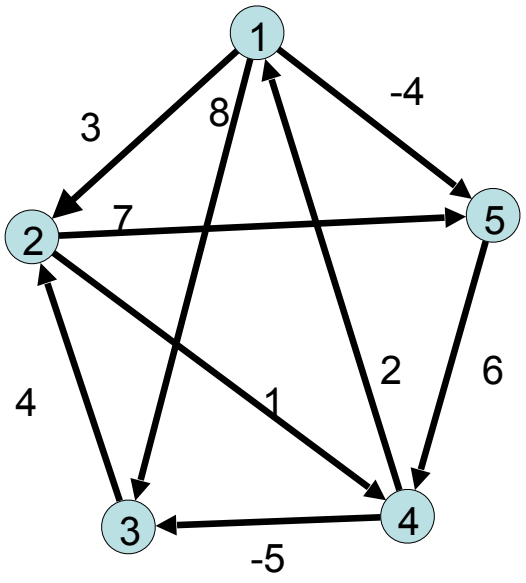
$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 3

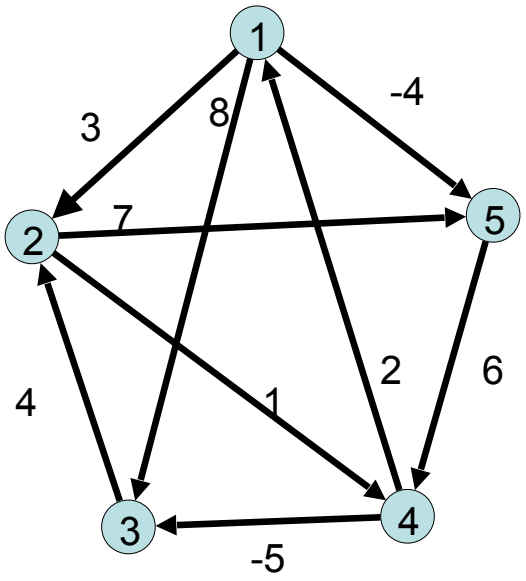
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 4

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 5 (final)

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Can We Do Better for Sparse Graphs?

- If graph is sparse, we can do better with Dijkstra from every vertex, provided weights are non-negative
  - Dijkstra: $O(|V||E| + |V|^2 \log(|V|))$ vs Floyd-Warshall $O(|V|^3)$

- Can we apply Dijkstra when graph has negative weight edges, but not negative cycles?
  - Graph reweighing

Graph Reweighing

Theorem. Given a function h : V → R, **reweigh** each edge (u,v) ∈ E by $w_h(u,v)=w(u,v)+h(u)–h(v)$.

Then, for any two vertices, all paths between them are reweighed by the same amount
==> shortest path in reweighed graph is also shortest in original path

Proof. Let $p = v_1 \to v_2 \to \cdots v_k$ be a path in the original graph G. We have

$$w(p) = w(v_1, v_2) + w(v_2, v_3) + \cdots + w(v_{k-1}, v_k)$$

On the reweighed graph G_h, we have

$$\begin{aligned} w_h(p) &= w_h(v_1, v_2) + w_h(v_2, v_3) + \cdots + w_h(v_{k-1}, v_k) \\ &= w(v_1, v_2) + h(v_1) - h(v_2) + w(v_2, v_3) + h(v_2) - h(v_3) + \cdots + w(v_{k-1}, v_k) + h(v_{k-1}) - h(v_k) \\ &= w(p) + h(v_1) - h(v_k) \end{aligned}$$

Same amount for every path that starts at v_1, ends at v_k!

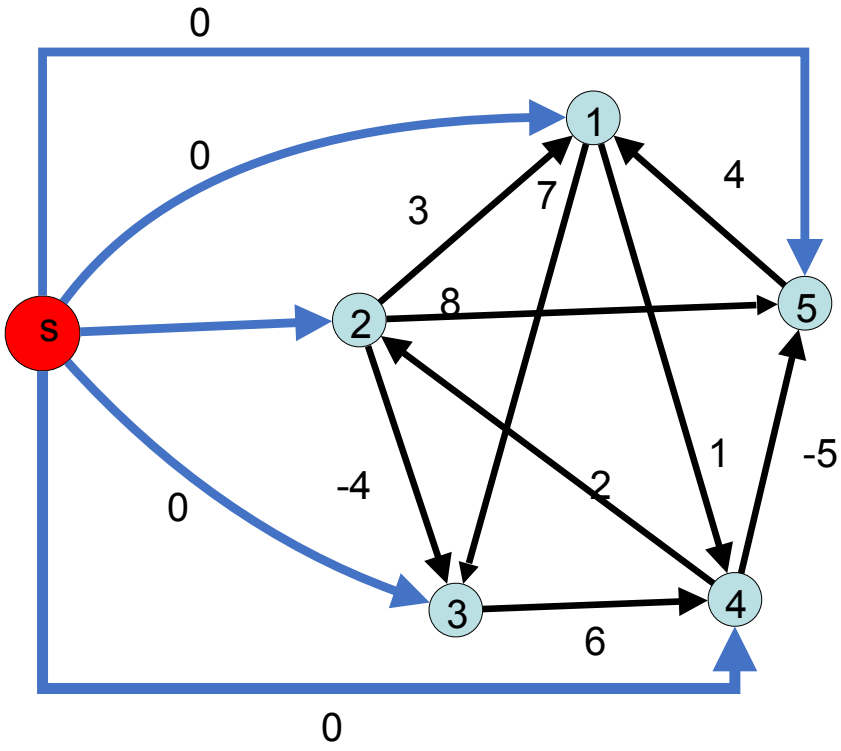
Can We Do Better for Sparse Graphs?

- So, want to find a reweighing function such that $w_h(u, v) \geq 0$ for all edges in E
  - Then, we could solve using Dijkstra's algorithm starting from each vertex
- **Johnson's** algorithm
  - Use an auxiliary shortest path problem solved with Bellman-Ford to using Bellman-Ford to find h(v) such that h(v) – h(u) ≤ w(u, v), or determine that a negative-weight cycle exists.
    • Time = O(|V| |E|)
  - Resulting problem with $w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$

Johnson's algorithm

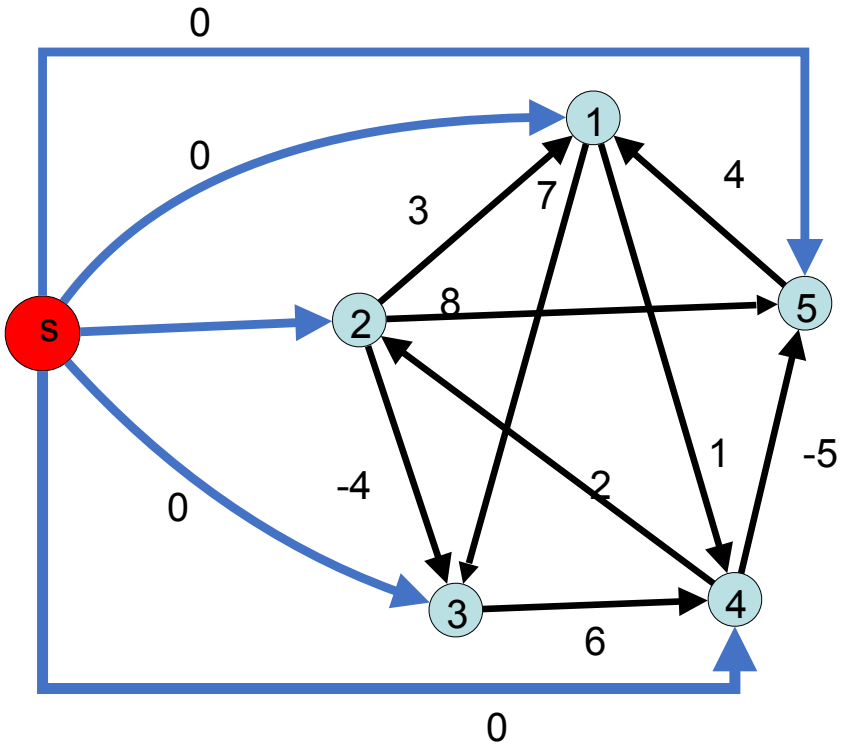
```
Johnson(G)
{   compute G', where V[G']=V[G]∪{s} and
                        E[G']=E[G]∪{(s,v):v∈V[G]
    if Bellman-Ford(G',w,s)=False
       then print "There is a neg-weight cycle"
    else for each vertex v ∈ V[G']
              set h(v)=δ(s,v) computed by Bellman-Ford algo.
              for each edge (u,v)∈E[G']:
                  w'(u,v)=w(u,v)+h(u)-h(v)
              for each vertex u ∈V[G']
                   run Dijkstra(G,w',u) to compute δ'(u,v)
              for each v ∈V[G]
                   d_uv=δ'(u,v)-h(u)+h(v)
    return D
}
```

Johnson's algorithm



Main idea: Add a dummy vertex with a directed edge of weight 0 to all other vertices

Johnson's algorithm

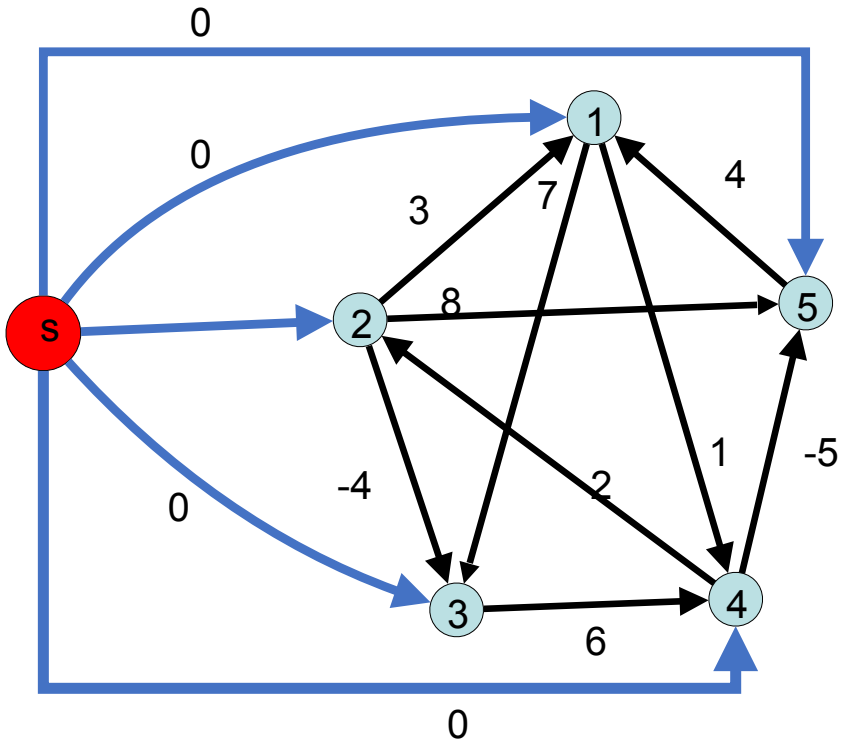


Use Bellman-Ford from s to get shortest distance to all other nodes:

Must be negative

	D[ ]	pred[ ]	Q [ ]
1	0	S	1
2	0	S	2
3	0	S	3
4	0	S	4
5	0	S	5

Johnson's algorithm



Pop 1

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	1
3	0	S	2
4	0	S	3
5	0	S	4

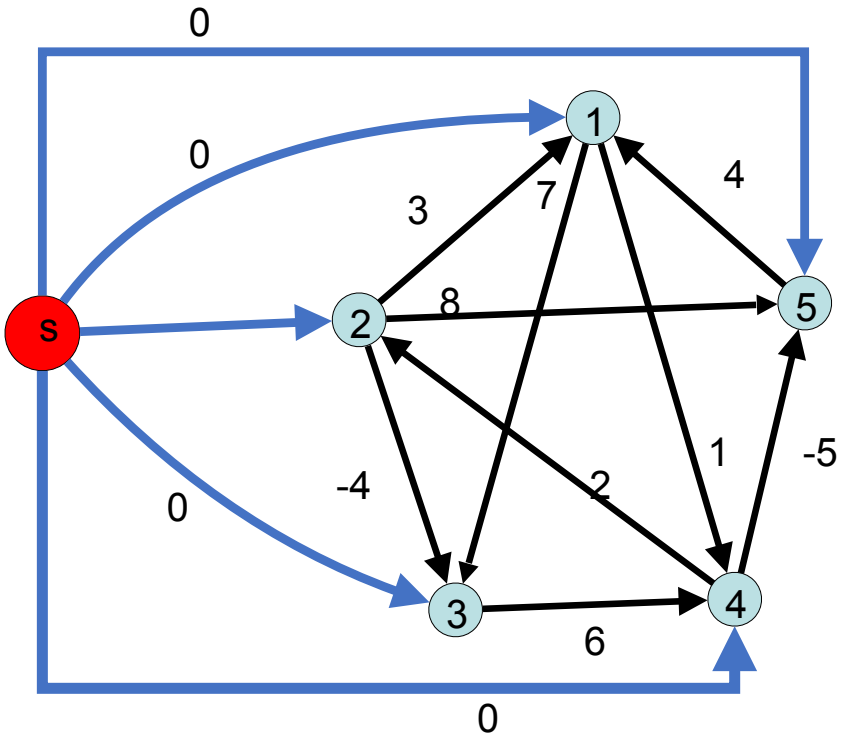
Pop 2

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	
3	-4	2	1
4	0	S	2
5	0	S	3

. . .

All-Pairs Shortest Paths

Johnson's algorithm



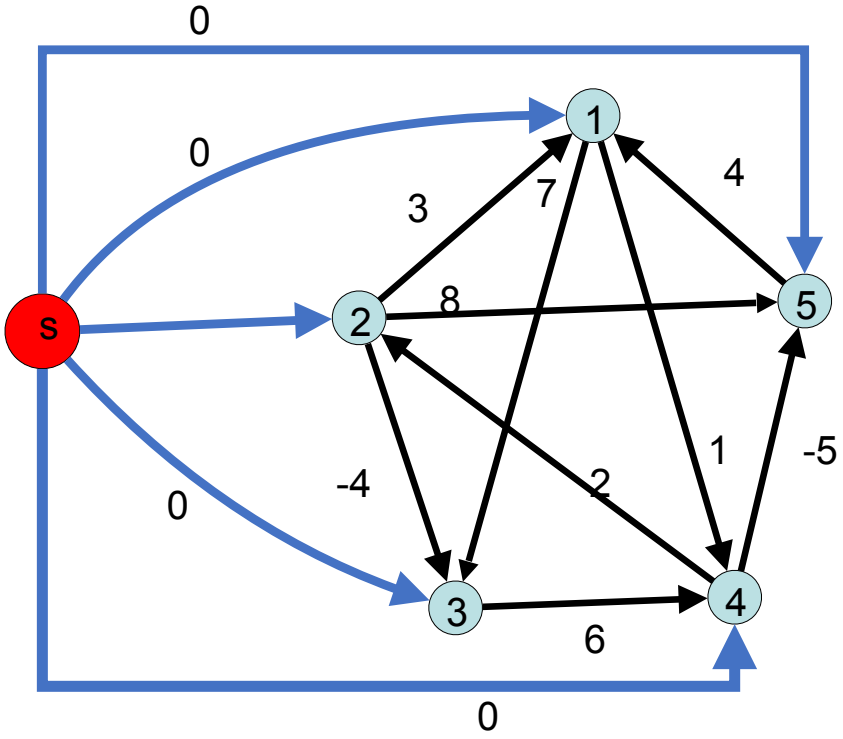
Pop 3

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	
3	-4	2	
4	0	S	1
5	0	S	2

Pop 4

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	
3	-4	2	
4	0	S	
5	-5	4	1

Johnson's algorithm



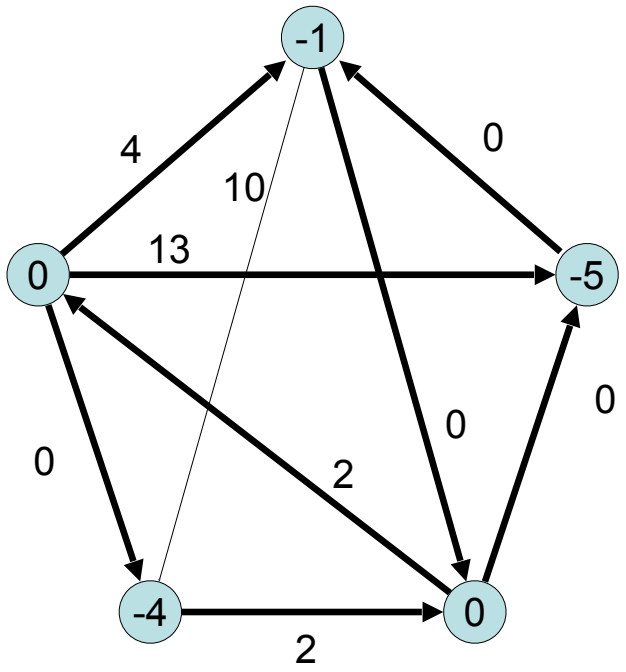
Pop 5

	D[ ]	pred[ ]	Q [ ]
1	-1	5	1
2	0	S	
3	-4	2	
4	0	S	
5	-5	4	

Pop 1

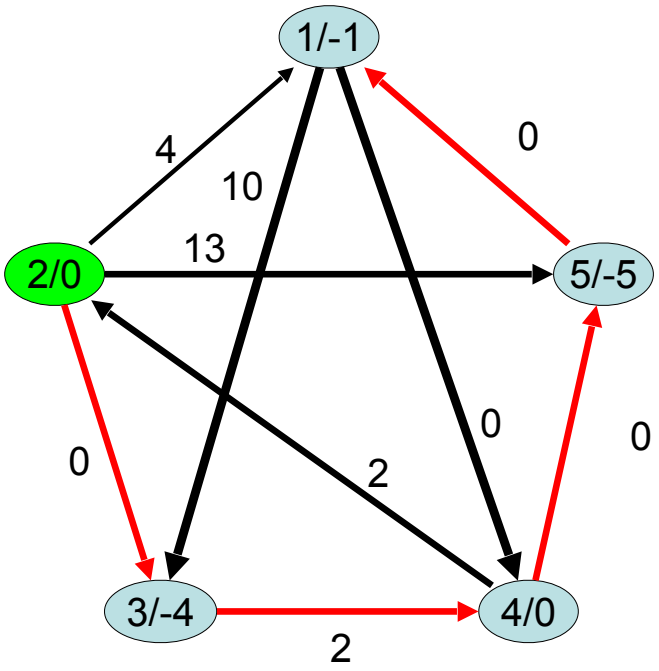
	D[ ]	pred[ ]	Q [ ]
1	-1	5	
2	0	S	
3	-4	2	
4	0	S	
5	-5	4	

Johnson's algorithm



Use Distance from previous problem for reweighing

Johnson's algorithm



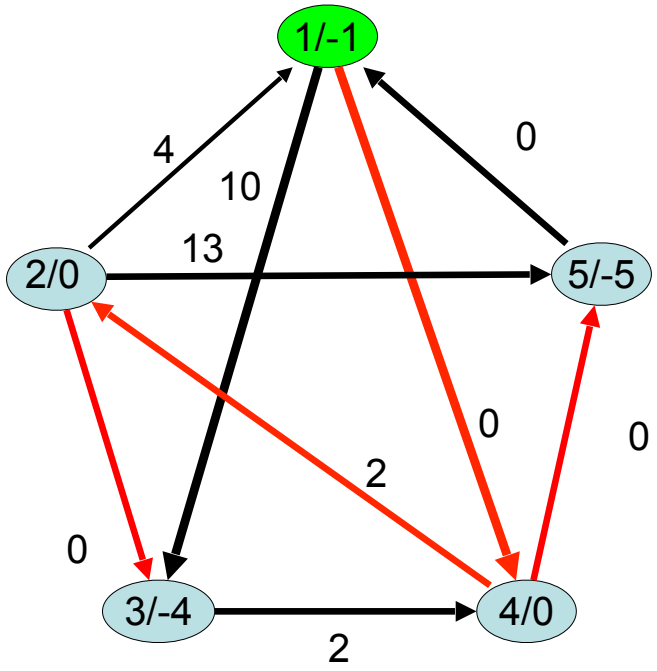
	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	0
3	Inf	Null	
4	Inf	Null	
5	inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	0
4	Inf	Null	
5	13	2	1

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	
4	2	3	0
5	13	2	1

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	
4	2	3	
5	2	4	1

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	
4	2	3	
5	2	4	1



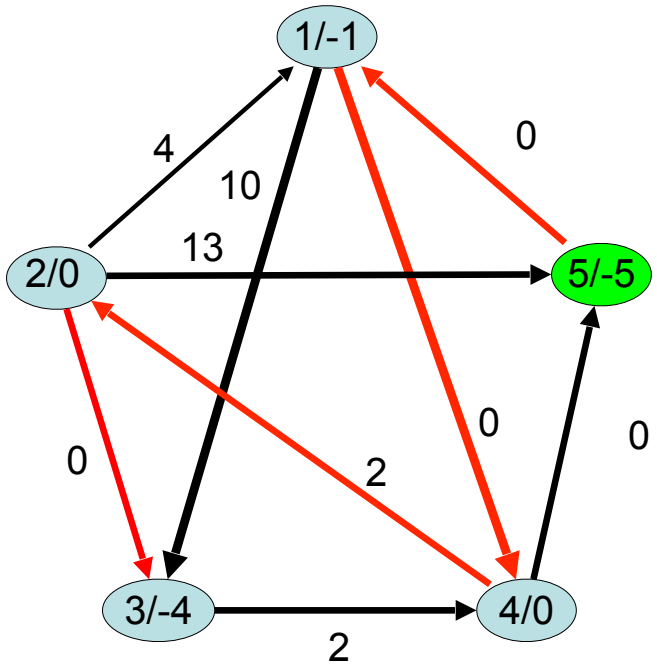
	D[ ]	pred[ ]	Q [ ]
1	0	Null	0
2	Inf	null	
3	Inf	Null	
4	Inf	Null	
5	inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	2	null	2
3	10	1	1
4	0	1	
5	0	5	0

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	null	
3	0	2	0
4	0	1	
5	0	5	

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	Inf	null	
3	10	1	1
4	0	1	0
5	Inf	null	

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	2	null	0
3	10	1	1
4	0	1	
5	0	5	



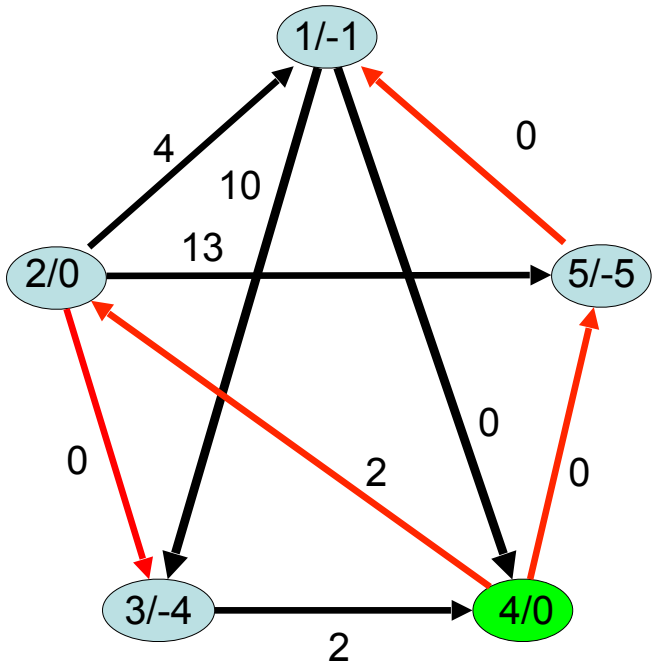
	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	Inf	Null	
4	Inf	Null	
5	0	Null	0

	D[ ]	pred[ ]	Q [ ]
1	0	5	0
2	Inf	null	
3	inf	Null	
4	inf	Null	
5	0	null	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	Inf	null	
3	10	1	1
4	0	1	0
5	0	null	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	0
3	10	1	1
4	0	1	
5	0	null	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	
3	2	2	0
4	0	1	
5	0	null	



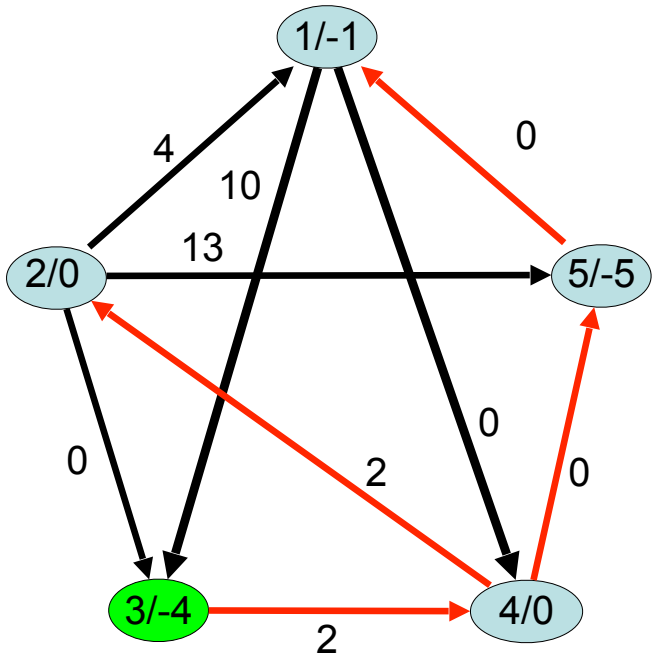
	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	Inf	Null	
4	0	Null	0
5	Inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	2	4	1
3	Inf	Null	
4	0	Null	
5	0	4	0

	D[ ]	pred[ ]	Q [ ]
1	0	5	0
2	2	4	1
3	Inf	Null	
4	0	Null	
5	0	4	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	0
3	10	1	1
4	0	Null	
5	0	4	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	
3	2	3	0
4	0	Null	
5	0	4	



	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	0	Null	0
4	inf	Null	
5	Inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	0	Null	
4	2	3	0
5	Inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	4	4	1
3	0	Null	
4	2	3	
5	2	4	0

	D[ ]	pred[ ]	Q [ ]
1	2	5	0
2	4	4	1
3	0	Null	
4	2	3	
5	2	4	

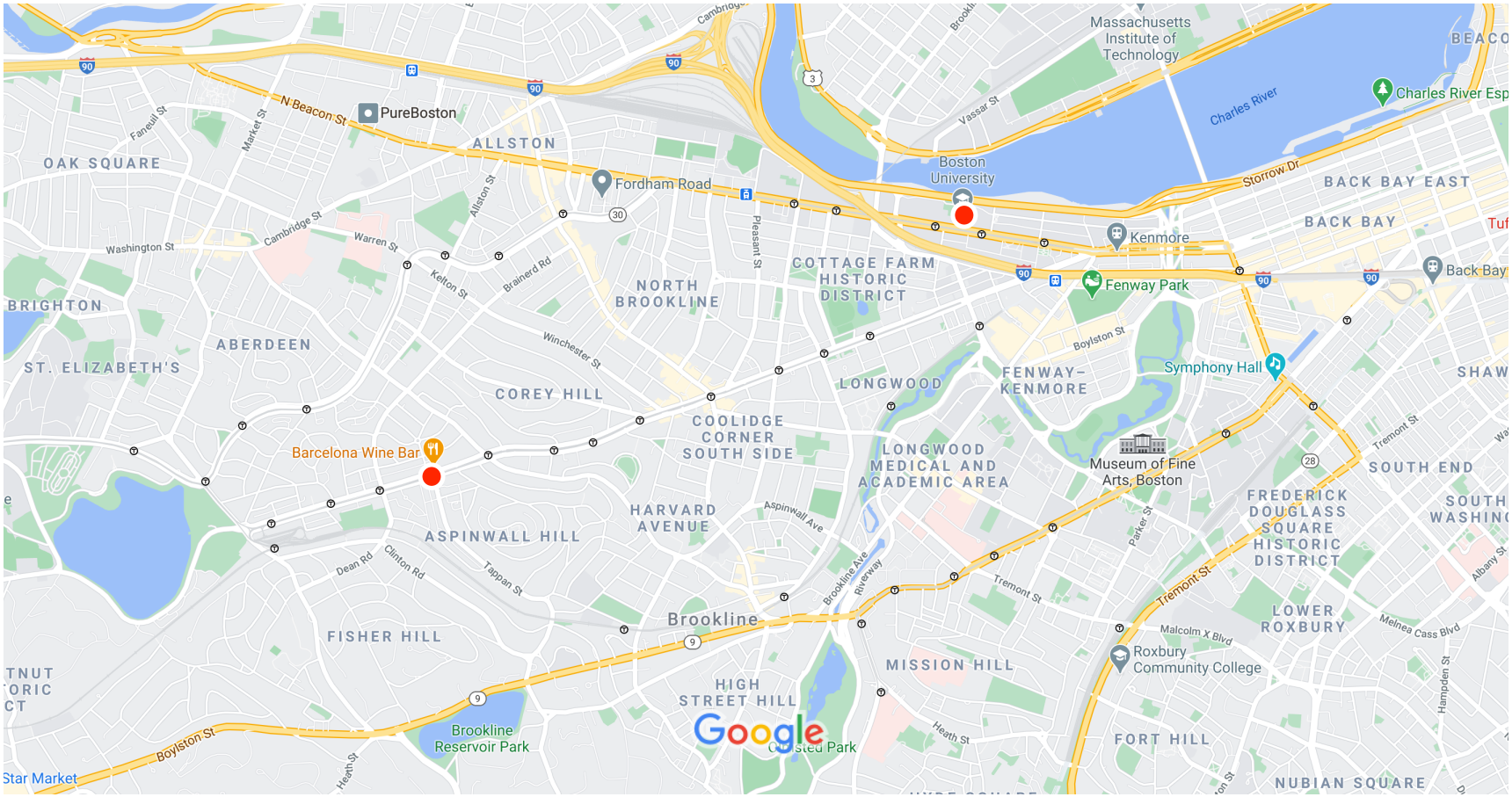
	D[ ]	pred[ ]	Q [ ]
1	2	5	
2	4	4	1
3	0	Null	
4	2	3	
5	2	4	

Problem: Shortest Path from s to a single t

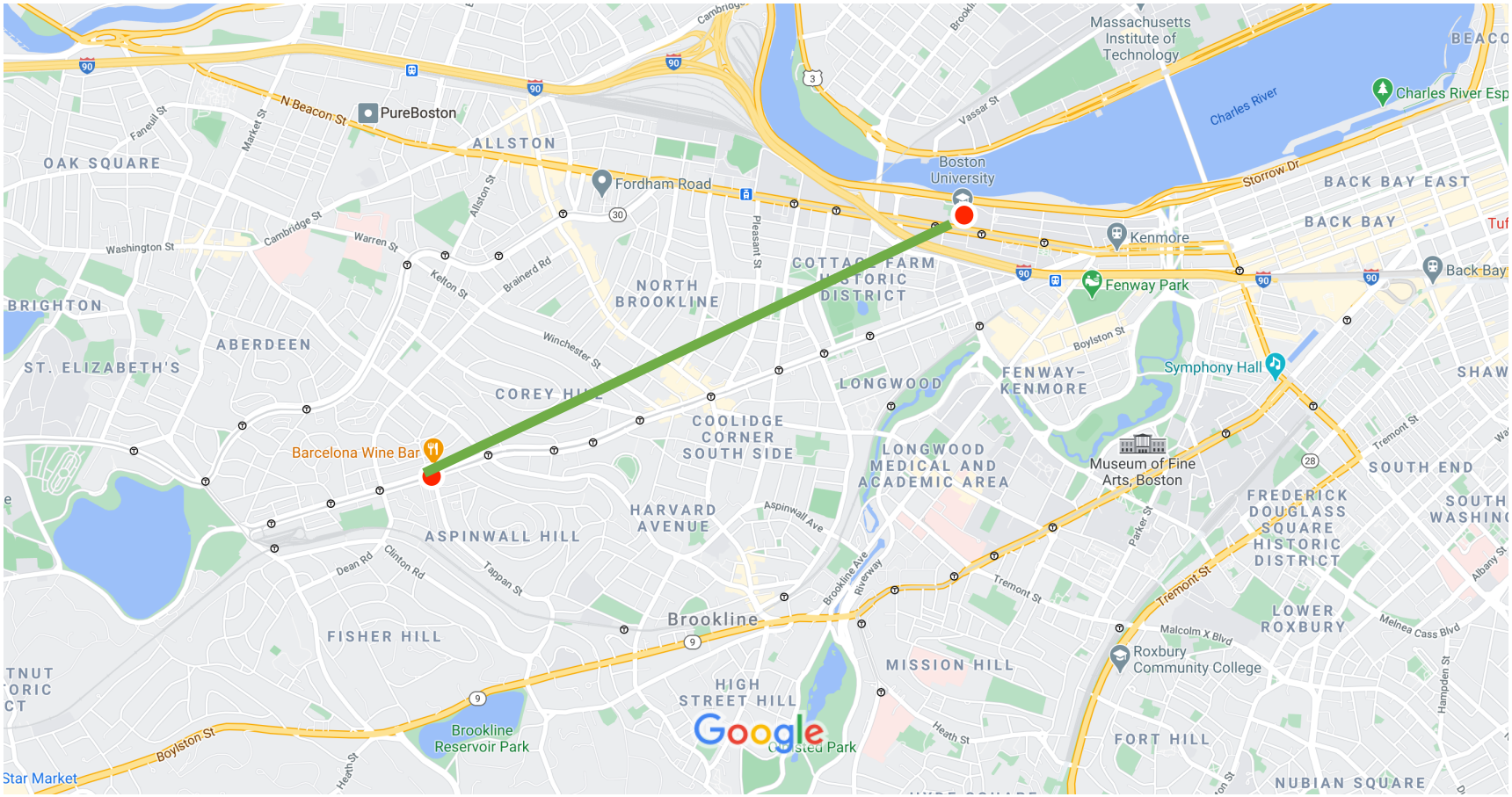
- We only want a path to a destination
  - But all algorithms compute paths from s to every destination
    - When can we stop?
    - Can we focus on getting only to where we want to go?
    - This is the Google maps or Waymo problem

- New algorithm: $A*$ search
  - Will search in direction of t for shortest path
  - A modification of Dijkstra to bias the search
  - Guaranteed optimal under certain conditions

Example


Massachusetts Institute of Technology
Charles River
Charles River Esp
PureBoston
N Beacon St
ALLSTON
OAK SQUARE
Fordham Road
Boston University
Storrow Dr
BACK BAY EAST
BACK BAY
Kenmore
Fenway Park
Back Bay
COTTAGE FARM HISTORIC DISTRICT
NORTH BROOKLINE
BRIGHTON
ABERDEEN
ST. ELIZABETH'S
COREY HILL
Winchester St
Boylston St
Symphony Hall
FENWAY–KENMORE
LONGWOOD
SHAW
Barcelona Wine Bar
COOLIDGE CORNER SOUTH SIDE
LONGWOOD MEDICAL AND ACADEMIC AREA
Museum of Fine Arts, Boston
SOUTH END
FREDERICK DOUGLASS SQUARE HISTORIC DISTRICT
HARVARD AVENUE
Aspinwall Ave
ASPINWALL HILL
Tremont St
Brookline
FISHER HILL
LOWER ROXBURY
Malcolm X Blvd
Roxbury Community College
MISSION HILL
HIGH STREET HILL
Brookline Reservoir Park
FORT HILL
NUBIAN SQUARE
Star Market
Google

Distance lower bound: Straight Path



Heuristic Distance Estimates

- Let $h : V \rightarrow \Re$ be a heuristic function that approximates the distance from vertex v to target t
- We call h **admissible** if: for each vertex v: h(v) ≤ D(v,t), where D(v,t) is the true shortest path distance
- We call h **consistent** if: For each (u,v) in E: h(u) ≤ w(u,v) + h(v).
  - For example, Euclidean distance to target on the plane
    - Euclidean distance is admissible and consistent

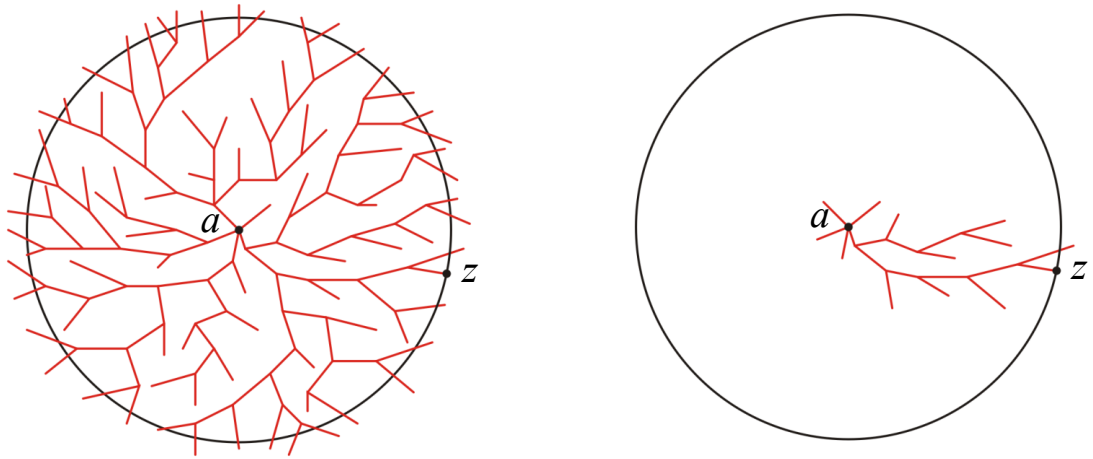
A* Search

- Simple idea: use heuristic h() to reweigh edges
  - $w_h(u, v) = w(u, v) - h(u) + h(v)$
    - Measures deviation from ideal distance estimate by h
    - Consistent heuristics guarantee non-negative weights
- Solve using Dijkstra's algorithm
- Result: If heuristic is consistent and admissible, we get an optimal shortest path
  - Dijkstra's distance is biased towards vertices that have shorter distance estimates

Comparison with Dijkstra's Algorithm

Graphically, we can suggest the behavior of the two algorithms as follows:

- Suppose we are moving from a to z:



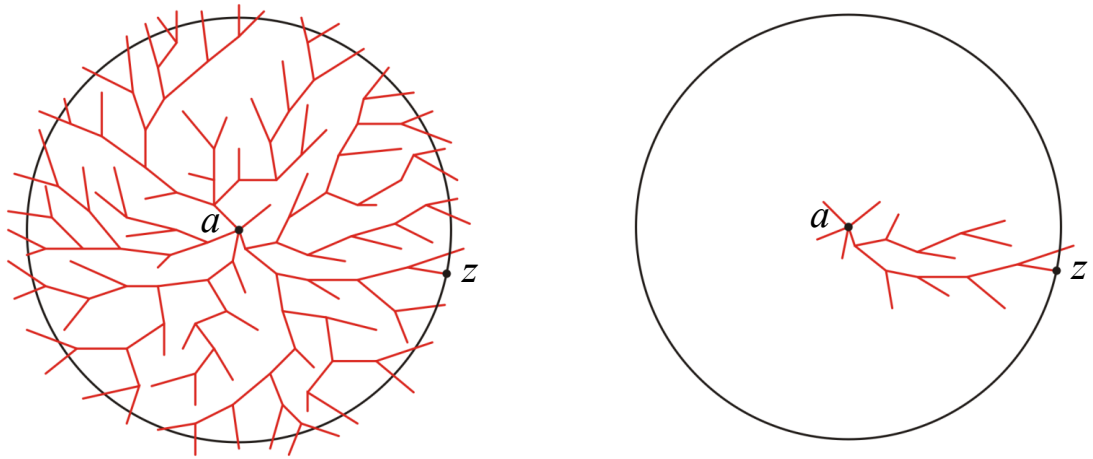
Representative search patterns for Dijkstra's and the A* search algorithms

Comparison with Dijkstra's Algorithm

Dijkstra's algorithm is the A* search algorithm when

using the heuristic distance $$h(u,v) = \begin{cases} 0 & u = v \\ 1 & u \neq v \end{cases}$$

- No vertex is better than any other vertex



Representative search patterns for Dijkstra's and the A* search algorithms

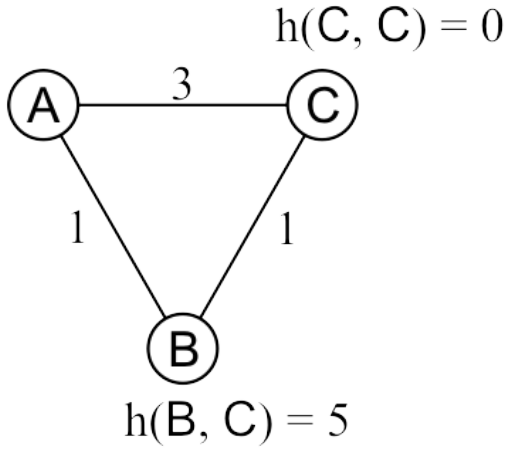
Algorithm Description

- Keep track of best distance from s, but use as key in the priority queue the optimistic distance estimate to destination t
- Initially, mark distances D[s] = 0, D[v] = ∞, $v \neq s$, and key[v] = D[v] + h(v,t), pred[v] = NULL. Insert into priority queue Q.
- While t is still in Q:
  - Remove_min from Q, as vertex u
  - If u = t, break. We have the path we want.
  - For all v such that $(u, v) \in E$, and v is still in Q, compute $D_v = D[u] + w(u, v)$.
    - If $D_v < D[v]$, update $D[v] = D_v$, pred[v] = u, key[v] = D_v + h(v,t), and Decrease_key of v in Q

- Note: We don't have to keep track of key[v] explicitly in Q if we know h(v,t)

Optimally Guarantees?

- If h() is not consistent or admissible, this algorithm can fail
- Consider simple figure, with s = A, t = C, and with heuristic h(B,C) = 5, h(C,C) = 0
- Starting from A,
  - D[C] = 3, key[C] = 3, pred[C] = A;
  - D[B] = 1, key[B] = 6, pred[B] = A;
- First vertex out of Q is C, so shortest path identified is A—> C
  - Clearly longer than A—>B—>C
- Problem? Heuristic not admissible (not optimistic)



Optimality Results

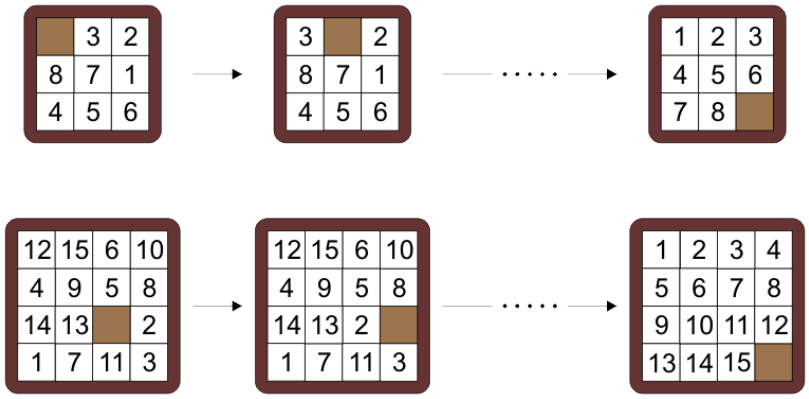
- Theorem: Let h() be admissible and consistent. If A* selects a path p from s to t, then p is the shortest path
  - Let w(p) be the distance of path p selected by A*, which is key[t] when A* ends
  - Assume for contradiction that another path p′ is shorter path to t, so w(p') < w(p)
  - Since h(v,t) is admissible, for any vertex v in p',

$$w(\{s, v]\}) + h(v, t) \leq w(p') < w(p)$$

  - Note, for any vertex v in p', key[v] < w(p); furthermore, key[t] = w(p) when t was removed from the priority queue Q to finish A*
  - Thus, all vertices v in p' should have been removed from Q before t with key[t] = w(p)
  - This means that t should have been removed from Q with key[t] = w(p'), which is a contradiction that A* ended with path p

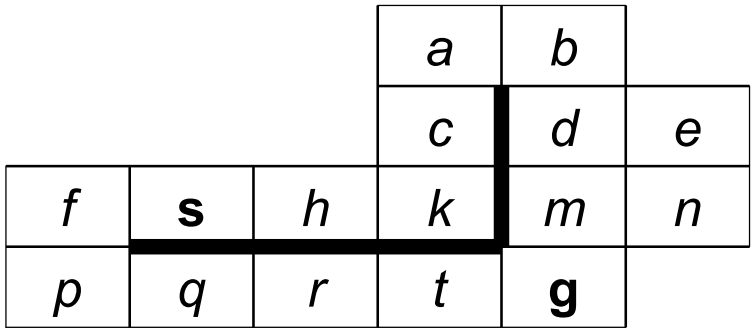
A* Search: Limitations

- Can use lots of memory!
  - Similar to Breadth-First-Search
  - Hence, often replaced by alternatives with bounded-memory
- Example: N-puzzle (9! And 16!) possible vertices in Priority Queue
  - Need efficient approaches
  - Not all vertices are reachable; only half can be solved… Still large!



A* Search: Example

- Problem: Find shortest path from start s to end g
  - There is a barrier!
  - Can only move horizontally, vertically as long as there is no barrier
  - Heuristic h(v,t) = sum of vertical and horizontal distance from v to g
    - Manhattan distance
    - e.g. h(f,g) = 5, h(k,g) = 2, h(e,g) = 3

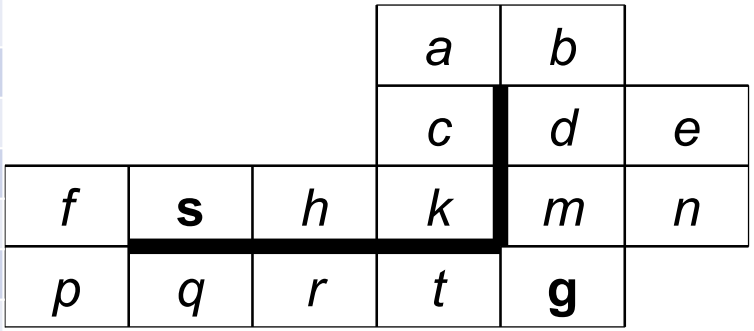


- Execute A* search on this

A* Search: Example

	D	Key	Pred
a	Inf	Inf	-1
b	Inf	Inf	-1
c	Inf	Inf	-1
d	Inf	Inf	-1
e	Inf	Inf	-1
f	Inf	Inf	-1
g	Inf	Inf	-1
h	Inf	Inf	-1
k	Inf	Inf	-1
m	Inf	Inf	-1
n	Inf	Inf	-1
p	Inf	Inf	-1
q	Inf	Inf	-1
r	Inf	Inf	-1
s	0	4	-1
t	Inf	Inf	-1

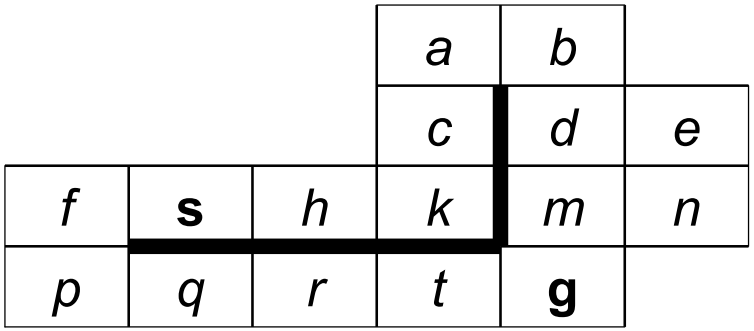
	D	Key	Pred
a	Inf	Inf	-1
b	Inf	Inf	-1
c	Inf	Inf	-1
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	Inf	Inf	-1
h	1	4	S
k	Inf	Inf	-1
m	Inf	Inf	-1
n	Inf	Inf	-1
p	Inf	Inf	-1
q	Inf	Inf	-1
r	Inf	Inf	-1
s	0	4	-1
t	Inf	Inf	-1



A* Search: Example

	D	Key	Pred
a	Inf	Inf	-1
b	Inf	Inf	-1
c	Inf	Inf	-1
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	Inf	Inf	-1
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	Inf	Inf	-1
q	Inf	Inf	-1
r	Inf	Inf	-1
s	0	4	-1
t	Inf	Inf	-1

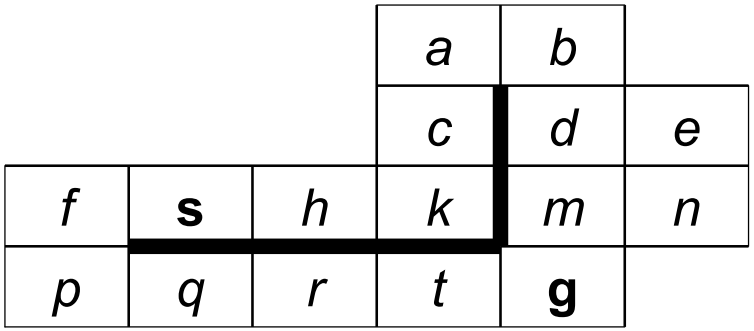
	D	Key	Pred
a	Inf	Inf	-1
b	Inf	Inf	-1
c	3	6	K
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	Inf	Inf	-1
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	Inf	Inf	-1
q	Inf	Inf	-1
r	Inf	Inf	-1
s	0	4	-1
t	Inf	Inf	-1



A* Search: Example

	D	Key	Pred
a	4	8	C
b	Inf	Inf	-1
c	3	6	K
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	Inf	Inf	-1
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	Inf	Inf	-1
q	Inf	Inf	-1
r	Inf	Inf	-1
s	0	4	-1
t	Inf	Inf	-1

	D	Key	Pred
a	4	8	C
b	Inf	Inf	-1
c	3	6	K
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	Inf	Inf	-1
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	2	6	F
q	Inf	Inf	-1
r	Inf	Inf	-1
s	0	4	-1
t	Inf	Inf	-1



A* Search: Example

	D	Key	Pred
a	4	8	C
b	Inf	Inf	-1
c	3	6	K
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	Inf	Inf	-1
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	2	6	F
q	3	6	P
r	Inf	Inf	-1
s	0	4	-1
t	Inf	Inf	-1

	D	Key	Pred
a	4	8	C
b	Inf	Inf	-1
c	3	6	K
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	Inf	Inf	-1
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	2	6	F
q	3	6	P
r	4	6	Q
s	0	4	-1
t	Inf	Inf	-1

	D	Key	Pred
a	4	8	C
b	Inf	Inf	-1
c	3	6	K
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
g	6	6	t
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	2	6	F
q	3	6	P
r	4	6	Q
s	0	4	-1
t	5	6	-1

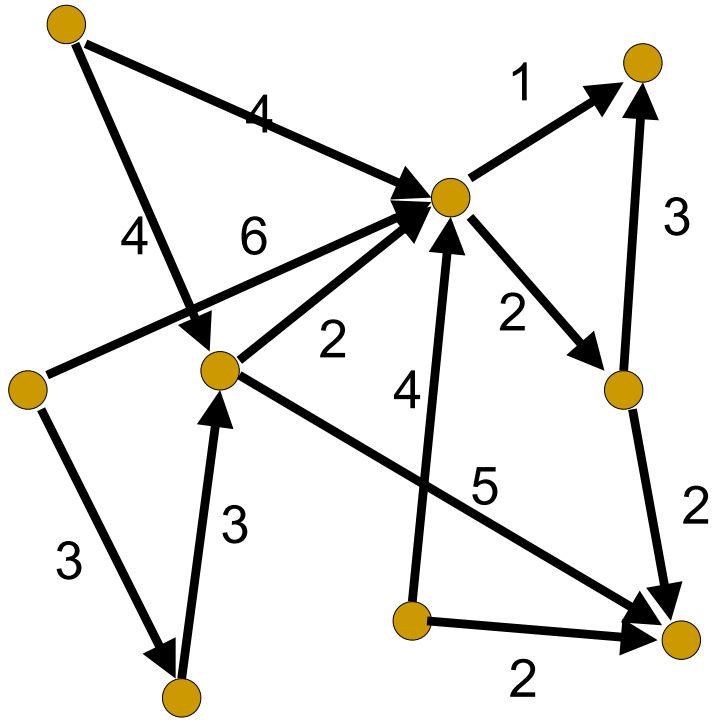
	D	Key	Pred
a	4	8	C
b	Inf	Inf	-1
c	3	6	K
d	Inf	Inf	-1
e	Inf	Inf	-1
f	1	6	S
h	1	4	S
k	2	4	h
m	Inf	Inf	-1
n	Inf	Inf	-1
p	2	6	F
q	3	6	P
r	4	6	Q
s	0	4	-1
t	5	6	-1

Graph Algorithms

- So far, we have studied problems with simple solutions
  - BFS, DFS: O(|V| + |E|) complexity for many variations of problems
  - MST, Shortest paths: O(|E| + |V| log(|V|)) for fastest worst case with good data structures
  - Complexity of solution is comparable to that of reading the input!
- We will consider harder graph problems next
  - Complexity still polynomial in problem size, but higher degree polynomial
- For instance, shortest path algorithms assume only one car on the road
  - But what if every car followed the same Waymo or Google Maps advice?
  - Shortest path = longest traffic jam!
  - Need algorithms that consider road capacity, multiple vehicles at the same time

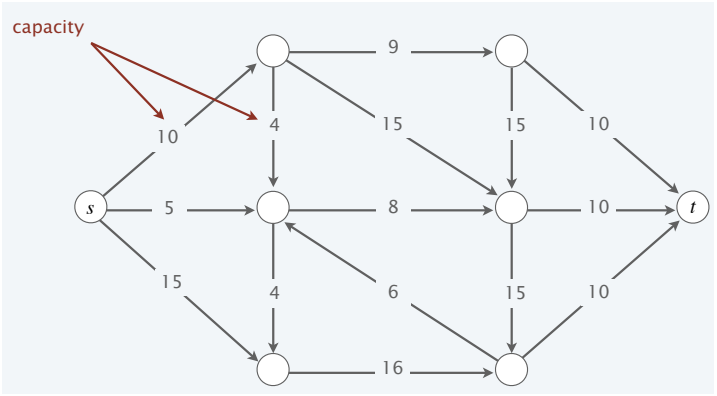
Directed, Capacitated Graphs

- A capacitated, directed graph $G = (V, E)$ is a directed graph along with a capacity function $c : E \rightarrow \mathfrak{R}^+$
- Capacities are positive
  - Represent maximum number of simultaneous units that can sue an edge



Flow Networks

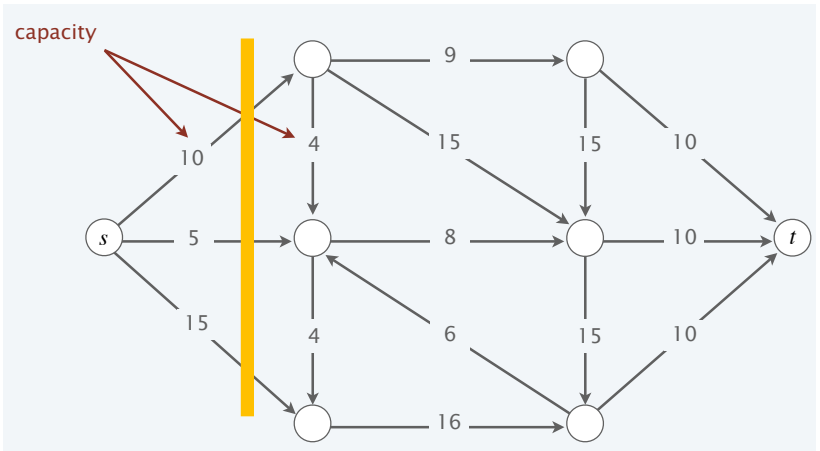
- A flow network is a tuple G = (V, E, s, t, c).
  - Directed graph (V, E) with source s ∈ V and sink t ∈ V.
  - Capacity c(e) ≥ 0 for each e ∈ E.
- Intuition. Material flowing through a transportation network
  - Material originates at source and is sent to sink.

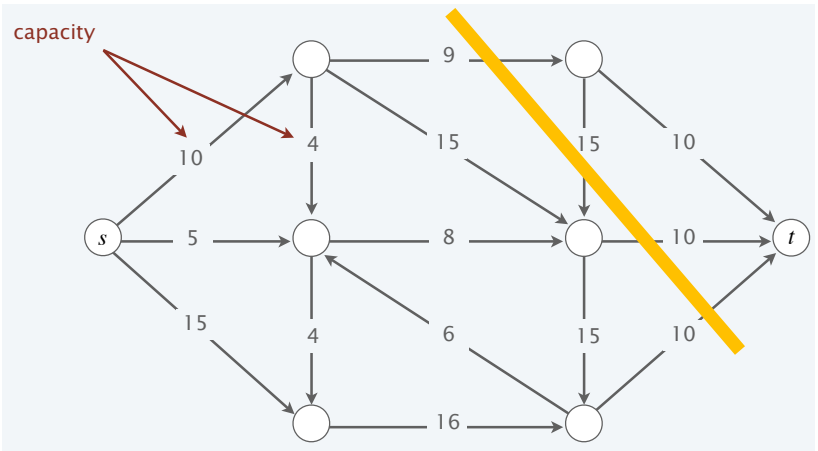


Cuts in Flow Networks

- An st-cut (cut) is a partition (A, B) of the nodes with s ∈ A and t ∈ B.
  - Capacity of st-cut (A,B) = sum of the capacities of edges from A to

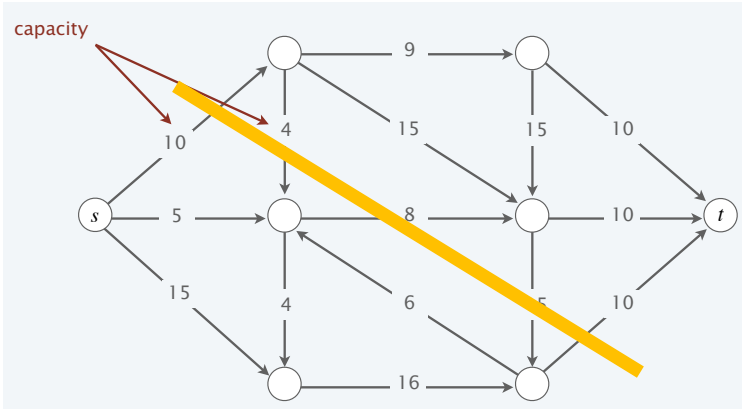
$$Cap(A,B) = \sum_{(u,v)\in E: u\in A, v\in B} c(u,v)$$

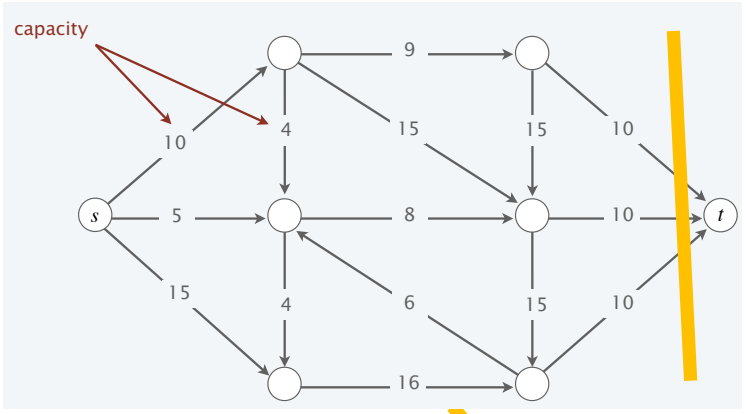


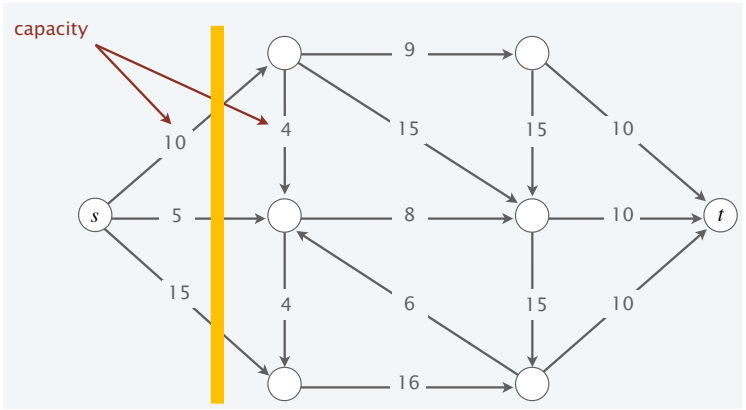


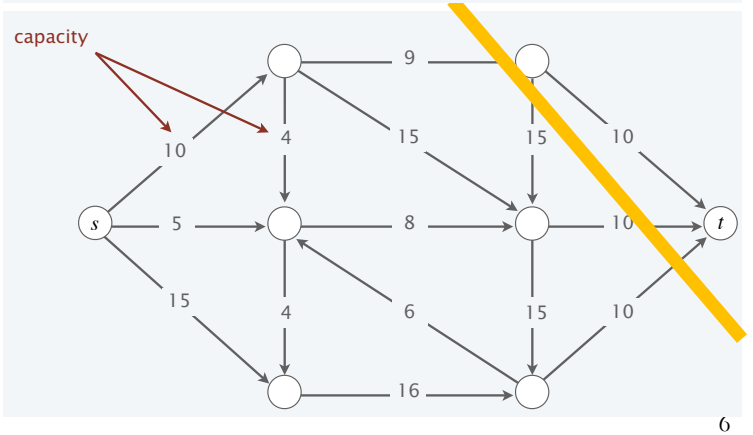
Minimum Cut Problem in Flow Networks

- Find the st-cut (cut) with minimum capacity



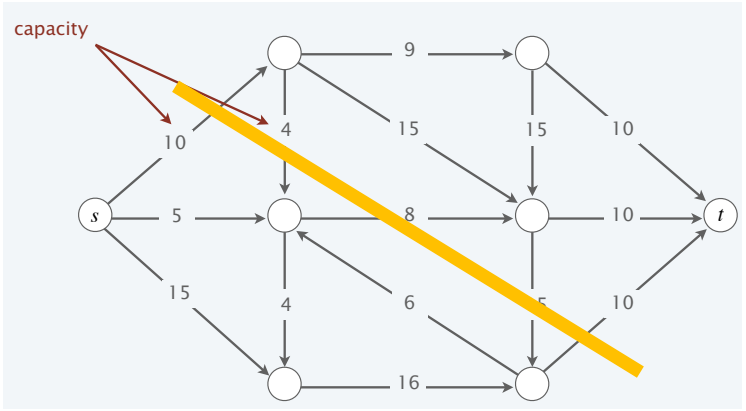


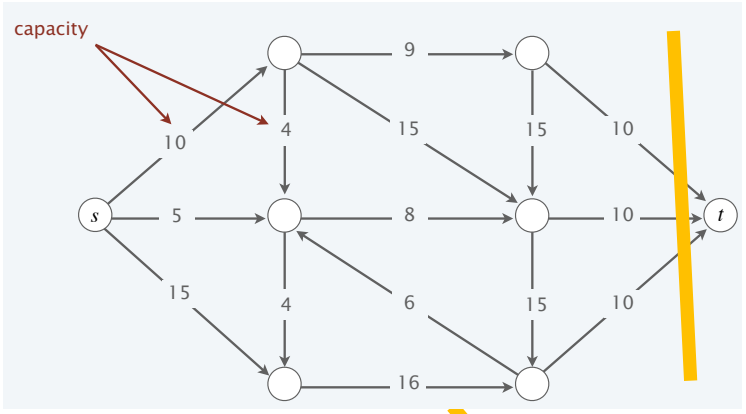


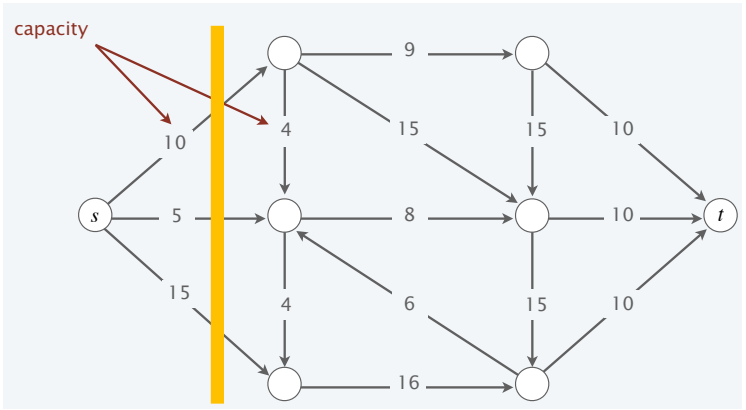


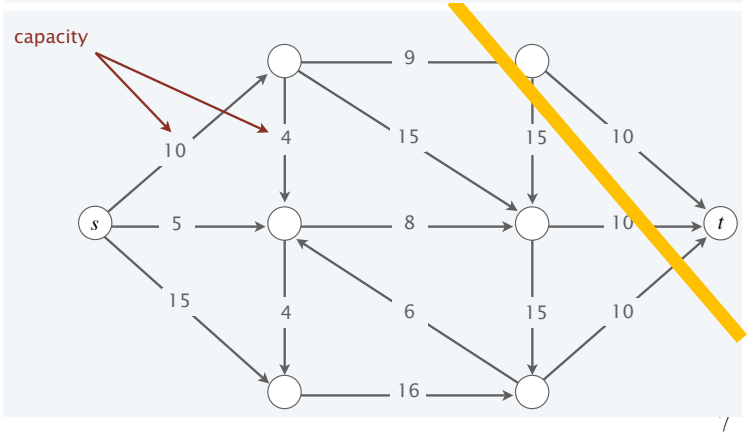
Minimum Cut Problem in Flow Networks

- Find the st-cut (cut) with minimum capacity







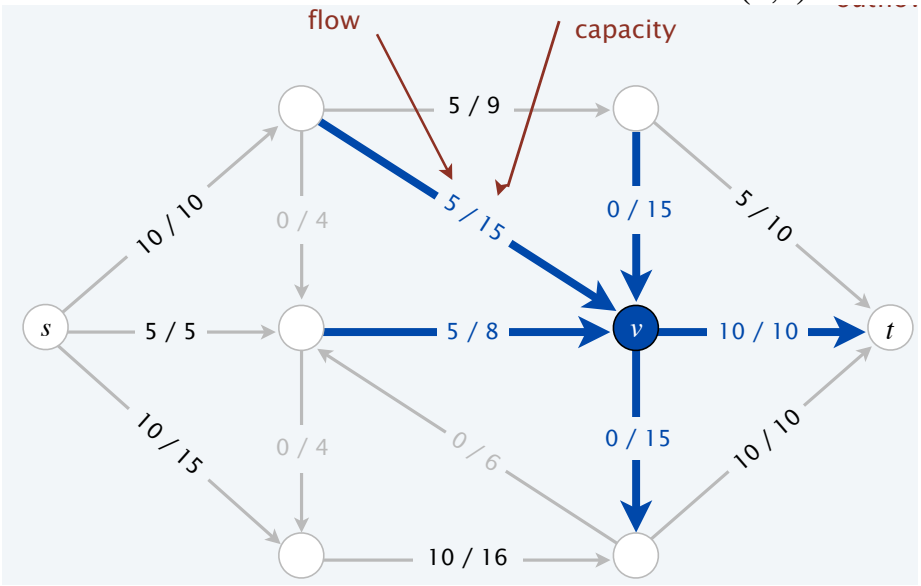


Flow in Flow Networks

- A feasible st-flow (flow) $f : E \rightarrow \Re^+$ is a function that satisfies

  $0 \leq f(u,v) \leq c(u,v)$ for all edges $(u\,.\,v) \in E$ (capacity satisfied)

  For every $v \in V - \{s,t\}$, flow is conserved: $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w)$



Maximum Flow in Flow Networks

- A feasible st-flow (flow) $f : E \to \Re^+$ is a function that satisfies

  $0 \le f(u,v) \le c(u,v)$ for all edges $(u\,.\,v) \in E$ (capacity satisfied)

  For every $v \in V - \{s,t\}$, flow is conserved: $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w)$

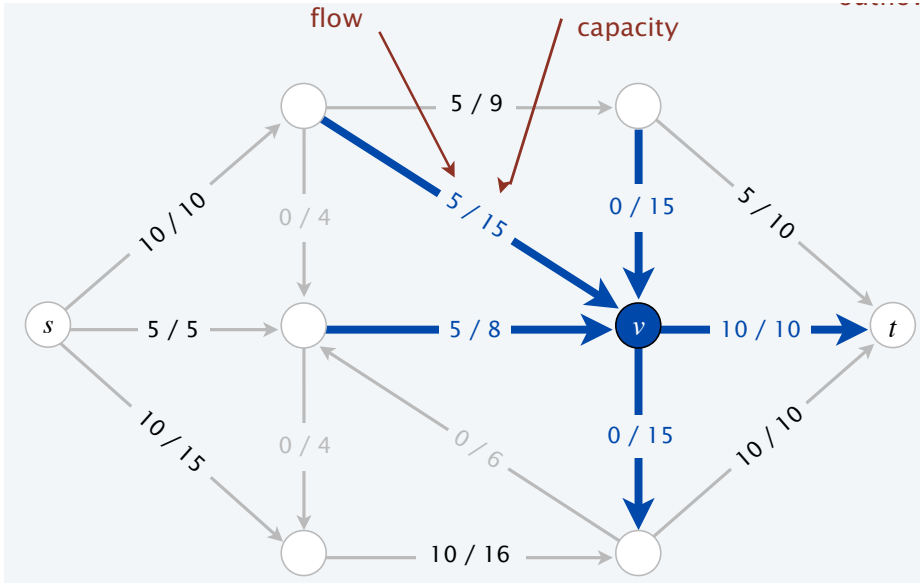
- Value of flow: net flow out of s

  $$val(f) = \sum_{(s,v)\in E} f(s,v) - \sum_{(v,s)\in E} f(v,s)$$

  Value is 25 on the right…

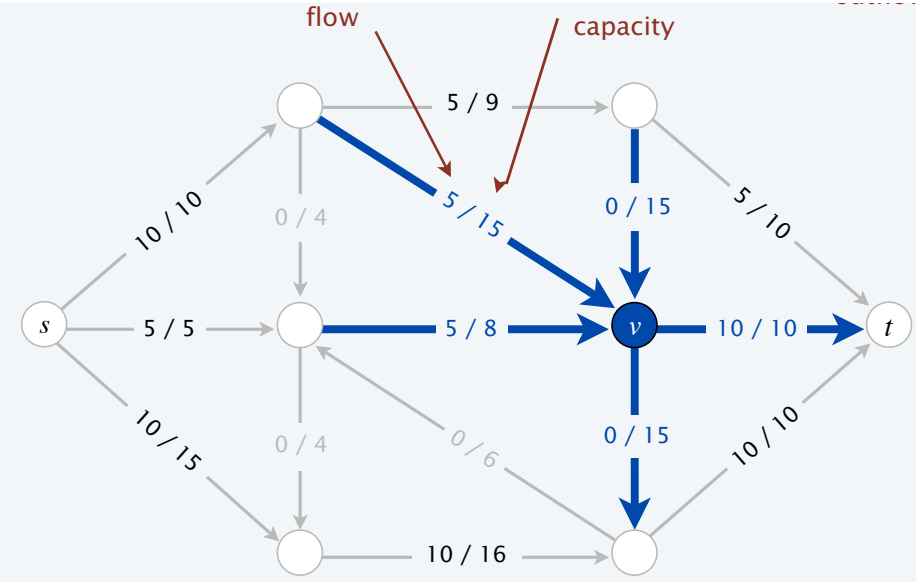
- Maximum flow problem: find flow f that maximizes $val(f)$

-



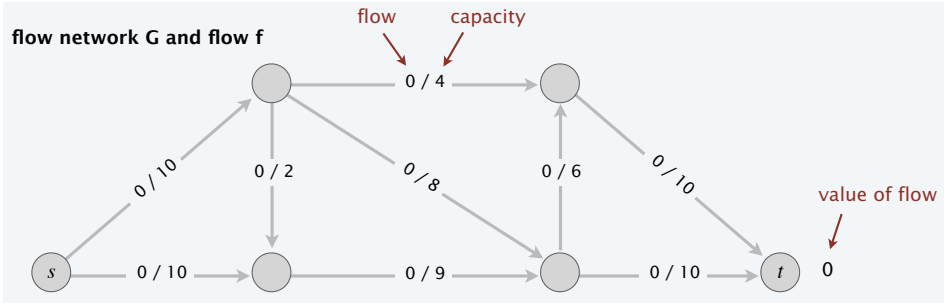
Max-flow, Min-cut are Related

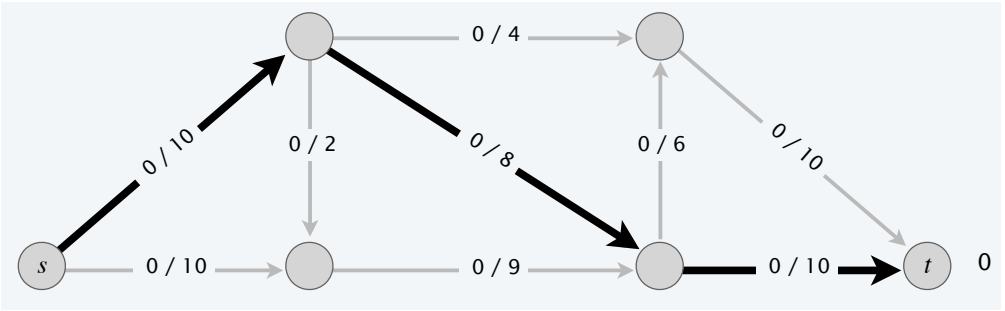
- Formally, they are dual of each other
- Result to come: optimal value of max-flow problem is the same as optimal value of min-cut problem
- Any s-t cut is an upper bound to the max flow problem
- Any feasible flow provides a lower bound to the min-cut problem



Initial Algorithm: Greedy approach

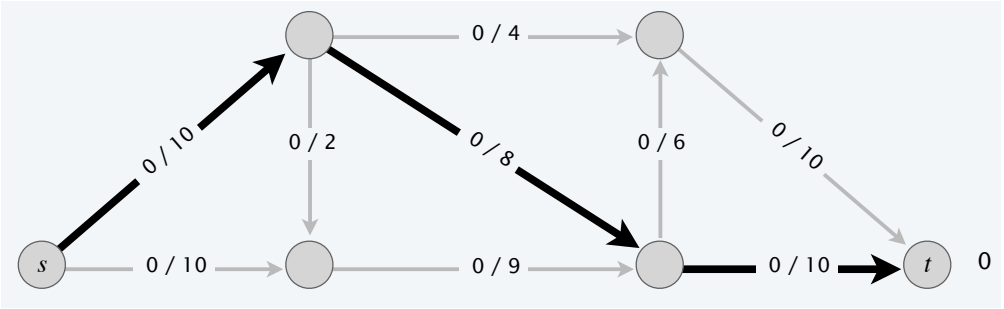
- Start with f (e) = 0 for each edge e ∈ E
- Find an s⇝t path P where each edge has f (e) < c(e)
- Augment flow along path P
- Repeat until you get stuck

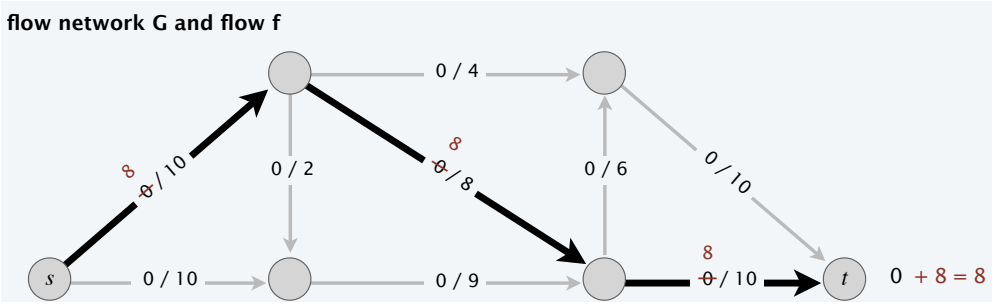




Initial Algorithm - 2

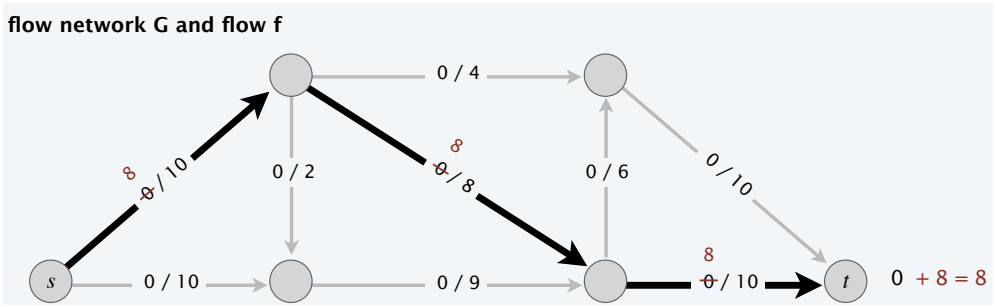
- Find path by BFS or DFS or any other search: O(|E|)
- Augment flow along path P: O(|V|)
  - Path capacity: minimum capacity along path
  - Add flow on path equal to path capacity

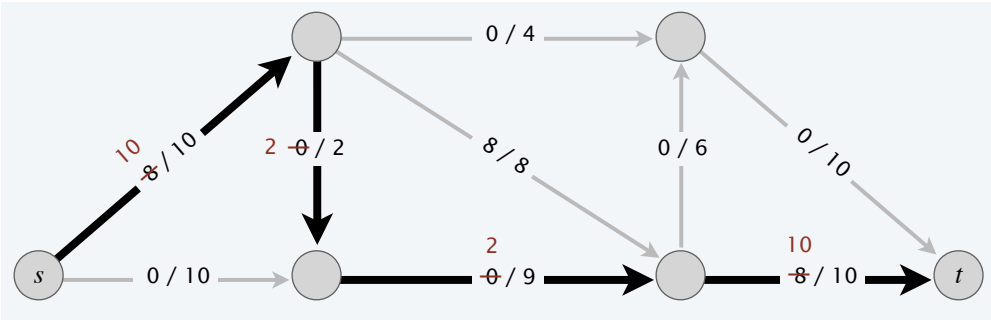




Initial Algorithm - 3

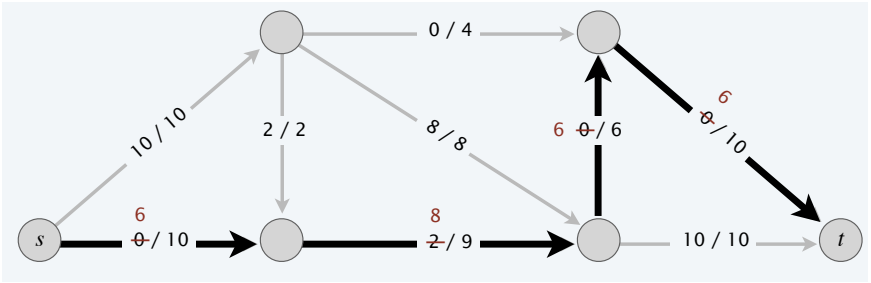
- Find another path by BFS or DFS or any other search using left-over capacity: O(|E|)
- Augment flow along path P: O(|E|)
  - Path capacity: minimum left-over capacity along path
  - Add flow on path equal to path capacity

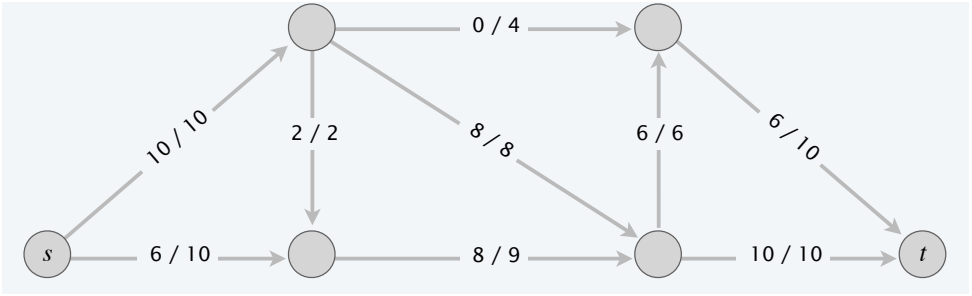




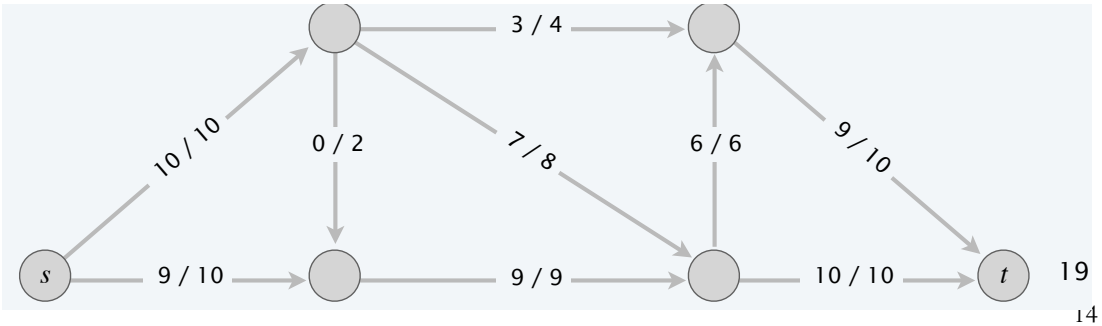
Initial Algorithm - 4

- Find another path by BFS or DFS or any other search using left-over capacity: O(|E|)
- No more paths … Max flow is 16 (Not!)



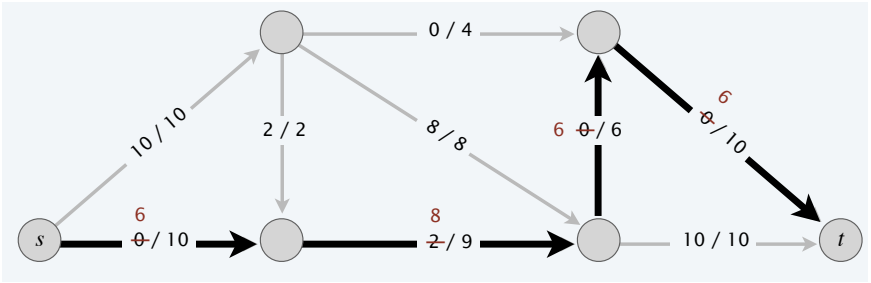


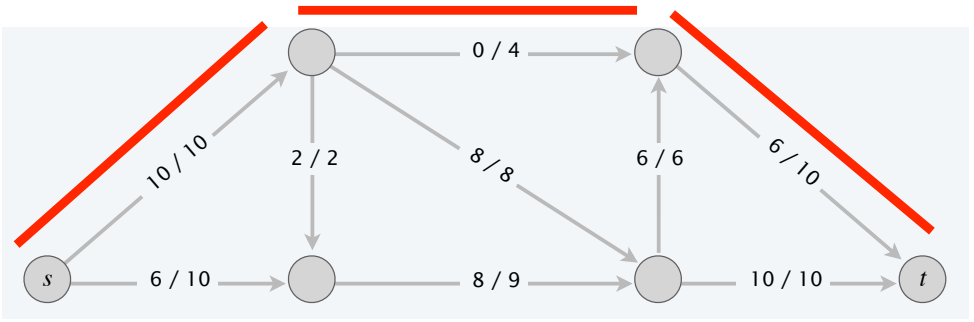
- Better solution: 19



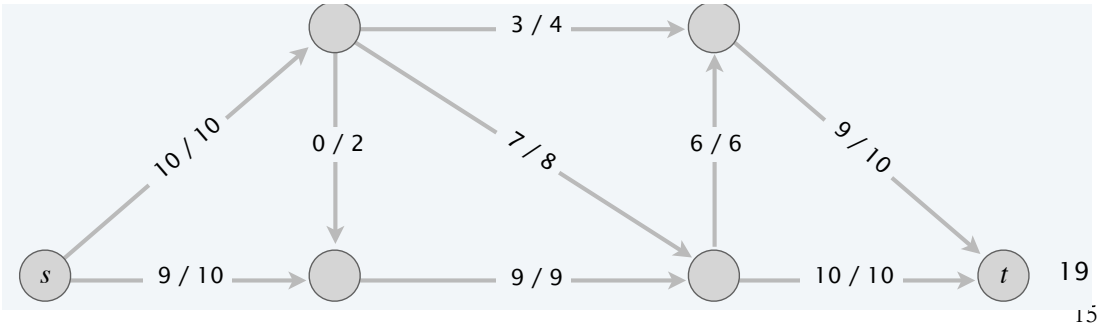
What happened? Too Greedy

- Greedy algorithm: no backtracking is insufficient
- Paths chose early may not be correct paths!



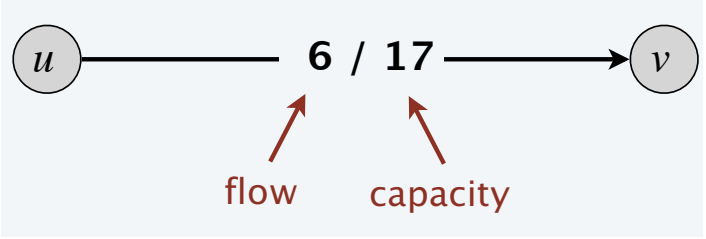


- Better solution: 19



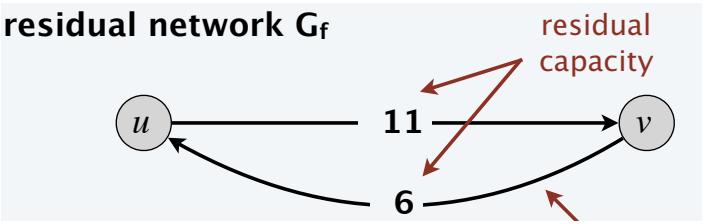
How to Backtrack

- Key concept: Residual network given a flow f
- Original edge. e = (u, v) $\in$ E
  - Flow f(u,v) > 0, capacity c(u,v)



- Want to create a way to "undo" assigned flow
  - Create a reverse edge (v,u) with capacity c(v,u) = f(u,v)
  - Flow on this reverse edge reverses the existing flow f(u,v)
- Residual network $G_f = (V, E_f)$
  - Capacity on residual network edges:

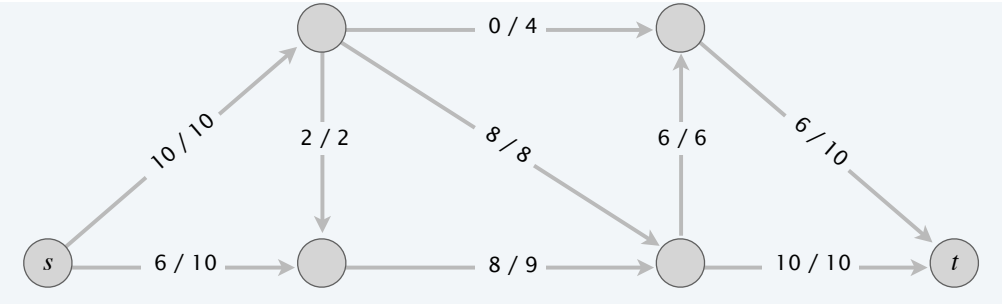
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(v,u) & (v,u) \in E \end{cases}$$



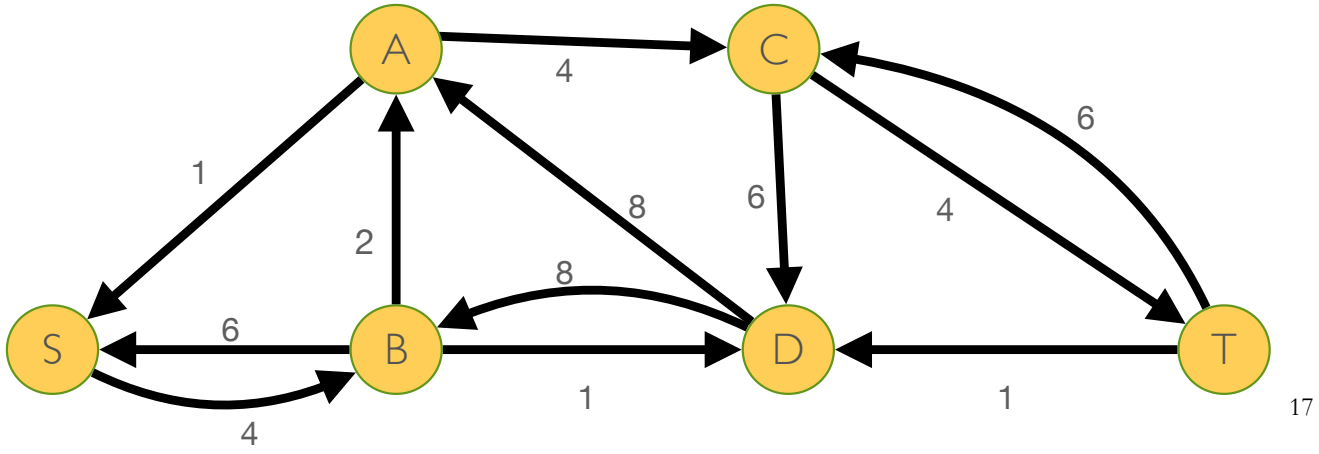
- $E_f = \{(u,v) \in E : f(u,v) < c(u,v)\} \cup \{(v,u) : (u,v) \in E, f(u,v) > 0\}$

Residual Network Example

- Previous solution using greedy algorithm



- Residual network has capacity



Augmenting Paths

- An augmenting path is a simple s—>t path in the residual network G_f
- The bottleneck capacity bottleneck(P, G_f) of an augmenting path P is the minimum residual capacity of any edge in P
- For bottleneck capacity δ, we can augment flow on P to modify f:
  - If $(u, v) \in P, (u, v) \in E$, augment $f'(u, v) = f(u, v) + \delta$
  - If $(v, u) \in P, (u, v) \in E$, this is a reverse edge; reduce $f'(u, v) = f(u, v) - \delta$
- Key property. Let f be a flow and let P be an augmenting path in G_f. Let δ be the flow on P (bottleneck capacity). Then, the augmented f' is a feasible flow in the original graph, and $val(f') = val(f) + \delta$

Ford-Fulkerson Algorithm ('56)

- Start with flow f(u,v) = 0, $(u, v) \in E$. Form the residual network $G_f = G$
- While there exists an s —> t path P in the residual network
  - Compute residual capacity δ on P, and augment flow f using flow δ on path P
  - Update residual network using new flow f, as G_f
- When no path can be found, return flow f

- Properties of algorithm:
  - At each iteration, flow increases
  - If capacities are integers, this must terminate: flow out of s increases by at least 1 at each iteration, and has upper limit as the sum of capacities of edges leaving s

Flows and Cuts

- Let f be any flow and let (A, B) be any st-cut. Then, the value of the flow f equals the net flow across the cut (A, B)

$$val(f) = \sum_{(u,v):u\in A,v\in B} f(u\,.\,v) - \sum_{(v,w):v\in B,w\in A} f(v,w)$$

- Why?

$$val(f) = \sum_{(s,v)\in E} f(s,v) - \sum_{(w,s)\in E} f(w,s) = \sum_{u\in A}\Big(\sum_{(u,v)\in E} f(u,v) - \sum_{(w,u)\in E} f(w,u)\Big)$$

(conservation of flow)

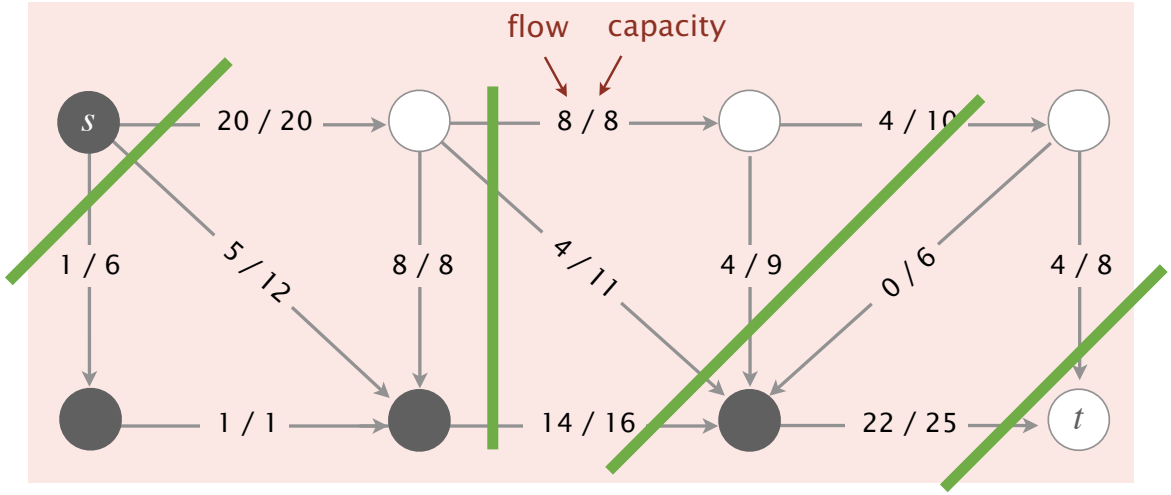
$$= \sum_{(u,v):u\in A,v\in B} f(u\,.\,v) - \sum_{(v,w):v\in B,w\in A} f(v,w)$$ (cancellations)

Flows and Cuts

- Let f be any flow and let (A, B) be any st-cut. Then, the value of the flow f equals the net flow across the cut (A, B)

$$val(f) = \sum_{(u,v):u\in A,v\in B} f(u\,.\,v) - \sum_{(v,w):v\in B,w\in A} f(v,w)$$

-



Flows and Cuts: Weak Duality

- Let f be any flow and let (A, B) be any st-cut. Then, the value of the flow f equals the net flow across the cut (A, B)

$$val(f) = \sum_{(u,v):u\in A,v\in B} f(u\,.\,v) - \sum_{(v,w):v\in B,w\in A} f(v,w)$$

- This implies that, for any feasible flow, and any st-cut (A,B), $val(f) \leq cap(A,B)$

$$val(f) = \sum_{(u,v):u\in A,v\in B} f(u\,.\,v) - \sum_{(v,w):v\in B,w\in A} f(v,w) \leq \sum_{(u,v):u\in A,v\in B} f(u\,.\,v)$$

$$\leq \sum_{(u,v):u\in A,v\in B} c(u\,.\,v) = cap(A,B)$$

Ford-Fulkerson Algorithm

- Proof of correctness (for integer capacities):

  Let f be flow at end, when no augmenting path left in the residual graph; let A be set of nodes reachable from s, and B = V - A. Then, (A,B) is a st-cut in the residual graph

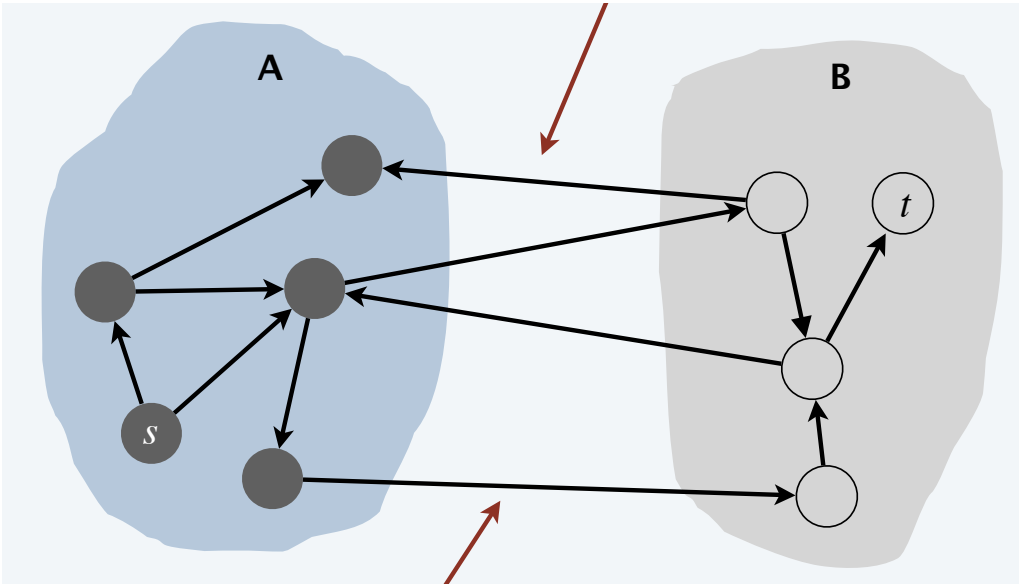
  This means every $(u, v) \in E, u \in A, v \in B$ has $f(u, v) = c(u, v)$, and flow on every $(v, w) \in E, w \in A, v \in B$ has $f(v, w) = 0$

  The max flow is bounded by the capacity of any st-cut, so max flow $\leq \sum_{(u,v):u \in A, v \in B} c(u, v)$

  But flow f has property that $\sum_{(u,v):u \in A, v \in B} f(u, v) = \sum_{(u,v):u \in A, v \in B} c(u, v)$ so it satisfies upper bound, hence it is as large as possible and maximal

Max-Cut, Min Flow

- Max-flow min-cut theorem: Value of a max flow = capacity of a min cut
- Augmenting path theorem: A flow f is a max flow if and only if there exists no augmenting paths in its residual graph
  - At optimality, edges from B to A must have 0 flow
  - At optimality, edges from A to B must have flow equal to capacity
  - Can compute a min cut from a max flow by letting A = nodes reachabl from s in residual graph of f



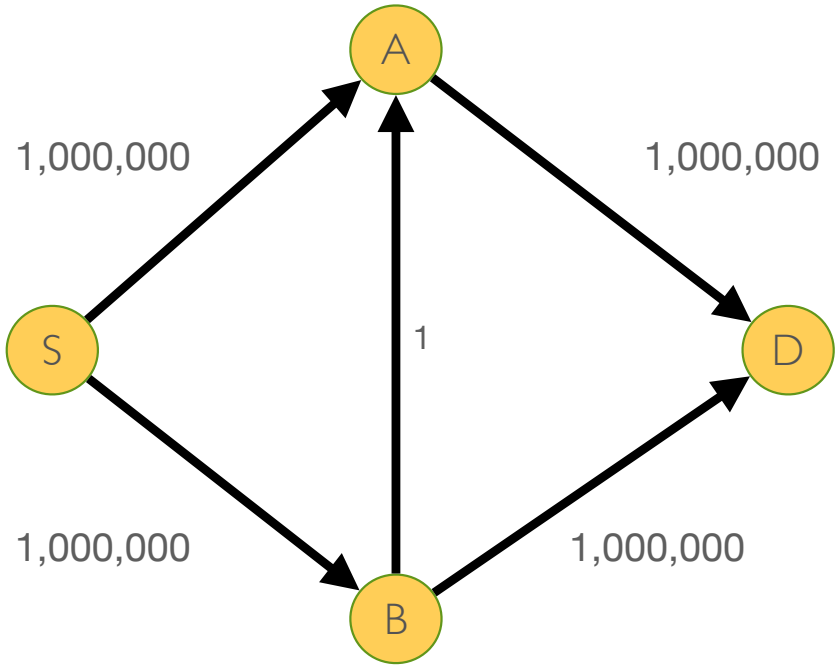
Ford-Fulkerson Algorithm

- Properties:
  - For integer capacities, all the flows in Ford-Fulkerson are integers
  - Each augmenting path iteration is O(|E|) (use BFS or DFS)
  - Each augmenting iteration increases flow by at least 1 unit
  - Let C be maximum capacity of any edge on graph. Then, value of max flow is less than |V|C

- Overall complexity O(|V||E| C)
  - Not a polynomial complexity (C can be very large…)

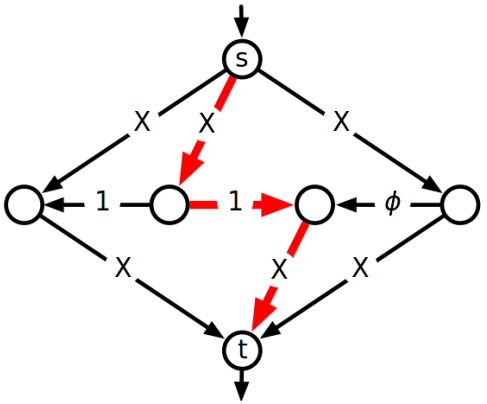
Ford-Fulkerson Algorithm

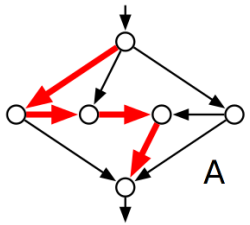
- Bad Case:
  - Optimal flow 2,000,000
  - Can do 2,000,000 iterations of Ford-Fulkerson if we pick wrong augmenting paths

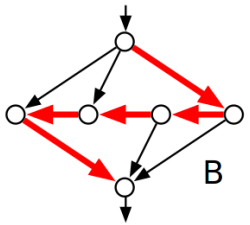


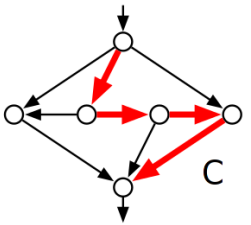
Ford-Fulkerson Algorithm

- Worse Case: real-valued capacities (Wikipedia)
  - $\phi = (\sqrt{5} - 1)/2$, $\phi^2 = 1 - \phi$
  - Can iterate forever; start with center path of capacity 1 then:
    - Augment B: cap ϕ
    - Augment C: cap ϕ
    - Augment B: cap ϕ^2
    - Augment A: cap ϕ^2
  - In limit, won't converge to max flow!







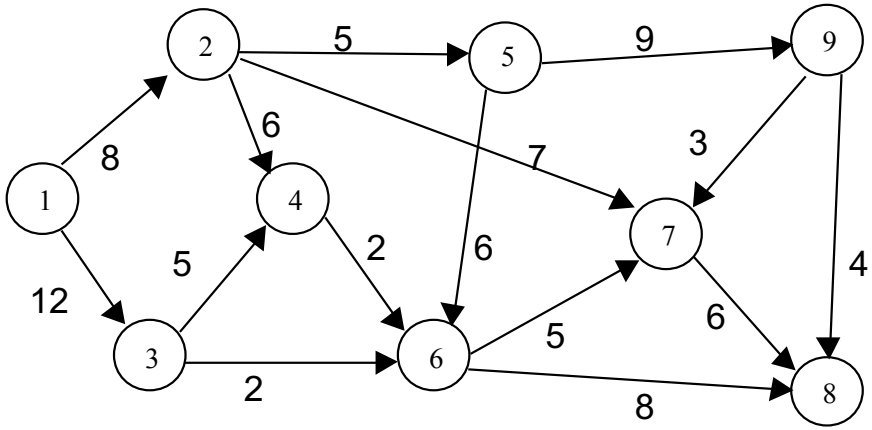


Easy Fix: Edmonds-Karp ('72)

- Choose augmenting paths in residual network that have least number of edges
  - Use BFS to find augmenting paths!
- Nice properties:
  - Length of augmenting path does not decrease in iterations
  - After at most |E| iterations, length of augmenting path must have increased
  - Maximum length of augmenting path is |V|-1
  - Bound on number of augmenting paths: $O(|V||E|)$
  - Bound on complexity: $O(|V||E|^2)$
- Another approach: choose augmentations with sufficient capacity
  - Capacity scaling used in best algorithms today

Example

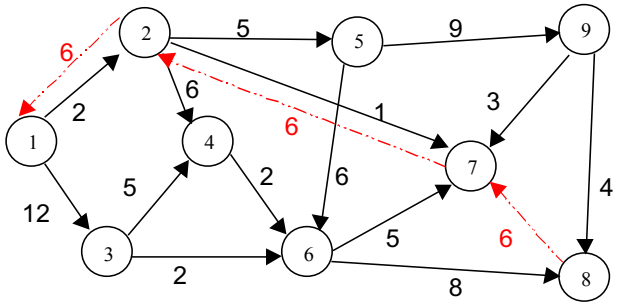
- Numbers are capacities, s = 1, t = 8



- BFS: augment 1—>2—>7—>8, capacity 6

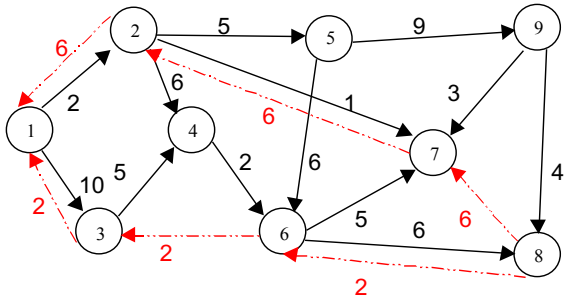
Example - 2

- Residual network



- BFS 2: augment 1—>3—>6—>8, capacity 2

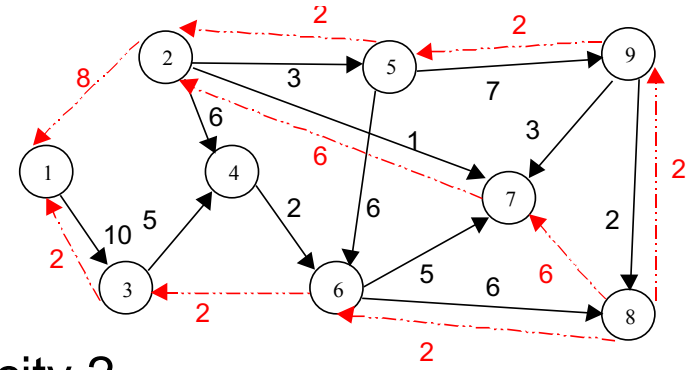
- Residual network



- BFS 3: augment 1—>2—>5—>9—>8, capacity 2

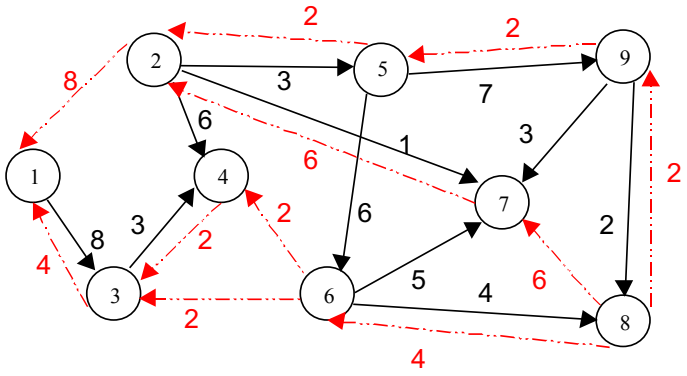
Example - 3

- Residual network



- BFS 4: augment 1—>3—>4—->6—>8, capacity 2

- Residual network



- No augmentation possible: Max-flow = 12
  Min Cut A = {1,3,4}